Test Your Knowledge: Answers

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

CHAPTER 14

Iterations and Comprehensions, Part 1

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

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

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

Iterators: A First Look

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

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

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

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

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

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

The Iteration Protocol: File Iterators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Manual Iteration: iter and next

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Other Built-in Type Iterators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

List Comprehensions: A First Look

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

In the previous chapter, we learned how to use `range` to change a list as we step across it:

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

>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

This works, but as I mentioned there, it may not be the optimal "best-practice" approach in Python. Today, the list comprehension expression makes many such prior use cases obsolete. Here, for example, we can replace the loop with a single expression that produces the desired result list:

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

The net result is the same, but it requires less coding on our part and is likely to run substantially faster. The list comprehension isn't exactly the same as the `for` loop statement version because it makes a *new* list object (which might matter if there are multiple references to the original list), but it's close enough for most applications and is a common and convenient enough approach to merit a closer look here.

List Comprehension Basics

We met the list comprehension briefly in Chapter 4. Syntactically, its syntax is derived from a construct in set theory notation that applies an operation to each item in a set, but you don't have to know set theory to use this tool. In Python, most people find that a list comprehension simply looks like a backward `for` loop.

To get a handle on the syntax, let's dissect the prior section's example in more detail:

```
>>> L = [x + 10 for x in L]
```

List comprehensions are written in square brackets because they are ultimately a way to construct a new list. They begin with an arbitrary expression that we make up, which uses a loop variable that we make up (`x + 10`). That is followed by what you should now recognize as the header of a `for` loop, which names the loop variable, and an iterable object (`for x in L`).

To run the expression, Python executes an iteration across `L` inside the interpreter, assigning `x` to each item in turn, and collects the results of running the items through the expression on the left side. The result list we get back is exactly what the list comprehension says—a new list containing `x + 10`, for every `x` in `L`.

Technically speaking, list comprehensions are never really required because we can always build up a list of expression results manually with `for` loops that append results as we go:

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[21, 22, 23, 24, 25]
```

In fact, this is exactly what the list comprehension does internally.

However, list comprehensions are more concise to write, and because this code pattern of building up result lists is so common in Python work, they turn out to be very handy in many contexts. Moreover, list comprehensions can run much faster than manual `for` loop statements (often roughly twice as fast) because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code; especially for larger data sets, there is a major performance advantage to using them.

Using List Comprehensions on Files

Let's work through another common use case for list comprehensions to explore them in more detail. Recall that the file object has a `readlines` method that loads the file into a list of line strings all at once:

```
>>> f = open('script1.py')
>>> lines = f.readlines()
```

```
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
```

This works, but the lines in the result all include the newline character (`\n`) at the end. For many programs, the newline character gets in the way—we have to be careful to avoid double-spacing when printing, and so on. It would be nice if we could get rid of these newlines all at once, wouldn't it?

Any time we start thinking about performing an operation on each item in a sequence, we're in the realm of list comprehensions. For example, assuming the variable `lines` is as it was in the prior interaction, the following code does the job by running each line in the list through the string `rstrip` method to remove whitespace on the right side (a `line[:-1]` slice would work, too, but only if we can be sure all lines are properly terminated):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This works as planned. Because list comprehensions are an iteration context just like `for` loop statements, though, we don't even have to open the file ahead of time. If we open it inside the expression, the list comprehension will automatically use the iteration protocol we met earlier in this chapter. That is, it will read one line from the file at a time by calling the file's `next` method, run the line through the `rstrip` expression, and add it to the result list. Again, we get what we ask for—the `rstrip` result of a line, for every line in the file:

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This expression does a lot implicitly, but we're getting a lot of work for free here—Python scans the file and builds a list of operation results automatically. It's also an efficient way to code this operation: because most of this work is done inside the Python interpreter, it is likely much faster than an equivalent `for` statement. Again, especially for large files, the speed advantages of list comprehensions can be significant.

Besides their efficiency, list comprehensions are also remarkably expressive. In our example, we can run any string operation on a file's lines as we iterate. Here's the list comprehension equivalent to the file iterator uppercase example we met earlier, along with a few others (the method chaining in the second of these examples works because string methods return a new string, to which we can apply another string method):

```
>>> [line.upper() for line in open('script1.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> [line.rstrip().upper() for line in open('script1.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(2 ** 33)']

>>> [line.split() for line in open('script1.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(2', '**','33)']]
```

```
>>> [line.replace(' ', '!') for line in open('script1.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(2!**!33)\n']

>>> [('sys' in line, line[0]) for line in open('script1.py')]
[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]
```

Extended List Comprehension Syntax

In fact, list comprehensions can be even more advanced in practice. As one particularly useful extension, the `for` loop nested in the expression can have an associated `if` clause to filter out of the result items for which the test is not true.

For example, suppose we want to repeat the prior section's file-scanning example, but we need to collect only lines that begin with the letter *p* (perhaps the first character on each line is an action code of some sort). Adding an `if` filter clause to our expression does the trick:

```
>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>> lines
['print(sys.path)', 'print(2 ** 33)']
```

Here, the `if` clause checks each line read from the file to see whether its first character is *p*; if not, the line is omitted from the result list. This is a fairly big expression, but it's easy to understand if we translate it to its simple `for` loop statement equivalent. In general, we can always translate a list comprehension to a `for` statement by appending as we go and further indenting each successive part:

```
>>> res = []
>>> for line in open('script1.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(2 ** 33)']
```

This `for` statement equivalent works, but it takes up four lines instead of one and probably runs substantially slower.

List comprehensions can become even more complex if we need them to—for instance, they may contain nested loops, coded as a series of `for` clauses. In fact, their full syntax allows for any number of `for` clauses, each of which can have an optional associated `if` clause (we'll be more formal about their syntax in Chapter 20).

For example, the following builds a list of the concatenation of `x + y` for every `x` in one string and every `y` in another. It effectively collects the permutation of the characters in two strings:

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Again, one way to understand this expression is to convert it to statement form by indenting its parts. The following is an equivalent, but likely slower, alternative way to achieve the same effect:

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Beyond this complexity level, though, list comprehension expressions can often become too compact for their own good. In general, they are intended for simple types of iterations; for more involved work, a simpler `for` statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it's probably not a good idea.

We'll revisit list comprehensions in Chapter 20, in the context of functional programming tools; as we'll see, they turn out to be just as related to functions as they are to looping statements.

Other Iteration Contexts

Later in the book, we'll see that user-defined classes can implement the iteration protocol too. Because of this, it's sometimes important to know which built-in tools make use of it—any tool that employs the iteration protocol will automatically work on any built-in type or user-defined class that provides it.

So far, I've been demonstrating iterators in the context of the `for` loop statement, because this part of the book is focused on statements. Keep in mind, though, that every tool that scans from left to right across objects uses the iteration protocol. This includes the `for` loops we've seen:

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

However, list comprehensions, the `in` membership test, the `map` built-in function, and other built-ins such as the `sorted` and `zip` calls also leverage the iteration protocol. When applied to a file, all of these use the file object's iterator automatically to scan line by line:

```
>>> uppers = [line.upper() for line in open('script1.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']
```

```
>>> map(str.upper, open('script1.py'))        # map is an iterable in 3.0
<map object at 0x02660710>

>>> list( map(str.upper, open('script1.py')) )
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> 'y = 2\n' in open('script1.py')
False
>>> 'x = 2\n' in open('script1.py')
True
```

We introduced the `map` call used here in the preceding chapter; it's a built-in that applies a function call to each item in the passed-in iterable object. `map` is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also *returns* an iterable object itself in Python 3.0, so we must wrap it in a `list` call to force it to give us all its values at once; more on this change later in this chapter. Because `map`, like the list comprehension, is related to both `for` loops and functions, we'll also explore both again in Chapters 19 and 20.

Python includes various additional built-ins that process iterables, too: `sorted` sorts items in an iterable, `zip` combines items from iterables, `enumerate` pairs items in an iterable with relative positions, `filter` selects items for which a function is true, and `reduce` runs pairs of items in an iterable through a function. All of these accept iterables, and `zip`, `enumerate`, and `filter` also return an iterable in Python 3.0, like `map`. Here they are in action running the file's iterator automatically to scan line by line:

```
>>> sorted(open('script1.py'))
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']

>>> list(zip(open('script1.py'), open('script1.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(2 ** 33)\n')]

>>> list(enumerate(open('script1.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
(3, 'print(2 ** 33)\n')]

>>> list(filter(bool, open('script1.py')))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('script1.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(2 ** 33)\n'
```

All of these are iteration tools, but they have unique roles. We met `zip` and `enumerate` in the prior chapter; `filter` and `reduce` are in Chapter 19's functional programming domain, so we'll defer details for now.

We first saw the `sorted` function used here at work in Chapter 4, and we used it for dictionaries in Chapter 8. `sorted` is a built-in that employs the iteration protocol—it's like the original list `sort` method, but it returns the new sorted list as a result and runs

on any iterable object. Notice that, unlike `map` and others, `sorted` returns an actual *list* in Python 3.0 instead of an iterable.

Other built-in functions support the iteration protocol as well (but frankly, are harder to cast in interesting examples related to files). For example, the `sum` call computes the sum of all the numbers in any iterable; the `any` and `all` built-ins return `True` if any or all items in an iterable are `True`, respectively; and `max` and `min` return the largest and smallest item in an iterable, respectively. Like `reduce`, all of the tools in the following examples accept any iterable as an argument and use the iteration protocol to scan it, but return a single result:

```
>>> sum([3, 2, 4, 1, 5, 0])                # sum expects numbers only
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

Strictly speaking, the `max` and `min` functions can be applied to files as well—they automatically use the iteration protocol to scan the file and pick out the lines with the highest and lowest string values, respectively (though I'll leave valid use cases to your imagination):

```
>>> max(open('script1.py'))                 # Line with max/min string value
'x = 2\n'
>>> min(open('script1.py'))
'import sys\n'
```

Interestingly, the iteration protocol is even more pervasive in Python today than the examples so far have demonstrated—*everything* in Python's built-in toolset that scans an object from left to right is defined to use the iteration protocol on the subject object. This even includes more esoteric tools such as the `list` and `tuple` built-in functions (which build new objects from iterables), the string `join` method (which puts a substring between strings contained in an iterable), and even sequence assignments. Consequently, all of these will also work on an open file and automatically read one line at a time:

```
>>> list(open('script1.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> tuple(open('script1.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n')

>>> '&&'.join(open('script1.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(2 ** 33)\n'

>>> a, b, c, d = open('script1.py')
>>> a, d
```

```
('import sys\n', 'print(2 ** 33)\n')

>>> a, *b = open('script1.py')                  # 3.0 extended form
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n'])
```

Earlier, we saw that the built-in `dict` call accepts an iterable `zip` result, too. For that matter, so does the `set` call, as well as the new *set* and *dictionary comprehension expressions* in Python 3.0, which we met in Chapters 4, 5, and 8:

```
>>> set(open('script1.py'))
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {line for line in open('script1.py')}
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {ix: line for ix, line in enumerate(open('script1.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(2 ** 33)\n'}
```

In fact, both set and dictionary comprehensions support the extended syntax of list comprehensions we met earlier in this chapter, including `if` tests:

```
>>> {line for line in open('script1.py') if line[0] == 'p'}
{'print(sys.path)\n', 'print(2 ** 33)\n'}

>>> {ix: line for (ix, line) in enumerate(open('script1.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(2 ** 33)\n'}
```

Like the list comprehension, both of these scan the file line by line and pick out lines that begin with the letter "p." They also happen to build sets and dictionaries in the end, but we get a lot of work "for free" by combining file iteration and comprehension syntax.

There's one last iteration context that's worth mentioning, although it's a bit of a preview: in Chapter 18, we'll learn that a special `*arg` form can be used in function calls to unpack a collection of values into individual arguments. As you can probably predict by now, this accepts any iterable, too, including files (see Chapter 18 for more details on the call syntax):

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4])                    # Unpacks into arguments
1&2&3&4

>>> f(*open('script1.py'))              # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(2 ** 33)
```

In fact, because this argument-unpacking syntax in calls accepts iterables, it's also possible to use the `zip` built-in to *unzip* zipped tuples, by making prior or nested `zip` results

arguments for another `zip` call (warning: you probably shouldn't read the following example if you plan to operate heavy machinery anytime soon!):

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))                  # Zip tuples: returns an iterable
[(1, 3), (2, 4)]
>>>
>>> A, B = zip(*zip(X, Y))           # Unzip a zip!
>>> A
(1, 2)
>>> B
(3, 4)
```

Still other tools in Python, such as the `range` built-in and dictionary view objects, return iterables instead of processing them. To see how these have been absorbed into the iteration protocol in Python 3.0 as well, we need to move on to the next section.

New Iterables in Python 3.0

One of the fundamental changes in Python 3.0 is that it has a stronger emphasis on iterators than 2.X. In addition to the iterators associated with built-in types such as files and dictionaries, the dictionary methods `keys`, `values`, and `items` return iterable objects in Python 3.0, as do the built-in functions `range`, `map`, `zip`, and `filter`. As shown in the prior section, the last three of these functions both return iterators and process them. All of these tools produce results on demand in Python 3.0, instead of constructing result lists as they do in 2.6.

Although this saves memory space, it can impact your coding styles in some contexts. In various places in this book so far, for example, we've had to wrap up various function and method call results in a `list(...)` call in order to force them to produce all their results at once:

```
>>> zip('abc', 'xyz')                # An iterable in Python 3.0 (a list in 2.6)
<zip object at 0x02E66710>

>>> list(zip('abc', 'xyz'))          # Force list of results in 3.0 to display
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

This isn't required in 2.6, because functions like `zip` return lists of results. In 3.0, though, they return iterable objects, producing results on demand. This means extra typing is required to display the results at the interactive prompt (and possibly in some other contexts), but it's an asset in larger programs—delayed evaluation like this conserves memory and avoids pauses while large result lists are computed. Let's take a quick look at some of the new 3.0 iterables in action.

The range Iterator

We studied the `range` built-in's basic behavior in the prior chapter. In 3.0, it returns an iterator that generates numbers in the range on demand, instead of building the result list in memory. This subsumes the older 2.X `xrange` (see the upcoming version skew note), and you must use `list(range(...))` to force an actual range list if one is needed (e.g., to display results):

```
C:\\misc> c:\python30\python
>>> R = range(10)                  # range returns an iterator, not a list
>>> R
range(0, 10)

>>> I = iter(R)                    # Make an iterator from the range
>>> next(I)                        # Advance to next result
0                                  # What happens in for loops, comprehensions, etc.
>>> next(I)
1
>>> next(I)
2

>>> list(range(10))                # To force a list if required
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Unlike the list returned by this call in 2.X, `range` objects in 3.0 support only iteration, indexing, and the `len` function. They do not support any other sequence operations (use `list(...)` if you require more list tools):

```
>>> len(R)                         # range also does len and indexing, but no others
10
>>> R[0]
0
>>> R[-1]
9

>>> next(I)                        # Continue taking from iterator, where left off
3
>>> I.__next__()                   # .next() becomes .__next__(), but use new next()
4
```

Version skew note: Python 2.X also has a built-in called `xrange`, which is like `range` but produces items on demand instead of building a list of results in memory all at once. Since this is exactly what the new iterator-based `range` does in Python 3.0, `xrange` is no longer available in 3.0—it has been subsumed. You may still see it in 2.X code, though, especially since `range` builds result lists there and so is not as efficient in its memory usage. As noted in a sidebar in the prior chapter, the `file.xreadlines()` method used to minimize memory use in 2.X has been dropped in Python 3.0 for similar reasons, in favor of file iterators.

The map, zip, and filter Iterators

Like `range`, the `map`, `zip`, and `filter` built-ins also become iterators in 3.0 to conserve space, rather than producing a result list all at once in memory. All three not only process iterables, as in 2.X, but also return iterable results in 3.0. Unlike `range`, though, they are their own iterators—after you step through their results once, they are exhausted. In other words, you can't have multiple iterators on their results that maintain different positions in those results.

Here is the case for the `map` built-in we met in the prior chapter. As with other iterators, you can force a list with `list(...)` if you really need one, but the default behavior can save substantial space in memory for large result sets:

```
>>> M = map(abs, (-1, 0, 1))            # map returns an iterator, not a list
>>> M
<map object at 0x0276B890>
>>> next(M)                             # Use iterator manually: exhausts results
1                                       # These do not support len() or indexing
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration

>>> for x in M: print(x)                # map iterator is now empty: one pass only
...

>>> M = map(abs, (-1, 0, 1))            # Make a new iterator to scan again
>>> for x in M: print(x)                # Iteration contexts auto call next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1)))          # Can force a real list if needed
[1, 0, 1]
```

The `zip` built-in, introduced in the prior chapter, returns iterators that work the same way:

```
>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip is the same: a one-pass iterator
>>> Z
<zip object at 0x02770EE0>

>>> list(Z)
[(1, 10), (2, 20), (3, 30)]

>>> for pair in Z: print(pair)          # Exhausted after one pass
...

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)          # Iterator used automatically or manually
...
(1, 10)
```

```
(2, 20)
(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)
```

The `filter` built-in, which we'll study in the next part of this book, is also analogous. It returns items in an iterable for which a passed-in function returns `True` (as we've learned, in Python `True` includes nonempty objects):

```
>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x0269C6D0>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']
```

Like most of the tools discussed in this section, `filter` both accepts an iterable to process and returns an iterable to generate results in 3.0.

Multiple Versus Single Iterators

It's interesting to see how the `range` object differs from the built-ins described in this section—it supports `len` and indexing, it is not its own iterator (you make one with `iter` when iterating manually), and it supports multiple iterators over its result that remember their positions independently:

```
>>> R = range(3)                        # range allows multiple iterators
>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)
>>> next(I1)
0
>>> next(I1)
1
>>> I2 = iter(R)                        # Two iterators on one range
>>> next(I2)
0
>>> next(I1)                            # I1 is at a different spot than I2
2
```

By contrast, `zip`, `map`, and `filter` do not support multiple active iterators on the same result:

```
>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                        # Two iterators on one zip
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                            # I2 is at same spot as I1!
```

```
(3, 12)

>>> M = map(abs, (-1, 0, 1))                # Ditto for map (and filter)
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)
StopIteration

>>> R = range(3)                            # But range allows many iterators
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)
0
```

When we code our own iterable objects with classes later in the book (Chapter 29), we'll see that multiple iterators are usually supported by returning new objects for the `iter` call; a single iterator generally means an object returns itself. In Chapter 20, we'll also find that *generator functions and expressions* behave like `map` and `zip` instead of `range` in this regard, supporting a single active iteration. In that chapter, we'll see some subtle implications of one-shot iterators in loops that attempt to scan multiple times.

Dictionary View Iterators

As we saw briefly in Chapter 8, in Python 3.0 the dictionary `keys`, `values`, and `items` methods return iterable *view* objects that generate result items one at a time, instead of producing result lists all at once in memory. View items maintain the same physical ordering as that of the dictionary and reflect changes made to the underlying dictionary. Now that we know more about iterators, here's the rest of the story:

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

>>> K = D.keys()                            # A view object in 3.0, not a list
>>> K
<dict_keys object at 0x026D83C0>

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

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

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

As for all iterators, you can always force a 3.0 dictionary view to build a real list by passing it to the `list` built-in. However, this usually isn't required except to display results interactively or to apply list operations like indexing:

```
>>> K = D.keys()
>>> list(K)                              # Can still force a real list if needed
['a', 'c', 'b']

>>> V = D.values()                       # Ditto for values() and items() views
>>> V
<dict_values object at 0x026D8260>
>>> list(V)
[1, 3, 2]

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

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

In addition, 3.0 dictionaries still have iterators themselves, which return successive keys. Thus, it's not often necessary to call `keys` directly in this context:

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

>>> for key in D: print(key, end=' ')    # Still no need to call keys() to iterate
...                                      # But keys is an iterator in 3.0 too!
a c b
```

Finally, remember again that because `keys` no longer returns a list, the traditional coding pattern for scanning a dictionary by sorted keys won't work in 3.0. Instead, convert keys views first with a `list` call, or use the `sorted` call on either a keys view or the dictionary itself, as follows:

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

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

Other Iterator Topics

We'll learn more about both list comprehensions and iterators in Chapter 20, in conjunction with functions, and again in Chapter 29 when we study classes. As you'll see later:

- User-defined functions can be turned into iterable *generator functions*, with `yield` statements.
- List comprehensions morph into iterable *generator expressions* when coded in parentheses.
- User-defined classes are made iterable with `__iter__` or `__getitem__` *operator overloading*.

In particular, user-defined iterators defined with classes allow arbitrary objects and operations to be used in any of the iteration contexts we've met here.

Chapter Summary

In this chapter, we explored concepts related to looping in Python. We took our first substantial look at the *iteration protocol* in Python—a way for nonsequence objects to take part in iteration loops—and at *list comprehensions*. As we saw, a list comprehension is an expression similar to a `for` loop that applies another expression to all the items in any iterable object. Along the way, we also saw other built-in iteration tools at work and studied recent iteration additions in Python 3.0.

This wraps up our tour of specific procedural statements and related tools. The next chapter closes out this part of the book by discussing documentation options for Python code; documentation is also part of the general syntax model, and it's an important component of well-written programs. In the next chapter, we'll also dig into a set of exercises for this part of the book before we turn our attention to larger structures such as functions. As usual, though, let's first exercise what we've learned here with a quiz.

Test Your Knowledge: Quiz

1. How are `for` loops and iterators related?
2. How are `for` loops and list comprehensions related?
3. Name four iteration contexts in the Python language.
4. What is the best way to read line by line from a text file today?
5. What sort of weapons would you expect to see employed by the Spanish Inquisition?

Test Your Knowledge: Answers

1. The `for` loop uses the *iteration protocol* to step through items in the object across which it is iterating. It calls the object's `__next__` method (run by the `next` built-in) on each iteration and catches the `StopIteration` exception to determine when to stop looping. Any object that supports this model works in a `for` loop and in other iteration contexts.
2. Both are iteration tools. List comprehensions are a concise and efficient way to perform a common `for` loop task: collecting the results of applying an expression to all items in an iterable object. It's always possible to translate a list comprehension to a `for` loop, and part of the list comprehension expression looks like the header of a `for` loop syntactically.
3. Iteration contexts in Python include the `for` loop; list comprehensions; the `map` built-in function; the `in` membership test expression; and the built-in functions `sorted`, `sum`, `any`, and `all`. This category also includes the `list` and `tuple` built-ins, string `join` methods, and sequence assignments, all of which use the iteration protocol (the `__next__` method) to step across iterable objects one item at a time.
4. The best way to read lines from a text file today is to not read it explicitly at all: instead, open the file within an iteration context such as a `for` loop or list comprehension, and let the iteration tool automatically scan one line at a time by running the file's `next` method on each iteration. This approach is generally best in terms of coding simplicity, execution speed, and memory space requirements.
5. I'll accept any of the following as correct answers: fear, intimidation, nice red uniforms, a comfy chair, and soft pillows.

CHAPTER 15

The Documentation Interlude

This part of the book concludes with a look at techniques and tools used for documenting Python code. Although Python code is designed to be readable, a few well-placed human-readable comments can do much to help others understand the workings of your programs. Python includes syntax and tools to make documentation easier.

Although this is something of a tools-related concept, the topic is presented here partly because it involves Python's syntax model, and partly as a resource for readers struggling to understand Python's toolset. For the latter purpose, I'll expand here on documentation pointers first given in Chapter 4. As usual, in addition to the chapter quiz this concluding chapter ends with some warnings about common pitfalls and a set of exercises for this part of the text.

Python Documentation Sources

By this point in the book, you're probably starting to realize that Python comes with an amazing amount of prebuilt functionality—built-in functions and exceptions, predefined object attributes and methods, standard library modules, and more. And we've really only scratched the surface of each of these categories.

One of the first questions that bewildered beginners often ask is: how do I find information on all the built-in tools? This section provides hints on the various documentation sources available in Python. It also presents *documentation strings* (docstrings) and the *PyDoc* system that makes use of them. These topics are somewhat peripheral to the core language itself, but they become essential knowledge as soon as your code reaches the level of the examples and exercises in this part of the book.

As summarized in Table 15-1, there are a variety of places to look for information on Python, with generally increasing verbosity. Because documentation is such a crucial tool in practical programming, we'll explore each of these categories in the sections that follow.

Table 15-1. Python documentation sources

Form	Role
# comments	In-file documentation
The `dir` function	Lists of attributes available in objects
Docstrings: `__doc__`	In-file documentation attached to objects
PyDoc: The `help` function	Interactive help for objects
PyDoc: HTML reports	Module documentation in a browser
The standard manual set	Official language and library descriptions
Web resources	Online tutorials, examples, and so on
Published books	Commercially available reference texts

Comments

Hash-mark comments are the most basic way to document your code. Python simply ignores all the text following a `#` (as long as it's not inside a string literal), so you can follow this character with words and descriptions meaningful to programmers. Such comments are accessible only in your source files, though; to code comments that are more widely available, you'll need to use docstrings.

In fact, current best practice generally dictates that docstrings are best for larger functional documentation (e.g., "my file does this"), and `#` comments are best limited to smaller code documentation (e.g., "this strange expression does that"). More on docstrings in a moment.

The dir Function

The built-in `dir` function is an easy way to grab a list of all the attributes available inside an object (i.e., its methods and simpler data items). It can be called on any object that has attributes. For example, to find out what's available in the standard library's `sys` module, import it and pass it to `dir` (these results are from Python 3.0; they might vary slightly on 2.6):

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding',
...more names omitted...]
```

Only some of the many names are displayed here; run these statements on your machine to see the full list.

To find out what attributes are provided in built-in object types, run `dir` on a literal (or existing instance) of the desired type. For example, to see list and string attributes, you can pass empty objects:

```
>>> dir([])
['__add__', '__class__', '__contains__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>> dir('')
['__add__', '__class__', '__contains__', ...more...
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', '
maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
...more names omitted...]
```

`dir` results for any built-in type include a set of attributes that are related to the implementation of that type (technically, operator overloading methods); they all begin and end with double underscores to make them distinct, and you can safely ignore them at this point in the book.

Incidentally, you can achieve the same effect by passing a type name to `dir` instead of a literal:

```
>>> dir(str) == dir('')          # Same result as prior example
True
>>> dir(list) == dir([])
True
```

This works because names like `str` and `list` that were once type converter functions are actually names of types in Python today; calling one of these invokes its constructor to generate an instance of that type. I'll have more to say about constructors and operator overloading methods when we discuss classes in Part VI.

The `dir` function serves as a sort of memory-jogger—it provides a list of attribute names, but it does not tell you anything about what those names mean. For such extra information, we need to move on to the next documentation source.

Docstrings: __doc__

Besides # comments, Python supports documentation that is automatically attached to objects and retained at runtime for inspection. Syntactically, such comments are coded as strings at the tops of module files and function and class statements, before any other executable code (# comments are OK before them). Python automatically stuffs the strings, known as *docstrings*, into the `__doc__` attributes of the corresponding objects.

User-defined docstrings

For example, consider the following file, *docstrings.py*. Its docstrings appear at the beginning of the file and at the start of a function and a class within it. Here, I've used triple-quoted block strings for multiline comments in the file and the function, but any sort of string will work. We haven't studied the `def` or `class` statements in detail yet, so ignore everything about them except the strings at their tops:

```
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
    """
    function documentation
    can we have your liver then?
    """
    return x ** 2          # square

class Employee:
    "class documentation"
    pass

print(square(4))
print(square.__doc__)
```

The whole point of this documentation protocol is that your comments are retained for inspection in `__doc__` attributes after the file is imported. Thus, to display the docstrings associated with the module and its objects, we simply import the file and print their `__doc__` attributes, where Python has saved the text:

```
>>> import docstrings
16

    function documentation
    can we have your liver then?

>>> print(docstrings.__doc__)

Module documentation
Words Go Here

>>> print(docstrings.square.__doc__)

    function documentation
    can we have your liver then?

>>> print(docstrings.Employee.__doc__)
    class documentation
```

Note that you will generally want to use `print` to print docstrings; otherwise, you'll get a single string with embedded newline characters.

You can also attach docstrings to *methods* of classes (covered in Part VI), but because these are just `def` statements nested in `class` statements, they're not a special case. To fetch the docstring of a method function inside a class within a module, you would simply extend the path to go through the class: `module.class.method.__doc__` (we'll see an example of method docstrings in Chapter 28).

Docstring standards

There is no broad standard about what should go into the text of a docstring (although some companies have internal standards). There have been various markup language and template proposals (e.g., HTML or XML), but they don't seem to have caught on in the Python world. And frankly, convincing Python programmers to document their code using handcoded HTML is probably not going to happen in our lifetimes!

Documentation tends to have a low priority amongst programmers in general. Usually, if you get any comments in a file at all, you count yourself lucky. I strongly encourage you to document your code liberally, though—it really is an important part of well-written programs. The point here is that there is presently no standard on the structure of docstrings; if you want to use them, anything goes today.

Built-in docstrings

As it turns out, built-in modules and objects in Python use similar techniques to attach documentation above and beyond the attribute lists returned by `dir`. For example, to see an actual human-readable description of a built-in module, import it and print its `__doc__` string:

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...more text omitted...
```

Functions, classes, and methods within built-in modules have attached descriptions in their `__doc__` attributes as well:

```
>>> print(sys.getrefcount.__doc__)
getrefcount(object) -> integer

Return the reference count of object.  The count returned is generally
one higher than you might expect, because it includes the (temporary)
...more text omitted...
```

You can also read about built-in functions via their docstrings:

```
>>> print(int.__doc__)
int(x[, base]) -> integer

Convert a string or number to an integer, if possible.  A floating
point argument will be truncated towards zero (this does not include a
...more text omitted...

>>> print(map.__doc__)
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
```

You can get a wealth of information about built-in tools by inspecting their docstrings this way, but you don't have to—the `help` function, the topic of the next section, does this automatically for you.

PyDoc: The help Function

The docstring technique proved to be so useful that Python now ships with a tool that makes docstrings even easier to display. The standard *PyDoc* tool is Python code that knows how to extract docstrings and associated structural information and format them into nicely arranged reports of various types. Additional tools for extracting and formatting docstrings are available in the open source domain (including tools that may support structured text—search the Web for pointers), but Python ships with PyDoc in its standard library.

There are a variety of ways to launch PyDoc, including command-line script options (see the Python library manual for details). Perhaps the two most prominent PyDoc interfaces are the built-in `help` function and the PyDoc GUI/HTML interface. The `help` function invokes PyDoc to generate a simple textual report (which looks much like a "manpage" on Unix-like systems):

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:

getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object.  The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    ...more omitted...
```

Note that you do not have to import `sys` in order to call `help`, but you do have to import `sys` to get help on `sys`; it expects an object reference to be passed in. For larger objects such as modules and classes, the `help` display is broken down into multiple sections, a few of which are shown here. Run this interactively to see the full report:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...more omitted...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it in builtins.
        ...more omitted...

DATA
    __stderr__ = <io.TextIOWrapper object at 0x0236E950>
    __stdin__ = <io.TextIOWrapper object at 0x02366550>
    __stdout__ = <io.TextIOWrapper object at 0x02366E30>
    ...more omitted...
```

Some of the information in this report is docstrings, and some of it (e.g., function call patterns) is structural information that PyDoc gleans automatically by inspecting objects' internals, when available. You can also use `help` on built-in functions, methods, and types. To get help for a built-in type, use the type name (e.g., `dict` for dictionary, `str` for string, `list` for list). You'll get a large display that describes all the methods available for that type:

```
>>> help(dict)
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary.
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 ...more omitted...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    ...more omitted...

>>> help(ord)
```

```
Help on built-in function ord in module builtins:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Finally, the `help` function works just as well on your modules as it does on built-ins. Here it is reporting on the *docstrings.py* file we coded earlier. Again, some of this is docstrings, and some is information automatically extracted by inspecting objects' structures:

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    function documentation
    can we have your liver then?

>>> help(docstrings.Employee)
Help on class Employee in module docstrings:

class Employee(builtins.object)
 |  class documentation
 |
 |  Data descriptors defined here:
 ...more omitted...

>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\misc\docstrings.py

DESCRIPTION
    Module documentation
    Words Go Here

CLASSES
    builtins.object
        Employee

    class Employee(builtins.object)
     |  class documentation
     |
     |  Data descriptors defined here:
     ...more omitted...

FUNCTIONS
    square(x)
        function documentation
```

```
            can we have your liver then?

    DATA
        spam = 40
```

PyDoc: HTML Reports

The `help` function is nice for grabbing documentation when working interactively. For a more grandiose display, however, PyDoc also provides a GUI interface (a simple but portable Python/tkinter script) and can render its report in HTML page format, viewable in any web browser. In this mode, PyDoc can run locally or as a remote server in client/server mode; reports contain automatically created hyperlinks that allow you to click your way through the documentation of related components in your application.

To start PyDoc in this mode, you generally first launch the search engine GUI captured in Figure 15-1. You can start this either by selecting the "Module Docs" item in Python's Start button menu on Windows, or by launching the *pydoc.py* script in Python's standard library directory: *Lib* on Windows (run `pydoc.py` with a `-g` command-line argument). Enter the name of a module you're interested in, and press the Enter key; PyDoc will march down your module import search path (`sys.path`) looking for references to the requested module.

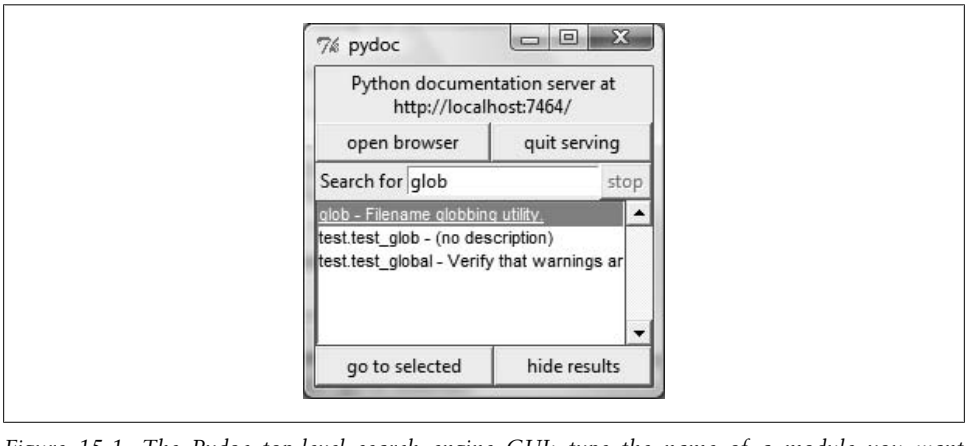

Figure 15-1. The Pydoc top-level search engine GUI: type the name of a module you want documentation for, press Enter, select the module, and then press "go to selected" (or omit the module name and press "open browser" to see all available modules).

Once you've found a promising entry, select it and click "go to selected." PyDoc will spawn a web browser on your machine to display the report rendered in HTML format. Figure 15-2 shows the information PyDoc displays for the built-in `glob` module.

Notice the hyperlinks in the Modules section of this page—you can click these to jump to the PyDoc pages for related (imported) modules. For larger pages, PyDoc also generates hyperlinks to sections within the page.

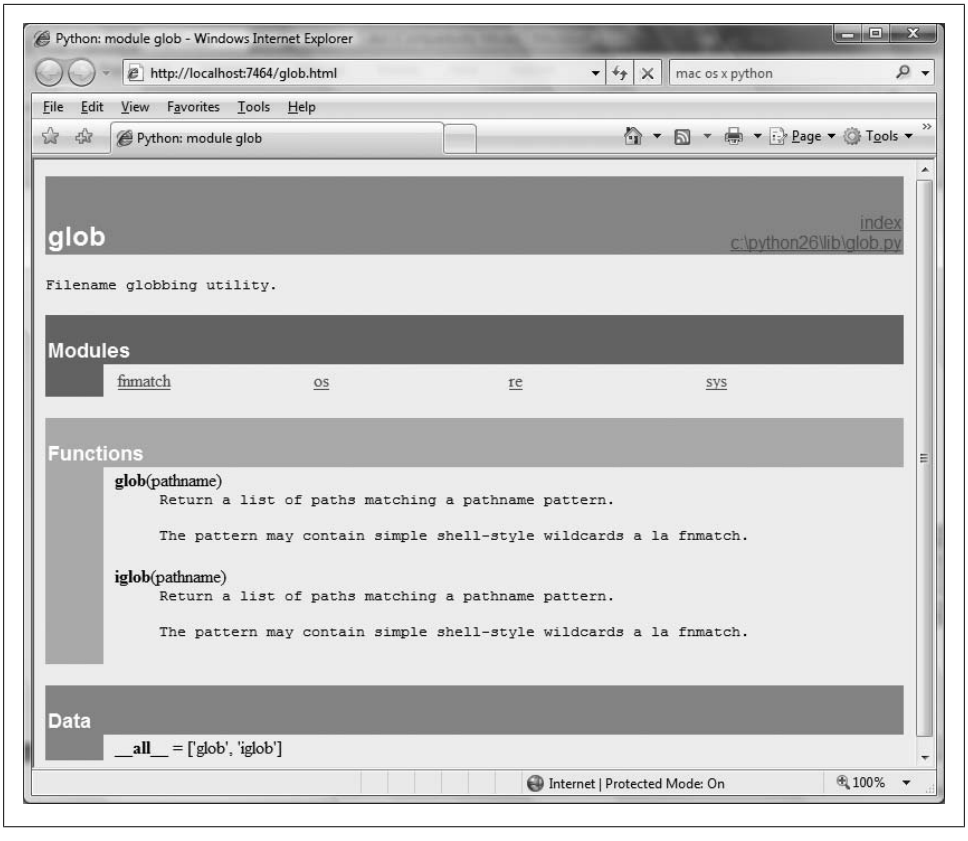

Figure 15-2. When you find a module in the Figure 15-1 GUI (such as this built-in standard library module) and press "go to selected," the module's documentation is rendered in HTML and displayed in a web browser window like this one.

Like the `help` function interface, the GUI interface works on user-defined modules as well as built-ins. Figure 15-3 shows the page generated for our *docstrings.py* module file.

PyDoc can be customized and launched in various ways we won't cover here; see its entry in Python's standard library manual for more details. The main thing to take away from this section is that PyDoc essentially gives you implementation reports "for free"—if you are good about using docstrings in your files, PyDoc does all the work of collecting and formatting them for display. PyDoc only helps for objects like functions and modules, but it provides an easy way to access a middle level of documentation for such tools—its reports are more useful than raw attribute lists, and less exhaustive than the standard manuals.

Cool PyDoc trick of the day: If you leave the module name empty in the top input field of the window in Figure 15-1 and press the "open browser" button, PyDoc will produce a web page containing a hyperlink to every module you can possibly import on your computer. This includes Python standard library modules, modules of third-party

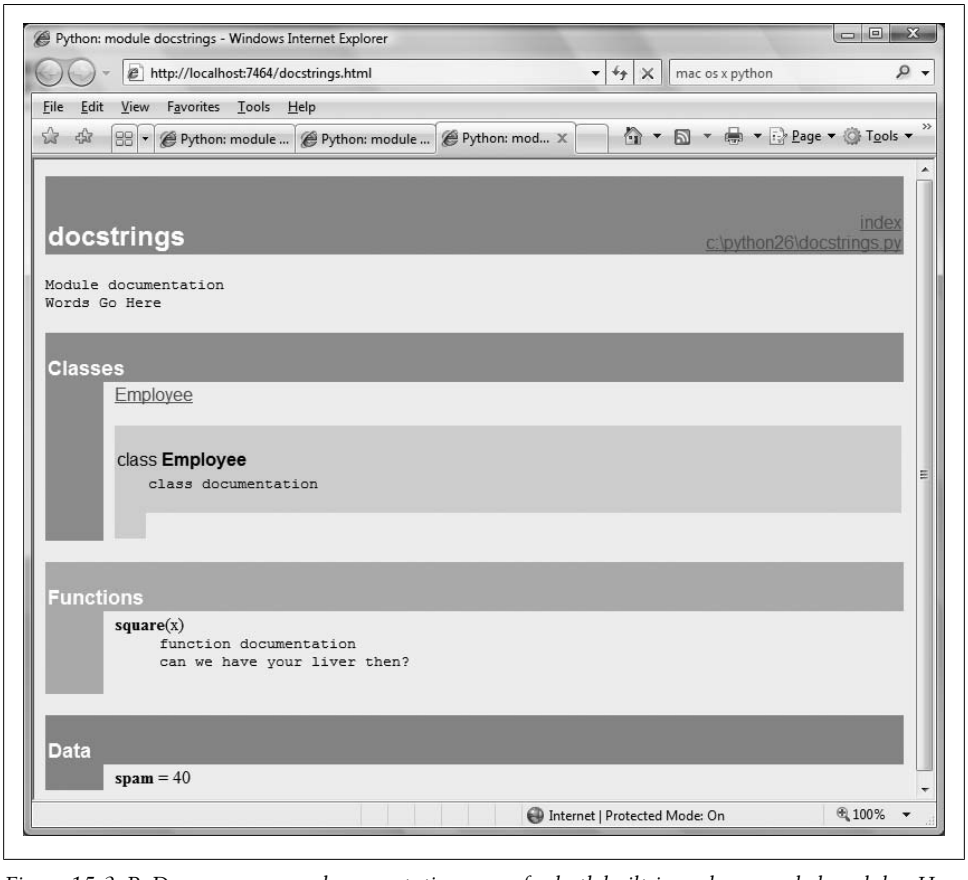

Figure 15-3. PyDoc can serve up documentation pages for both built-in and user-coded modules. Here is the page for a user-defined module, showing all its documentation strings (docstrings) extracted from the source file.

extensions you may have installed, user-defined modules on your import search path, and even statically or dynamically linked-in C-coded modules. Such information is hard to come by otherwise without writing code that inspects a set of module sources.

PyDoc can also be run to save the HTML documentation for a module in a file for later viewing or printing; see its documentation for pointers. Also, note that PyDoc might not work well if run on scripts that read from standard input—PyDoc imports the target module to inspect its contents, and there may be no connection for standard input text when it is run in GUI mode. Modules that can be imported without immediate input requirements will always work under PyDoc, though.

The Standard Manual Set

For the complete and most up-to-date description of the language and its toolset, Python's standard manuals stand ready to serve. Python's manuals ship in HTML and other formats, and they are installed with the Python system on Windows—they are available in your Start button's menu for Python, and they can also be opened from the Help menu within IDLE. You can also fetch the manual set separately from *http://www.python.org* in a variety of formats, or read them online at that site (follow the Documentation link). On Windows, the manuals are a compiled help file to support searches, and the online versions at the Python website include a web-based search page.

When opened, the Windows format of the manuals displays a root page like that in Figure 15-4. The two most important entries here are most likely the Library Reference (which documents built-in types, functions, exceptions, and standard library modules) and the Language Reference (which provides a formal description of language-level details). The tutorial listed on this page also provides a brief introduction for newcomers.

Figure 15-4. Python's standard manual set, available online at http://www.python.org, from IDLE's Help menu, and in the Windows Start button menu. It's a searchable help file on Windows, and there is a search engine for the online version. Of these, the Library Reference is the one you'll want to use most of the time.

Web Resources

At the official Python website (*http://www.python.org*), you'll find links to various Python resources, some of which cover special topics or domains. Click the Documentation link to access an online tutorial and the Beginners Guide to Python. The site also lists non-English Python resources.

You will find numerous Python wikis, blogs, websites, and a host of other resources on the Web today. To sample the online community, try searching for a term like "Python programming" in Google.

Published Books

As a final resource, you can choose from a large collection of reference books for Python. Bear in mind that books tend to lag behind the cutting edge of Python changes, partly because of the work involved in writing, and partly because of the natural delays built into the publishing cycle. Usually, by the time a book comes out, it's three or more months behind the current Python state. Unlike standard manuals, books are also generally not free.

Still, for many, the convenience and quality of a professionally published text is worth the cost. Moreover, Python changes so slowly that books are usually still relevant years after they are published, especially if their authors post updates on the Web. See the Preface for pointers to other Python books.

Common Coding Gotchas

Before the programming exercises for this part of the book, let's run through some of the most common mistakes beginners make when coding Python statements and programs. Many of these are warnings I've thrown out earlier in this part of the book, collected here for ease of reference. You'll learn to avoid these pitfalls once you've gained a bit of Python coding experience, but a few words now might help you avoid falling into some of these traps initially:

- **Don't forget the colons**. Always remember to type a `:` at the end of compound statement headers (the first line of an `if`, `while`, `for`, etc.). You'll probably forget at first (I did, and so have most of my 3,000 Python students over the years), but you can take some comfort from the fact that it will soon become an unconscious habit.
- **Start in column 1**. Be sure to start top-level (unnested) code in column 1. That includes unnested code typed into module files, as well as unnested code typed at the interactive prompt.

- **Blank lines matter at the interactive prompt**. Blank lines in compound statements are always ignored in module files, but when you're typing code at the interactive prompt, they end the statement. In other words, blank lines tell the interactive command line that you've finished a compound statement; if you want to continue, don't hit the Enter key at the `...` prompt (or in IDLE) until you're really done.
- **Indent consistently**. Avoid mixing tabs and spaces in the indentation of a block, unless you know what your text editor does with tabs. Otherwise, what you see in your editor may not be what Python sees when it counts tabs as a number of spaces. This is true in any block-structured language, not just Python—if the next programmer has her tabs set differently, she will not understand the structure of your code. It's safer to use all tabs or all spaces for each block.
- **Don't code C in Python**. A reminder for C/C++ programmers: you don't need to type parentheses around tests in `if` and `while` headers (e.g., `if (X==1):`). You can, if you like (any expression can be enclosed in parentheses), but they are fully superfluous in this context. Also, do not terminate all your statements with semicolons; it's technically legal to do this in Python as well, but it's totally useless unless you're placing more than one statement on a single line (the end of a line normally terminates a statement). And remember, don't embed assignment statements in `while` loop tests, and don't use `{}` around blocks (indent your nested code blocks consistently instead).
- **Use simple `for` loops instead of `while` or `range`**. Another reminder: a simple `for` loop (e.g., `for x in seq:`) is almost always simpler to code and quicker to run than a `while`- or `range`-based counter loop. Because Python handles indexing internally for a simple `for`, it can sometimes be twice as fast as the equivalent `while`. Avoid the temptation to count things in Python!
- **Beware of mutables in assignments**. I mentioned this in Chapter 11: you need to be careful about using mutables in a multiple-target assignment (`a = b = []`), as well as in an augmented assignment (`a += [1, 2]`). In both cases, in-place changes may impact other variables. See Chapter 11 for details.
- **Don't expect results from functions that change objects in-place**. We encountered this one earlier, too: in-place change operations like the `list.append` and `list.sort` methods introduced in Chapter 8 do not return values (other than `None`), so you should call them without assigning the result. It's not uncommon for beginners to say something like `mylist = mylist.append(X)` to try to get the result of an `append`, but what this actually does is assign `mylist` to `None`, not to the modified list (in fact, you'll lose your reference to the list altogether).

  A more devious example of this pops up in Python 2.X code when trying to step through dictionary items in a sorted fashion. It's fairly common to see code like `for k in D.keys().sort():`. This almost works—the `keys` method builds a keys list, and the `sort` method orders it—but because the `sort` method returns `None`, the loop fails because it is ultimately a loop over `None` (a nonsequence). This fails even

sooner in Python 3.0, because dictionary keys are views, not lists! To code this correctly, either use the newer `sorted` built-in function, which returns the sorted list, or split the method calls out to statements: `Ks = list(D.keys())`, then `Ks.sort()`, and finally, `for k in Ks:`. This, by the way, is one case where you'll still want to call the `keys` method explicitly for looping, instead of relying on the dictionary iterators—iterators do not sort.

- **Always use parentheses to call a function**. You must add parentheses after a function name to call it, whether it takes arguments or not (e.g., use `function()`, not `function`). In Part IV, we'll see that functions are simply objects that have a special operation—a call that you trigger with the parentheses.

  In classes, this problem seems to occur most often with files; it's common to see beginners type `file.close` to close a file, rather than `file.close()`. Because it's legal to reference a function without calling it, the first version with no parentheses succeeds silently, but it does not close the file!

- **Don't use extensions or paths in imports and reloads**. Omit directory paths and file suffixes in `import` statements (e.g., say `import mod`, not `import mod.py`). (We discussed module basics in Chapter 3 and will continue studying modules in Part V.) Because modules may have other suffixes besides *.py* (*.pyc*, for instance), hardcoding a particular suffix is not only illegal syntax, but doesn't make sense. Any platform-specific directory path syntax comes from module search path settings, not the `import` statement.

Chapter Summary

This chapter took us on a tour of program documentation—both documentation we write ourselves for our own programs, and documentation available for built-in tools. We met docstrings, explored the online and manual resources for Python reference, and learned how PyDoc's `help` function and web page interface provide extra sources of documentation. Because this is the last chapter in this part of the book, we also reviewed common coding mistakes to help you avoid them.

In the next part of this book, we'll start applying what we already know to larger program constructs: functions. Before moving on, however, be sure to work through the set of lab exercises for this part of the book that appear at the end of this chapter. And even before that, let's run through this chapter's quiz.

Test Your Knowledge: Quiz

1. When should you use documentation strings instead of hash-mark comments?
2. Name three ways you can view documentation strings.

3. How can you obtain a list of the available attributes in an object?
4. How can you get a list of all available modules on your computer?
5. Which Python book should you purchase after this one?

Test Your Knowledge: Answers

1. Documentation strings (docstrings) are considered best for larger, functional documentation, describing the use of modules, functions, classes, and methods in your code. Hash-mark comments are today best limited to micro-documentation about arcane expressions or statements. This is partly because docstrings are easier to find in a source file, but also because they can be extracted and displayed by the PyDoc system.
2. You can see docstrings by printing an object's `__doc__` attribute, by passing it to PyDoc's `help` function, and by selecting modules in PyDoc's GUI search engine in client/server mode. Additionally, PyDoc can be run to save a module's documentation in an HTML file for later viewing or printing.
3. The built-in `dir(X)` function returns a list of all the attributes attached to any object.
4. Run the PyDoc GUI interface, leave the module name blank, and select "open browser"; this opens a web page containing a link to every module available to your programs.
5. Mine, of course. (Seriously, the Preface lists a few recommended follow-up books, both for reference and for application tutorials.)

Test Your Knowledge: Part III Exercises

Now that you know how to code basic program logic, the following exercises will ask you to implement some simple tasks with statements. Most of the work is in exercise 4, which lets you explore coding alternatives. There are always many ways to arrange statements, and part of learning Python is learning which arrangements work better than others.

See Part III in Appendix B for the solutions.

1. *Coding basic loops*.
   a. Write a `for` loop that prints the ASCII code of each character in a string named `S`. Use the built-in function `ord(character)` to convert each character to an ASCII integer. (Test it interactively to see how it works.)
   b. Next, change your loop to compute the sum of the ASCII codes of all the characters in a string.

c. Finally, modify your code again to return a new list that contains the ASCII codes of each character in the string. Does the expression `map(ord, S)` have a similar effect? (Hint: see Chapter 14.)

2. *Backslash characters*. What happens on your machine when you type the following code interactively?

```
for i in range(50):
    print('hello %d\n\a' % i)
```

Beware that if it's run outside of the IDLE interface this example may beep at you, so you may not want to run it in a crowded lab. IDLE prints odd characters instead of beeping (see the backslash escape characters in Table 7-2).

3. *Sorting dictionaries*. In Chapter 8, we saw that dictionaries are unordered collections. Write a `for` loop that prints a dictionary's items in sorted (ascending) order. (Hint: use the dictionary `keys` and list `sort` methods, or the newer `sorted` built-in function.)

4. *Program logic alternatives*. Consider the following code, which uses a `while` loop and `found` flag to search a list of powers of 2 for the value of 2 raised to the fifth power (32). It's stored in a module file called *power.py*.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = False
i = 0
while not found and i < len(L):
   if 2 ** X == L[i]:
       found = True
   else:
       i = i+1

if found:
    print('at index', i)
else:
    print(X, 'not found')

C:\book\tests> python power.py
at index 5
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps outlined here to improve it (for all the transformations, you may either type your code interactively or store it in a script file run from the system command line—using a file makes this exercise much easier):

a. First, rewrite this code with a `while` loop `else` clause to eliminate the `found` flag and final `if` statement.

b. Next, rewrite the example to use a `for` loop with an `else` clause, to eliminate the explicit list-indexing logic. (Hint: to get the index of an item, use the list `index` method—`L.index(X)` returns the offset of the first `X` in list `L`.)

c. Next, remove the loop completely by rewriting the example with a simple `in` operator membership expression. (See Chapter 8 for more details, or type this to test: `2 in [1,2,3]`.)

d. Finally, use a `for` loop and the list `append` method to generate the powers-of-2 list (`L`) instead of hardcoding a list literal.

Deeper thoughts:

e. Do you think it would improve performance to move the `2 ** X` expression outside the loops? How would you code that?

f. As we saw in exercise 1, Python includes a `map(function, list)` tool that can generate a powers-of-2 list, too: `map(lambda x: 2 ** x, range(7))`. Try typing this code interactively; we'll meet `lambda` more formally in Chapter 19.

PART IV

Functions

CHAPTER 16
Function Basics

In Part III, we looked at basic procedural statements in Python. Here, we'll move on to explore a set of additional statements that we can use to create functions of our own.

In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program. Functions also can compute a result value and let us specify parameters that serve as function inputs, which may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts.

More fundamentally, functions are the alternative to programming by cutting and pasting—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we only have one copy to update, not many.

Functions are the most basic program structure Python provides for maximizing *code reuse* and minimizing *code redundancy*. As we'll see, functions are also a design tool that lets us split complex systems into manageable parts. Table 16-1 summarizes the primary function-related tools we'll study in this part of the book.

Table 16-1. Function-related statements and expressions

Statement	Examples
Calls	`myfunc('spam', 'eggs', meat=ham)`
def, return	`def adder(a, b=1, *c):` `  return a + b + c[0]`
global	`def changer():` `  global x; x = 'new'`
nonlocal	`def changer():` `  nonlocal x; x = 'new'`
yield	`def squares(x):` `  for i in range(x): yield i ** 2`
lambda	`funcs = [lambda x: x**2, lambda x: x*3]`

Why Use Functions?

Before we get into the details, let's establish a clear picture of what functions are all about. Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*. As a brief introduction, functions serve two primary development roles:

Maximizing code reuse and minimizing redundancy
: As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, all the code we've been writing has run immediately. Functions allow us to group and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many places, Python functions are the most basic *factoring* tool in the language: they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

Procedural decomposition
: Functions also provide a tool for splitting systems into pieces that have well-defined roles. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If you were programming a pizza-making robot, functions would help you divide the overall "make pizza" task into chunks—one function for each subtask in the process. It's easier to implement the smaller tasks in isolation than it is to implement the entire process at once. In general, functions are about *procedure*—how to do something, rather than what you're doing it to. We'll see why this distinction matters in Part VI, when we start making new object with classes.

In this part of the book, we'll explore the tools used to code functions in Python: function basics, scope rules, and argument passing, along with a few related concepts such as generators and functional tools. Because its importance begins to become more apparent at this level of coding, we'll also revisit the notion of polymorphism introduced earlier in the book. As you'll see, functions don't imply much new syntax, but they do lead us to some bigger programming ideas.

Coding Functions

Although it wasn't made very formal, we've already used some functions in earlier chapters. For instance, to make a file object, we called the built-in `open` function; similarly, we used the `len` built-in function to ask for the number of items in a collection object.

In this chapter, we will explore how to write *new* functions in Python. Functions we write behave the same way as the built-ins we've already seen: they are called in

expressions, are passed values, and return results. But writing new functions requires the application of a few additional ideas that haven't yet been introduced. Moreover, functions behave very differently in Python than they do in compiled languages like C. Here is a brief introduction to the main concepts behind Python functions, all of which we will study in this part of the book:

- **`def` is executable code**. Python functions are written with a new statement, the `def`. Unlike functions in compiled languages such as C, `def` is an executable statement—your function does not exist until Python reaches and runs the `def`. In fact, it's legal (and even occasionally useful) to nest `def` statements inside `if` statements, `while` loops, and even other `def`s. In typical operation, `def` statements are coded in module files and are naturally run to generate functions when a module file is first imported.
- **`def` creates an object and assigns it to a name**. When Python reaches and runs a `def` statement, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magic about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined *attributes* attached to them to record data.
- **`lambda` creates an object but returns it as a result**. Functions may also be created with the `lambda` expression, a feature that allows us to in-line function definitions in places where a `def` statement won't work syntactically (this is a more advanced concept that we'll defer until Chapter 19).
- **`return` sends a result object back to the caller**. When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a `return` statement; the returned value becomes the result of the function call.
- **`yield` sends a result object back to the caller, but remembers where it left off**. Functions known as *generators* may also use the `yield` statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time. This is another advanced topic covered later in this part of the book.
- **`global` declares module-level variables that are to be assigned**. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a `global` statement. More generally, names are always looked up in *scopes*—places where variables are stored—and assignments bind names to scopes.
- **`nonlocal` declares enclosing function variables that are to be assigned**. Similarly, the `nonlocal` statement added in Python 3.0 allows a function to assign a name that exists in the scope of a syntactically enclosing `def` statement. This allows

enclosing functions to serve as a place to retain *state*—information remembered when a function is called—without using shared global names.

- **Arguments are passed by assignment (object reference)**. In Python, arguments are passed to functions by assignment (which, as we've learned, means by object reference). As you'll see, in Python's model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects can change objects shared by the caller.
- **Arguments, return values, and variables are not declared**. As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As one consequence, a single function can often be applied to a variety of object types—any objects that sport a compatible *interface* (methods and expressions) will do, regardless of their specific types.

If some of the preceding words didn't sink in, don't worry—we'll explore all of these concepts with real code in this part of the book. Let's get started by expanding on some of these ideas and looking at a few examples.

def Statements

The `def` statement creates a function object and assigns it to a name. Its general format is as follows:

```
def <name>(arg1, arg2,... argN):
    <statements>
```

As with all compound Python statements, `def` consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon). The statement block becomes the function's *body*—that is, the code Python executes each time the function is called.

The `def` header line specifies a function *name* that is assigned the function object, along with a list of zero or more *arguments* (sometimes called *parameters*) in parentheses. The argument names in the header are assigned to the objects passed in parentheses at the point of call.

Function bodies often contain a `return` statement:

```
def <name>(arg1, arg2,... argN):
    ...
    return <value>
```

The Python `return` statement can show up anywhere in a function body; it ends the function call and sends a result back to the caller. The `return` statement consists of an object expression that gives the function's result. The `return` statement is optional; if it's not present, the function exits when the control flow falls off the end of the function

body. Technically, a function without a `return` statement returns the `None` object automatically, but this return value is usually ignored.

Functions may also contain `yield` statements, which are designed to produce a series of values over time, but we'll defer discussion of these until we survey generator topics in Chapter 20.

def Executes at Runtime

The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in Python is *runtime*; there is no such thing as a separate compile time.) Because it's a statement, a `def` can appear anywhere a statement can—even nested in other statements. For instance, although `def`s normally are run when the module enclosing them is imported, it's also completely legal to nest a function `def` inside an `if` statement to select between alternative definitions:

```
if test:
    def func():             # Define func this way
        ...
else:
    def func():             # Or else this way
        ...
...
func()                      # Call the version selected and built
```

One way to understand this code is to realize that the `def` is much like an `=` statement: it simply assigns a name at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, `def`s are not evaluated until they are reached and run, and the code *inside* `def`s is not evaluated until the functions are later called.

Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func             # Assign function object
othername()                  # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just *objects*; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary *attributes* to be attached to record information for later use:

```
def func(): ...              # Create function object
func()                       # Call object
func.attr = value            # Attach attributes
```

A First Example: Definitions and Calls

Apart from such runtime concepts (which tend to seem most unique to programmers with backgrounds in traditional compiled languages), Python functions are straightforward to use. Let's code a first real example to demonstrate the basics. As you'll see, there are two sides to the function picture: a *definition* (the `def` that creates a function) and a *call* (an expression that tells Python to run the function's body).

Definition

Here's a definition typed interactively that defines a function called `times`, which returns the product of its two arguments:

```
>>> def times(x, y):          # Create and assign function
...     return x * y          # Body executed when called
...
```

When Python reaches and runs this `def`, it creates a new function object that packages the function's code and assigns the object to the name `times`. Typically, such a statement is coded in a module file and runs when the enclosing file is imported; for something this small, though, the interactive prompt suffices.

Calls

After the `def` has run, you can call (run) the function in your program by adding parentheses after the function's name. The parentheses may optionally contain one or more object arguments, to be passed (assigned) to the names in the function's header:

```
>>> times(2, 4)               # Arguments in parentheses
8
```

This expression passes two arguments to `times`. As mentioned previously, arguments are passed by assignment, so in this case the name `x` in the function header is assigned the value `2`, `y` is assigned the value `4`, and the function's body is run. For this function, the body is just a `return` statement that sends back the result as the value of the call expression. The returned object was printed here interactively (as in most languages, `2 * 4` is `8` in Python), but if we needed to use it later we could instead assign it to a variable. For example:

```
>>> x = times(3.14, 4)        # Save the result object
>>> x
12.56
```

Now, watch what happens when the function is called a third time, with very different kinds of objects passed in:

```
>>> times('Ni', 4)            # Functions are "typeless"
'NiNiNiNi'
```

This time, our function means something completely different (Monty Python reference again intended). In this third call, a string and an integer are passed to `x` and `y`, instead of two numbers. Recall that * works on both numbers and sequences; because we never declare the types of variables, arguments, or return values in Python, we can use `times` to either *multiply* numbers or *repeat* sequences.

In other words, what our `times` function means and does depends on what we pass into it. This is a core idea in Python (and perhaps the key to using the language well), which we'll explore in the next section.

Polymorphism in Python

As we just saw, the very meaning of the expression `x * y` in our simple `times` function depends completely upon the kinds of objects that `x` and `y` are—thus, the same function can perform multiplication in one instance and repetition in another. Python leaves it up to the *objects* to do something reasonable for the syntax. Really, * is just a dispatch mechanism that routes control to the objects being processed.

This sort of type-dependent behavior is known as *polymorphism*, a term we first met in Chapter 4 that essentially means that the meaning of an operation depends on the objects being operated upon. Because it's a dynamically typed language, polymorphism runs rampant in Python. In fact, every operation is a polymorphic operation in Python: printing, indexing, the * operator, and much more.

This is deliberate, and it accounts for much of the language's conciseness and flexibility. A single function, for instance, can generally be applied to a whole category of object types automatically. As long as those objects support the expected interface (a.k.a. protocol), the function can process them. That is, if the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function's logic.

Even in our simple `times` function, this means that *any* two objects that support a * will work, no matter what they may be, and no matter when they are coded. This function will work on two numbers (performing multiplication), or a string and a number (performing repetition), or any other combination of objects supporting the expected interface—even class-based objects we have not even coded yet.

Moreover, if the objects passed in do *not* support this expected interface, Python will detect the error when the * expression is run and raise an exception automatically. It's therefore pointless to code error checking ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for.

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is *not supposed to care* about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types that

may be coded in the future. Although it is possible to test for types with tools like the `type` built-in function, doing so breaks your code's flexibility. By and large, we code to object *interfaces* in Python, not data types.

Of course, this polymorphic model of programming means we have to test our code to detect errors, rather than providing type declarations a compiler can use to detect some types of errors for us ahead of time. In exchange for an initial bit of testing, though, we radically reduce the amount of code we have to write and radically increase our code's flexibility. As you'll learn, it's a net win in practice.

A Second Example: Intersecting Sequences

Let's look at a second function example that does something a bit more useful than multiplying arguments and further illustrates function basics.

In Chapter 13, we coded a `for` loop that collected items held in common in two strings. We noted there that the code wasn't as useful as it could be because it was set up to work only on specific variables and could not be rerun later. Of course, we could copy the code and paste it into each place where it needs to be run, but this solution is neither good nor general—we'd still have to edit each copy to support different sequence names, and changing the algorithm would then require changing multiple copies.

Definition

By now, you can probably guess that the solution to this dilemma is to package the `for` loop inside a function. Doing so offers a number of advantages:

- Putting the code in a function makes it a tool that you can run as many times as you like.
- Because callers can pass in arbitrary arguments, functions are general enough to work on any two sequences (or other iterables) you wish to intersect.
- When the logic is packaged in a function, you only have to change code in one place if you ever need to change the way the intersection works.
- Coding the function in a module file means it can be imported and reused by any program run on your machine.

In effect, wrapping the code in a function makes it a general intersection utility:

```
def intersect(seq1, seq2):
    res = []                        # Start empty
    for x in seq1:                  # Scan seq1
        if x in seq2:               # Common item?
            res.append(x)           # Add to end
    return res
```

The transformation from the simple code of Chapter 13 to this function is straightforward; we've just nested the original logic under a `def` header and made the objects on

which it operates passed-in parameter names. Because this function computes a result, we've also added a `return` statement to send a result object back to the caller.

Calls

Before you can call a function, you have to make it. To do this, run its `def` statement, either by typing it interactively or by coding it in a module file and importing the file. Once you've run the `def`, you can call the function by passing any two sequence objects in parentheses:

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)             # Strings
['S', 'A', 'M']
```

Here, we've passed in two strings, and we get back a list containing the characters in common. The algorithm the function uses is simple: "for every item in the first argument, if that item is also in the second argument, append the item to the result." It's a little shorter to say that in Python than in English, but it works out the same.

To be fair, our intersect function is fairly slow (it executes nested loops), isn't really mathematical intersection (there may be duplicates in the result), and isn't required at all (as we've seen, Python's set data type provides a built-in intersection operation). Indeed, the function could be replaced with a single list comprehension expression, as it exhibits the classic loop collector code pattern:

```
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

As a function basics example, though, it does the job—this single piece of code can apply to an entire range of object types, as the next section explains.

Polymorphism Revisited

Like all functions in Python, `intersect` is polymorphic. That is, it works on arbitrary types, as long as they support the expected object interface:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Mixed types
>>> x                                     # Saved result object
[1]
```

This time, we passed in different types of objects to our function—a list and a tuple (mixed types)—and it still picked out the common items. Because you don't have to specify the types of arguments ahead of time, the `intersect` function happily iterates through any kind of sequence objects you send it, as long as they support the expected interfaces.

For `intersect`, this means that the first argument has to support the `for` loop, and the second has to support the `in` membership test. Any two such objects will work, regardless of their specific types—that includes physically stored sequences like strings

and lists; all the iterable objects we met in Chapter 14, including files and dictionaries; and even any class-based objects we code that apply operator overloading techniques (we'll discuss these later in the book).*

Here again, if we pass in objects that do not support these interfaces (e.g., numbers), Python will automatically detect the mismatch and raise an exception for us—which is exactly what we want, and the best we could do on our own if we coded explicit type tests. By not coding type tests and allowing Python to detect the mismatches for us, we both reduce the amount of code we need to write and increase our code's flexibility.

Local Variables

Probably the most interesting part of this example is its names. It turns out that the variable `res` inside `intersect` is what in Python is called a *local variable*—a name that is visible only to code inside the function `def` and that exists only while the function runs. In fact, because all names *assigned* in any way inside a function are classified as local variables by default, nearly all the names in `intersect` are local variables:

- `res` is obviously assigned, so it is a local variable.
- Arguments are passed by assignment, so `seq1` and `seq2` are, too.
- The `for` loop assigns items to a variable, so the name `x` is also local.

All these local variables appear when the function is called and disappear when the function exits—the `return` statement at the end of `intersect` sends back the result *object*, but the *name* `res` goes away. To fully explore the notion of locals, though, we need to move on to Chapter 17.

Chapter Summary

This chapter introduced the core ideas behind function definition—the syntax and operation of the `def` and `return` statements, the behavior of function call expressions, and the notion and benefits of polymorphism in Python functions. As we saw, a `def` statement is executable code that creates a function object at runtime; when the function is later called, objects are passed into it by assignment (recall that assignment means object reference in Python, which, as we learned in Chapter 6, really means pointer internally), and computed values are sent back by `return`. We also began

* This code will always work if we intersect files' contents obtained with `file.readlines()`. It may not work to intersect lines in open input files directly, though, depending on the file object's implementation of the `in` operator or general iteration. Files must generally be rewound (e.g., with a `file.seek(0)` or another `open`) after they have been read to end-of-file once. As we'll see in Chapter 29 when we study operator overloading, classes implement the `in` operator either by providing the specific `__contains__` method or by supporting the general iteration protocol with the `__iter__` or older `__getitem__` methods; if coded, classes can define what iteration means for their data.

exploring the concepts of local variables and scopes in this chapter, but we'll save all the details on those topics for Chapter 17. First, though, a quick quiz.

Test Your Knowledge: Quiz

1. What is the point of coding functions?
2. At what time does Python create a function?
3. What does a function return if it has no `return` statement in it?
4. When does the code nested inside the function definition statement run?
5. What's wrong with checking the types of objects passed into a function?

Test Your Knowledge: Answers

1. Functions are the most basic way of avoiding code *redundancy* in Python—factoring code into functions means that we have only one copy of an operation's code to update in the future. Functions are also the basic unit of code *reuse* in Python—wrapping code in functions makes it a reusable tool, callable in a variety of programs. Finally, functions allow us to divide a complex system into manageable parts, each of which may be developed individually.
2. A function is created when Python reaches and runs the `def` statement; this statement creates a function object and assigns it the function's name. This normally happens when the enclosing module file is imported by another module (recall that imports run the code in a file from top to bottom, including any `def`s), but it can also occur when a `def` is typed interactively or nested in other statements, such as `if`s.
3. A function returns the `None` object by default if the control flow falls off the end of the function body without running into a `return` statement. Such functions are usually called with expression statements, as assigning their `None` results to variables is generally pointless.
4. The function body (the code nested inside the function definition statement) is run when the function is later called with a call expression. The body runs anew each time the function is called.
5. Checking the types of objects passed into a function effectively breaks the function's flexibility, constraining the function to work on specific types only. Without such checks, the function would likely be able to process an entire range of object types—any objects that support the interface expected by the function will work. (The term *interface* means the set of methods and expression operators the function's code runs.)

CHAPTER 17
Scopes

Chapter 16 introduced basic function definitions and calls. As we saw, Python's basic function model is simple to use, but even simple function examples quickly led us to questions about the meaning of variables in our code. This chapter moves on to present the details behind Python's *scopes*—the places where variables are defined and looked up. As we'll see, the place where a name is assigned in our code is crucial to determining what the name means. We'll also find that scope usage can have a major impact on program maintenance effort; overuse of globals, for example, is a generally bad thing.

Python Scope Basics

Now that you're ready to start writing your own functions, we need to get more formal about what names mean in Python. When you use a name in a program, Python creates, changes, or looks up the name in what is known as a *namespace*—a place where names live. When we talk about the search for a name's value in relation to code, the term *scope* refers to a namespace: that is, the location of a name's assignment in your code determines the scope of the name's visibility to your code.

Just about everything related to names, including scope classification, happens at assignment time in Python. As we've seen, names in Python spring into existence when they are first assigned values, and they must be assigned before they are used. Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., *bind* it to) a particular namespace. In other words, the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility.

Besides packaging code, functions add an extra namespace layer to your programs—by default, all names assigned inside a function are associated with that function's namespace, and no other. This means that:

- Names defined inside a `def` can only be seen by the code within that `def`. You cannot even refer to such names from outside the function.

- Names defined inside a `def` do not clash with variables outside the `def`, even if the same names are used elsewhere. A name `X` assigned outside a given `def` (i.e., in a different `def` or at the top level of a module file) is a completely different variable from a name `X` assigned inside that `def`.

In all cases, the scope of a variable (where it can be used) is always determined by where it is assigned in your source code and has nothing to do with which functions call which. In fact, as we'll learn in this chapter, variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a `def`, it is *local* to that function.
- If a variable is assigned in an enclosing `def`, it is *nonlocal* to nested functions.
- If a variable is assigned outside all `def`s, it is *global* to the entire file.

We call this *lexical scoping* because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls.

For example, in the following module file, the `X = 99` assignment creates a *global* variable named `X` (visible everywhere in this file), but the `X = 88` assignment creates a *local* variable `X` (visible only within the `def` statement):

```
X = 99

def func():
    X = 88
```

Even though both variables are named `X`, their scopes make them different. The net effect is that function scopes help to avoid name clashes in your programs and help to make functions more self-contained program units.

Scope Rules

Before we started writing functions, all the code we wrote was at the top level of a module (i.e., not nested in a `def`), so the names we used either lived in the module itself or were built-ins predefined by Python (e.g., `open`). Functions provide nested namespaces (scopes) that localize the names they use, such that names inside a function won't clash with those outside it (in a module or another function). Again, functions define a *local scope*, and modules define a *global scope*. The two scopes are related as follows:

- **The enclosing module is a global scope**. Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. Global variables become attributes of a module object to the outside world but can be used as simple variables within a module file.
- **The global scope spans a single file only**. Don't be fooled by the word "global" here—names at the top level of a file are only global to code within that single file. There is really no notion of a single, all-encompassing global file-based scope in

Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear "global" in Python, think "module."

- **Each call to a function creates a new local scope**. Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live. You can think of each `def` statement (and `lambda` expression) as defining a new local scope, but because Python allows functions to call themselves to loop (an advanced technique known as *recursion*), the local scope in fact technically corresponds to a function call—in other words, each call creates a new local namespace. Recursion is useful when processing structures whose shapes can't be predicted ahead of time.
- **Assigned names are local unless declared global or nonlocal**. By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a `global` statement inside the function. If you need to assign a name that lives in an enclosing `def`, as of Python 3.0 you can do so by declaring it in a `nonlocal` statement.
- **All other names are enclosing function locals, globals, or built-ins**. Names not assigned a value in the function definition are assumed to be enclosing scope locals (in an enclosing `def`), globals (in the enclosing module's namespace), or built-ins (in the predefined `__builtin__` module Python provides).

There are a few subtleties to note here. First, keep in mind that code typed at the *interactive command prompt* follows these same rules. You may not know it yet, but code run interactively is really entered into a built-in module called `__main__`; this module works just like a module file, but results are echoed as you go. Because of this, interactively created names live in a module, too, and thus follow the normal scope rules: they are global to the interactive session. You'll learn more about modules in the next part of this book.

Also note that *any type of assignment* within a function classifies a name as local. This includes `=` statements, module names in `import`, function names in `def`, function argument names, and so on. If you assign a name in any way within a `def`, it will become a local to that function.

Conversely, *in-place changes* to objects do not classify names as locals; only actual name assignments do. For instance, if the name `L` is assigned to a list at the top level of a module, a statement `L = X` within a function will classify `L` as a local, but `L.append(X)` will not. In the latter case, we are changing the list object that `L` references, not `L` itself—`L` is found in the global scope as usual, and Python happily modifies it without requiring a `global` (or `nonlocal`) declaration. As usual, it helps to keep the distinction between names and objects clear: changing an object is not an assignment to a name.

Name Resolution: The LEGB Rule

If the prior section sounds confusing, it really boils down to three simple rules. With a `def` statement:

- Name references search at most four scopes: local, then enclosing functions (if any), then global, then built-in.
- Name assignments create or change local names by default.
- `global` and `nonlocal` declarations map assigned names to enclosing module and function scopes.

In other words, all names assigned inside a function `def` statement (or a `lambda`, an expression we'll meet later) are locals by default. Functions can freely use names assigned in syntactically enclosing functions and the global scope, but they must declare such nonlocals and globals in order to change them.

Python's name-resolution scheme is sometimes called the *LEGB rule*, after the scope names:

- When you use an unqualified name inside a function, Python searches up to four scopes—the local (*L*) scope, then the local scopes of any enclosing (*E*) `def`s and `lambda`s, then the global (*G*) scope, and then the built-in (*B*) scope—and stops at the first place the name is found. If the name is not found during this search, Python reports an error. As we learned in Chapter 6, names must be assigned before they can be used.
- When you assign a name in a function (instead of just referring to it in an expression), Python always creates or changes the name in the local scope, unless it's declared to be global or nonlocal in that function.
- When you assign a name outside any function (i.e., at the top level of a module file, or at the interactive prompt), the local scope is the same as the global scope—the module's namespace.

Figure 17-1 illustrates Python's four scopes. Note that the second scope lookup layer, *E*—the scopes of enclosing `def`s or `lambda`s—can technically correspond to more than one lookup layer. This case only comes into play when you nest functions within functions, and it is addressed by the `nonlocal` statement.*

Also keep in mind that these rules apply only to simple *variable* names (e.g., `spam`). In Parts V and VI, we'll see that qualified *attribute* names (e.g., `object.spam`) live in particular objects and follow a completely different set of lookup rules than those

* The scope lookup rule was called the "LGB rule" in the first edition of this book. The enclosing `def` "E" layer was added later in Python to obviate the task of passing in enclosing scope names explicitly with default arguments—a topic usually of marginal interest to Python beginners that we'll defer until later in this chapter. Since this scope is addressed by the `nonlocal` statement in Python 3.0, I suppose the lookup rule might now be better named "LNGB," but backward compatibility matters in books, too!

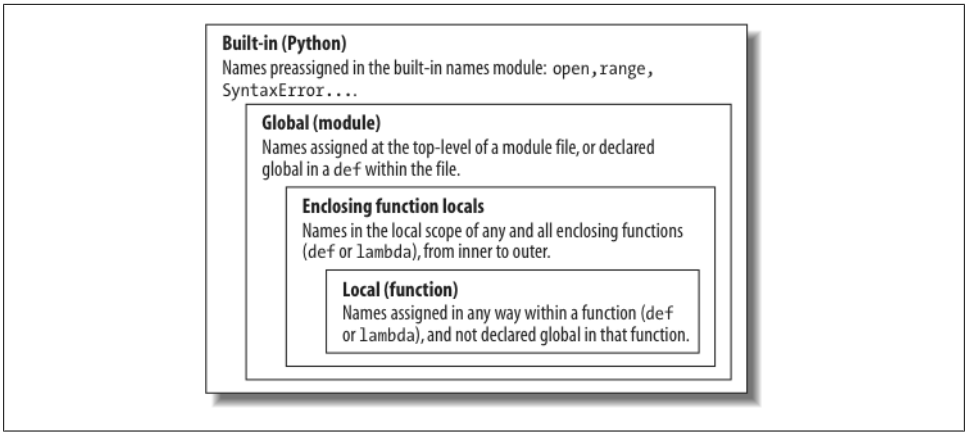

Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

covered here. References to attribute names following periods (`.`) search one or more *objects*, not scopes, and may invoke something called "inheritance"; more on this in Part VI of this book.

Scope Example

Let's look at a larger example that demonstrates scope ideas. Suppose we wrote the following code in a module file:

```
# Global scope
X = 99                  # X and func assigned in module: global

def func(Y):            # Y and Z assigned in function: locals
    # Local scope
    Z = X + Y           # X is a global
    return Z

func(1)                 # func in module: result=100
```

This module and the function it contains use a number of names to do their business. Using Python's scope rules, we can classify the names as follows:

Global names: `X`, `func`
: `X` is global because it's assigned at the top level of the module file; it can be referenced inside the function without being declared global. `func` is global for the same reason; the `def` statement assigns a function object to the name `func` at the top level of the module.

Local names: `Y`, `Z`

`Y` and `Z` are local to the function (and exist only while the function runs) because they are both assigned values in the function definition: `Z` by virtue of the `=` statement, and `Y` because arguments are always passed by assignment.

The whole point behind this name-segregation scheme is that local variables serve as temporary names that you need only while a function is running. For instance, in the preceding example, the argument `Y` and the addition result `Z` exist only inside the function; these names don't interfere with the enclosing module's namespace (or any other function, for that matter).

The local/global distinction also makes functions easier to understand, as most of the names a function uses appear in the function itself, not at some arbitrary place in a module. Also, because you can be sure that local names will not be changed by some remote function in your program, they tend to make programs easier to debug and modify.

The Built-in Scope

We've been talking about the built-in scope in the abstract, but it's a bit simpler than you may think. Really, the built-in scope is just a built-in module called `builtins`, but you have to import `builtins` to query built-ins because the name `builtins` is not itself built-in....

No, I'm serious! The built-in scope is implemented as a standard library module named `builtins`, but that name itself is not placed in the built-in scope, so you have to import it in order to inspect it. Once you do, you can run a `dir` call to see which names are predefined. In Python 3.0:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
    ...many more names omitted...
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

The names in this list constitute the built-in scope in Python; roughly the first half are built-in exceptions, and the second half are built-in functions. Also in this list are the special names `None`, `True`, and `False`, though they are treated as reserved words. Because Python automatically searches this module last in its LEGB lookup, you get all the names in this list "for free;" that is, you can use them without importing any modules. Thus, there are really two ways to refer to a built-in function—by taking advantage of the LEGB rule, or by manually importing the `builtins` module:

```
>>> zip                          # The normal way
<class 'zip'>
```

```
>>> import builtins              # The hard way
>>> builtins.zip
<class 'zip'>
```

The second of these approaches is sometimes useful in advanced work. The careful reader might also notice that because the LEGB lookup procedure takes the first occurrence of a name that it finds, names in the local scope may override variables of the same name in both the global and built-in scopes, and global names may override built-ins. A function can, for instance, create a local variable called `open` by assigning to it:

```
def hider():
    open = 'spam'               # Local variable, hides built-in
    ...
    open('data.txt')            # This won't open a file now in this scope!
```

However, this will hide the built-in function called `open` that lives in the built-in (outer) scope. It's also usually a bug, and a nasty one at that, because Python will not issue a warning message about it (there are times in advanced programming where you may really want to replace a built-in name by redefining it in your code).

Functions can similarly hide global variables of the same name with locals:

```
X = 88                          # Global X

def func():
    X = 99                      # Local X: hides global

func()
print(X)                        # Prints 88: unchanged
```

Here, the assignment within the function creates a local `X` that is a completely different variable from the global `X` in the module outside the function. Because of this, there is no way to change a name outside a function without adding a `global` (or `nonlocal`) declaration to the `def`, as described in the next section.

Version skew note: Actually, the tongue twisting gets a bit worse. The Python 3.0 `builtins` module used here is named `__builtin__` in Python 2.6. And just for fun, the name `__builtins__` (with the "s") is preset in most global scopes, including the interactive session, to reference the module known as `builtins` (a.k.a. `__builtin__` in 2.6).

That is, after importing `builtins`, `__builtins__ is builtins` is `True` in 3.0, and `__builtins__ is __builtin__` is `True` in 2.6. The net effect is that we can inspect the built-in scope by simply running `dir(__builtins__)` with no import in both 3.0 and 2.6, but we are advised to use `builtins` for real work in 3.0. Who said documenting this stuff was easy?

Breaking the Universe in Python 2.6

Here's another thing you can do in Python that you probably shouldn't—because the names `True` and `False` in 2.6 are just variables in the built-in scope and are not reserved, it's possible to reassign them with a statement like `True = False`. Don't worry, you won't actually break the logical consistency of the universe in so doing! This statement merely redefines the word `True` for the single scope in which it appears. All other scopes still find the originals in the built-in scope.

For more fun, though, in Python 2.6 you could say `__builtin__.True = False`, to reset `True` to `False` for the entire Python process. Alas, this type of assignment has been disallowed in Python 3.0, because `True` and `False` are treated as actual reserved words, just like `None`. In 2.6, though, it sends IDLE into a strange panic state that resets the user code process.

This technique can be useful, however, both to illustrate the underlying namespace model and for tool writers who must change built-ins such as `open` to customized functions. Also, note that third-party tools such as PyChecker will warn about common programming mistakes, including accidental assignment to built-in names (this is known as "shadowing" a built-in in PyChecker).

The global Statement

The `global` statement and its `nonlocal` cousin are the only things that are remotely like declaration statements in Python. They are not type or size declarations, though; they are *namespace declarations*. The `global` statement tells Python that a function plans to change one or more global names—i.e., names that live in the enclosing module's scope (namespace).

We've talked about `global` in passing already. Here's a summary:

- Global names are variables assigned at the top level of the enclosing module file.
- Global names must be declared only if they are assigned within a function.
- Global names may be referenced within a function without being declared.

In other words, `global` allows us to change names that live outside a `def` at the top level of a module file. As we'll see later, the `nonlocal` statement is almost identical but applies to names in the enclosing `def`'s local scope, rather than names in the enclosing module.

The `global` statement consists of the keyword `global`, followed by one or more names separated by commas. All the listed names will be mapped to the enclosing module's scope when assigned or referenced within the function body. For instance:

```
X = 88                          # Global X

def func():
    global X
    X = 99                      # Global X: outside def
```

```
func()
print(X)                    # Prints 99
```

We've added a `global` declaration to the example here, such that the `X` inside the `def` now refers to the `X` outside the `def`; they are the same variable this time. Here is a slightly more involved example of `global` at work:

```
y, z = 1, 2                 # Global variables in module
def all_global():
    global x                # Declare globals assigned
    x = y + z               # No need to declare y, z: LEGB rule
```

Here, `x`, `y`, and `z` are all globals inside the function `all_global`. `y` and `z` are global because they aren't assigned in the function; `x` is global because it was listed in a `global` statement to map it to the module's scope explicitly. Without the `global` here, `x` would be considered local by virtue of the assignment.

Notice that `y` and `z` are not declared global; Python's LEGB lookup rule finds them in the module automatically. Also, notice that `x` might not exist in the enclosing module before the function runs; in this case, the assignment in the function creates `x` in the module.

Minimize Global Variables

By default, names assigned in functions are locals, so if you want to change names outside functions you have to write extra code (e.g., `global` statements). This is by design—as is common in Python, you have to say more to do the potentially "wrong" thing. Although there are times when globals are useful, variables assigned in a `def` are local by default because that is normally the best policy. Changing globals can lead to well-known software engineering problems: because the variables' values are dependent on the order of calls to arbitrarily distant functions, programs can become difficult to debug.

Consider this module file, for example:

```
X = 99
def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

Now, imagine that it is your job to modify or reuse this module file. What will the value of `X` be here? Really, that question has no meaning unless it's qualified with a point of reference in time—the value of `X` is timing-dependent, as it depends on which function was called last (something we can't tell from this file alone).

The net effect is that to understand this code, you have to trace the flow of control through the *entire program*. And, if you need to reuse or modify the code, you have to keep the entire program in your head all at once. In this case, you can't really use one of these functions without bringing along the other. They are dependent on (that is, *coupled* with) the global variable. This is the problem with globals—they generally make code more difficult to understand and use than code consisting of self-contained functions that rely on locals.

On the other hand, short of using object-oriented programming and classes, global variables are probably the most straightforward way to retain shared state information (information that a function needs to remember for use the next time it is called) in Python—local variables disappear when the function returns, but globals do not. Other techniques, such as default mutable arguments and enclosing function scopes, can achieve this, too, but they are more complex than pushing values out to the global scope for retention.

Some programs designate a single module to collect globals; as long as this is expected, it is not as harmful. In addition, programs that use multithreading to do parallel processing in Python commonly depend on global variables—they become shared memory between functions running in parallel threads, and so act as a communication device.†

For now, though, especially if you are relatively new to programming, avoid the temptation to use globals whenever you can—try to communicate with passed-in arguments and return values instead. Six months from now, both you and your coworkers will be happy you did.

Minimize Cross-File Changes

Here's another scope-related issue: although we *can* change variables in another file directly, we usually shouldn't. Module files were introduced in Chapter 3 and are covered in more depth in the next part of this book. To illustrate their relationship to scopes, consider these two module files:

```
# first.py
X = 99                  # This code doesn't know about second.py

# second.py
import first
print(first.X)          # Okay: references a name in another file
first.X = 88            # But changing it can be too subtle and implicit
```

† *Multithreading* runs function calls in parallel with the rest of the program and is supported by Python's standard library modules `_thread`, `threading`, and `queue` (`thread`, `threading`, and `Queue` in Python 2.6). Because all threaded functions run in the same process, global scopes often serve as shared memory between them. Threading is commonly used for long-running tasks in GUIs, to implement nonblocking operations in general and to leverage CPU capacity. It is also beyond this book's scope; see the Python library manual, as well as the follow-up texts listed in the Preface (such as O'Reilly's *Programming Python*), for more details.

The first defines a variable X, which the second prints and then changes by assignment. Notice that we must import the first module into the second file to get to its variable at all—as we've learned, each module is a self-contained namespace (package of variables), and we must import one module to see inside it from another. That's the main point about modules: by segregating variables on a per-file basis, they avoid name collisions across files.

Really, though, in terms of this chapter's topic, the global scope of a module file *becomes* the attribute namespace of the module object once it is imported—importers automatically have access to all of the file's global variables, because a file's global scope morphs into an object's attribute namespace when it is imported.

After importing the first module, the second module prints its variable and then assigns it a new value. Referencing the module's variable to print it is fine—this is how modules are linked together into a larger system normally. The problem with the assignment, however, is that it is far too implicit: whoever's charged with maintaining or reusing the first module probably has no clue that some arbitrarily far-removed module on the import chain can change X out from under him at runtime. In fact, the second module may be in a completely different directory, and so difficult to notice at all.

Although such cross-file variable changes are always possible in Python, they are usually much more subtle than you will want. Again, this sets up too strong a *coupling* between the two files—because they are both dependent on the value of the variable X, it's difficult to understand or reuse one file without the other. Such implicit cross-file dependencies can lead to inflexible code at best, and outright bugs at worst.

Here again, the best prescription is generally to not do this—the best way to communicate across file boundaries is to call functions, passing in arguments and getting back return values. In this specific case, we would probably be better off coding an *accessor function* to manage the change:

```
# first.py
X = 99

def setX(new):
    global X
    X = new

# second.py
import first
first.setX(88)
```

This requires more code and may seem like a trivial change, but it makes a huge difference in terms of readability and maintainability—when a person reading the first module by itself sees a function, that person will know that it is a point of *interface* and will expect the change to the X. In other words, it removes the element of surprise that is rarely a good thing in software projects. Although we cannot prevent cross-file changes from happening, common sense dictates that they should be minimized unless widely accepted across the program.

Other Ways to Access Globals

Interestingly, because global-scope variables morph into the attributes of a loaded module object, we can emulate the `global` statement by importing the enclosing module and assigning to its attributes, as in the following example module file. Code in this file imports the enclosing module, first by name, and then by indexing the `sys.modules` loaded modules table (more on this table in Chapter 21):

```
# thismod.py

var = 99                            # Global variable == module attribute

def local():
    var = 0                         # Change local var

def glob1():
    global var                      # Declare global (normal)
    var += 1                        # Change global var

def glob2():
    var = 0                         # Change local var
    import thismod                  # Import myself
    thismod.var += 1                # Change global var

def glob3():
    var = 0                         # Change local var
    import sys                      # Import system table
    glob = sys.modules['thismod']   # Get module object (or use __name__)
    glob.var += 1                   # Change global var

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)
```

When run, this adds 3 to the global variable (only the first function does not impact it):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

This works, and it illustrates the equivalence of globals to module attributes, but it's much more work than using the `global` statement to make your intentions explicit.

As we've seen, `global` allows us to change names in a module outside a function. It has a cousin named `nonlocal` that can be used to change names in enclosing functions, too, but to understand how that can be useful, we first need to explore enclosing functions in general.

Scopes and Nested Functions

So far, I've omitted one part of Python's scope rules on purpose, because it's relatively rare to encounter it in practice. However, it's time to take a deeper look at the letter *E* in the LEGB lookup rule. The *E* layer is fairly new (it was added in Python 2.2); it takes the form of the local scopes of any and all enclosing function `def`s. Enclosing scopes are sometimes also called *statically nested scopes*. Really, the nesting is a lexical one—nested scopes correspond to physically and syntactically nested code structures in your program's source code.

Nested Scope Details

With the addition of nested function scopes, variable lookup rules become slightly more complex. Within a function:

- A reference (`X`) looks for the name `X` first in the current local scope (function); then in the local scopes of any lexically enclosing functions in your source code, from inner to outer; then in the current global scope (the module file); and finally in the built-in scope (the module `builtins`). `global` declarations make the search begin in the global (module file) scope instead.
- An assignment (`X = value`) creates or changes the name `X` in the current local scope, by default. If `X` is declared *global* within the function, the assignment creates or changes the name `X` in the enclosing module's scope instead. If, on the other hand, `X` is declared *nonlocal* within the function, the assignment changes the name `X` in the closest enclosing function's local scope.

Notice that the `global` declaration still maps variables to the enclosing module. When nested functions are present, variables in enclosing functions may be referenced, but they require `nonlocal` declarations to be changed.

Nested Scope Examples

To clarify the prior section's points, let's illustrate with some real code. Here is what an enclosing function scope looks like:

```
X = 99                      # Global scope name: not used

def f1():
    X = 88                  # Enclosing def local
    def f2():
        print(X)            # Reference made in nested def
    f2()

f1()                        # Prints 88: enclosing def local
```

First off, this is legal Python code: the `def` is simply an executable statement, which can appear anywhere any other statement can—including nested in another `def`. Here, the

nested `def` runs while a call to the function `f1` is running; it generates a function and assigns it to the name `f2`, a local variable within `f1`'s local scope. In a sense, `f2` is a temporary function that lives only during the execution of (and is visible only to code in) the enclosing `f1`.

But notice what happens inside `f2`: when it prints the variable `X`, it refers to the `X` that lives in the enclosing `f1` function's local scope. Because functions can access names in all physically enclosing `def` statements, the `X` in `f2` is automatically mapped to the `X` in `f1`, by the LEGB lookup rule.

This enclosing scope lookup works even if the enclosing function has already returned. For example, the following code defines a function that makes and returns another function:

```
def f1():
    X = 88
    def f2():
        print(X)          # Remembers X in enclosing def scope
    return f2             # Return f2 but don't call it

action = f1()             # Make, return function
action()                  # Call it now: prints 88
```

In this code, the call to `action` is really running the function we named `f2` when `f1` ran. `f2` remembers the enclosing scope's `X` in `f1`, even though `f1` is no longer active.

Factory functions

Depending on whom you ask, this sort of behavior is also sometimes called a *closure* or *factory* function. These terms refer to a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory. Although classes (described in Part VI of this book) are usually best at remembering state because they make it explicit with attribute assignments, such functions provide an alternative.

For instance, factory functions are sometimes used by programs that need to generate event handlers on the fly in response to conditions at runtime (e.g., user inputs that cannot be anticipated). Look at the following function, for example:

```
>>> def maker(N):
...     def action(X):                  # Make and return action
...         return X ** N               # action retains N from enclosing scope
...     return action
...
```

This defines an outer function that simply generates and returns a nested function, without calling it. If we call the outer function:

```
>>> f = maker(2)                        # Pass 2 to N
>>> f
<function action at 0x014720B0>
```

what we get back is a reference to the generated nested function—the one created by running the nested `def`. If we now call what we got back from the outer function:

```
>>> f(3)                          # Pass 3 to X, N remembers 2: 3 ** 2
9
>>> f(4)                          # 4 ** 2
16
```

it invokes the nested function—the one called `action` within `maker`. The most unusual part of this is that the nested function remembers integer `2`, the value of the variable `N` in `maker`, even though `maker` has returned and exited by the time we call `action`. In effect, `N` from the enclosing local scope is retained as state information attached to `action`, and we get back its argument squared.

If we now call the outer function again, we get back a new nested function with different state information attached. That is, we get the argument cubed instead of squared, but the original still squares as before:

```
>>> g = maker(3)                  # g remembers 3, f remembers 2
>>> g(3)                          # 3 ** 3
27
>>> f(3)                          # 3 ** 2
9
```

This works because each call to a factory function like this gets its own set of state information. In our case, the function we assign to name `g` remembers `3`, and `f` remembers `2`, because each has its own state information retained by the variable `N` in `maker`.

This is an advanced technique that you're unlikely to see very often in most code, except among programmers with backgrounds in functional programming languages. On the other hand, enclosing scopes are often employed by `lambda` function-creation expressions (discussed later in this chapter)—because they are expressions, they are almost always nested within a `def`. Moreover, function nesting is commonly used for *decorators* (explored in Chapter 38)—in some cases, it's the most reasonable coding pattern.

As a general rule, *classes* are better at "memory" like this because they make the state retention explicit in attributes. Short of using classes, though, globals, enclosing scope references like these, and default arguments are the main ways that Python functions can retain state information. To see how they compete, Chapter 18 provides complete coverage of defaults, but the next section gives enough of an introduction to get us started.

Retaining enclosing scopes' state with defaults

In earlier versions of Python, the sort of code in the prior section failed because nested `def`s did not do anything about scopes—a reference to a variable within `f2` would search only the local (`f2`), then global (the code outside `f1`), and then built-in scopes. Because it skipped the scopes of enclosing functions, an error would result. To work around this, programmers typically used *default argument values* to pass in and remember the objects in an enclosing scope:

```
def f1():
    x = 88
    def f2(x=x):                # Remember enclosing scope X with defaults
        print(x)
    f2()

f1()                            # Prints 88
```

This code works in all Python releases, and you'll still see this pattern in some existing Python code. In short, the syntax `arg = val` in a `def` header means that the argument `arg` will default to the value `val` if no real value is passed to `arg` in a call.

In the modified `f2` here, the `x=x` means that the argument `x` will default to the value of `x` in the enclosing scope—because the second `x` is evaluated before Python steps into the nested `def`, it still refers to the `x` in `f1`. In effect, the default remembers what `x` was in `f1` (i.e., the object `88`).

That's fairly complex, and it depends entirely on the timing of default value evaluations. In fact, the nested scope lookup rule was added to Python to make defaults unnecessary for this role—today, Python automatically remembers any values required in the enclosing scope for use in nested `def`s.

Of course, the best prescription for most code is simply to avoid nesting `def`s within `def`s, as it will make your programs much simpler. The following is an equivalent of the prior example that banishes the notion of nesting. Notice the forward reference in this code—it's OK to call a function defined after the function that calls it, as long as the second `def` runs before the first function is actually called. Code inside a `def` is never evaluated until the function is actually called:

```
>>> def f1():
...     x = 88                 # Pass x along instead of nesting
...     f2(x)                  # Forward reference okay
...
>>> def f2(x):
...     print(x)
...
>>> f1()
88
```

If you avoid nesting this way, you can almost forget about the nested scopes concept in Python, unless you need to code in the factory function style discussed earlier—at least, for `def` statements. `lambda`s, which almost naturally appear nested in `def`s, often rely on nested scopes, as the next section explains.

Nested scopes and lambdas

While they're rarely used in practice for `def`s themselves, you are more likely to care about nested function scopes when you start coding `lambda` expressions. We won't cover `lambda` in depth until Chapter 19, but in short, it's an expression that generates a new function to be called later, much like a `def` statement. Because it's an expression,

though, it can be used in places that `def` cannot, such as within list and dictionary literals.

Like a `def`, a `lambda` expression introduces a new local scope for the function it creates. Thanks to the enclosing scopes lookup layer, `lambda`s can see all the variables that live in the functions in which they are coded. Thus, the following code works, but only because the nested scope rules are applied:

```
def func():
    x = 4
    action = (lambda n: x ** n)          # x remembered from enclosing def
    return action

x = func()
print(x(2))                              # Prints 16, 4 ** 2
```

Prior to the introduction of nested function scopes, programmers used defaults to pass values from an enclosing scope into `lambda`s, just as for `def`s. For instance, the following works on all Python releases:

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)     # Pass x in manually
    return action
```

Because `lambda`s are expressions, they naturally (and even normally) nest inside enclosing `def`s. Hence, they are perhaps the biggest beneficiaries of the addition of enclosing function scopes in the lookup rules; in most cases, it is no longer necessary to pass values into `lambda`s with defaults.

Scopes versus defaults with loop variables

There is one notable exception to the rule I just gave: if a `lambda` or `def` defined within a function is nested inside a loop, and the nested function references an enclosing scope variable that is changed by that loop, all functions generated within the loop will have the same value—the value the referenced variable had in the last loop iteration.

For instance, the following attempts to build up a list of functions that each remember the current variable `i` from the enclosing scope:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                      # Tries to remember each i
...         acts.append(lambda x: i ** x)       # All remember same last i!
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
```

This doesn't quite work, though—because the enclosing scope variable is looked up when the nested functions are later *called*, they all effectively remember the same value

(the value the loop variable had on the *last* loop iteration). That is, we get back 4 to the power of 2 for each function in the list, because `i` is the same in all of them:

```
>>> acts[0](2)                          # All are 4 ** 2, value of last i
16
>>> acts[2](2)                          # This should be 2 ** 2
16
>>> acts[4](2)                          # This should be 4 ** 2
16
```

This is the one case where we still have to explicitly retain enclosing scope values with default arguments, rather than enclosing scope references. That is, to make this sort of code work, we must pass in the *current* value of the enclosing scope's variable with a default. Because defaults are evaluated when the nested function is *created* (not when it's later *called*), each remembers its own value for `i`:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                     # Use defaults instead
...         acts.append(lambda x, i=i: i ** x) # Remember current i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2)                          # 0 ** 2
0
>>> acts[2](2)                          # 2 ** 2
4
>>> acts[4](2)                          # 4 ** 2
16
```

This is a fairly obscure case, but it can come up in practice, especially in code that generates callback handler functions for a number of widgets in a GUI (e.g., button-press handlers). We'll talk more about defaults in Chapter 18 and `lambda`s in Chapter 19, so you may want to return and review this section later.‡

Arbitrary scope nesting

Before ending this discussion, I should note that scopes may nest arbitrarily, but only enclosing function `def` statements (not classes, described in Part VI) are searched:

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print(x)          # Found in f1's local scope!
...         f3()
```

‡ In the section "Function Gotchas" on page 518 at the end of this part of the book, we'll also see that there is an issue with using mutable objects like lists and dictionaries for default arguments (e.g., `def f(a=[])`)—because defaults are implemented as single objects attached to functions, mutable defaults retain state from call to call, rather then being initialized anew on each call. Depending on whom you ask, this is either considered a feature that supports state retention, or a strange wart on the language. More on this at the end of Chapter 20.

```
...     f2()
...
>>> f1()
99
```

Python will search the local scopes of *all* enclosing `def`s, from inner to outer, after the referencing function's local scope and before the module's global scope or built-ins. However, this sort of code is even less likely to pop up in practice. In Python, we say *flat is better than nested*—except in very limited contexts, your life (and the lives of your coworkers) will generally be better if you minimize nested function definitions.

The nonlocal Statement

In the prior section we explored the way that nested functions can *reference* variables in an enclosing function's scope, even if that function has already returned. It turns out that, as of Python 3.0, we can also *change* such enclosing scope variables, as long as we declare them in `nonlocal` statements. With this statement, nested `def`s can have both read and write access to names in enclosing functions.

The `nonlocal` statement is a close cousin to `global`, covered earlier. Like `global`, `nonlocal` declares that a name will be changed in an enclosing scope. Unlike `global`, though, `nonlocal` applies to a name in an enclosing function's scope, not the global module scope outside all `def`s. Also unlike `global`, `nonlocal` names must already exist in the enclosing function's scope when declared—they can exist only in enclosing functions and cannot be created by a first assignment in a nested `def`.

In other words, `nonlocal` both allows assignment to names in enclosing function scopes and limits scope lookups for such names to enclosing `def`s. The net effect is a more direct and reliable implementation of changeable scope information, for programs that do not desire or need classes with attributes.

nonlocal Basics

Python 3.0 introduces a new `nonlocal` statement, which has meaning only inside a function:

```
def func():
    nonlocal name1, name2, ...
```

This statement allows a nested function to change one or more names defined in a syntactically enclosing function's scope. In Python 2.X (including 2.6), when one function `def` is nested in another, the nested function can reference any of the names defined by assignment in the enclosing `def`'s scope, but it cannot change them. In 3.0, declaring the enclosing scopes' names in a `nonlocal` statement enables nested functions to assign and thus change such names as well.

This provides a way for enclosing functions to provide *writeable* state information, remembered when the nested function is later called. Allowing the state to change

makes it more useful to the nested function (imagine a counter in the enclosing scope, for instance). In 2.X, programmers usually achieve similar goals by using classes or other schemes. Because nested functions have become a more common coding pattern for state retention, though, `nonlocal` makes it more generally applicable.

Besides allowing names in enclosing `def`s to be changed, the `nonlocal` statement also forces the issue for references—just like the `global` statement, `nonlocal` causes searches for the names listed in the statement to begin in the enclosing `def`s' scopes, not in the local scope of the declaring function. That is, `nonlocal` also means "skip my local scope entirely."

In fact, the names listed in a `nonlocal` *must* have been previously defined in an enclosing `def` when the `nonlocal` is reached, or an error is raised. The net effect is much like `global`: `global` means the names reside in the enclosing module, and `nonlocal` means they reside in an enclosing `def`. `nonlocal` is even more strict, though—scope search is restricted to *only* enclosing `def`s. That is, nonlocal names can appear only in enclosing `def`s, not in the module's global scope or built-in scopes outside the `def`s.

The addition of `nonlocal` does not alter name reference scope rules in general; they still work as before, per the "LEGB" rule described earlier. The `nonlocal` statement mostly serves to allow names in enclosing scopes to be changed rather than just referenced. However, `global` and `nonlocal` statements do both restrict the lookup rules somewhat, when coded in a function:

- `global` makes scope lookup begin in the enclosing module's scope and allows names there to be assigned. Scope lookup continues on to the built-in scope if the name does not exist in the module, but assignments to global names always create or change them in the module's scope.
- `nonlocal` restricts scope lookup to just enclosing `def`s, requires that the names already exist there, and allows them to be assigned. Scope lookup does not continue on to the global or built-in scopes.

In Python 2.6, references to enclosing `def` scope names are allowed, but not assignment. However, you can still use classes with explicit attributes to achieve the same changeable state information effect as nonlocals (and you may be better off doing so in some contexts); globals and function attributes can sometimes accomplish similar goals as well. More on this in a moment; first, let's turn to some working code to make this more concrete.

nonlocal in Action

On to some examples, all run in 3.0. References to enclosing `def` scopes work as they do in 2.6. In the following, `tester` builds and returns the function `nested`, to be called later, and the `state` reference in `nested` maps the local scope of `tester` using the normal scope lookup rules:

```
C:\\misc>c:\python30\python

>>> def tester(start):
...     state = start             # Referencing nonlocals works normally
...     def nested(label):
...         print(label, state)   # Remembers state in enclosing scope
...     return nested
...
>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 0
```

Changing a name in an enclosing `def`'s scope is not allowed by default, though; this is the normal case in 2.6 as well:

```
>>> def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...         state += 1            # Cannot change by default (or in 2.6)
...     return nested
...
>>> F = tester(0)
>>> F('spam')
UnboundLocalError: local variable 'state' referenced before assignment
```

Using nonlocal for changes

Now, under 3.0, if we declare `state` in the `tester` scope as `nonlocal` within `nested`, we get to change it inside the nested function, too. This works even though `tester` has returned and exited by the time we call the returned `nested` function through the name `F`:

```
>>> def tester(start):
...     state = start             # Each call gets its own state
...     def nested(label):
...         nonlocal state        # Remembers state in enclosing scope
...         print(label, state)
...         state += 1            # Allowed to change it if nonlocal
...     return nested
...
>>> F = tester(0)
>>> F('spam')                     # Increments state on each call
spam 0
>>> F('ham')
ham 1
>>> F('eggs')
eggs 2
```

As usual with enclosing scope references, we can call the `tester` factory function multiple times to get multiple copies of its state in memory. The `state` object in the enclosing scope is essentially attached to the `nested` function object returned; each call makes a

new, distinct `state` object, such that updating one function's state won't impact the other. The following continues the prior listing's interaction:

```
>>> G = tester(42)              # Make a new tester that starts at 42
>>> G('spam')
spam 42

>>> G('eggs')                   # My state information updated to 43
eggs 43

>>> F('bacon')                  # But F's is where it left off: at 3
bacon 3                         # Each call has different state information
```

Boundary cases

There are a few things to watch out for. First, unlike the `global` statement, `nonlocal` names really *must* have previously been assigned in an enclosing `def`'s scope when a `nonlocal` is evaluated, or else you'll get an error—you cannot create them dynamically by assigning them anew in the enclosing scope:

```
>>> def tester(start):
...     def nested(label):
...         nonlocal state       # Nonlocals must already exist in enclosing def!
...         state = 0
...         print(label, state)
...     return nested
...
SyntaxError: no binding for nonlocal 'state' found

>>> def tester(start):
...     def nested(label):
...         global state         # Globals don't have to exist yet when declared
...         state = 0            # This creates the name in the module now
...         print(label, state)
...     return nested
...
>>> F = tester(0)
>>> F('abc')
abc 0
>>> state
0
```

Second, `nonlocal` restricts the scope lookup to just enclosing `def`s; nonlocals are not looked up in the enclosing module's global scope or the built-in scope outside all `def`s, even if they are already there:

```
>>> spam = 99
>>> def tester():
...     def nested():
...         nonlocal spam        # Must be in a def, not the module!
...         print('Current=', spam)
...         spam += 1
...     return nested
...
SyntaxError: no binding for nonlocal 'spam' found
```

These restrictions make sense once you realize that Python would not otherwise generally know which enclosing scope to create a brand new name in. In the prior listing, should `spam` be assigned in `tester`, or the module outside? Because this is ambiguous, Python must resolve nonlocals at function *creation* time, not function *call* time.

Why nonlocal?

Given the extra complexity of nested functions, you might wonder what the fuss is about. Although it's difficult to see in our small examples, state information becomes crucial in many programs. There are a variety of ways to "remember" information across function and method calls in Python. While there are tradeoffs for all, `nonlocal` does improve this story for enclosing scope references—the `nonlocal` statement allows multiple copies of changeable state to be retained in memory and addresses simple state-retention needs where classes may not be warranted.

As we saw in the prior section, the following code allows state to be retained and modified in an enclosing scope. Each call to `tester` creates a little self-contained *package of changeable information*, whose names do not clash with any other part of the program:

```
def tester(start):
    state = start                    # Each call gets its own state
    def nested(label):
        nonlocal state               # Remembers state in enclosing scope
        print(label, state)
        state += 1                   # Allowed to change it if nonlocal
    return nested

F = tester(0)
F('spam')
```

Unfortunately, this code only works in Python 3.0. If you are using Python 2.6, other options are available, depending on your goals. The next two sections present some alternatives.

Shared state with globals

One usual prescription for achieving the `nonlocal` effect in 2.6 and earlier is to simply move the state out to the *global scope* (the enclosing module):

```
>>> def tester(start):
...     global state                # Move it out to the module to change it
...     state = start               # global allows changes in module scope
...     def nested(label):
...         global state
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('spam')                       # Each call increments shared global state
```

```
spam 0
>>> F('eggs')
eggs 1
```

This works in this case, but it requires `global` declarations in both functions and is prone to name collisions in the global scope (what if "state" is already being used?). A worse, and more subtle, problem is that it only allows for a *single shared copy* of the state information in the module scope—if we call `tester` again, we'll wind up resetting the module's `state` variable, such that prior calls will see their `state` overwritten:

```
>>> G = tester(42)                  # Resets state's single copy in global scope
>>> G('toast')
toast 42

>>> G('bacon')
bacon 43

>>> F('ham')                        # Oops -- my counter has been overwritten!
ham 44
```

As shown earlier, when using `nonlocal` instead of `global`, each call to `tester` remembers its own unique copy of the `state` object.

State with classes (preview)

The other prescription for changeable state information in 2.6 and earlier is to use *classes with attributes* to make state information access more explicit than the implicit magic of scope lookup rules. As an added benefit, each instance of a class gets a fresh copy of the state information, as a natural byproduct of Python's object model.

We haven't explored classes in detail yet, but as a brief preview, here is a reformulation of the `tester`/`nested` functions used earlier as a class—state is recorded in objects explicitly as they are created. To make sense of this code, you need to know that a `def` within a `class` like this works exactly like a `def` outside of a `class`, except that the function's `self` argument automatically receives the implied subject of the call (an instance object created by calling the class itself):

```
>>> class tester:                   # Class-based alternative (see Part VI)
...     def __init__(self, start):  # On object construction,
...         self.state = start      # save state explicitly in new object
...     def nested(self, label):
...         print(label, self.state)    # Reference state explicitly
...         self.state += 1             # Changes are always allowed
...
>>> F = tester(0)                   # Create instance, invoke __init__
>>> F.nested('spam')                # F is passed to self
spam 0
>>> F.nested('ham')
ham 1

>>> G = tester(42)                  # Each instance gets new copy of state
>>> G.nested('toast')               # Changing one does not impact others
toast 42
```

```
>>> G.nested('bacon')
bacon 43

>>> F.nested('eggs')                    # F's state is where it left off
eggs 2
>>> F.state                             # State may be accessed outside class
3
```

With just slightly more magic, which we'll delve into later in this book, we could also make our class look like a callable function using operator overloading. `__call__` intercepts direct calls on an instance, so we don't need to call a named method:

```
>>> class tester:
...     def __init__(self, start):
...         self.state = start
...     def __call__(self, label):          # Intercept direct instance calls
...         print(label, self.state)        # So .nested() not required
...         self.state += 1
...
>>> H = tester(99)
>>> H('juice')                          # Invokes __call__
juice 99
>>> H('pancakes')
pancakes 100
```

Don't sweat the details in this code too much at this point in the book; we'll explore classes in depth in Part VI and will look at specific operator overloading tools like `__call__` in Chapter 29, so you may wish to file this code away for future reference. The point here is that classes can make state information more obvious, by leveraging explicit attribute assignment instead of scope lookups.

While using classes for state information is generally a good rule of thumb to follow, they might be overkill in cases like this, where state is a single counter. Such trivial state cases are more common than you might think; in such contexts, nested `def`s are sometimes more lightweight than coding classes, especially if you're not familiar with OOP yet. Moreover, there are some scenarios in which nested `def`s may actually work better than classes (see the description of *method decorators* in Chapter 38 for an example that is far beyond this chapter's scope).

State with function attributes

As a final state-retention option, we can also sometimes achieve the same effect as nonlocals with *function attributes*—user-defined names attached to functions directly. Here's a final version of our example based on this technique—it replaces a nonlocal with an attribute attached to the nested function. Although this scheme may not be as intuitive to some, it also allows the state variable to be accessed *outside* the nested function (with nonlocals, we can only see state variables within the nested `def`):

```
>>> def tester(start):
...     def nested(label):
...         print(label, nested.state)      # nested is in enclosing scope
...         nested.state += 1               # Change attr, not nested itself
```

```
...     nested.state = start              # Initial state after func defined
...     return nested
...
>>> F = tester(0)
>>> F('spam')                            # F is a 'nested' with state attached
spam 0
>>> F('ham')
ham 1
>>> F.state                              # Can access state outside functions too
2
>>>
>>> G = tester(42)                       # G has own state, doesn't overwrite F's
>>> G('eggs')
eggs 42
>>> F('ham')
ham 2
```

This code relies on the fact that the function name `nested` is a local variable in the `tester` scope enclosing `nested`; as such, it can be referenced freely inside `nested`. This code also relies on the fact that changing an object in-place is not an assignment to a name; when it increments `nested.state`, it is changing part of the object `nested` references, not the name `nested` itself. Because we're not really assigning a name in the enclosing scope, no `nonlocal` is needed.

As you can see, globals, nonlocals, classes, and function attributes all offer state-retention options. Globals only support shared data, classes require a basic knowledge of OOP, and both classes and function attributes allow state to be accessed outside the nested function itself. As usual, the best tool for your program depends upon your program's goals.

Chapter Summary

In this chapter, we studied one of two key concepts related to functions: *scopes* (how variables are looked up when they are used). As we learned, variables are considered local to the function definitions in which they are assigned, unless they are specifically declared to be global or nonlocal. We also studied some more advanced scope concepts here, including nested function scopes and function attributes. Finally, we looked at some general design ideas, such as the need to avoid globals and cross-file changes.

In the next chapter, we're going to continue our function tour with the second key function-related concept: argument passing. As we'll find, arguments are passed into a function by assignment, but Python also provides tools that allow functions to be flexible in how items are passed. Before we move on, let's take this chapter's quiz to review the scope concepts we've covered here.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> X = 'Spam'
>>> def func():
...     print(X)
...
>>> func()
```

2. What is the output of this code, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI!'
...
>>> func()
>>> print(X)
```

3. What does this code print, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     print(X)
...
>>> func()
>>> print(X)
```

4. What output does this code produce? Why?

```
>>> X = 'Spam'
>>> def func():
...     global X
...     X = 'NI'
...
>>> func()
>>> print(X)
```

5. What about this code—what's the output, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     def nested():
...         print(X)
...     nested()
...
>>> func()
>>> X
```

6. How about this example: what is its output in Python 3.0, and why?

```
>>> def func():
...     X = 'NI'
...     def nested():
...         nonlocal X
...         X = 'Spam'
...     nested()
...     print(X)
...
>>> func()
```

7. Name three or more ways to retain state information in a Python function.

Test Your Knowledge: Answers

1. The output here is `'Spam'`, because the function references a global variable in the enclosing module (because it is not assigned in the function, it is considered global).
2. The output here is `'Spam'` again because assigning the variable inside the function makes it a local and effectively hides the global of the same name. The `print` statement finds the variable unchanged in the global (module) scope.
3. It prints `'NI'` on one line and `'Spam'` on another, because the reference to the variable within the function finds the assigned local and the reference in the `print` statement finds the global.
4. This time it just prints `'NI'` because the global declaration forces the variable assigned inside the function to refer to the variable in the enclosing global scope.
5. The output in this case is again `'NI'` on one line and `'Spam'` on another, because the `print` statement in the nested function finds the name in the enclosing function's local scope, and the `print` at the end finds the variable in the global scope.
6. This example prints `'Spam'`, because the `nonlocal` statement (available in Python 3.0 but not 2.6) means that the assignment to `X` inside the nested function changes `X` in the enclosing function's local scope. Without this statement, this assignment would classify `X` as local to the nested function, making it a different variable; the code would then print `'NI'` instead.
7. Although the values of local variables go away when a function returns, you can make a Python function retain state information by using shared global variables, enclosing function scope references within nested functions, or using default argument values. Function attributes can sometimes allow state to be attached to the function itself, instead of looked up in scopes. Another alternative, using OOP with classes, sometimes supports state retention better than any of the scope-based techniques because it makes it explicit with attribute assignments; we'll explore this option in Part VI.

CHAPTER 18
Arguments

Chapter 17 explored the details behind Python's *scopes*—the places where variables are defined and looked up. As we learned, the place where a name is defined in our code determines much of its meaning. This chapter continues the function story by studying the concepts in Python *argument passing*—the way that objects are sent to functions as inputs. As we'll see, arguments (a.k.a. parameters) are assigned to names in a function, but they have more to do with object references than with variable scopes. We'll also find that Python provides extra tools, such as keywords, defaults, and arbitrary argument collectors, that allow for wide flexibility in the way arguments are sent to a function.

Argument-Passing Basics

Earlier in this part of the book, I noted that arguments are passed by *assignment*. This has a few ramifications that aren't always obvious to beginners, which I'll expand on in this section. Here is a rundown of the key points in passing arguments to functions:

- **Arguments are passed by automatically assigning objects to local variable names**. Function arguments—references to (possibly) shared objects sent by the caller—are just another instance of Python assignment at work. Because references are implemented as pointers, all arguments are, in effect, passed by pointer. Objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller**. Argument names in the function header become new, local names when the function runs, in the scope of the function. There is no aliasing between function argument names and variable names in the scope of the caller.
- **Changing a mutable object argument in a function may impact the caller**. On the other hand, as arguments are simply assigned to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Mutable arguments can be input and output for functions.

For more details on *references*, see Chapter 6; everything we learned there also applies to function arguments, though the assignment to argument names is automatic and implicit.

Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the C language's argument-passing model in practice:

- **Immutable arguments are effectively passed "by value."** Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in-place anyhow, the effect is much like making a copy.
- **Mutable arguments are effectively passed "by pointer."** Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers—mutable objects can be changed in-place in the function, much like C arrays.

Of course, if you've never used C, Python's argument-passing mode will seem simpler still—it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.

Arguments and Shared References

To illustrate argument-passing properties at work, consider the following code:

```
>>> def f(a):                  # a is assigned to (references) passed object
...     a = 99                 # Changes local variable a only
...
>>> b = 88
>>> f(b)                       # a and b both reference same 88 initially
>>> print(b)                   # b is not changed
88
```

In this example the variable `a` is assigned the object `88` at the moment the function is called with `f(b)`, but `a` lives only within the called function. Changing `a` inside the function has no effect on the place where the function is called; it simply resets the local variable `a` to a completely different object.

That's what is meant by a lack of name *aliasing*—assignment to an argument name inside a function (e.g., `a=99`) does not magically change a variable like `b` in the scope of the function call. Argument names may share passed objects initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends.

At least, that's the case for assignment to argument *names* themselves. When arguments are passed *mutable* objects like lists and dictionaries, we also need to be aware that in-place changes to such *objects* may live on after a function exits, and hence impact callers. Here's an example that demonstrates this behavior:

```
>>> def changer(a, b):          # Arguments assigned references to objects
...     a = 2                   # Changes local name's value only
...     b[0] = 'spam'           # Changes shared object in-place
...
>>> X = 1
>>> L = [1, 2]                  # Caller
>>> changer(X, L)               # Pass immutable and mutable objects
>>> X, L                        # X is unchanged, L is different!
(1, ['spam', 2])
```

In this code, the `changer` function assigns values to argument `a` itself, and to a component of the object referenced by argument `b`. These two assignments within the function are only slightly different in syntax but have radically different results:

- Because `a` is a local variable name in the function's scope, the first assignment has no effect on the caller—it simply changes the local variable `a` to reference a completely different object, and does not change the binding of the name `X` in the caller's scope. This is the same as in the prior example.
- Argument `b` is a local variable name, too, but it is passed a mutable object (the list that `L` references in the caller's scope). As the second assignment is an in-place object change, the result of the assignment to `b[0]` in the function impacts the value of `L` after the function returns.

Really, the second assignment statement in `changer` doesn't change `b`—it changes part of the object that `b` currently references. This in-place change impacts the caller only because the changed object outlives the function call. The name `L` hasn't changed either—it still references the same, changed object—but it seems as though `L` differs after the call because the value it references has been modified within the function.

Figure 18-1 illustrates the name/object bindings that exist immediately after the function has been called, and before its code has run.

If this example is still confusing, it may help to notice that the effect of the automatic assignments of the passed-in arguments is the same as running a series of simple assignment statements. In terms of the first argument, the assignment has no effect on the caller:

```
>>> X = 1
>>> a = X               # They share the same object
>>> a = 2               # Resets 'a' only, 'X' is still 1
>>> print(X)
1
```

The assignment through the second argument does affect a variable at the call, though, because it is an in-place object change:

```
>>> L = [1, 2]
>>> b = L               # They share the same object
>>> b[0] = 'spam'       # In-place change: 'L' sees the change too
>>> print(L)
['spam', 2]
```

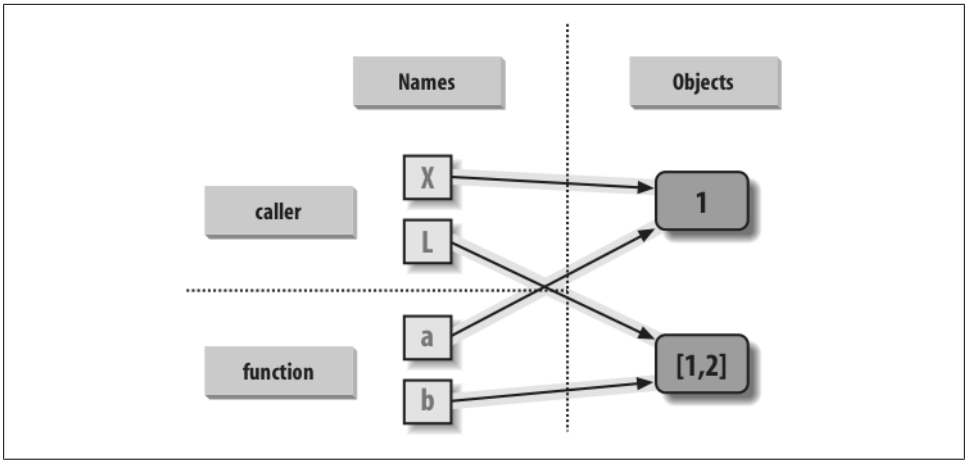

Figure 18-1. References: arguments. Because arguments are passed by assignment, argument names in the function may share objects with variables in the scope of the call. Hence, in-place changes to mutable arguments in a function can impact the caller. Here, a and b in the function initially reference the objects referenced by variables X and L when the function is first called. Changing the list through variable b makes L appear different after the call returns.

If you recall our discussions about shared mutable objects in Chapters 6 and 9, you'll recognize the phenomenon at work: changing a mutable object in-place can impact other references to that object. Here, the effect is to make one of the arguments work like both an input and an *output* of the function.

Avoiding Mutable Argument Changes

This behavior of in-place changes to mutable arguments isn't a bug—it's simply the way argument passing works in Python. Arguments are passed to functions by reference (a.k.a. pointer) by default because that is what we normally want. It means we can pass large objects around our programs without making multiple copies along the way, and we can easily update these objects as we go. In fact, as we'll see in Part VI, Python's class model *depends* upon changing a passed-in "self" argument in-place, to update object state.

If we don't want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects, as we learned in Chapter 6. For function arguments, we can always copy the list at the point of call:

```
L = [1, 2]
changer(X, L[:])        # Pass a copy, so our 'L' does not change
```

We can also copy within the function itself, if we never want to change passed-in objects, regardless of how the function is called:

```
def changer(a, b):
    b = b[:]            # Copy input list so we don't impact caller
```

```
a = 2
b[0] = 'spam'          # Changes our list copy only
```

Both of these copying schemes don't stop the function from changing the object—they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to immutable objects to force the issue. Tuples, for example, throw an exception when changes are attempted:

```
L = [1, 2]
changer(X, tuple(L))    # Pass a tuple, so changes are errors
```

This scheme uses the built-in `tuple` function, which builds a new tuple out of all the items in a sequence (really, any iterable). It's also something of an extreme—because it forces the function to be written to never change passed-in arguments, this solution might impose more limitations on the function than it should, and so should generally be avoided (you never know when changing arguments might come in handy for other calls in the future). Using this technique will also make the function lose the ability to call any list-specific methods on the argument, including methods that do not change the object in-place.

The main point to remember here is that functions might update mutable objects like lists and dictionaries passed into them. This isn't necessarily a problem if it's expected, and often serves useful purposes. Moreover, functions that change passed-in mutable objects in place are probably designed and intended to do so—the change is likely part of a well-defined API that you shouldn't violate by making copies.

However, you do have to be aware of this property—if objects change out from under you unexpectedly, check whether a called function might be responsible, and make copies when objects are passed if needed.

Simulating Output Parameters

We've already discussed the `return` statement and used it in a few examples. Here's another way to use this statement: because `return` can send back any sort of object, it can return *multiple values* by packaging them in a tuple or other collection type. In fact, although Python doesn't support what some languages label "call-by-reference" argument passing, we can usually simulate it by returning tuples and assigning the results back to the original argument names in the caller:

```
>>> def multiple(x, y):
...     x = 2                  # Changes local names only
...     y = [3, 4]
...     return x, y            # Return new values in a tuple
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)     # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

It looks like the code is returning two values here, but it's really just one—a two-item tuple with the optional surrounding parentheses omitted. After the call returns, we can use tuple assignment to unpack the parts of the returned tuple. (If you've forgotten why this works, flip back to "Tuples" on page 225 in Chapter 4, Chapter 9, and "Assignment Statements" on page 279 in Chapter 11.) The net effect of this coding pattern is to simulate the output parameters of other languages by explicit assignments. `X` and `L` change after the call, but only because the code said so.

Unpacking arguments in Python 2.X: The preceding example unpacks a tuple returned by the function with tuple assignment. In Python 2.6, it's also possible to automatically unpack tuples in arguments passed to a function. In 2.6, a function defined by this header:

```
def f((a, (b, c))):
```

can be called with tuples that match the expected structure: `f((1, (2, 3)))` assigns `a`, `b`, and `c` to `1`, `2`, and `3`, respectively. Naturally, the passed tuple can also be an object created before the call (`f(T)`). This `def` syntax is no longer supported in Python 3.0. Instead, code this function as:

```
def f(T): (a, (b, c)) = T
```

to unpack in an explicit assignment statement. This explicit form works in both 3.0 and 2.6. Argument unpacking is an obscure and rarely used feature in Python 2.X. Moreover, a function header in 2.6 supports only the tuple form of sequence assignment; more general sequence assignments (e.g., `def f((a, [b, c])):`) fail on syntax errors in 2.6 as well and require the explicit assignment form.

Tuple unpacking argument syntax is also disallowed by 3.0 in `lambda` function argument lists: see the sidebar "Why You Will Care: List Comprehensions and map" on page 491 for an example. Somewhat asymmetrically, tuple unpacking assignment is still automatic in 3.0 `for` loops targets, though; see Chapter 13 for examples.

Special Argument-Matching Modes

As we've just seen, arguments are always passed by *assignment* in Python; names in the `def` header are assigned to passed-in objects. On top of this model, though, Python provides additional tools that alter the way the argument objects in a call are *matched* with argument names in the header prior to assignment. These tools are all optional, but they allow us to write functions that support more flexible calling patterns, and you may encounter some libraries that require them.

By default, arguments are matched by position, from left to right, and you must pass exactly as many arguments as there are argument names in the function header. However, you can also specify matching by name, default values, and collectors for extra arguments.

The Basics

Before we go into the syntactic details, I want to stress that these special modes are optional and only have to do with matching objects to names; the underlying passing mechanism after the matching takes place is still assignment. In fact, some of these tools are intended more for people writing libraries than for application developers. But because you may stumble across these modes even if you don't code them yourself, here's a synopsis of the available tools:

Positionals: matched from left to right
: The normal case, which we've mostly been using so far, is to match passed argument values to argument names in a function header by position, from left to right.

Keywords: matched by argument name
: Alternatively, callers can specify which argument in the function is to receive a value by using the argument's name in the call, with the `name=value` syntax.

Defaults: specify values for arguments that aren't passed
: Functions themselves can specify default values for arguments to receive if the call passes too few values, again using the `name=value` syntax.

Varargs collecting: collect arbitrarily many positional or keyword arguments
: Functions can use special arguments preceded with one or two * characters to collect an arbitrary number of extra arguments (this feature is often referred to as *varargs*, after the varargs feature in the C language, which also supports variable-length argument lists).

Varargs unpacking: pass arbitrarily many positional or keyword arguments
: Callers can also use the * syntax to unpack argument collections into discrete, separate arguments. This is the inverse of a * in a function header—in the header it means collect arbitrarily many arguments, while in the call it means pass arbitrarily many arguments.

Keyword-only arguments: arguments that must be passed by name
: In Python 3.0 (but not 2.6), functions can also specify arguments that must be passed by name with keyword arguments, not by position. Such arguments are typically used to define configuration options in addition to actual arguments.

Matching Syntax

Table 18-1 summarizes the syntax that invokes the special argument-matching modes.

Table 18-1. Function argument-matching forms

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*sequence)`	Caller	Pass all objects in sequence as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in `dict` as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*args, name)`	Function	Arguments that must be passed by keyword only in calls (3.0)
`def func(*, name=value)`		

These special matching modes break down into function calls and definitions as follows:

- In a *function call* (the first four rows of the table), simple values are matched by position, but using the `name=value` form tells Python to match by name to arguments instead; these are called *keyword arguments*. Using a `*sequence` or `**dict` in a call allows us to package up arbitrarily many positional or keyword objects in sequences and dictionaries, respectively, and unpack them as separate, individual arguments when they are passed to the function.
- In a *function header* (the rest of the table), a simple `name` is matched by position or name depending on how the caller passes it, but the `name=value` form specifies a *default value*. The `*name` form collects any extra unmatched positional arguments in a tuple, and the `**name` form collects extra keyword arguments in a dictionary. In Python 3.0 and later, any normal or defaulted argument names following a `*name` or a bare * are *keyword-only* arguments and must be passed by keyword in calls.

Of these, keyword arguments and defaults are probably the most commonly used in Python code. We've informally used both of these earlier in this book:

- We've already used *keywords* to specify options to the 3.0 `print` function, but they are more general—keywords allow us to label any argument with its name, to make calls more informational.

- We met *defaults* earlier, too, as a way to pass in values from the enclosing function's scope, but they are also more general—they allow us to make any argument optional, providing its default value in a function definition.

As we'll see, the combination of defaults in a function header and keywords in a call further allows us to pick and choose which defaults to override.

In short, special argument-matching modes let you be fairly liberal about how many arguments must be passed to a function. If a function specifies defaults, they are used if you pass *too few* arguments. If a function uses the * variable argument list forms, you can pass *too many* arguments; the * names collect the extra arguments in data structures for processing in the function.

The Gritty Details

If you choose to use and combine the special argument-matching modes, Python will ask you to follow these ordering rules:

- In a function *call*, arguments must appear in this order: any positional arguments (`value`), followed by a combination of any keyword arguments (`name=value`) and the `*sequence` form, followed by the `**dict` form.
- In a function *header*, arguments must appear in this order: any normal arguments (`name`), followed by any default arguments (`name=value`), followed by the `*name` (or * in 3.0) form if present, followed by any `name` or `name=value` keyword-only arguments (in 3.0), followed by the `**name` form.

In both the call and header, the `**arg` form must appear last if present. If you mix arguments in any other order, you will get a syntax error because the combinations can be ambiguous. The steps that Python internally carries out to match arguments before assignment can roughly be described as follows:

1. Assign nonkeyword arguments by position.
2. Assign keyword arguments by matching names.
3. Assign extra nonkeyword arguments to `*name` tuple.
4. Assign extra keyword arguments to `**name` dictionary.
5. Assign default values to unassigned arguments in header.

After this, Python checks to make sure each argument is passed just one value; if not, an error is raised. When all matching is complete, Python assigns argument names to the objects passed to them.

The actual matching algorithm Python uses is a bit more complex (it must also account for keyword-only arguments in 3.0, for instance), so we'll defer to Python's standard language manual for a more exact description. It's not required reading, but tracing Python's matching algorithm may help you to understand some convoluted cases, especially when modes are mixed.

In Python 3.0, argument names in a function header can also have *annotation* values, specified as `name:value` (or `name:value=default` when defaults are present). This is simply additional syntax for arguments and does not augment or change the argument-ordering rules described here. The function itself can also have an annotation value, given as `def f()->value`. See the discussion of function annotation in Chapter 19 for more details.

Keyword and Default Examples

This is all simpler in code than the preceding descriptions may imply. If you don't use any special matching syntax, Python matches names by position from left to right, like most other languages. For instance, if you define a function that requires three arguments, you must call it with three arguments:

```
>>> def f(a, b, c): print(a, b, c)
...
```

Here, we pass them by position—`a` is matched to `1`, `b` is matched to `2`, and so on (this works the same in Python 3.0 and 2.6, but extra tuple parentheses are displayed in 2.6 because we're using 3.0 `print` calls):

```
>>> f(1, 2, 3)
1 2 3
```

Keywords

In Python, though, you can be more specific about what goes where when you call a function. Keyword arguments allow us to match by *name*, instead of by position:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

The `c=3` in this call, for example, means send `3` to the argument named `c`. More formally, Python matches the name `c` in the call to the argument named `c` in the function definition's header, and then passes the value `3` to that argument. The net effect of this call is the same as that of the prior call, but notice that the left-to-right order of the arguments no longer matters when keywords are used because arguments are matched by name, not by position. It's even possible to combine positional and keyword arguments in a single call. In this case, all positionals are matched first from left to right in the header, before keywords are matched by name:

```
>>> f(1, c=3, b=2)
1 2 3
```

When most people see this the first time, they wonder why one would use such a tool. Keywords typically have two roles in Python. First, they make your calls a bit more self-documenting (assuming that you use better argument names than `a`, `b`, and `c`). For example, a call of this form:

```
func(name='Bob', age=40, job='dev')
```

is much more meaningful than a call with three naked values separated by commas—the keywords serve as labels for the data in the call. The second major use of keywords occurs in conjunction with defaults, which we turn to next.

Defaults

We talked about defaults in brief earlier, when discussing nested function scopes. In short, defaults allow us to make selected function arguments optional; if not passed a value, the argument is assigned its default before the function runs. For example, here is a function that requires one argument and defaults two:

```
>>> def f(a, b=2, c=3): print(a, b, c)
...
```

When we call this function, we must provide a value for `a`, either by position or by keyword; however, providing values for `b` and `c` is optional. If we don't pass values to `b` and `c`, they default to `2` and `3`, respectively:

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

If we pass two values, only `c` gets its default, and with three values, no defaults are used:

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Finally, here is how the keyword and default features interact. Because they subvert the normal left-to-right positional mapping, keywords allow us to essentially skip over arguments with defaults:

```
>>> f(1, c=6)
1 2 6
```

Here, `a` gets `1` by position, `c` gets `6` by keyword, and `b`, in between, defaults to `2`.

Be careful not to confuse the special `name=value` syntax in a function header and a function call; in the call it means a match-by-name keyword argument, while in the header it specifies a default for an optional argument. In both cases, this is not an assignment statement (despite its appearance); it is special syntax for these two contexts, which modifies the default argument-matching mechanics.

Combining keywords and defaults

Here is a slightly larger example that demonstrates keywords and defaults in action. In the following, the caller must always pass at least two arguments (to match `spam` and `eggs`), but the other two are optional. If they are omitted, Python assigns `toast` and `ham` to the defaults specified in the header:

```
def func(spam, eggs, toast=0, ham=0):    # First 2 required
    print((spam, eggs, toast, ham))

func(1, 2)                               # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                   # Output: (1, 0, 0, 1)
func(spam=1, eggs=0)                     # Output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)            # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                         # Output: (1, 2, 3, 4)
```

Notice again that when keyword arguments are used in the call, the order in which the arguments are listed doesn't matter; Python matches by name, not by position. The caller must supply values for `spam` and `eggs`, but they can be matched by position or by name. Again, keep in mind that the form `name=value` means different things in the call and the `def`: a keyword in the call and a default in the header.

Arbitrary Arguments Examples

The last two matching extensions, * and **, are designed to support functions that take any number of arguments. Both can appear in either the function definition or a function call, and they have related purposes in the two locations.

Collecting arguments

The first use, in the function definition, collects unmatched *positional* arguments into a tuple:

```
>>> def f(*args): print(args)
...
```

When this function is called, Python collects all the positional arguments into a new tuple and assigns the variable `args` to that tuple. Because it is a normal tuple object, it can be indexed, stepped through with a `for` loop, and so on:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

The ** feature is similar, but it only works for *keyword* arguments—it collects them into a new dictionary, which can then be processed with normal dictionary tools. In a sense, the ** form allows you to convert from keywords to dictionaries, which you can then step through with `keys` calls, dictionary iterators, and the like:

```
>>> def f(**args): print(args)
...
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Finally, function headers can combine normal arguments, the *, and the ** to implement wildly flexible call signatures. For instance, in the following, `1` is passed to `a` by position, `2` and `3` are collected into the `pargs` positional tuple, and `x` and `y` wind up in the `kargs` keyword dictionary:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

In fact, these features can be combined in even more complex ways that may seem ambiguous at first glance—an idea we will revisit later in this chapter. First, though, let's see what happens when * and ** are coded in function calls instead of definitions.

Unpacking arguments

In recent Python releases, we can use the * syntax when we call a function, too. In this context, its meaning is the inverse of its meaning in the function definition—it unpacks a collection of arguments, rather than building a collection of arguments. For example, we can pass four arguments to a function in a tuple and let Python unpack them into individual arguments:

```
>>> def func(a, b, c, d): print(a, b, c, d)
...
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)
1 2 3 4
```

Similarly, the ** syntax in a function call unpacks a dictionary of key/value pairs into separate keyword arguments:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)
1 2 3 4
```

Again, we can combine normal, positional, and keyword arguments in the call in very flexible ways:

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})
1 2 4 4

>>> func(1, *(2, 3), **{'d': 4})
1 2 3 4

>>> func(1, c=3, *(2,), **{'d': 4})
```

```
1 2 3 4

>>> func(1, *(2, 3), d=4)
1 2 3 4

>>> f(1, *(2,), c=3, **{'d':4})
1 2 3 4
```

This sort of code is convenient when you cannot predict the number of arguments that will be passed to a function when you write your script; you can build up a collection of arguments at runtime instead and call the function generically this way. Again, don't confuse the */** syntax in the function header and the function call—in the header it collects any number of arguments, while in the call it unpacks any number of arguments.

As we saw in Chapter 14, the `*pargs` form in a call is an *iteration context*, so technically it accepts any iterable object, not just tuples or other sequences as shown in the examples here. For instance, a file object works after the `*`, and unpacks its lines into individual arguments (e.g., `func(*open('fname'))`).

This generality is supported in both Python 3.0 and 2.6, but it holds true only for *calls*—a `*pargs` in a call allows any iterable, but the same form in a `def` header always bundles extra arguments into a *tuple*. This header behavior is similar in spirit and syntax to the `*` in Python 3.0 extended sequence unpacking assignment forms we met in Chapter 11 (e.g., `x, *y = z`), though that feature always creates lists, not tuples.

Applying functions generically

The prior section's examples may seem obtuse, but they are used more often than you might expect. Some programs need to call arbitrary functions in a generic fashion, without knowing their names or arguments ahead of time. In fact, the real power of the special "varargs" call syntax is that you don't need to know how many arguments a function call requires before you write a script. For example, you can use `if` logic to select from a set of functions and argument lists, and call any of them generically:

```
if <test>:
    action, args = func1, (1,)          # Call func1 with 1 arg in this case
else:
    action, args = func2, (1, 2, 3)     # Call func2 with 3 args here
...
action(*args)                           # Dispatch generically
```

More generally, this varargs call syntax is useful any time you cannot predict the arguments list. If your user selects an arbitrary function via a user interface, for instance, you may be unable to hardcode a function call when writing your script. To work around this, simply build up the arguments list with sequence operations, and call it with starred names to unpack the arguments:

```
>>> args = (2,3)
>>> args += (4,)
>>> args
(2, 3, 4)
>>> func(*args)
```

Because the arguments list is passed in as a tuple here, the program can build it at runtime. This technique also comes in handy for functions that test or time other functions. For instance, in the following code we support any function with any arguments by passing along whatever arguments were sent in:

```
def tracer(func, *pargs, **kargs):          # Accept arbitrary arguments
    print('calling:', func.__name__)
    return func(*pargs, **kargs)            # Pass along arbitrary arguments

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))
```

When this code is run, arguments are collected by the tracer and then *propagated* with varargs call syntax:

```
calling: func
10
```

We'll see larger examples of such roles later in this book; see especially the sequence timing example in Chapter 20 and the various decorator tools we will code in Chapter 38.

The defunct apply built-in (Python 2.6)

Prior to Python 3.0, the effect of the `*args` and `**args` varargs call syntax could be achieved with a built-in function named `apply`. This original technique has been removed in 3.0 because it is now redundant (3.0 cleans up many such dusty tools that have been subsumed over the years). It's still available in Python 2.6, though, and you may come across it in older 2.X code.

In short, the following are equivalent prior to Python 3.0:

```
func(*pargs, **kargs)           # Newer call syntax: func(*sequence, **dict)

apply(func, pargs, kargs)       # Defunct built-in:  apply(func, sequence, dict)
```

For example, consider the following function, which accepts any number of positional or keyword arguments:

```
>>> def echo(*args, **kwargs): print(args, kwargs)
...
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

In Python 2.6, we can call it generically with `apply`, or with the call syntax that is now required in 3.0:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}

>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}

>>> echo(*pargs, **kargs)
(1, 2) {'a': 3, 'b': 4}
```

The unpacking call syntax form is newer than the `apply` function, is preferred in general, and is required in 3.0. Apart from its symmetry with the `*pargs` and `**kargs` collector forms in `def` headers, and the fact that it requires fewer keystrokes overall, the newer call syntax also allows us to pass along additional arguments without having to manually extend argument sequences or dictionaries:

```
>>> echo(0, c=5, *pargs, **kargs)      # Normal, keyword, *sequence, **dictionary
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

That is, the call syntax form is *more general*. Since it's required in 3.0, you should now disavow all knowledge of `apply` (unless, of course, it appears in 2.X code you must use or maintain...).

Python 3.0 Keyword-Only Arguments

Python 3.0 generalizes the ordering rules in function headers to allow us to specify *keyword-only arguments*—arguments that must be passed by keyword only and will never be filled in by a positional argument. This is useful if we want a function to both process any number of arguments and accept possibly optional configuration options.

Syntactically, keyword-only arguments are coded as named arguments that appear after `*args` in the arguments list. All such arguments must be passed using keyword syntax in the call. For example, in the following, `a` may be passed by name or position, `b` collects any extra positional arguments, and `c` must be passed by keyword only:

```
>>> def kwonly(a, *b, c):
...     print(a, b, c)
...
>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() needs keyword-only argument c
```

We can also use a * character by itself in the arguments list to indicate that a function does not accept a variable-length argument list but still expects all arguments following the * to be passed as keywords. In the next function, `a` may be passed by position or name again, but `b` and `c` must be keywords, and no extra positionals are allowed:

```
>>> def kwonly(a, *, b, c):
...     print(a, b, c)
...
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
>>> kwonly(1)
TypeError: kwonly() needs keyword-only argument b
```

You can still use defaults for keyword-only arguments, even though they appear after the * in the function header. In the following code, `a` may be passed by name or position, and `b` and `c` are optional but must be passed by keyword if used:

```
>>> def kwonly(a, *, b='spam', c='ham'):
...     print(a, b, c)
...
>>> kwonly(1)
1 spam ham
>>> kwonly(1, c=3)
1 spam 3
>>> kwonly(a=1)
1 spam ham
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)
```

In fact, keyword-only arguments with defaults are optional, but those without defaults effectively become required keywords for the function:

```
>>> def kwonly(a, *, b, c='spam'):
...     print(a, b, c)
...
>>> kwonly(1, b='eggs')
1 eggs spam
>>> kwonly(1, c='eggs')
TypeError: kwonly() needs keyword-only argument b
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)

>>> def kwonly(a, *, b=1, c, d=2):
...     print(a, b, c, d)
...
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() needs keyword-only argument c
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
```

Ordering rules

Finally, note that keyword-only arguments must be specified after a single star, not two—named arguments cannot appear after the **`**args`** arbitrary keywords form, and a ** can't appear by itself in the arguments list. Both attempts generate a syntax error:

```
>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax
```

This means that in a function *header*, keyword-only arguments must be coded before the **`**args`** arbitrary keywords form and after the **`*args`** arbitrary positional form, when both are present. Whenever an argument name appears before **`*args`**, it is a possibly default positional argument, not keyword-only:

```
>>> def f(a, *b, **d, c=6): print(a, b, c, d)          # Keyword-only before **!
SyntaxError: invalid syntax

>>> def f(a, *b, c=6, **d): print(a, b, c, d)          # Collect args in header
...
>>> f(1, 2, 3, x=4, y=5)                               # Default used
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, 2, 3, x=4, y=5, c=7)                          # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, 2, 3, c=7, x=4, y=5)                          # Anywhere in keywords
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> def f(a, c=6, *b, **d): print(a, b, c, d)          # c is not keyword-only!
...
>>> f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

In fact, similar ordering rules hold true in function *calls*: when keyword-only arguments are passed, they must appear before a **`**args`** form. The keyword-only argument can be coded either before or after the **`*args`**, though, and may be included in **`**args`**:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d)          # KW-only between * and **
...
>>> f(1, *(2, 3), **dict(x=4, y=5))                    # Unpack args at call
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5), c=7)               # Keywords before **args!
SyntaxError: invalid syntax

>>> f(1, *(2, 3), c=7, **dict(x=4, y=5))               # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, c=7, *(2, 3), **dict(x=4, y=5))               # After or before *
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5, c=7))               # Keyword-only in **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Trace through these cases on your own, in conjunction with the general argument-ordering rules described formally earlier. They may appear to be worst cases in the artificial examples here, but they can come up in real practice, especially for people who write libraries and tools for other Python programmers to use.

Why keyword-only arguments?

So why care about keyword-only arguments? In short, they make it easier to allow a function to accept both any number of positional arguments to be processed, and configuration options passed as keywords. While their use is optional, without keyword-only arguments extra work may be required to provide defaults for such options and to verify that no superfluous keywords were passed.

Imagine a function that processes a set of passed-in objects and allows a tracing flag to be passed:

```
process(X, Y, Z)                  # use flag's default
process(X, Y, notify=True)        # override flag default
```

Without keyword-only arguments we have to use both `*args` and `**args` and manually inspect the keywords, but with keyword-only arguments less code is required. The following guarantees that no positional argument will be incorrectly matched against `notify` and requires that it be a keyword if passed:

```
def process(*args, notify=False): ...
```

Since we're going to see a more realistic example of this later in this chapter, in "Emulating the Python 3.0 print Function" on page 457, I'll postpone the rest of this story until then. For an additional example of keyword-only arguments in action, see the iteration options timing case study in Chapter 20. And for additional function definition enhancements in Python 3.0, stay tuned for the discussion of function annotation syntax in Chapter 19.

The min Wakeup Call!

Time for something more realistic. To make this chapter's concepts more concrete, let's work through an exercise that demonstrates a practical application of argument-matching tools.

Suppose you want to code a function that is able to compute the minimum value from an arbitrary set of arguments and an arbitrary set of object data types. That is, the function should accept zero or more arguments, as many as you wish to pass. Moreover, the function should work for all kinds of Python object types: numbers, strings, lists, lists of dictionaries, files, and even `None`.

The first requirement provides a natural example of how the * feature can be put to good use—we can collect arguments into a tuple and step over each of them in turn with a simple `for` loop. The second part of the problem definition is easy: because every

object type supports comparisons, we don't have to specialize the function per type (an application of polymorphism); we can simply compare objects blindly and let Python worry about what sort of comparison to perform.

Full Credit

The following file shows three ways to code this operation, at least one of which was suggested by a student in one of my courses:

- The first function fetches the first argument (`args` is a tuple) and traverses the rest by slicing off the first (there's no point in comparing an object to itself, especially if it might be a large structure).
- The second version lets Python pick off the first and rest of the arguments automatically, and so avoids an index and slice.
- The third converts from a tuple to a list with the built-in `list` call and employs the list `sort` method.

The `sort` method is coded in C, so it can be quicker than the other approaches at times, but the linear scans of the first two techniques will make them faster most of the time.* The file *mins.py* contains the code for all three solutions:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)              # Or, in Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print(min1(3,4,1,2))
```

* Actually, this is fairly complicated. The Python `sort` routine is coded in C and uses a highly optimized algorithm that attempts to take advantage of partial ordering in the items to be sorted. It's named "timsort" after Tim Peters, its creator, and in its documentation it claims to have "supernatural performance" at times (pretty good, for a sort!). Still, sorting is an inherently exponential operation (it must chop up the sequence and put it back together many times), and the other versions simply perform one linear left-to-right scan. The net effect is that sorting is quicker if the arguments are partially ordered, but is likely to be slower otherwise. Even so, Python performance can change over time, and the fact that sorting is implemented in the C language can help greatly; for an exact analysis, you should time the alternatives with the `time` or `timeit` modules we'll meet in Chapter 20.

```
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

All three solutions produce the same result when the file is run. Try typing a few calls interactively to experiment with these on your own:

```
% python mins.py
1
aa
[1, 1]
```

Notice that none of these three variants tests for the case where no arguments are passed in. They could, but there's no point in doing so here—in all three solutions, Python will automatically raise an exception if no arguments are passed in. The first variant raises an exception when we try to fetch item 0, the second when Python detects an argument list mismatch, and the third when we try to return item 0 at the end.

This is exactly what we want to happen—because these functions support any data type, there is no valid sentinel value that we could pass back to designate an error. There are exceptions to this rule (e.g., if you have to run expensive actions before you reach the error), but in general it's better to assume that arguments will work in your functions' code and let Python raise errors for you when they do not.

Bonus Points

You can get can get bonus points here for changing these functions to compute the *maximum*, rather than minimum, values. This one's easy: the first two versions only require changing `<` to `>`, and the third simply requires that we return `tmp[-1]` instead of `tmp[0]`. For an extra point, be sure to set the function name to "max" as well (though this part is strictly optional).

It's also possible to generalize a single function to compute either a minimum *or* a maximum value, by evaluating comparison expression strings with a tool like the `eval` built-in function (see the library manual) or passing in an arbitrary comparison function. The file *minmax.py* shows how to implement the latter scheme:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y                # See also: lambda
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))       # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

% python minmax.py
```

```
1
6
```

Functions are another kind of object that can be passed into a function like this one. To make this a `max` (or other) function, for example, we could simply pass in the right sort of `test` function. This may seem like extra work, but the main point of generalizing functions this way (instead of cutting and pasting to change just a single character) is that we'll only have one version to change in the future, not two.

The Punch Line...

Of course, all this was just a coding exercise. There's really no reason to code `min` or `max` functions, because both are built-ins in Python! We met them briefly in Chapter 5 in conjunction with numeric tools, and again in Chapter 14 when exploring iteration contexts. The built-in versions work almost exactly like ours, but they're coded in C for optimal speed and accept either a single iterable or multiple arguments. Still, though it's superfluous in this context, the general coding pattern we used here might be useful in other scenarios.

Generalized Set Functions

Let's look at a more useful example of special argument-matching modes at work. At the end of Chapter 16, we wrote a function that returned the intersection of two sequences (it picked out items that appeared in both). Here is a version that intersects an arbitrary number of sequences (one or more) by using the varargs matching form `*args` to collect all the passed-in arguments. Because the arguments come in as a tuple, we can process them in a simple `for` loop. Just for fun, we'll code a `union` function that also accepts an arbitrary number of arguments to collect items that appear in any of the operands:

```
def intersect(*args):
    res = []
    for x in args[0]:                   # Scan first sequence
        for other in args[1:]:          # For all other args
            if x not in other: break    # Item in each one?
        else:                           # No: break out of loop
            res.append(x)               # Yes: add items to end
    return res

def union(*args):
    res = []
    for seq in args:                    # For all args
        for x in seq:                   # For all nodes
            if not x in res:
                res.append(x)           # Add new items to result
    return res
```

Because these are tools worth reusing (and they're too big to retype interactively), we'll store the functions in a module file called *inter2.py* (if you've forgotten how modules and imports work, see the introduction in Chapter 3, or stay tuned for in-depth coverage in Part V). In both functions, the arguments passed in at the call come in as the `args` tuple. As in the original `intersect`, both work on any kind of sequence. Here, they are processing strings, mixed types, and more than two sequences:

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)          # Two operands
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))                 # Mixed types
[1]

>>> intersect(s1, s2, s3)                     # Three operands
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```

I should note that because Python now has a *set object type* (described in Chapter 5), none of the set-processing examples in this book are strictly required anymore; they are included only as demonstrations of coding techniques. Because it's constantly improving, Python has an uncanny way of conspiring to make my book examples obsolete over time!

Emulating the Python 3.0 print Function

To round out the chapter, let's look at one last example of argument matching at work. The code you'll see here is intended for use in Python 2.6 or earlier (it works in 3.0, too, but is pointless there): it uses both the `*args` arbitrary positional tuple and the `**args` arbitrary keyword-arguments dictionary to simulate most of what the Python 3.0 `print` function does.

As we learned in Chapter 11, this isn't actually required, because 2.6 programmers can always enable the 3.0 `print` function with an import of this form:

```
from __future__ import print_function
```

To demonstrate argument matching in general, though, the following file, *print30.py*, does the same job in a small amount of reusable code:

```
"""
Emulate most of the 3.0 print function for use in 2.X
call signature: print30(*args, sep=' ', end='\n', file=None)
"""
import sys

def print30(*args, **kargs):
    sep  = kargs.get('sep', ' ')            # Keyword arg defaults
    end  = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

To test it, import this into another file or the interactive prompt, and use it like the 3.0 `print` function. Here is a test script, *testprint30.py* (notice that the function must be called "print30", because "print" is a reserved word in 2.6):

```
from print30 import print30
print30(1, 2, 3)
print30(1, 2, 3, sep='')                    # Suppress separator
print30(1, 2, 3, sep='...')
print30(1, [2], (3,), sep='...')            # Various object types

print30(4, 5, 6, sep='', end='')            # Suppress newline
print30(7, 8, 9)
print30()                                   # Add newline (or blank line)

import sys
print30(1, 2, 3, sep='??', end='.\n', file=sys.stderr)    # Redirect to file
```

When run under 2.6, we get the same results as 3.0's `print` function:

```
C:\misc> c:\python26\python testprint30.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9

1??2??3.
```

Although pointless in 3.0, the results are the same when run there. As usual, the generality of Python's design allows us to prototype or develop concepts in the Python language itself. In this case, argument-matching tools are as flexible in Python code as they are in Python's internal implementation.

Using Keyword-Only Arguments

It's interesting to notice that this example could be coded with Python 3.0 keyword-only arguments, described earlier in this chapter, to automatically validate configuration arguments:

```
# Use keyword-only args

def print30(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

This version works the same as the original, and it's a prime example of how keyword-only arguments come in handy. The original version assumes that all positional arguments are to be printed, and all keywords are for options only. That's almost sufficient, but any extra keyword arguments are silently ignored. A call like the following, for instance, will generate an exception with the keyword-only form:

```
>>> print30(99, name='bob')
TypeError: print30() got an unexpected keyword argument 'name'
```

but will silently ignore the `name` argument in the original version. To detect superfluous keywords manually, we could use `dict.pop()` to delete fetched entries, and check if the dictionary is not empty. Here is an equivalent to the keyword-only version:

```
# Use keyword args deletion with defaults

def print30(*args, **kargs):
    sep  = kargs.pop('sep', ' ')
    end  = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('extra keywords: %s' % kargs)
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

This works as before, but it now catches extraneous keyword arguments, too:

```
>>> print30(99, name='bob')
TypeError: extra keywords: {'name': 'bob'}
```

This version of the function runs under Python 2.6, but it requires four more lines of code than the keyword-only version. Unfortunately, the extra code is required in this case—the keyword-only version only works on 3.0, which negates most of the reason that I wrote this example in the first place (a 3.0 emulator that only works on 3.0 isn't incredibly useful!). In programs written to run on 3.0, though, keyword-only arguments can simplify a specific category of functions that accept both arguments and options. For another example of 3.0 keyword-only arguments, be sure to see the upcoming iteration timing case study in Chapter 20.

Why You Will Care: Keyword Arguments

As you can probably tell, advanced argument-matching modes can be complex. They are also entirely optional; you can get by with just simple positional matching, and it's probably a good idea to do so when you're starting out. However, because some Python tools make use of them, some general knowledge of these modes is important.

For example, keyword arguments play an important role in `tkinter`, the de facto standard GUI API for Python (this module's name is `Tkinter` in Python 2.6). We touch on `tkinter` only briefly at various points in this book, but in terms of its call patterns, keyword arguments set configuration options when GUI components are built. For instance, a call of the form:

```
from tkinter import *
widget = Button(text="Press me", command=someFunction)
```

creates a new button and specifies its text and callback function, using the `text` and `command` keyword arguments. Since the number of configuration options for a widget can be large, keyword arguments let you pick and choose which to apply. Without them, you might have to either list all the possible options by position or hope for a judicious positional argument defaults protocol that would handle every possible option arrangement.

Many built-in functions in Python expect us to use keywords for usage-mode options as well, which may or may not have defaults. As we learned in Chapter 8, for instance, the `sorted` built-in:

```
sorted(iterable, key=None, reverse=False)
```

expects us to pass an iterable object to be sorted, but also allows us to pass in optional keyword arguments to specify a dictionary sort key and a reversal flag, which default to `None` and `False`, respectively. Since we normally don't use these options, they may be omitted to use defaults.

Chapter Summary

In this chapter, we studied the second of two key concepts related to functions: *arguments* (how objects are passed into a function). As we learned, arguments are passed into a function by assignment, which means by object reference, which really means

by pointer. We also studied some more advanced extensions, including default and keyword arguments, tools for using arbitrarily many arguments, and keyword-only arguments in 3.0. Finally, we saw how mutable arguments can exhibit the same behavior as other shared references to objects—unless the object is explicitly copied when it's sent in, changing a passed-in mutable in a function can impact the caller.

The next chapter continues our look at functions by exploring some more advanced function-related ideas: function annotations, `lambda`s, and functional tools such as `map` and `filter`. Many of these concepts stem from the fact that functions are normal objects in Python, and so support some advanced and very flexible processing modes. Before diving into those topics, however, take this chapter's quiz to review the argument ideas we've studied here.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> def func(a, b=4, c=5):
...     print(a, b, c)
...
>>> func(1, 2)
```

2. What is the output of this code, and why?

```
>>> def func(a, b, c=5):
...     print(a, b, c)
...
>>> func(1, c=3, b=2)
```

3. How about this code: what is its output, and why?

```
>>> def func(a, *pargs):
...     print(a, pargs)
...
>>> func(1, 2, 3)
```

4. What does this code print, and why?

```
>>> def func(a, **kargs):
...     print(a, kargs)
...
>>> func(a=1, c=3, b=2)
```

5. One last time: what is the output of this code, and why?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)
...
>>> func(1, *(5,6))
```

6. Name three or more ways that functions can communicate results to a caller.

Test Your Knowledge: Answers

1. The output here is `'1 2 5'`, because `1` and `2` are passed to `a` and `b` by position, and `c` is omitted in the call and defaults to `5`.
2. The output this time is `'1 2 3'`: `1` is passed to `a` by position, and `b` and `c` are passed `2` and `3` by name (the left-to-right order doesn't matter when keyword arguments are used like this).
3. This code prints `'1 (2, 3)'`, because `1` is passed to `a` and the `*pargs` collects the remaining positional arguments into a new tuple object. We can step through the extra positional arguments tuple with any iteration tool (e.g., `for arg in pargs: ...`).
4. This time the code prints `'1, {'c': 3, 'b': 2}'`, because `1` is passed to `a` by name and the `**kargs` collects the remaining keyword arguments into a dictionary. We could step through the extra keyword arguments dictionary by key with any iteration tool (e.g., `for key in kargs: ...`).
5. The output here is `'1 5 6 4'`: `1` matches `a` by position, `5` and `6` match `b` and `c` by `*name` positionals (`6` overrides `c`'s default), and `d` defaults to `4` because it was not passed a value.
6. Functions can send back results with `return` statements, by changing passed-in mutable arguments, and by setting global variables. Globals are generally frowned upon (except for very special cases, like multithreaded programs) because they can make code more difficult to understand and use. `return` statements are usually best, but changing mutables is fine, if expected. Functions may also communicate with system devices such as files and sockets, but these are beyond our scope here.

CHAPTER 19
Advanced Function Topics

This chapter introduces a collection of more advanced function-related topics: recursive functions, function attributes and annotations, the `lambda` expression, and functional programming tools such as `map` and `filter`. These are all somewhat advanced tools that, depending on your job description, you may not encounter on a regular basis. Because of their roles in some domains, though, a basic understanding can be useful; `lambda`s, for instance, are regular customers in GUIs.

Part of the art of using functions lies in the interfaces between them, so we will also explore some general function design principles here. The next chapter continues this advanced theme with an exploration of generator functions and expressions and a revival of list comprehensions in the context of the functional tools we will study here.

Function Design Concepts

Now that we've had a chance to study function basics in Python, let's begin this chapter with a few words of context. When you start using functions in earnest, you're faced with choices about how to glue components together—for instance, how to decompose a task into purposeful functions (known as *cohesion*), how your functions should communicate (called *coupling*), and so on. You also need to take into account concepts such as the size of your functions, because they directly impact code usability. Some of this falls into the category of structured analysis and design, but it applies to Python code as to any other.

We introduced some ideas related to function and module coupling in the Chapter 17 when studying scopes, but here is a review of a few general guidelines for function beginners:

- **Coupling: use arguments for inputs and `return` for outputs**. Generally, you should strive to make a function independent of things outside of it. Arguments and `return` statements are often the best ways to isolate external dependencies to a small number of well-known places in your code.

- **Coupling: use global variables only when truly necessary**. Global variables (i.e., names in the enclosing module) are usually a poor way for functions to communicate. They can create dependencies and timing issues that make programs difficult to debug and change.
- **Coupling: don't change mutable arguments unless the caller expects it**. Functions can change parts of passed-in mutable objects, but (as with global variables) this creates lots of coupling between the caller and callee, which can make a function too specific and brittle.
- **Cohesion: each function should have a single, unified purpose**. When designed well, each of your functions should do one thing—something you can summarize in a simple declarative sentence. If that sentence is very broad (e.g., "this function implements my whole program"), or contains lots of conjunctions (e.g., "this function gives employee raises *and* submits a pizza order"), you might want to think about splitting it into separate and simpler functions. Otherwise, there is no way to reuse the code behind the steps mixed together in the function.
- **Size: each function should be relatively small**. This naturally follows from the preceding goal, but if your functions start spanning multiple pages on your display, it's probably time to split them. Especially given that Python code is so concise to begin with, a long or deeply nested function is often a symptom of design problems. Keep it simple, and keep it short.
- **Coupling: avoid changing variables in another module file directly**. We introduced this concept in Chapter 17, and we'll revisit it in the next part of the book when we focus on modules. For reference, though, remember that changing variables across file boundaries sets up a coupling between modules similar to how global variables couple functions—the modules become difficult to understand and reuse. Use accessor functions whenever possible, instead of direct assignment statements.

Figure 19-1 summarizes the ways functions can talk to the outside world; inputs may come from items on the left side, and results may be sent out in any of the forms on the right. Good function designers prefer to use only arguments for inputs and `return` statements for outputs, whenever possible.

Of course, there are plenty of exceptions to the preceding design rules, including some related to Python's OOP support. As you'll see in Part VI, Python classes *depend* on changing a passed-in mutable object—class functions set attributes of an automatically passed-in argument called `self` to change per-object state information (e.g., `self.name='bob'`). Moreover, if classes are not used, global variables are often the most straightforward way for functions in modules to retain state between calls. Side effects are dangerous only if they're unexpected.

In general though, you should strive to minimize external dependencies in functions and other program components. The more *self-contained* a function is, the easier it will be to understand, reuse, and modify.

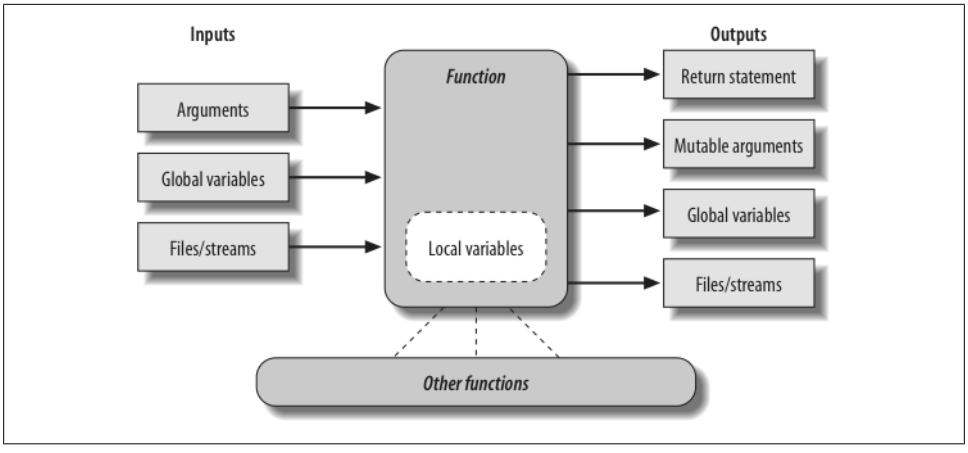

Figure 19-1. Function execution environment. Functions may obtain input and produce output in a variety of ways, though functions are usually easier to understand and maintain if you use arguments for input and return statements and anticipated mutable argument changes for output. In Python 3, outputs may also take the form of declared nonlocal names that exist in an enclosing function scope.

Recursive Functions

While discussing scope rules near the start of Chapter 17, we briefly noted that Python supports *recursive functions*—functions that call themselves either directly or indirectly in order to loop. Recursion is a somewhat advanced topic, and it's relatively rare to see in Python. Still, it's a useful technique to know about, as it allows programs to traverse structures that have arbitrary and unpredictable shapes. Recursion is even an alternative for simple loops and iterations, though not necessarily the simplest or most efficient one.

Summation with Recursion

Let's look at some examples. To sum a list (or other sequence) of numbers, we can either use the built-in `sum` function or write a more custom version of our own. Here's what a custom summing function might look like when coded with recursion:

```
>>> def mysum(L):
...     if not L:
...         return 0
...     else:
...         return L[0] + mysum(L[1:])          # Call myself

>>> mysum([1, 2, 3, 4, 5])
15
```

At each level, this function calls itself recursively to compute the sum of the rest of the list, which is later added to the item at the front. The recursive loop ends and zero is returned when the list becomes empty. When using recursion like this, each open level

of call to the function has its own copy of the function's local scope on the runtime call stack—here, that means `L` is different in each level.

If this is difficult to understand (and it often is for new programmers), try adding a `print` of `L` to the function and run it again, to trace the current list at each call level:

```
>>> def mysum(L):
...     print(L)                                # Trace recursive levels
...     if not L:                               # L shorter at each level
...         return 0
...     else:
...         return L[0] + mysum(L[1:])
...
>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

As you can see, the list to be summed grows smaller at each recursive level, until it becomes empty—the termination of the recursive loop. The sum is computed as the recursive calls unwind.

Coding Alternatives

Interestingly, we can also use Python's `if`/`else` ternary expression (described in Chapter 12) to save some code real-estate here. We can also generalize for any summable type (which is easier if we assume at least one item in the input, as we did in Chapter 18's minimum value example) and use Python 3.0's extended sequence assignment to make the first/rest unpacking simpler (as covered in Chapter 11):

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])           # Use ternary expression

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:])  # Any type, assume one

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest)    # Use 3.0 ext seq assign
```

The latter two of these fail for empty lists but allow for sequences of any object type that supports `+`, not just numbers:

```
>>> mysum([1])                          # mysum([]) fails in last 2
1
>>> mysum([1, 2, 3, 4, 5])
15
>>> mysum(('s', 'p', 'a', 'm'))         # But various types now work
'spam'
```

```
>>> mysum(['spam', 'ham', 'eggs'])
'spamhameggs'
```

If you study these three variants, you'll find that the latter two also work on a single string argument (e.g., `mysum ('spam')`), because strings are sequences of one-character strings; the third variant works on arbitary iterables, including open input files, but the others do not because they index; and the function header `def mysum(first, * rest)`, although similar to the third variant, wouldn't work at all, because it expects individual arguments, not a single iterable.

Keep in mind that recursion can be direct, as in the examples so far, or *indirect*, as in the following (a function that calls another function, which calls back to its caller). The net effect is the same, though there are two function calls at each level instead of one:

```
>>> def mysum(L):
...     if not L: return 0
...     return nonempty(L)                  # Call a function that calls me
...
>>> def nonempty(L):
...     return L[0] + mysum(L[1:])          # Indirectly recursive
...
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Loop Statements Versus Recursion

Though recursion works for summing in the prior sections' examples, it's probably overkill in this context. In fact, recursion is not used nearly as often in Python as in more esoteric languages like Prolog or Lisp, because Python emphasizes simpler procedural statements like loops, which are usually more natural. The `while`, for example, often makes things a bit more concrete, and it doesn't require that a function be defined to allow recursive calls:

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
...     sum += L[0]
...     L = L[1:]
...
>>> sum
15
```

Better yet, `for` loops iterate for us automatically, making recursion largely extraneous in most cases (and, in all likelihood, less efficient in terms of memory space and execution time):

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
...
>>> sum
15
```

With looping statements, we don't require a fresh copy of a local scope on the call stack for each iteration, and we avoid the speed costs associated with function calls in general. (Stay tuned for Chapter 20's timer case study for ways to compare the execution times of alternatives like these.)

Handling Arbitrary Structures

On the other hand, recursion (or equivalent explicit stack-based algorithms, which we'll finesse here) can be required to traverse arbitrarily shaped structures. As a simple example of recursion's role in this context, consider the task of computing the sum of all the numbers in a nested sublists structure like this:

```
[1, [2, [3, 4], 5], 6, [7, 8]]          # Arbitrarily nested sublists
```

Simple looping statements won't work here because this not a linear iteration. Nested looping statements do not suffice either, because the sublists may be nested to arbitrary depth and in an arbitrary shape. Instead, the following code accommodates such general nesting by using recursion to visit sublists along the way:

```
def sumtree(L):
    tot = 0
    for x in L:                          # For each item at this level
        if not isinstance(x, list):
            tot += x                     # Add numbers directly
        else:
            tot += sumtree(x)            # Recur for sublists
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]       # Arbitrary nesting
print(sumtree(L))                        # Prints 36


# Pathological cases

print(sumtree([1, [2, [3, [4, [5]]]]]))  # Prints 15 (right-heavy)

print(sumtree([[[[[1], 2], 3], 4], 5]))  # Prints 15 (left-heavy)
```

Trace through the test cases at the bottom of this script to see how recursion traverses their nested lists. Although this example is artificial, it is representative of a larger class of programs; inheritance trees and module import chains, for example, can exhibit similarly general structures. In fact, we will use recursion again in such roles in more realistic examples later in this book:

- In Chapter 24's *reloadall.py*, to traverse import chains
- In Chapter 28's *classtree.py*, to traverse class inheritance trees
- In Chapter 30's *lister.py*, to traverse class inheritance trees again

Although you should generally prefer looping statements to recursion for linear iterations on the grounds of simplicity and efficiency, we'll find that recursion is essential in scenarios like those in these later examples.

Moreover, you sometimes need to be aware of the potential of *unintended* recursion in your programs. As you'll also see later in the book, some operator overloading methods in classes such as `__setattr__` and `__getattribute__` have the potential to recursively loop if used incorrectly. Recursion is a powerful tool, but it tends to be best when expected!

Function Objects: Attributes and Annotations

Python functions are more flexible than you might think. As we've seen in this part of the book, functions in Python are much more than code-generation specifications for a compiler—Python functions are full-blown *objects*, stored in pieces of memory all their own. As such, they can be freely passed around a program and called indirectly. They also support operations that have little to do with calls at all—attribute storage and annotation.

Indirect Function Calls

Because Python functions are objects, you can write programs that process them generically. Function objects may be assigned to other names, passed to other functions, embedded in data structures, returned from one function to another, and more, as if they were simple numbers or strings. Function objects also happen to support a special operation: they can be called by listing arguments in parentheses after a function expression. Still, functions belong to the same general category as other objects.

We've seen some of these generic use cases for functions in earlier examples, but a quick review helps to underscore the object model. For example, there's really nothing special about the name used in a `def` statement: it's just a variable assigned in the current scope, as if it had appeared on the left of an `=` sign. After a `def` runs, the function name is simply a reference to an object—you can *reassign* that object to other names freely and call it through any reference:

```
>>> def echo(message):                  # Name echo assigned to function object
...     print(message)
...
>>> echo('Direct call')                 # Call object through original name
Direct call

>>> x = echo                            # Now x references the function too
>>> x('Indirect call!')                 # Call object through name by adding ()
Indirect call!
```

Because arguments are passed by assigning objects, it's just as easy to *pass* functions to other functions as arguments. The callee may then call the passed-in function just by adding arguments in parentheses:

```
>>> def indirect(func, arg):
...     func(arg)                          # Call the passed-in object by adding ()
...
>>> indirect(echo, 'Argument call!')       # Pass the function to another function
Argument call!
```

You can even stuff function objects into data structures, as though they were integers or strings. The following, for example, *embeds* the function twice in a list of tuples, as a sort of actions table. Because Python compound types like these can contain any sort of object, there's no special case here, either:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
...     func(arg)                          # Call functions embedded in containers
...
Spam!
Ham!
```

This code simply steps through the `schedule` list, calling the `echo` function with one argument each time through (notice the tuple-unpacking assignment in the `for` loop header, introduced in Chapter 13). As we saw in Chapter 17's examples, functions can also be created and *returned* for use elsewhere:

```
>>> def make(label):                      # Make a function but don't call it
...     def echo(message):
...         print(label + ':' + message)
...     return echo
...
>>> F = make('Spam')                      # Label in enclosing scope is retained
>>> F('Ham!')                             # Call the function that make returned
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Python's universal object model and lack of type declarations make for an incredibly flexible programming language.

Function Introspection

Because they are objects, we can also process functions with normal object tools. In fact, functions are more flexible than you might expect. For instance, once we make a function, we can call it as usual:

```
>>> def func(a):
...     b = 'spam'
...     return b * a
...
>>> func(8)
'spamspamspamspamspamspamspamspam'
```

But the call expression is just one operation defined to work on function objects. We can also inspect their attributes generically (the following is run in Python 3.0, but 2.6 results are similar):

```
>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Introspection tools allow us to explore implementation details too—functions have attached *code objects*, for example, which provide details on aspects such as the functions' local variables and arguments:

```
>>> func.__code__
<code object func at 0x0257C9B0, file "<stdin>", line 1>

>>> dir(func.__code__)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
...more omitted...
'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab',
'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

Tool writers can make use of such information to manage functions (in fact, we will too in Chapter 38, to implement validation of function arguments in decorators).

Function Attributes

Function objects are not limited to the system-defined attributes listed in the prior section, though. As we learned in Chapter 17, it's possible to attach arbitrary user-defined attributes to them as well:

```
>>> func
<function func at 0x0257C738>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
__str__', '__subclasshook__', 'count', 'handles']
```

As we saw in that chapter, such attributes can be used to attach *state information* to function objects directly, instead of using other techniques such as globals, nonlocals, and classes. Unlike nonlocals, such attributes are accessible anywhere the function itself is. In a sense, this is also a way to emulate "static locals" in other languages—variables whose names are local to a function, but whose values are retained after a function exits. Attributes are related to objects instead of scopes, but the net effect is similar.

Function Annotations in 3.0

In Python 3.0 (but not 2.6), it's also possible to attach *annotation information*—arbitrary user-defined data about a function's arguments and result—to a function object. Python provides special syntax for specifying annotations, but it doesn't do anything with them itself; annotations are completely optional, and when present are simply attached to the function object's `__annotations__` attribute for use by other tools.

We met Python 3.0's keyword-only arguments in the prior chapter; annotations generalize function header syntax further. Consider the following nonannotated function, which is coded with three arguments and returns a result:

```
>>> def func(a, b, c):
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Syntactically, function annotations are coded in `def` header lines, as arbitrary expressions associated with arguments and return values. For arguments, they appear after a colon immediately following the argument's name; for return values, they are written after a `->` following the arguments list. This code, for example, annotates all three of the prior function's arguments, as well as its return value:

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Calls to an annotated function work as usual, but when annotations are present Python collects them in a *dictionary* and attaches it to the function object itself. Argument names become keys, the return value annotation is stored under key "return" if coded, and the values of annotation keys are assigned to the results of the annotation expressions:

```
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Because they are just Python objects attached to a Python object, annotations are straightforward to process. The following annotates just two of three arguments and steps through the attached annotations generically:

```
>>> def func(a: 'spam', b, c: 99):
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func.__annotations__
{'a': 'spam', 'c': 99}

>>> for arg in func.__annotations__:
...    print(arg, '=>', func.__annotations__[arg])
...
a => spam
c => 99
```

There are two fine points to note here. First, you can still use *defaults* for arguments if you code annotations—the annotation (and its `:` character) appear before the default (and its `=` character). In the following, for example, `a: 'spam' = 4` means that argument `a` defaults to `4` and is annotated with the string `'spam'`:

```
>>> def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func()                          # 4 + 5 + 6   (all defaults)
15
>>> func(1, c=10)                   # 1 + 5 + 10  (keywords work normally)
16
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Second, note that the *blank spaces* in the prior example are all optional—you can use spaces between components in function headers or not, but omitting them might degrade your code's readability to some observers:

```
>>> def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:
...     return a + b + c
...
>>> func(1, 2)                      # 1 + 2 + 6
9
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Annotations are a new feature in 3.0, and some of their potential uses remain to be uncovered. It's easy to imagine annotations being used to specify constraints for argument types or values, though, and larger APIs might use this feature as a way to register function interface information. In fact, we'll see a potential application in Chapter 38, where we'll look at annotations as an alternative to *function decorator arguments* (a more general concept in which information is coded outside the function header and so is not limited to a single role). Like Python itself, annotation is a tool whose roles are shaped by your imagination.

Finally, note that annotations work only in `def` statements, not `lambda` expressions, because `lambda`'s syntax already limits the utility of the functions it defines. Coincidentally, this brings us to our next topic.

Anonymous Functions: lambda

Besides the `def` statement, Python also provides an expression form that generates function objects. Because of its similarity to a tool in the Lisp language, it's called `lambda`.* Like `def`, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why `lambda`s are sometimes known as *anonymous* (i.e., unnamed) functions. In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.

lambda Basics

The `lambda`'s general form is the keyword `lambda`, followed by one or more arguments (exactly like the arguments list you enclose in parentheses in a `def` header), followed by an expression after a colon:

```
lambda argument1, argument2,... argumentN :expression using arguments
```

Function objects returned by running `lambda` expressions work exactly the same as those created and assigned by `def`s, but there are a few differences that make `lambda`s useful in specialized roles:

- **`lambda` is an expression, not a statement**. Because of this, a `lambda` can appear in places a `def` is not allowed by Python's syntax—inside a list literal or a function call's arguments, for example. As an expression, `lambda` returns a value (a new function) that can optionally be assigned a name. In contrast, the `def` statement always assigns the new function to the name in the header, instead of returning it as a result.
- **`lambda`'s body is a single expression, not a block of statements**. The `lambda`'s body is similar to what you'd put in a `def` body's `return` statement; you simply type the result as a naked expression, instead of explicitly returning it. Because it is limited to an expression, a `lambda` is less general than a `def`—you can only squeeze so much logic into a `lambda` body without using statements such as `if`. This is by design, to limit program nesting: `lambda` is designed for coding simple functions, and `def` handles larger tasks.

* The `lambda` tends to intimidate people more than it should. This reaction seems to stem from the name "lambda" itself—a name that comes from the Lisp language, which got it from lambda calculus, which is a form of symbolic logic. In Python, though, it's really just a keyword that introduces the expression syntactically. Obscure mathematical heritage aside, `lambda` is simpler to use than you may think.

Apart from those distinctions, `def`s and `lambda`s do the same sort of work. For instance, we've seen how to make a function with a `def` statement:

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
```

But you can achieve the same effect with a `lambda` expression by explicitly assigning its result to a name through which you can later call the function:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Here, `f` is assigned the function object the `lambda` expression creates; this is how `def` works, too, but its assignment is automatic.

Defaults work on `lambda` arguments, just like in a `def`:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

The code in a `lambda` body also follows the same scope lookup rules as code inside a `def`. `lambda` expressions introduce a local scope much like a nested `def`, which automatically sees names in enclosing functions, the module, and the built-in scope (via the LEGB rule):

```
>>> def knights():
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x)        # Title in enclosing def
...     return action                               # Return a function
...
>>> act = knights()
>>> act('robin')
'Sir robin'
```

In this example, prior to Release 2.2, the value for the name `title` would typically have been passed in as a default argument value instead; flip back to the scopes coverage in Chapter 17 if you've forgotten why.

Why Use lambda?

Generally speaking, `lambda`s come in handy as a sort of function shorthand that allows you to embed a function's definition within the code that uses it. They are entirely optional (you can always use `def`s instead), but they tend to be simpler coding constructs in scenarios where you just need to embed small bits of executable code.

For instance, we'll see later that callback handlers are frequently coded as inline `lambda` expressions embedded directly in a registration call's arguments list, instead of being defined with a `def` elsewhere in a file and referenced by name (see the sidebar "Why You Will Care: Callbacks" on page 479 for an example).

lambdas are also commonly used to code *jump tables*, which are lists or dictionaries of actions to be performed on demand. For example:

```
L = [lambda x: x ** 2,          # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]          # A list of 3 callable functions

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

The lambda expression is most useful as a shorthand for def, when you need to stuff small pieces of executable code into places where statements are illegal syntactically. This code snippet, for example, builds up a list of three functions by embedding lambda expressions inside a list literal; a def won't work inside a list literal like this because it is a statement, not an expression. The equivalent def coding would require temporary function names and function definitions outside the context of intended use:

```
def f1(x): return x ** 2
def f2(x): return x ** 3        # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]                # Reference by name

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

In fact, you can do the same sort of thing with dictionaries and other data structures in Python to build up more general sorts of action tables. Here's another example to illustrate, at the interactive prompt:

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
...  'got':     (lambda: 2 * 4),
...  'one':     (lambda: 2 ** 6)}[key]()
8
```

Here, when Python makes the temporary dictionary, each of the nested lambdas generates and leaves behind a function to be called later. Indexing by key fetches one of those functions, and parentheses force the fetched function to be called. When coded this way, a dictionary becomes a more general multiway branching tool than what I could show you in Chapter 12's coverage of if statements.

To make this work without lambda, you'd need to instead code three def statements somewhere else in your file, outside the dictionary in which the functions are to be used, and reference the functions by name:

```
>>> def f1(): return 2 + 2
...
>>> def f2(): return 2 * 4
...
```

```
>>> def f3(): return 2 ** 6
...
>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
64
```

This works, too, but your `def`s may be arbitrarily far away in your file, even if they are just little bits of code. The *code proximity* that `lambda`s provide is especially useful for functions that will only be used in a single context—if the three functions here are not useful anywhere else, it makes sense to embed their definitions within the dictionary as `lambda`s. Moreover, the `def` form requires you to make up names for these little functions that may clash with other names in this file (perhaps unlikely, but always possible).

`lambda`s also come in handy in function-call argument lists as a way to inline temporary function definitions not used anywhere else in your program; we'll see some examples of such other uses later in this chapter, when we study `map`.

How (Not) to Obfuscate Your Python Code

The fact that the body of a `lambda` has to be a single expression (not a series of statements) would seem to place severe limits on how much logic you can pack into a `lambda`. If you know what you're doing, though, you can code most statements in Python as expression-based equivalents.

For example, if you want to print from the body of a `lambda` function, simply say `sys.stdout.write(str(x)+'\n')`, instead of `print(x)` (recall from Chapter 11 that this is what `print` really does). Similarly, to nest logic in a `lambda`, you can use the `if`/`else` ternary expression introduced in Chapter 12, or the equivalent but trickier `and`/`or` combination also described there. As you learned earlier, the following statement:

```
if a:
    b
else:
    c
```

can be emulated by either of these roughly equivalent expressions:

```
b if a else c
((a and b) or c)
```

Because expressions like these can be placed inside a `lambda`, they may be used to implement selection logic within a `lambda` function:

```
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

Furthermore, if you need to perform loops within a `lambda`, you can also embed things like `map` calls and list comprehension expressions (tools we met in earlier chapters and will revisit in this and the next chapter):

```
>>> import sys
>>> showall = lambda x: list(map(sys.stdout.write, x))          # Use list in 3.0

>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])
spam
toast
eggs

>>> showall = lambda x: [sys.stdout.write(line) for line in x]

>>> t = showall(('bright\n', 'side\n', 'of\n', 'life\n'))
bright
side
of
life
```

Now that I've shown you these tricks, I am required by law to ask you to please only use them as a last resort. Without due care, they can lead to unreadable (a.k.a. *obfuscated*) Python code. In general, simple is better than complex, explicit is better than implicit, and full statements are better than arcane expressions. That's why `lambda` is limited to expressions. If you have larger logic to code, use `def`; `lambda` is for small pieces of inline code. On the other hand, you may find these techniques useful in moderation.

Nested lambdas and Scopes

`lambda`s are the main beneficiaries of nested function scope lookup (the E in the LEGB scope rule we studied in Chapter 17). In the following, for example, the `lambda` appears inside a `def`—the typical case—and so can access the value that the name `x` had in the enclosing function's scope at the time that the enclosing function was called:

```
>>> def action(x):
...     return (lambda y: x + y)          # Make and return function, remember x
...
>>> act = action(99)
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)                               # Call what action returned
101
```

What wasn't illustrated in the prior discussion of nested function scopes is that a `lambda` also has access to the names in any enclosing `lambda`. This case is somewhat obscure, but imagine if we recoded the prior `def` with a `lambda`:

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
```

```
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

Here, the nested `lambda` structure makes a function that makes a function when called. In both cases, the nested `lambda`'s code has access to the variable `x` in the enclosing `lambda`. This works, but it's fairly convoluted code; in the interest of readability, nested `lambda`s are generally best avoided.

Why You Will Care: Callbacks

Another very common application of `lambda` is to define inline callback functions for Python's `tkinter` GUI API (this module is named `Tkinter` in Python 2.6). For example, the following creates a button that prints a message on the console when pressed, assuming `tkinter` is available on your computer (it is by default on Windows and other OSs):

```
import sys
from tkinter import Button, mainloop     # Tkinter in 2.6
x = Button(
        text ='Press me',
        command=(lambda:sys.stdout.write('Spam\n')))
x.pack()
mainloop()
```

Here, the callback handler is registered by passing a function generated with a `lambda` to the `command` keyword argument. The advantage of `lambda` over `def` here is that the code that handles a button press is right here, embedded in the button-creation call.

In effect, the `lambda` *defers* execution of the handler until the event occurs: the `write` call happens on button presses, not when the button is created.

Because the nested function scope rules apply to `lambda`s as well, they are also easier to use as callback handlers, as of Python 2.2—they automatically see names in the functions in which they are coded and no longer require passed-in defaults in most cases. This is especially handy for accessing the special `self` instance argument that is a local variable in enclosing class method functions (more on classes in Part VI):

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.onPress("spam")))
    def onPress(self, message):
        ...use message...
```

In prior releases, even `self` had to be passed in to a `lambda` with defaults.

Mapping Functions over Sequences: map

One of the more common things programs do with lists and other sequences is apply an operation to each item and collect the results. For instance, updating all the counters in a list can be done easily with a `for` loop:

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)                # Add 10 to each item
...
>>> updated
[11, 12, 13, 14]
```

But because this is such a common operation, Python actually provides a built-in that does most of the work for you. The `map` function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results. For example:

```
>>> def inc(x): return x + 10                 # Function to be run
...
>>> list(map(inc, counters))                  # Collect results
[11, 12, 13, 14]
```

We met `map` briefly in Chapters 13 and 14, as a way to apply a built-in function to items in an iterable. Here, we make better use of it by passing in a user-defined function to be applied to each item in the list—`map` calls `inc` on each list item and collects all the return values into a new list. Remember that `map` is an iterable in Python 3.0, so a `list` call is used to force it to produce all its results for display here; this isn't necessary in 2.6.

Because `map` expects a function to be passed in, it also happens to be one of the places where `lambda` commonly appears:

```
>>> list(map((lambda x: x + 3), counters))    # Function expression
[4, 5, 6, 7]
```

Here, the function adds 3 to each item in the `counters` list; as this little function isn't needed elsewhere, it was written inline as a `lambda`. Because such uses of `map` are equivalent to `for` loops, with a little extra code you can always code a general mapping utility yourself:

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
```

Assuming the function `inc` is still as it was when it was shown previously, we can map it across a sequence with the built-in or our equivalent:

```
>>> list(map(inc, [1, 2, 3]))                 # Built-in is an iterator
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])                     # Ours builds a list (see generators)
[11, 12, 13]
```

However, as `map` is a built-in, it's always available, always works the same way, and has some performance benefits (as we'll prove in the next chapter, it's usually faster than a manually coded `for` loop). Moreover, `map` can be used in more advanced ways than

shown here. For instance, given multiple sequence arguments, it sends items taken from sequences in parallel as distinct arguments to the function:

```
>>> pow(3, 4)                           # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))  # 1**2, 2**3, 3**4
[1, 8, 81]
```

With multiple sequences, `map` expects an N-argument function for N sequences. Here, the `pow` function takes two arguments on each call—one from each sequence passed to `map`. It's not much extra work to simulate this multiple-sequence generality in code, too, but we'll postpone doing so until later in the next chapter, after we've met some additional iteration tools.

The `map` call is similar to the list comprehension expressions we studied in Chapter 14 and will meet again in the next chapter, but `map` applies a *function* call to each item instead of an arbitrary *expression*. Because of this limitation, it is a somewhat less general tool. However, in some cases `map` may be faster to run than a list comprehension (e.g., when mapping a built-in function), and it may also require less coding.

Functional Programming Tools: filter and reduce

The `map` function is the simplest representative of a class of Python built-ins used for *functional programming*—tools that apply functions to sequences and other iterables. Its relatives filter out items based on a test function (`filter`) and apply functions to pairs of items and running results (`reduce`). Because they return iterables, `range` and `filter` both require `list` calls to display all their results in 3.0. For example, the following `filter` call picks out items in a sequence that are greater than zero:

```
>>> list(range(-5, 5))                          # An iterator in 3.0
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x > 0), range(-5, 5)))   # An iterator in 3.0
[1, 2, 3, 4]
```

Items in the sequence or iterable for which the function returns a true result are added to the result list. Like `map`, this function is roughly equivalent to a `for` loop, but it is built-in and fast:

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
```

`reduce`, which is a simple built-in function in 2.6 but lives in the `functools` module in 3.0, is more complex. It accepts an iterator to process, but it's not an iterator itself—it

returns a single result. Here are two `reduce` calls that compute the sum and product of the items in a list:

```
>>> from functools import reduce       # Import in 3.0, not in 2.6

>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

At each step, `reduce` passes the current sum or product, along with the next item from the list, to the passed-in `lambda` function. By default, the first item in the sequence initializes the starting value. To illustrate, here's the `for` loop equivalent to the first of these calls, with the addition hardcoded inside the loop:

```
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

Coding your own version of `reduce` is actually fairly straightforward. The following function emulates most of the built-in's behavior and helps demystify its operation in general:

```
>>> def myreduce(function, sequence):
...     tally = sequence[0]
...     for next in sequence[1:]:
...         tally = function(tally, next)
...     return tally
...
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

The built-in `reduce` also allows an optional third argument placed before the items in the sequence to serve as a default result when the sequence is empty, but we'll leave this extension as a suggested exercise.

If this coding technique has sparked your interest, you might also be interested in the standard library `operator` module, which provides functions that correspond to built-in expressions and so comes in handy for some uses of functional tools (see Python's library manual for more details on this module):

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])          # Function-based +
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Together with `map`, `filter` and `reduce` support powerful functional programming techniques. Some observers might also extend the functional programming toolset in Python to include `lambda`, discussed earlier, as well as list comprehensions—a topic we will return to in the next chapter.

Chapter Summary

This chapter took us on a tour of advanced function-related concepts: recursive functions; function annotations; `lambda` expression functions; functional tools such as `map`, `filter`, and `reduce`; and general function design ideas. The next chapter continues the advanced topics motif with a look at generators and a reprisal of iterators and list comprehensions—tools that are just as related to functional programming as to looping statements. Before you move on, though, make sure you've mastered the concepts covered here by working through this chapter's quiz.

Test Your Knowledge: Quiz

1. How are `lambda` expressions and `def` statements related?
2. What's the point of using `lamba`?
3. Compare and contrast `map`, `filter`, and `reduce`.
4. What are function annotations, and how are they used?
5. What are recursive functions, and how are they used?
6. What are some general design guidelines for coding functions?

Test Your Knowledge: Answers

1. Both `lambda` and `def` create function objects to be called later. Because `lambda` is an expression, though, it returns a function object instead of assigning it to a name, and it can be used to nest a function definition in places where a `def` will not work syntactically. A `lambda` only allows for a single implicit return value expression, though; because it does not support a block of statements, it is not ideal for larger functions.
2. `lambda`s allow us to "inline" small units of executable code, defer its execution, and provide it with state in the form of default arguments and enclosing scope variables. Using a `lambda` is never required; you can always code a `def` instead and reference the function by name. `lambda`s come in handy, though, to embed small pieces of deferred code that are unlikely to be used elsewhere in a program. They commonly appear in callback-based program such as GUIs, and they have a natural affinity with function tools like `map` and `filter` that expect a processing function.

3. These three built-in functions all apply another function to items in a sequence (iterable) object and collect results. `map` passes each item to the function and collects all results, `filter` collects items for which the function returns a `True` value, and `reduce` computes a single value by applying the function to an accumulator and successive items. Unlike the other two, `reduce` is available in the `functools` module in 3.0, not the built-in scope.
4. Function annotations, available in 3.0 and later, are syntactic embellishments of a function's arguments and result, which are collected into a dictionary assigned to the function's `__annotations__` attribute. Python places no semantic meaning on these annotations, but simply packages them for potential use by other tools.
5. Recursive functions call themselves either directly or indirectly in order to loop. They may be used to traverse arbitrarily shaped structures, but they can also be used for iteration in general (though the latter role is often more simply and efficiently coded with looping statements).
6. Functions should generally be small, as self-contained as possible, have a single unified purpose, and communicate with other components through input arguments and return values. They may use mutable arguments to communicate results too if changes are expected, and some types of programs imply other communication mechanisms.

CHAPTER 20

Iterations and Comprehensions, Part 2

This chapter continues the advanced function topics theme, with a reprisal of the comprehension and iteration concepts introduced in Chapter 14. Because list comprehensions are as much related to the prior chapter's functional tools (e.g., `map` and `filter`) as they are to `for` loops, we'll revisit them in this context here. We'll also take a second look at iterators in order to study generator functions and their generator expression relatives—user-defined ways to produce results on demand.

Iteration in Python also encompasses user-defined classes, but we'll defer that final part of this story until Part VI, when we study operator overloading. As this is the last pass we'll make over built-in iteration tools, though, we will summarize the various tools we've met thus far, and time the relative performance of some of them. Finally, because this is the last chapter in the part of the book, we'll close with the usual sets of "gotchas" and exercises to help you start coding the ideas you've read about.

List Comprehensions Revisited: Functional Tools

In the prior chapter, we studied functional programming tools like `map` and `filter`, which map operations over sequences and collect results. Because this is such a common task in Python coding, Python eventually sprouted a new expression—the *list comprehension*—that is even more flexible than the tools we just studied. In short, list comprehensions apply an arbitrary *expression* to items in an iterable, rather than applying a function. As such, they can be more general tools.

We met list comprehensions in Chapter 14, in conjunction with looping statements. Because they're also related to functional programming tools like the `map` and `filter` calls, though, we'll resurrect the topic here for one last look. Technically, this feature is not tied to functions—as we'll see, list comprehensions can be a more general tool than `map` and `filter`—but it is sometimes best understood by analogy to function-based alternatives.

List Comprehensions Versus map

Let's work through an example that demonstrates the basics. As we saw in Chapter 7, Python's built-in `ord` function returns the ASCII integer code of a single character (the `chr` built-in is the converse—it returns the character for an ASCII integer code):

```
>>> ord('s')
115
```

Now, suppose we wish to collect the ASCII codes of *all* characters in an entire string. Perhaps the most straightforward approach is to use a simple `for` loop and append the results to a list:

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

Now that we know about `map`, though, we can achieve similar results with a single function call without having to manage list construction in the code:

```
>>> res = list(map(ord, 'spam'))          # Apply function to sequence
>>> res
[115, 112, 97, 109]
```

However, we can get the same results from a list comprehension expression—while `map` maps a *function* over a sequence, list comprehensions map an *expression* over a sequence:

```
>>> res = [ord(x) for x in 'spam']        # Apply expression to sequence
>>> res
[115, 112, 97, 109]
```

List comprehensions collect the results of applying an arbitrary expression to a sequence of values and return them in a new list. Syntactically, list comprehensions are enclosed in square brackets (to remind you that they construct lists). In their simple form, within the brackets you code an expression that names a variable followed by what looks like a `for` loop header that names the same variable. Python then collects the expression's results for each iteration of the implied loop.

The effect of the preceding example is similar to that of the manual `for` loop and the `map` call. List comprehensions become more convenient, though, when we wish to apply an arbitrary expression to a sequence:

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here, we've collected the squares of the numbers 0 through 9 (we're just letting the interactive prompt print the resulting list; assign it to a variable if you need to retain it). To do similar work with a `map` call, we would probably need to invent a little function to implement the square operation. Because we won't need this function elsewhere,

we'd typically (but not necessarily) code it inline, with a `lambda`, instead of using a `def` statement elsewhere:

```
>>> list(map((lambda x: x ** 2), range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This does the same job, and it's only a few keystrokes longer than the equivalent list comprehension. It's also only marginally more complex (at least, once you understand the `lambda`). For more advanced kinds of expressions, though, list comprehensions will often require considerably less typing. The next section shows why.

Adding Tests and Nested Loops: filter

List comprehensions are even more general than shown so far. For instance, as we learned in Chapter 14, you can code an `if` clause after the `for` to add selection logic. List comprehensions with `if` clauses can be thought of as analogous to the `filter` built-in discussed in the prior chapter—they skip sequence items for which the `if` clause is not true.

To demonstrate, here are both schemes picking up even numbers from 0 to 4; like the `map` list comprehension alternative of the prior section, the `filter` version here must invent a little `lambda` function for the test expression. For comparison, the equivalent `for` loop is shown here as well:

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> list(filter((lambda x: x % 2 == 0), range(5)))
[0, 2, 4]

>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         res.append(x)
...
>>> res
[0, 2, 4]
```

All of these use the modulus (remainder of division) operator, `%`, to detect even numbers: if there is no remainder after dividing a number by 2, it must be even. The `filter` call here is not much longer than the list comprehension either. However, we can combine an `if` clause and an arbitrary expression in our list comprehension, to give it the effect of a `filter` *and* a `map`, in a single expression:

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

This time, we collect the squares of the even numbers from 0 through 9: the `for` loop skips numbers for which the attached `if` clause on the right is false, and the expression on the left computes the squares. The equivalent `map` call would require a lot more work

on our part—we would have to combine `filter` selections with `map` iteration, making for a noticeably more complex expression:

```
>>> list( map((lambda x: x**2), filter((lambda x: x % 2 == 0), range(10))) )
[0, 4, 16, 36, 64]
```

In fact, list comprehensions are more general still. You can code any number of nested `for` loops in a list comprehension, and each may have an optional associated `if` test. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2] ...
             for targetN in iterableN [if conditionN] ]
```

When `for` clauses are nested within a list comprehension, they work like equivalent nested `for` loop statements. For example, the following:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

has the same effect as this substantially more verbose equivalent:

```
>>> res = []
>>> for x in [0, 1, 2]:
...     for y in [100, 200, 300]:
...         res.append(x + y)
...
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Although list comprehensions construct lists, remember that they can iterate over any sequence or other iterable type. Here's a similar bit of code that traverses strings instead of lists of numbers, and so collects concatenation results:

```
>>> [x + y for x in 'spam' for y in 'SPAM']
['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM',
'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']
```

Finally, here is a much more complex list comprehension that illustrates the effect of attached `if` selections on nested `for` clauses:

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

This expression permutes even numbers from 0 through 4 with odd numbers from 0 through 4. The `if` clauses filter out items in each sequence iteration. Here is the equivalent statement-based code:

```
>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         for y in range(5):
...             if y % 2 == 1:
...                 res.append((x, y))
...
```

```
>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Recall that if you're confused about what a complex list comprehension does, you can always nest the list comprehension's `for` and `if` clauses inside each other (indenting successively further to the right) to derive the equivalent statements. The result is longer, but perhaps clearer.

The `map` and `filter` equivalent would be wildly complex and deeply nested, so I won't even try showing it here. I'll leave its coding as an exercise for Zen masters, ex-Lisp programmers, and the criminally insane....

List Comprehensions and Matrixes

Not all list comprehensions are so artificial, of course. Let's look at one more application to stretch a few synapses. One basic way to code matrixes (a.k.a. multidimensional arrays) in Python is with nested list structures. The following, for example, defines two 3 × 3 matrixes as lists of nested lists:

```
>>> M = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]

>>> N = [[2, 2, 2],
...      [3, 3, 3],
...      [4, 4, 4]]
```

Given this structure, we can always index rows, and columns within rows, using normal index operations:

```
>>> M[1]
[4, 5, 6]

>>> M[1][2]
6
```

List comprehensions are powerful tools for processing such structures, though, because they automatically scan rows and columns for us. For instance, although this structure stores the matrix by rows, to collect the second column we can simply iterate across the rows and pull out the desired column, or iterate through positions in the rows and index as we go:

```
>>> [row[1] for row in M]
[2, 5, 8]

>>> [M[row][1] for row in (0, 1, 2)]
[2, 5, 8]
```

Given positions, we can also easily perform tasks such as pulling out a diagonal. The following expression uses `range` to generate the list of offsets and then indexes with the row and column the same, picking out `M[0][0]`, then `M[1][1]`, and so on (we assume the matrix has the same number of rows and columns):

```
>>> [M[i][i] for i in range(len(M))]
[1, 5, 9]
```

Finally, with a bit of creativity, we can also use list comprehensions to combine multiple matrixes. The following first builds a flat list that contains the result of multiplying the matrixes pairwise, and then builds a nested list structure having the same values by nesting list comprehensions:

```
>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]
[2, 4, 6, 12, 15, 18, 28, 32, 36]

>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

This last expression works because the row iteration is an outer loop: for each row, it runs the nested column iteration to build up one row of the result matrix. It's equivalent to this statement-based code:

```
>>> res = []
>>> for row in range(3):
...     tmp = []
...     for col in range(3):
...         tmp.append(M[row][col] * N[row][col])
...     res.append(tmp)
...
>>> res
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Compared to these statements, the list comprehension version requires only one line of code, will probably run substantially faster for large matrixes, and just might make your head explode! Which brings us to the next section.

Comprehending List Comprehensions

With such generality, list comprehensions can quickly become, well, incomprehensible, especially when nested. Consequently, my advice is typically to use simple `for` loops when getting started with Python, and `map` or comprehensions in isolated cases where they are easy to apply. The "keep it simple" rule applies here, as always: code conciseness is a much less important goal than code readability.

However, in this case, there is currently a substantial performance advantage to the extra complexity: based on tests run under Python today, `map` calls are roughly twice as fast as equivalent `for` loops, and list comprehensions are usually slightly faster than `map` calls.* This speed difference is generally due to the fact that `map` and list

* These performance generalizations can depend on call patterns, as well as changes and optimizations in Python itself. Recent Python releases have sped up the simple `for` loop statement, for example. Usually, though, list comprehensions are still substantially faster than `for` loops and even faster than `map` (though `map` can still win for built-in functions). To time these alternatives yourself, see the standard library's `time` module's `time.clock` and `time.time` calls, the newer `timeit` module added in Release 2.4, or this chapter's upcoming section "Timing Iteration Alternatives" on page 509.

comprehensions run at C language speed inside the interpreter, which is much faster than stepping through Python `for` loop code within the PVM.

Because `for` loops make logic more explicit, I recommend them in general on the grounds of simplicity. However, `map` and list comprehensions are worth knowing and using for simpler kinds of iterations, and if your application's speed is an important consideration. In addition, because `map` and list comprehensions are both expressions, they can show up syntactically in places that `for` loop statements cannot, such as in the bodies of `lambda` functions, within list and dictionary literals, and more. Still, you should try to keep your `map` calls and list comprehensions simple; for more complex tasks, use full statements instead.

Why You Will Care: List Comprehensions and map

Here's a more realistic example of list comprehensions and `map` in action (we solved this problem with list comprehensions in Chapter 14, but we'll revive it here to add `map`-based alternatives). Recall that the file `readlines` method returns lines with `\n` end-of-line characters at the ends:

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

If you don't want the end-of-line characters, you can slice them off all the lines in a single step with a list comprehension or a `map` call (`map` results are iterables in Python 3.0, so we must run them through `list` to see all their results at once):

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']

>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']

>>> list(map((lambda line: line.rstrip()), open('myfile')))
['aaa', 'bbb', 'ccc']
```

The last two of these make use of *file iterators* (which essentially means that you don't need a method call to grab all the lines in iteration contexts such as these). The `map` call is slightly longer than the list comprehension, but neither has to manage result list construction explicitly.

A list comprehension can also be used as a sort of column projection operation. Python's standard SQL database API returns query results as a list of tuples much like the following—the list is the table, tuples are rows, and items in tuples are column values:

```
listoftuple = [('bob', 35, 'mgr'), ('mel', 40, 'dev')]
```

A `for` loop could pick up all the values from a selected column manually, but `map` and list comprehensions can do it in a single step, and faster:

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]

>>> list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

The first of these makes use of *tuple assignment* to unpack row tuples in the list, and the second uses indexing. In Python 2.6 (but not in 3.0—see the note on 2.6 argument unpacking in Chapter 18), `map` can use tuple unpacking on its argument, too:

```
# 2.6 only
>>> list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

See other books and resources for more on Python's database API.

Beside the distinction between running functions versus expressions, the biggest difference between `map` and list comprehensions in Python 3.0 is that `map` is an *iterator*, generating results on demand; to achieve the same memory economy, list comprehensions must be coded as generator expressions (one of the topics of this chapter).

Iterators Revisited: Generators

Python today supports procrastination much more than it did in the past—it provides tools that produce results only when needed, instead of all at once. In particular, two language constructs delay result creation whenever possible:

- *Generator functions* are coded as normal `def` statements but use `yield` statements to return results one at a time, suspending and resuming their state between each.
- *Generator expressions* are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list.

Because neither constructs a result list all at once, they save memory space and allow computation time to be split across result requests. As we'll see, both of these ultimately perform their delayed-results magic by implementing the *iteration protocol* we studied in Chapter 14.

Generator Functions: yield Versus return

In this part of the book, we've learned about coding normal functions that receive input parameters and send back a single result immediately. It is also possible, however, to write functions that may send back a value and later be resumed, picking up where they left off. Such functions are known as *generator functions* because they generate a sequence of values over time.

Generator functions are like normal functions in most respects, and in fact are coded with normal `def` statements. However, when created, they are automatically made to implement the iteration protocol so that they can appear in iteration contexts. We studied iterators in Chapter 14; here, we'll revisit them to see how they relate to generators.

State suspension

Unlike normal functions that return a value and exit, generator functions automatically suspend and resume their execution and state around the point of value generation. Because of that, they are often a useful alternative to both computing an entire series of values up front and manually saving and restoring state in classes. Because the state that generator functions retain when they are suspended includes their entire local scope, their local variables retain information and make it available when the functions are resumed.

The chief code difference between generator and normal functions is that a generator *yields* a value, rather than *returning* one—the `yield` statement suspends the function and sends a value back to the caller, but retains enough state to enable the function to resume from where it left off. When resumed, the function continues execution immediately after the last `yield` run. From the function's perspective, this allows its code to produce a series of values over time, rather than computing them all at once and sending them back in something like a list.

Iteration protocol integration

To truly understand generator functions, you need to know that they are closely bound up with the notion of the iteration protocol in Python. As we've seen, iterable objects define a `__next__` method, which either returns the next item in the iteration, or raises the special `StopIteration` exception to end the iteration. An object's iterator is fetched with the `iter` built-in function.

Python `for` loops, and all other iteration contexts, use this iteration protocol to step through a sequence or value generator, if the protocol is supported; if not, iteration falls back on repeatedly indexing sequences instead.

To support this protocol, functions containing a `yield` statement are compiled specially as *generators*. When called, they return a generator object that supports the iteration interface with an automatically created method named `__next__` to resume execution. Generator functions may also have a `return` statement that, along with falling off the end of the `def` block, simply terminates the generation of values—technically, by raising a `StopIteration` exception after any normal function exit actions. From the caller's perspective, the generator's `__next__` method resumes the function and runs until either the next `yield` result is returned or a `StopIteration` is raised.

The net effect is that generator functions, coded as `def` statements containing `yield` statements, are automatically made to support the iteration protocol and thus may be used in any iteration context to produce results over time and on demand.

As noted in Chapter 14, in Python 2.6 and earlier, iterable objects define a method named `next` instead of `__next__`. This includes the generator objects we are using here. In 3.0 this method is renamed to `__next__`. The `next` built-in function is provided as a convenience and portability tool: `next(I)` is the same as `I.__next__()` in 3.0 and `I.next()` in 2.6. Prior to 2.6, programs simply call `I.next()` instead to iterate manually.

Generator functions in action

To illustrate generator basics, let's turn to some code. The following code defines a generator function that can be used to generate the squares of a series of numbers over time:

```
>>> def gensquares(N):
...     for i in range(N):
...         yield i ** 2        # Resume here later
...
```

This function yields a value, and so returns to its caller, each time through the loop; when it is resumed, its prior state is restored and control picks up again immediately after the `yield` statement. For example, when it's used in the body of a `for` loop, control returns to the function after its `yield` statement each time through the loop:

```
>>> for i in gensquares(5):      # Resume the function
...     print(i, end=' : ')      # Print last yielded value
...
0 : 1 : 4 : 9 : 16 :
>>>
```

To end the generation of values, functions either use a `return` statement with no value or simply allow control to fall off the end of the function body.

If you want to see what is going on inside the `for`, call the generator function directly:

```
>>> x = gensquares(4)
>>> x
<generator object at 0x0086C378>
```

You get back a generator object that supports the iteration protocol we met in Chapter 14—the generator object has a `__next__` method that starts the function, or resumes it from where it last yielded a value, and raises a `StopIteration` exception when the end of the series of values is reached. For convenience, the `next(X)` built-in calls an object's `X.__next__()` method for us:

```
>>> next(x)                      # Same as x.__next__() in 3.0
0
>>> next(x)                      # Use x.next() or next() in 2.6
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
```

```
Traceback (most recent call last):
...more text omitted...
StopIteration
```

As we learned in Chapter 14, `for` loops (and other iteration contexts) work with generators in the same way—by calling the `__next__` method repeatedly, until an exception is caught. If the object to be iterated over does not support this protocol, `for` loops instead use the indexing protocol to iterate.

Note that in this example, we could also simply build the list of yielded values all at once:

```
>>> def buildsquares(n):
...     res = []
...     for i in range(n): res.append(i ** 2)
...     return res
...
>>> for x in buildsquares(5): print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
```

For that matter, we could use any of the `for` loop, `map`, or list comprehension techniques:

```
>>> for x in [n ** 2 for n in range(5)]:
...     print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :

>>> for x in map((lambda n: n ** 2), range(5)):
...     print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
```

However, generators can be better in terms of both memory use and performance. They allow functions to avoid doing all the work up front, which is especially useful when the result lists are large or when it takes a lot of computation to produce each value. Generators distribute the time required to produce the series of values among loop iterations.

Moreover, for more advanced uses, generators can provide a simpler alternative to manually saving the state between iterations in class objects—with generators, variables accessible in the function's scopes are saved and restored automatically.† We'll discuss class-based iterators in more detail in Part VI.

† Interestingly, generator functions are also something of a "poor man's" *multithreading* device—they interleave a function's work with that of its caller, by dividing its operation into steps run between `yield`s. Generators are not threads, though: the program is explicitly directed to and from the function within a single thread of control. In one sense, threading is more general (producers can run truly independently and post results to a queue), but generators may be simpler to code. See the second footnote in Chapter 17 for a brief introduction to Python multithreading tools. Note that because control is routed explicitly at `yield` and `next` calls, generators are also not *backtracking*, but are more strongly related to *coroutines*—formal concepts that are both beyond this chapter's scope.

Extended generator function protocol: send versus next

In Python 2.5, a `send` method was added to the generator function protocol. The `send` method advances to the next item in the series of results, just like `__next__`, but also provides a way for the caller to communicate with the generator, to affect its operation.

Technically, `yield` is now an expression form that returns the item passed to `send`, not a statement (though it can be called either way—as `yield X`, or `A = (yield X)`). The expression must be enclosed in parentheses unless it's the only item on the right side of the assignment statement. For example, `X = yield Y` is OK, as is `X = (yield Y) + 42`.

When this extra protocol is used, values are sent into a generator `G` by calling `G.send(`*`value`*`)`. The generator's code is then resumed, and the `yield` expression in the generator returns the value passed to `send`. If the regular `G.__next__()` method (or its `next(G)` equivalent) is called to advance, the `yield` simply returns `None`. For example:

```
>>> def gen():
...     for i in range(10):
...         X = yield i
...         print(X)
...
>>> G = gen()
>>> next(G)                 # Must call next() first, to start generator
0
>>> G.send(77)              # Advance, and send value to yield expression
77
1
>>> G.send(88)
88
2
>>> next(G)                 # next() and X.__next__() send None
None
3
```

The `send` method can be used, for example, to code a generator that its caller can terminate by sending a termination code, or redirect by passing a new position. In addition, generators in 2.5 also support a `throw(`*`type`*`)` method to raise an exception inside the generator at the latest `yield`, and a `close` method that raises a special `GeneratorExit` exception inside the generator to terminate the iteration. These are advanced features that we won't delve into in more detail here; see reference texts and Python's standard manuals for more information.

Note that while Python 3.0 provides a `next(X)` convenience built-in that calls the `X.__next__()` method of an object, other generator methods, like `send`, must be called as methods of generator objects directly (e.g., `G.send(X)`). This makes sense if you realize that these extra methods are implemented on built-in generator objects only, whereas the `__next__` method applies to all iterable objects (both built-in types and user-defined classes).

Generator Expressions: Iterators Meet Comprehensions

In all recent versions of Python, the notions of iterators and list comprehensions are combined in a new feature of the language, *generator expressions*. Syntactically, generator expressions are just like normal list comprehensions, but they are enclosed in parentheses instead of square brackets:

```
>>> [x ** 2 for x in range(4)]          # List comprehension: build a list
[0, 1, 4, 9]

>>> (x ** 2 for x in range(4))          # Generator expression: make an iterable
<generator object at 0x011DC648>
```

In fact, at least on a function basis, coding a list comprehension is essentially the same as wrapping a generator expression in a `list` built-in call to force it to produce all its results in a list at once:

```
>>> list(x ** 2 for x in range(4))      # List comprehension equivalence
[0, 1, 4, 9]
```

Operationally, however, generator expressions are very different—instead of building the result list in memory, they return a generator object, which in turn supports the *iteration protocol* to yield one piece of the result list at a time in any iteration context:

```
>>> G = (x ** 2 for x in range(4))
>>> next(G)
0
>>> next(G)
1
>>> next(G)
4
>>> next(G)
9
>>> next(G)

Traceback (most recent call last):
...more text omitted...
StopIteration
```

We don't typically see the `next` iterator machinery under the hood of a generator expression like this because `for` loops trigger it for us automatically:

```
>>> for num in (x ** 2 for x in range(4)):
...     print('%s, %s' % (num, num / 2.0))
...
0, 0.0
1, 0.5
4, 2.0
9, 4.5
```

As we've already learned, every iteration context does this, including the `sum`, `map`, and `sorted` built-in functions; list comprehensions; and other iteration contexts we learned about in Chapter 14, such as the `any`, `all`, and `list` built-in functions.

Notice that the parentheses are not required around a generator expression if they are the sole item enclosed in other parentheses, like those of a function call. Extra parentheses are required, however, in the second call to `sorted`:

```
>>> sum(x ** 2 for x in range(4))
14

>>> sorted(x ** 2 for x in range(4))
[0, 1, 4, 9]

>>> sorted((x ** 2 for x in range(4)), reverse=True)
[9, 4, 1, 0]

>>> import math
>>> list( map(math.sqrt, (x ** 2 for x in range(4))) )
[0.0, 1.0, 2.0, 3.0]
```

Generator expressions are primarily a memory-space optimization—they do not require the entire result list to be constructed all at once, as the square-bracketed list comprehension does. They may also run slightly slower in practice, so they are probably best used only for very large result sets. A more authoritative statement about performance, though, will have to await the timing script we'll code later in this chapter.

Generator Functions Versus Generator Expressions

Interestingly, the same iteration can often be coded with either a generator function or a generator expression. The following *generator expression*, for example, repeats each character in a string four times:

```
>>> G = (c * 4 for c in 'SPAM')          # Generator expression
>>> list(G)                              # Force generator to produce all results
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

The equivalent *generator function* requires slightly more code, but as a multistatement function it will be able to code more logic and use more state information if needed:

```
>>> def timesfour(S):                    # Generator function
...     for c in S:
...         yield c * 4
...
>>> G = timesfour('spam')
>>> list(G)                              # Iterate automatically
['ssss', 'pppp', 'aaaa', 'mmmm']
```

Both expressions and functions support both automatic and manual iteration—the prior `list` call iterates automatically, and the following iterate manually:

```
>>> G = (c * 4 for c in 'SPAM')
>>> I = iter(G)                          # Iterate manually
>>> next(I)
'SSSS'
>>> next(I)
'PPPP'
```

```
>>> G = timesfour('spam')
>>> I = iter(G)
>>> next(I)
'ssss'
>>> next(I)
'pppp'
```

Notice that we make new generators here to iterate again—as explained in the next section, generators are one-shot iterators.

Generators Are Single-Iterator Objects

Both generator functions and generator expressions are their own iterators and thus support just *one active iteration*—unlike some built-in types, you can't have multiple iterators of either positioned at different locations in the set of results. For example, using the prior section's generator expression, a generator's iterator is the generator itself (in fact, calling `iter` on a generator is a no-op):

```
>>> G = (c * 4 for c in 'SPAM')
>>> iter(G) is G                        # My iterator is myself: G has __next__
True
```

If you iterate over the results stream manually with multiple iterators, they will all point to the same position:

```
>>> G = (c * 4 for c in 'SPAM')         # Make a new generator
>>> I1 = iter(G)                        # Iterate manually
>>> next(I1)
'SSSS'
>>> next(I1)
'PPPP'
>>> I2 = iter(G)                        # Second iterator at same position!
>>> next(I2)
'AAAA'
```

Moreover, once any iteration runs to completion, all are exhausted—we have to make a new generator to start again:

```
>>> list(I1)                            # Collect the rest of I1's items
['MMMM']
>>> next(I2)                            # Other iterators exhausted too
StopIteration

>>> I3 = iter(G)                        # Ditto for new iterators
>>> next(I3)
StopIteration

>>> I3 = iter(c * 4 for c in 'SPAM')    # New generator to start over
>>> next(I3)
'SSSS'
```

The same holds true for generator functions—the following `def` statement-based equivalent supports just one active iterator and is exhausted after one pass:

```
>>> def timesfour(S):
...     for c in S:
...         yield c * 4
...
>>> G = timesfour('spam')                  # Generator functions work the same way
>>> iter(G) is G
True
>>> I1, I2 = iter(G), iter(G)
>>> next(I1)
'ssss'
>>> next(I1)
'pppp'
>>> next(I2)                               # I2 at same position as I1
'aaaa'
```

This is different from the behavior of some built-in types, which support multiple iterators and passes and reflect their in-place changes in active iterators:

```
>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2)                               # Lists support multiple iterators
1
>>> del L[2:]                              # Changes reflected in iterators
>>> next(I1)
StopIteration
```

When we begin coding class-based iterators in Part VI, we'll see that it's up to us to decide how any iterations we wish to support for our objects, if any.

Emulating zip and map with Iteration Tools

To demonstrate the power of iteration tools in action, let's turn to some more advanced use case examples. Once you know about list comprehensions, generators, and other iteration tools, it turns out that emulating many of Python's functional built-ins is both straightforward and instructive.

For example, we've already seen how the built-in `zip` and `map` functions combine iterables and project functions across them, respectively. With multiple sequence arguments, `map` projects the function across items taken from each sequence in much the same way that `zip` pairs them up:

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

```
# zip pairs items, truncates at shortest

>>> list(zip([-2, -1, 0, 1, 2]))          # Single sequence: 1-ary tuples
[(-2,), (-1,), (0,), (1,), (2,)]

>>> list(zip([1, 2, 3], [2, 3, 4, 5]))    # N sequences: N-ary tuples
[(1, 2), (2, 3), (3, 4)]

# map passes paired itenms to a function, truncates

>>> list(map(abs, [-2, -1, 0, 1, 2]))     # Single sequence: 1-ary function
[2, 1, 0, 1, 2]

>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))   # N sequences: N-ary function
[1, 8, 81]
```

Though they're being used for different purposes, if you study these examples long enough, you might notice a relationship between `zip` results and mapped function arguments that our next example can exploit.

Coding your own map(func, ...)

Although the `map` and `zip` built-ins are fast and convenient, it's always possible to emulate them in code of our own. In the preceding chapter, for example, we saw a function that emulated the `map` built-in for a single sequence argument. It doesn't take much more work to allow for multiple sequences, as the built-in does:

```
# map(func, seqs...) workalike with zip

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

This version relies heavily upon the special `*args` argument-passing syntax—it collects multiple sequence (really, iterable) arguments, unpacks them as `zip` arguments to combine, and then unpacks the paired `zip` results as arguments to the passed-in function. That is, we're using the fact that the zipping is essentially a nested operation in mapping. The test code at the bottom applies this to both one and two sequences to produce this output (the same we would get with the built-in `map`):

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Really, though, the prior version exhibits the classic *list comprehension pattern*, building a list of operation results within a `for` loop. We can code our map more concisely as an equivalent one-line list comprehension:

```
# Using a list comprehension

def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

When this is run the result is the same as before, but the code is more concise and might run faster (more on performance in the section "Timing Iteration Alternatives" on page 509). Both of the preceding `mymap` versions build result lists all at once, though, and this can waste memory for larger lists. Now that we know about *generator functions and expressions*, it's simple to recode both these alternatives to produce results on demand instead:

```
# Using generators: yield and (...)

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        yield func(*args)

def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))
```

These versions produce the same results but return generators designed to support the iteration protocol—the first yields one result at a time, and the second returns a generator expression's result to do the same. They produce the same results if we wrap them in `list` calls to force them to produce their values all at once:

```
print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

No work is really done here until the `list` calls force the generators to run, by activating the iteration protocol. The generators returned by these functions themselves, as well as that returned by the Python 3.0 flavor of the `zip` built-in they use, produce results only on demand.

Coding your own zip(...) and map(None, ...)

Of course, much of the magic in the examples shown so far lies in their use of the `zip` built-in to pair arguments from multiple sequences. You'll also note that our `map` workalikes are really emulating the behavior of the Python 3.0 `map`—they truncate at the length of the shortest sequence, and they do not support the notion of padding results when lengths differ, as `map` does in Python 2.X with a `None` argument:

```
C:\misc> c:\python26\python
>>> map(None, [1, 2, 3], [2, 3, 4, 5])
[(1, 2), (2, 3), (3, 4), (None, 5)]
>>> map(None, 'abc', 'xyz123')
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Using iteration tools, we can code workalikes that emulate both truncating `zip` and 2.6's padding `map`—these turn out to be nearly the same in code:

```
# zip(seqs...) and 2.6 map(None, seqs...) workalikes

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res  = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res  = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Both of the functions coded here work on any type of iterable object, because they run their arguments through the `list` built-in to force result generation (e.g., files would work as arguments, in addition to sequences like strings). Notice the use of the `all` and `any` built-ins here—these return `True` if all and any items in an iterable are `True` (or equivalently, nonempty), respectively. These built-ins are used to stop looping when any or all of the listified arguments become empty after deletions.

Also note the use of the Python 3.0 *keyword-only* argument, `pad`; unlike the 2.6 `map`, our version will allow any pad object to be specified (if you're using 2.6, use a `**kargs` form to support this option instead; see Chapter 18 for details). When these functions are run, the following results are printed—a `zip`, and two padding `map`s:

```
[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]
```

These functions aren't amenable to list comprehension translation because their loops are too specific. As before, though, while our `zip` and `map` workalikes currently build and return result lists, it's just as easy to turn them into *generators* with `yield` so that they each return one piece of their result set at a time. The results are the same as before, but we need to use `list` again to force the generators to yield their values for display:

```
# Using generators: yield

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)
```

```
def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))
print(list(mymapPad(S1, S2)))
print(list(mymapPad(S1, S2, pad=99)))
```

Finally, here's an alternative implementation of our `zip` and `map` emulators—rather than deleting arguments from lists with the `pop` method, the following versions do their job by calculating the minimum and maximum *argument lengths*. Armed with these lengths, it's easy to code nested list comprehensions to step through argument index ranges:

```
# Alternate implementation with lengths

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index  = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Because these use `len` and indexing, they assume that arguments are sequences or similar, not arbitrary iterables. The outer comprehensions here step through argument index ranges, and the inner comprehensions (passed to `tuple`) step through the passed-in sequences to pull out arguments in parallel. When they're run, the results are as before.

Most strikingly, generators and iterators seem to run rampant in this example. The arguments passed to `min` and `max` are generator expressions, which run to completion before the nested comprehensions begin iterating. Moreover, the nested list comprehensions employ two levels of delayed evaluation—the Python 3.0 `range` built-in is an iterable, as is the generator expression argument to `tuple`.

In fact, no results are produced here until the square brackets of the list comprehensions request values to place in the result list—they force the comprehensions and generators to run. To turn these functions themselves into generators instead of list builders, use parentheses instead of square brackets again. Here's the case for our `zip`:

```
# Using generators: (...)

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
```

```
        return (tuple(S[i] for S in seqs) for i in range(minlen))

print(list(myzip(S1, S2)))
```

In this case, it takes a `list` call to activate the generators and iterators to produce their results. Experiment with these on your own for more details. Developing further coding alternatives is left as a suggested exercise (see also the sidebar "Why You Will Care: One-Shot Iterations" for investigation of one such option).

Why You Will Care: One-Shot Iterations

In Chapter 14, we saw how some built-ins (like `map`) support only a single traversal and are empty after it occurs, and I promised to show you an example of how that can become subtle but important in practice. Now that we've studied a few more iteration topics, I can make good on this promise. Consider the following clever alternative coding for this chapter's `zip` emulation examples, adapted from one in Python's manuals:

```
def myzip(*args):
    iters = map(iter, args)
    while iters:
        res = [next(i) for i in iters]
        yield tuple(res)
```

Because this code uses `iter` and `next`, it works on any type of iterable. Note that there is no reason to catch the `StopIteration` raised by the `next(it)` inside the comprehension here when any one of the arguments' iterators is exhausted—allowing it to pass ends this generator function and has the same effect that a `return` statement would. The `while iters:` suffices to loop if at least one argument is passed, and avoids an infinite loop otherwise (the list comprehension would always return an empty list).

This code works fine in Python 2.6 as is:

```
>>> list(myzip('abc', 'lmnop'))
[('a', 'l'), ('b', 'm'), ('c', 'n')]
```

But it falls into an infinite loop and fails in Python 3.0, because the 3.0 `map` returns a one-shot iterable object instead of a list as in 2.6. In 3.0, as soon as we've run the list comprehension inside the loop once, `iters` will be empty (and `res` will be `[]`) forever. To make this work in 3.0, we need to use the `list` built-in function to create an object that can support multiple iterations:

```
def myzip(*args):
    iters = list(map(iter, args))
    ...rest as is...
```

Run this on your own to trace its operation. The lesson here: wrapping `map` calls in `list` calls in 3.0 is not just for display!

Value Generation in Built-in Types and Classes

Finally, although we've focused on coding value generators ourselves in this section, don't forget that many built-in types behave in similar ways—as we saw in Chapter 14, for example, dictionaries have iterators that produce keys on each iteration:

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

Like the values produced by handcoded generators, dictionary keys may be iterated over both manually and with automatic iteration tools including `for` loops, `map` calls, list comprehensions, and the many other contexts we met in Chapter 14:

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

As we've also seen, for file iterators, Python simply loads lines from the file on demand:

```
>>> for line in open('temp.txt'):
...     print(line, end='')
...
Tis but
a flesh wound.
```

While built-in type iterators are bound to a specific type of value generation, the concept is similar to generators we code with expressions and functions. Iteration contexts like `for` loops accept any iterable, whether user-defined or built-in.

Although beyond the scope of this chapter, it is also possible to implement arbitrary user-defined generator objects with *classes* that conform to the iteration protocol. Such classes define a special `__iter__` method run by the `iter` built-in function that returns an object having a `__next__` method run by the `next` built-in function (a `__getitem__` indexing method is also available as a fallback option for iteration).

The instance objects created from such a class are considered iterable and may be used in `for` loops and all other iteration contexts. With classes, though, we have access to richer logic and data structuring options than other generator constructs can offer.

The iterator story won't really be complete until we've seen how it maps to classes, too. For now, we'll have to settle for postponing its conclusion until we study class-based iterators in Chapter 29.

3.0 Comprehension Syntax Summary

We've been focusing on list comprehensions and generators in this chapter, but keep in mind that there are two other comprehension expression forms: set and dictionary comprehensions are also available as of Python 3.0. We met these briefly in Chapters 5 and 8, but with our new knowledge of comprehensions and generators, you should now be able to grasp these 3.0 extensions in full:

- For *sets*, the new literal form `{1, 3, 2}` is equivalent to `set([1, 3, 2])`, and the new set comprehension syntax `{f(x) for x in S if P(x)}` is like the generator expression `set(f(x) for x in S if P(x))`, where `f(x)` is an arbitrary expression.
- For *dictionaries*, the new dictionary comprehension syntax `{key: val for (key, val) in zip(keys, vals)}` works like the form `dict(zip(keys, vals))`, and `{x: f(x) for x in items}` is like the generator expression `dict((x, f(x)) for x in items)`.

Here's a summary of all the comprehension alternatives in 3.0. The last two are new and are not available in 2.6:

```
>>> [x * x for x in range(10)]              # List comprehension: builds list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]        # like list(generator expr)

>>> (x * x for x in range(10))              # Generator expression: produces items
<generator object at 0x009E7328>            # Parens are often optional

>>> {x * x for x in range(10)}              # Set comprehension, new in 3.0
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}        # {x, y} is a set in 3.0 too

>>> {x: x * x for x in range(10)}           # Dictionary comprehension, new in 3.0
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Comprehending Set and Dictionary Comprehensions

In a sense, set and dictionary comprehensions are just syntactic sugar for passing generator expressions to the type names. Because both accept any iterable, a generator works well here:

```
>>> {x * x for x in range(10)}              # Comprehension
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> set(x * x for x in range(10))           # Generator and type name
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> {x: x * x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

As for list comprehensions, though, we can always build the result objects with manual code, too. Here are statement-based equivalents of the last two comprehensions:

```
>>> res = set()
>>> for x in range(10):                    # Set comprehension equivalent
...     res.add(x * x)
...
>>> res
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> res = {}
>>> for x in range(10):                    # Dict comprehension equivalent
...     res[x] = x * x
...
>>> res
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Notice that although both forms accept iterators, they have no notion of generating results on demand—both forms build objects all at once. If you mean to produce keys and values upon request, a generator expression is more appropriate:

```
>>> G = ((x, x * x) for x in range(10))
>>> next(G)
(0, 0)
>>> next(G)
(1, 1)
```

Extended Comprehension Syntax for Sets and Dictionaries

Like list comprehensions and generator expressions, both set and dictionary comprehensions support nested associated `if` clauses to filter items out of the result—the following collect squares of even items (i.e., items having no remainder for division by 2) in a range:

```
>>> [x * x for x in range(10) if x % 2 == 0]          # Lists are ordered
[0, 4, 16, 36, 64]
>>> {x * x for x in range(10) if x % 2 == 0}          # But sets are not
{0, 16, 4, 64, 36}
>>> {x: x * x for x in range(10) if x % 2 == 0}       # Neither are dict keys
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}
```

Nested `for` loops work as well, though the unordered and no-duplicates nature of both types of objects can make the results a bit less straightforward to decipher:

```
>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]]     # Lists keep duplicates
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]}     # But sets do not
{8, 9, 5, 6, 7}
>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]}      # Neither do dict keys
{1: 6, 2: 6, 3: 6}
```

Like list comprehensions, the set and dictionary varieties can also iterate over any type of iterator—lists, strings, files, ranges, and anything else that supports the iteration protocol:

```
>>> {x + y for x in 'ab' for y in 'cd'}
{'bd', 'ac', 'ad', 'bc'}
```

```
>>> {x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}
{'bd': (98, 100), 'ac': (97, 99), 'ad': (97, 100), 'bc': (98, 99)}

>>> {k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'sausagesausage', 'spamspam'}

>>> {k.upper(): k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'SAUSAGE': 'sausagesausage', 'SPAM': 'spamspam'}
```

For more details, experiment with these tools on your own. They may or may not have a performance advantage over the generator or `for` loop alternatives, but we would have to time their performance explicitly to be sure—which seems a natural segue to the next section.

Timing Iteration Alternatives

We've met quite a few iteration alternatives in this book. To summarize, let's work through a larger case study that pulls together some of the things we've learned about iteration and functions.

I've mentioned a few times that list comprehensions have a speed performance advantage over `for` loop statements, and that `map` performance can be better or worse depending on call patterns. The generator expressions of the prior sections tend to be slightly slower than list comprehensions, though they minimize memory space requirements.

All that's true today, but relative performance can vary over time because Python's internals are constantly being changed and optimized. If you want to verify their performance for yourself, you need to time these alternatives on your own computer and your own version of Python.

Timing Module

Luckily, Python makes it easy to time code. To see how the iteration options stack up, let's start with a simple but general timer utility function coded in a module file, so it can be used in a variety of programs:

```
# File mytimer.py

import time
reps = 1000
repslist = range(reps)

def timer(func, *pargs, **kargs):
    start = time.clock()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = time.clock() - start
    return (elapsed, ret)
```

Operationally, this module times calls to any function with any positional and keyword arguments by fetching the start time, calling the function a fixed number of times, and subtracting the start time from the stop time. Points to notice:

- Python's `time` module gives access to the current time, with precision that varies per platform. On Windows, this call is claimed to give microsecond granularity and so is very accurate.
- The `range` call is hoisted out of the timing loop, so its construction cost is not charged to the timed function in Python 2.6. In 3.0 `range` is an iterator, so this step isn't required (but doesn't hurt).
- The `reps` count is a global that importers can change if needed: `mytimer.reps = N`.

When complete, the total elapsed time for all calls is returned in a tuple, along with the timed function's final return value so callers can verify its operation.

From a larger perspective, because this function is coded in a module file, it becomes a generally useful tool anywhere we wish to import it. You'll learn more about modules and imports in the next part of this book, but you've already seen enough of the basics to make sense of this code—simply import the module and call the function to use this file's timer (and see Chapter 3's coverage of module attributes if you need a refresher).

Timing Script

Now, to time iteration tool speed, run the following script—it uses the timer module we just wrote to time the relative speeds of the various list construction techniques we've studied:

```
# File timeseqs.py

import sys, mytimer                          # Import timer function
reps = 10000
repslist = range(reps)                       # Hoist range out in 2.6

def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repslist]

def mapCall():
    return list(map(abs, repslist))          # Use list in 3.0 only

def genExpr():
    return list(abs(x) for x in repslist)    # list forces results

def genFunc():
    def gen():
```

```
        for x in repslist:
            yield abs(x)
    return list(gen())

print(sys.version)
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    elapsed, result = mytimer.timer(test)
    print ('-' * 33)
    print ('%-9s: %.5f => [%s...%s]' %
           (test.__name__, elapsed, result[0], result[-1]))
```

This script tests five alternative ways to build lists of results and, as shown, executes on the order of 10 million steps for each—that is, each of the five tests builds a list of 10,000 items 1,000 times.

Notice how we have to run the generator expression and function results through the built-in `list` call to force them to yield all of their values; if we did not, we would just produce generators that never do any real work. In Python 3.0 (only) we must do the same for the `map` result, since it is now an iterable object as well. Also notice how the code at the bottom steps through a tuple of four function objects and prints the `__name__` of each: as we've seen, this is a built-in attribute that gives a function's name.

Timing Results

When the script of the prior section is run under Python 3.0, I get the following results on my Windows Vista laptop—`map` is slightly faster than list comprehensions, both are substantially quicker than `for` loops, and generator expressions and functions place in the middle:

```
C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
---------------------------------
forLoop  : 2.64441 => [0...9999]
---------------------------------
listComp : 1.60110 => [0...9999]
---------------------------------
mapCall  : 1.41977 => [0...9999]
---------------------------------
genExpr  : 2.21758 => [0...9999]
---------------------------------
genFunc  : 2.18696 => [0...9999]
```

If you study this code and its output long enough, you'll notice that generator expressions run slower than list comprehensions. Although wrapping a generator expression in a `list` call makes it functionally equivalent to a square-bracketed list comprehension, the internal implementations of the two expressions appear to differ (though we're also effectively timing the `list` call for the generator test):

```
return [abs(x) for x in range(size)]          # 1.6 seconds
return list(abs(x) for x in range(size))      # 2.2 seconds: differs internally
```

Interestingly, when I ran this on Windows XP with Python 2.5 for the prior edition of this book, the results were relatively similar—list comprehensions were nearly twice as fast as equivalent `for` loop statements, and `map` was slightly quicker than list comprehensions when mapping a built-in function such as `abs` (absolute value). I didn't test generator functions then, and the output format wasn't quite as grandiose:

```
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement         => 6.10899996758
listComprehension    => 3.51499986649
mapFunction          => 2.73399996758
generatorExpression  => 4.11600017548
```

The fact that the actual 2.5 test times listed here are over two times as slow as the output I showed earlier is likely due to my using a quicker laptop for the more recent test, not due to improvements in Python 3.0. In fact, all the 2.6 results for this script are slightly quicker than 3.0 on this same machine if the `list` call is removed from the `map` test to avoid creating the results list twice (try this on your own to verify).

Watch what happens, though, if we change this script to perform a real operation on each iteration, such as addition, instead of calling a trivial built-in function like `abs` (the omitted parts of the following are the same as before):

```
# File timeseqs.py
...
...
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res

def listComp():
    return [x + 10 for x in repslist]

def mapCall():
    return list(map((lambda x: x + 10), repslist))          # list in 3.0 only

def genExpr():
    return list(x + 10 for x in repslist)                   # list in 2.6 + 3.0

def genFunc():
    def gen():
        for x in repslist:
            yield x + 10
    return list(gen())
...
...
```

Now the need to call a user-defined function for the `map` call makes it slower than the `for` loop statements, despite the fact that the looping statements version is larger in terms of code. On Python 3.0:

```
C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
```

```
--------------------------------
forLoop  : 2.60754 => [10...10009]
--------------------------------
listComp : 1.57585 => [10...10009]
--------------------------------
mapCall  : 3.10276 => [10...10009]
--------------------------------
genExpr  : 1.96482 => [10...10009]
--------------------------------
genFunc  : 1.95340 => [10...10009]
```

The Python 2.5 results on a slower machine were again relatively similar in the prior edition, but twice as slow due to test machine differences:

```
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement         => 5.25699996948
listComprehension    => 2.68400001526
mapFunction          => 5.96900010109
generatorExpression  => 3.37400007248
```

Because the interpreter optimizes so much internally, performance analysis of Python code like this is a very tricky affair. It's virtually impossible to guess which method will perform the best—the best you can do is time your own code, on your computer, with your version of Python. In this case, all we should say for certain is that on this Python, using a user-defined function in `map` calls can slow performance by at least a factor of 2, and that list comprehensions run quickest for this test.

As I've mentioned before, however, performance should not be your primary concern when writing Python code—the first thing you should do to optimize Python code is to not optimize Python code! Write for *readability and simplicity* first, then optimize later, if and only if needed. It could very well be that any of the five alternatives is quick enough for the data sets your program needs to process; if so, program clarity should be the chief goal.

Timing Module Alternatives

The timing module of the prior section works, but it's a bit primitive on multiple fronts:

- It always uses the `time.clock` call to time code. While that option is best on Windows, the `time.time` call may provide better resolution on some Unix platforms.
- Adjusting the number of repetitions requires changing module-level globals—a less than ideal arrangement if the `timer` function is being used and shared by multiple importers.
- As is, the timer works by running the test function a large number of times. To account for random system load fluctuations, it might be better to select the *best* time among all the tests, instead of the *total* time.

The following alternative implements a more sophisticated timer module that addresses all three points by selecting a timer call based on platform, allowing the repeat count

to be passed in as a keyword argument named `_reps`, and providing a best-of-N alternative timing function:

```
# File mytimer.py (2.6 and 3.0)

"""
timer(spam, 1, 2, a=3, b=4, _reps=1000) calls and times spam(1, 2, a=3)
_reps times, and returns total time for all runs, with final result;

best(spam, 1, 2, a=3, b=4, _reps=50) runs best-of-N timer to filter out
any system load variation, and returns best time among _reps tests
"""

import time, sys
if sys.platform[:3] == 'win':
    timefunc = time.clock               # Use time.clock on Windows
else:
    timefunc = time.time                # Better resolution on some Unix platforms

def trace(*args): pass                  # Or: print args

def timer(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000)    # Passed-in or default reps
    trace(func, pargs, kargs, _reps)
    repslist = range(_reps)             # Hoist range out for 2.6 lists
    start = timefunc()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)

def best(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 50)
    best = 2 ** 32
    for i in range(_reps):
        (time, ret) = timer(func, *pargs, _reps=1, **kargs)
        if time < best: best = time
    return (best, ret)
```

This module's docstring at the top of the file describes its intended usage. It uses dictionary `pop` operations to remove the `_reps` argument from arguments intended for the test function and provide it with a default, and it traces arguments during development if you change its `trace` function to `print`. To test with this new timer module on either Python 3.0 or 2.6, change the timing script as follows (the omitted code in the test functions of this version use the `x + 1` operation for each test, as coded in the prior section):

```
# File timeseqs.py

import sys, mytimer
reps = 10000
repslist = range(reps)

def forLoop(): ...
```

```
def listComp(): ...

def mapCall(): ...

def genExpr(): ...

def genFunc(): ...

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (forLoop, listComp, mapCall, genExpr, genFunc):
        elapsed, result = tester(test)
        print ('-' * 35)
        print ('%-9s: %.5f => [%s...%s]' %
               (test.__name__, elapsed, result[0], result[-1]))
```

When run under Python 3.0, the timing results are essentially the same as before, and relatively the same for both to the total-of-N and best-of-N timing techniques—running tests many times seems to do as good a job filtering out system load fluctuations as taking the best case, but the best-of-N scheme may be better when testing a long-running function. The results on my machine are as follows:

```
C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----------------------------------
forLoop  : 2.35371 => [10...10009]
-----------------------------------
listComp : 1.29640 => [10...10009]
-----------------------------------
mapCall  : 3.16556 => [10...10009]
-----------------------------------
genExpr  : 1.97440 => [10...10009]
-----------------------------------
genFunc  : 1.95072 => [10...10009]
<best>
-----------------------------------
forLoop  : 0.00193 => [10...10009]
-----------------------------------
listComp : 0.00124 => [10...10009]
-----------------------------------
mapCall  : 0.00268 => [10...10009]
-----------------------------------
genExpr  : 0.00164 => [10...10009]
-----------------------------------
genFunc  : 0.00165 => [10...10009]
```

The times reported by the best-of-N timer here are small, of course, but they might become significant if your program iterates many times over large data sets. At least in terms of relative performance, list comprehensions appear best in most cases; `map` is only slightly better when built-ins are applied.

Using keyword-only arguments in 3.0

We can also make use of Python 3.0 *keyword-only arguments* here to simplify the timer module's code. As we learned in Chapter 19, keyword-only arguments are ideal for configuration options such as our functions' `_reps` argument. They must be coded after a * and before a ** in the function header, and in a function call they must be passed by keyword and appear before the ** if used. Here's a keyword-only-based alternative to the prior module. Though simpler, it compiles and runs under Python 3.X only, not 2.6:

```
# File mytimer.py (3.X only)

"""
Use 3.0 keyword-only default arguments, instead of ** and dict pops.
No need to hoist range() out of test in 3.0: a generator, not a list
"""

import time, sys
trace = lambda *args: None  # or print
timefunc = time.clock if sys.platform == 'win32' else time.time

def timer(func, *pargs, _reps=1000, **kargs):
    trace(func, pargs, kargs, _reps)
    start = timefunc()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)

def best(func, *pargs, _reps=50, **kargs):
    best = 2 ** 32
    for i in range(_reps):
        (time, ret) = timer(func, *pargs, _reps=1, **kargs)
        if time < best: best = time
    return (best, ret)
```

This version is used the same way as and produces results identical to the prior version, not counting negligible test time differences from run to run:

```
C:\misc> c:\python30\python timeseqs.py
...same results as before...
```

In fact, for variety we can also test this version of the module from the interactive prompt, completely independent of the sequence timer script—it's a general-purpose tool:

```
C:\misc> c:\python30\python
>>> from mytimer import timer, best
>>>
>>> def power(X, Y): return X ** Y            # Test function
...
>>> timer(power, 2, 32)                       # Total time, last result
(0.002625403507987747, 4294967296)
>>> timer(power, 2, 32, _reps=1000000)        # Override defult reps
```

```
(1.1822605247314932, 4294967296)
>>> timer(power, 2, 100000)[0]                  # 2 ** 100,000 tot time @1,000 reps
2.2496919999608878

>>> best(power, 2, 32)                          # Best time, last result
(5.587302297271189e-06, 4294967296)
>>> best(power, 2, 100000)[0]                   # 2 ** 100,000 best time
0.0019937589833460834
>>> best(power, 2, 100000, _reps=500)[0]        # Override default reps
0.0019845399345541637
```

For trivial functions like the one tested in this interactive session, the costs of the timer's code are probably as significant as those of the timed function, so you should not take timer results too absolutely (we are timing more than just `X ** Y` here). The timer's results can help you judge relative speeds of coding alternatives, though, and may be more meaningful for longer-running operations like the following—calculating 2 to the power one million takes an order of magnitude (power of 10) longer than the preceding `2**100,000`:

```
>>> timer(power, 2, 1000000, _reps=1)[0]        # 2 ** 1,000,000: total time
0.088112804839710179
>>> timer(power, 2, 1000000, _reps=10)[0]
0.40922470593329763

>>> best(power, 2, 1000000, _reps=1)[0]         # 2 ** 1,000,000: best time
0.086550036387279761
>>> best(power, 2, 1000000, _reps=10)[0]        # 10 is sometimes as good as 50
0.029616752967200455
>>> best(power, 2, 1000000, _reps=50)[0]        # Best resolution
0.029486918030102061
```

Again, although the times measured here are small, the differences can be significant in programs that compute powers often.

See Chapter 19 for more on keyword-only arguments in 3.0; they can simplify code for configurable tools like this one but are not backward compatible with 2.X Pythons. If you want to compare 2.X and 3.X speed, for example, or support programmers using either Python line, the prior version is likely a better choice. If you're using Python 2.6, the above session runs the same with the prior version of the timer module.

Other Suggestions

For more insight, try modifying the repetition counts used by these modules, or explore the alternative `timeit` module in Python's standard library, which automates timing of code, supports command-line usage modes, and finesses some platform-specific issues. Python's manuals document its use.

You might also want to look at the `profile` standard library module for a complete source code profiler tool—we'll learn more about it in Chapter 35 in the context of development tools for large projects. In general, you should profile code to isolate bottlenecks before recoding and timing alternatives as we've done here.

It might be useful as well to experiment with using the new `str.format` method in Python 2.6 and 3.0 instead of the `%` formatting expression (which could potentially be deprecated in the future!), by changing the timing script's formatted `print` lines as follows:

```
print('<%s>' % tester.__name__)              # From expression

    print('<{0}>'.format(tester.__name__))       # To method call

        print ('%-9s: %.5f => [%s...%s]' %
               (test.__name__, elapsed, result[0], result[-1]))

        print('{0:<9}: {1:.5f} => [{2}...{3}]'.format(
                test.__name__, elapsed, result[0], result[-1]))
```

You can judge the difference between these techniques yourself.

If you feel ambitious, you might also try modifying or emulating the timing script to measure the speed of the 3.0 *set and dictionary comprehensions* illustrated in this chapter, and their `for` loop equivalents. Since using them is much less common in Python programs than building lists of results, we'll leave this task in the suggested exercise column (and please, no wagering...).

Finally, keep the timing module we wrote here filed away for future reference—we'll repurpose it to measure performance of alternative numeric square root operations in an exercise at the end of this chapter. If you're interested in pursuing this topic further, we'll also experiment with techniques for timing dictionary comprehensions versus `for` loops interactively.

Function Gotchas

Now that we've reached the end of the function story, let's review some common pitfalls. Functions have some jagged edges that you might not expect. They're all obscure, and a few have started to fall away from the language completely in recent releases, but most have been known to trip up new users.

Local Names Are Detected Statically

As you know, Python classifies names assigned in a function as *locals* by default; they live in the function's scope and exist only while the function is running. What you may not realize is that Python detects locals statically, when it compiles the `def`'s code, rather than by noticing assignments as they happen at runtime. This leads to one of the most common oddities posted on the Python newsgroup by beginners.

Normally, a name that isn't assigned in a function is looked up in the enclosing module:

```
>>> X = 99
>>> def selector():          # X used but not assigned
...     print(X)             # X found in global scope
...
>>> selector()
99
```

Here, the `X` in the function resolves to the `X` in the module. But watch what happens if you add an assignment to `X` after the reference:

```
>>> def selector():
...     print(X)             # Does not yet exist!
...     X = 88               # X classified as a local name (everywhere)
...                          # Can also happen for "import X", "def X"...
>>> selector()
...error text omitted...
UnboundLocalError: local variable 'X' referenced before assignment
```

You get the name usage error shown here, but the reason is subtle. Python reads and compiles this code when it's typed interactively or imported from a module. While compiling, Python sees the assignment to `X` and decides that `X` will be a local name everywhere in the function. But when the function is actually run, because the assignment hasn't yet happened when the `print` executes, Python says you're using an undefined name. According to its name rules, it should say this; the local `X` is used before being assigned. In fact, any assignment in a function body makes a name local. Imports, `=`, nested `def`s, nested classes, and so on are all susceptible to this behavior.

The problem occurs because assigned names are treated as locals everywhere in a function, not just after the statements where they are assigned. Really, the previous example is ambiguous at best: was the intention to print the global `X` and then create a local `X`, or is this a genuine programming error? Because Python treats `X` as a local everywhere, it is viewed as an error; if you really mean to print the global `X`, you need to declare it in a `global` statement:

```
>>> def selector():
...     global X                # Force X to be global (everywhere)
...     print(X)
...     X = 88
...
>>> selector()
99
```

Remember, though, that this means the assignment also changes the global `X`, not a local `X`. Within a function, you can't use both local and global versions of the same simple name. If you really meant to print the global and then set a local of the same name, you'd need to import the enclosing module and use module attribute notation to get to the global version:

```
>>> X = 99
>>> def selector():
...     import __main__         # Import enclosing module
...     print(__main__.X)       # Qualify to get to global version of name
...     X = 88                  # Unqualified X classified as local
```

```
...     print(X)                    # Prints local version of name
...
>>> selector()
99
88
```

Qualification (the `.X` part) fetches a value from a namespace object. The interactive namespace is a module called `__main__`, so `__main__.X` reaches the global version of `X`. If that isn't clear, check out Chapter 17.

In recent versions Python has improved on this story somewhat by issuing for this case the more specific "unbound local" error message shown in the example listing (it used to simply raise a generic name error); this gotcha is still present in general, though.

Defaults and Mutable Objects

Default argument values are evaluated and saved when a `def` statement is run, not when the resulting function is called. Internally, Python saves one object per default argument attached to the function itself.

That's usually what you want—because defaults are evaluated at `def` time, it lets you save values from the enclosing scope, if needed. But because a default retains an object between calls, you have to be careful about changing mutable defaults. For instance, the following function uses an empty list as a default value, and then changes it in-place each time the function is called:

```
>>> def saver(x=[]):                # Saves away a list object
...     x.append(1)                 # Changes same object each time!
...     print(x)
...
>>> saver([2])                      # Default not used
[2, 1]
>>> saver()                         # Default used
[1]
>>> saver()                         # Grows on each call!
[1, 1]
>>> saver()
[1, 1, 1]
```

Some see this behavior as a feature—because mutable default arguments retain their state between function calls, they can serve some of the same roles as *static* local function variables in the C language. In a sense, they work sort of like global variables, but their names are local to the functions and so will not clash with names elsewhere in a program.

To most observers, though, this seems like a gotcha, especially the first time they run into it. There are better ways to retain state between calls in Python (e.g., using classes, which will be discussed in Part VI).

Moreover, mutable defaults are tricky to remember (and to understand at all). They depend upon the timing of default object construction. In the prior example, there is

just one list object for the default value—the one created when the `def` is executed. You don't get a new list every time the function is called, so the list grows with each new append; it is not reset to empty on each call.

If that's not the behavior you want, simply make a copy of the default at the start of the function body, or move the default value expression into the function body. As long as the value resides in code that's actually executed each time the function runs, you'll get a new object each time through:

```
>>> def saver(x=None):
...     if x is None:               # No argument passed?
...         x = []                  # Run code to make a new list
...     x.append(1)                 # Changes new list object
...     print(x)
...
>>> saver([2])
[2, 1]
>>> saver()                        # Doesn't grow here
[1]
>>> saver()
[1]
```

By the way, the `if` statement in this example could *almost* be replaced by the assignment `x = x or []`, which takes advantage of the fact that Python's `or` returns one of its operand objects: if no argument was passed, `x` would default to `None`, so the `or` would return the new empty list on the right.

However, this isn't exactly the same. If an empty list were passed in, the `or` expression would cause the function to extend and return a newly created list, rather than extending and returning the passed-in list like the `if` version. (The expression becomes `[] or []`, which evaluates to the new empty list on the right; see the section "Truth Tests" on page 320 if you don't recall why). Real program requirements may call for either behavior.

Today, another way to achieve the effect of mutable defaults in a possibly less confusing way is to use the *function attributes* we discussed in Chapter 19:

```
>>> def saver():
...     saver.x.append(1)
...     print(saver.x)
...
>>> saver.x = []
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

The function name is global to the function itself, but it need not be declared because it isn't changed directly within the function. This isn't used in exactly the same way,

but when coded like this, the attachment of an object to the function is much more explicit (and arguably less magical).

Functions Without returns

In Python functions, `return` (and `yield`) statements are optional. When a function doesn't return a value explicitly, the function exits when control falls off the end of the function body. Technically, all functions return a value; if you don't provide a `return` statement, your function returns the `None` object automatically:

```
>>> def proc(x):
...     print(x)                    # No return is a None return
...
>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Functions such as this without a `return` are Python's equivalent of what are called "procedures" in some languages. They're usually invoked as statements, and the `None` results are ignored, as they do their business without computing a useful result.

This is worth knowing, because Python won't tell you if you try to use the result of a function that doesn't return one. For instance, assigning the result of a list `append` method won't raise an error, but you'll get back `None`, not the modified list:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)          # append is a "procedure"
>>> print(list)                    # append changes list in-place
None
```

As mentioned in "Common Coding Gotchas" on page 387 in Chapter 15, such functions do their business as a side effect and are usually designed to be run as statements, not expressions.

Enclosing Scope Loop Variables

We described this gotcha in Chapter 17's discussion of enclosing function scopes, but as a reminder, be careful about relying on enclosing function scope lookup for variables that are changed by enclosing loops—all such references will remember the value of the *last* loop iteration. Use defaults to save loop variable values instead (see Chapter 17 for more details on this topic).

Chapter Summary

This chapter wrapped up our coverage of built-in comprehension and iteration tools. It explored list comprehensions in the context of functional tools and presented generator functions and expressions as additional iteration protocol tools. As a finale, we

also measured the performance of iteration alternatives, and we closed with a review of common function-related mistakes to help you avoid pitfalls.

This concludes the functions part of this book. In the next part, we will study *modules*—the topmost organizational structure in Python, and the structure in which our functions always live. After that, we will explore classes, tools that are largely packages of functions with special first arguments. As we'll see, user-defined classes can implement objects that tap into the iteration protocol, just like the generators and iterables we met here. Everything we have learned in this part of the book will apply when functions pop up later in the context of class methods.

Before moving on to modules, though, be sure to work through this chapter's quiz and the exercises for this part of the book, to practice what we've learned about functions here.

Test Your Knowledge: Quiz

1. What is the difference between enclosing a list comprehension in square brackets and parentheses?
2. How are generators and iterators related?
3. How can you tell if a function is a generator function?
4. What does a `yield` statement do?
5. How are `map` calls and list comprehensions related? Compare and contrast the two.

Test Your Knowledge: Answers

1. List comprehensions in square brackets produce the result list all at once in memory. When they are enclosed in parentheses instead, they are actually generator expressions—they have a similar meaning but do not produce the result list all at once. Instead, generator expressions return a generator object, which yields one item in the result at a time when used in an iteration context.
2. Generators are objects that support the iteration protocol—they have a `__next__` method that repeatedly advances to the next item in a series of results and raises an exception at the end of the series. In Python, we can code generator functions with `def`, generator expressions with parenthesized list comprehensions, and generator objects with classes that define a special method named `__iter__` (discussed later in the book).
3. A generator function has a `yield` statement somewhere in its code. Generator functions are otherwise identical to normal functions syntactically, but they are compiled specially by Python so as to return an iterable object when called.

4. When present, this statement makes Python compile the function specially as a generator; when called, the function returns a generator object that supports the iteration protocol. When the `yield` statement is run, it sends a result back to the caller and suspends the function's state; the function can then be resumed after the last `yield` statement, in response to a `next` built-in or `__next__` method call issued by the caller. Generator functions may also have a `return` statement, which terminates the generator.
5. The `map` call is similar to a list comprehension—both build a new list by collecting the results of applying an operation to each item in a sequence or other iterable, one item at a time. The main difference is that `map` applies a function call to each item, and list comprehensions apply arbitrary expressions. Because of this, list comprehensions are more general; they can apply a function call expression like `map`, but `map` requires a function to apply other kinds of expressions. List comprehensions also support extended syntax such as nested `for` loops and `if` clauses that subsume the `filter` built-in.

Test Your Knowledge: Part IV Exercises

In these exercises, you're going to start coding more sophisticated programs. Be sure to check the solutions in "Part IV, Functions" on page 1111 in Appendix B, and be sure to start writing your code in module files. You won't want to retype these exercises from scratch if you make a mistake.

1. *The basics*. At the Python interactive prompt, write a function that prints its single argument to the screen and call it interactively, passing a variety of object types: string, integer, list, dictionary. Then, try calling it without passing any argument. What happens? What happens when you pass two arguments?
2. *Arguments*. Write a function called `adder` in a Python module file. The function should accept two arguments and return the sum (or concatenation) of the two. Then, add code at the bottom of the file to call the `adder` function with a variety of object types (two strings, two lists, two floating points), and run this file as a script from the system command line. Do you have to print the call statement results to see results on your screen?
3. *varargs*. Generalize the `adder` function you wrote in the last exercise to compute the sum of an arbitrary number of arguments, and change the calls to pass more or fewer than two arguments. What type is the return value sum? (Hints: a slice such as `S[:0]` returns an empty sequence of the same type as `S`, and the `type` built-in function can test types; but see the manually coded `min` examples in Chapter 18 for a simpler approach.) What happens if you pass in arguments of different types? What about passing in dictionaries?

4. *Keywords*. Change the `adder` function from exercise 2 to accept and sum/concatenate three arguments: `def adder(good, bad, ugly)`. Now, provide default values for each argument, and experiment with calling the function interactively. Try passing one, two, three, and four arguments. Then, try passing keyword arguments. Does the call `adder(ugly=1, good=2)` work? Why? Finally, generalize the new `adder` to accept and sum/concatenate an arbitrary number of keyword arguments. This is similar to what you did in exercise 3, but you'll need to iterate over a dictionary, not a tuple. (Hint: the `dict.keys` method returns a list you can step through with a `for` or `while`, but be sure to wrap it in a `list` call to index it in 3.0!)
5. Write a function called `copyDict(dict)` that copies its dictionary argument. It should return a new dictionary containing all the items in its argument. Use the dictionary `keys` method to iterate (or, in Python 2.2, step over a dictionary's keys without calling `keys`). Copying sequences is easy (`X[:]` makes a top-level copy); does this work for dictionaries, too?
6. Write a function called `addDict(dict1, dict2)` that computes the union of two dictionaries. It should return a new dictionary containing all the items in both its arguments (which are assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either. Test your function by writing it in a file and running the file as a script. What happens if you pass lists instead of dictionaries? How could you generalize your function to handle this case, too? (Hint: see the `type` built-in function used earlier.) Does the order of the arguments passed in matter?
7. *More argument-matching examples*. First, define the following six functions (either interactively or in a module file that can be imported):

```
def f1(a, b): print(a, b)           # Normal args
def f2(a, *b): print(a, b)          # Positional varargs

def f3(a, **b): print(a, b)         # Keyword varargs

def f4(a, *b, **c): print(a, b, c)  # Mixed modes

def f5(a, b=2, c=3): print(a, b, c) # Defaults

def f6(a, b=2, *c): print(a, b, c)  # Defaults and positional varargs
```

   Now, test the following calls interactively, and try to explain each result; in some cases, you'll probably need to fall back on the matching algorithm shown in Chapter 18. Do you think mixing matching modes is a good idea in general? Can you think of cases where it would be useful?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)
```

```
>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

8. *Primes revisited*. Recall the following code snippet from Chapter 13, which simplistically determines whether a positive integer is prime:

```
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')
```

   Package this code as a reusable function in a module file (`y` should be a passed-in argument), and add some calls to the function at the bottom of your file. While you're at it, experiment with replacing the first line's `//` operator with `/` to see how true division changes the `/` operator in Python 3.0 and breaks this code (refer back to Chapter 5 if you need a refresher). What can you do about negatives, and the values `0` and `1`? How about speeding this up? Your outputs should look something like this:

```
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
```

9. *List comprehensions*. Write code to build a new list containing the square roots of all the numbers in this list: `[2, 4, 9, 16, 25]`. Code this as a `for` loop first, then as a `map` call, and finally as a list comprehension. Use the `sqrt` function in the built-in `math` module to do the calculation (i.e., import `math` and say `math.sqrt(x)`). Of the three, which approach do you like best?

10. *Timing tools*. In Chapter 5, we saw three ways to compute square roots: `math.sqrt(X)`, `X ** .5`, and `pow(X, .5)`. If your programs run a lot these, their relative performance might become important. To see which is quickest, repurpose the *timerseqs.py* script we wrote in this chapter to time each of these three tools. Use the *mytimer.py* timer module with the `best` function (you can use either the 3.0-ony keyword-only variant, or the 2.6/3.0 version). You might also want to repackage the testing code in this script for better reusability—by passing a test functions tuple to a general tester function, for example (for this exercise a copy-and-modify approach is fine). Which of the three square root tools seems to run fastest on your machine and Python in general? Finally, how might you go about interactively timing the speed of dictionary comprehensions versus `for` loops?

PART V

Modules

CHAPTER 21

Modules: The Big Picture

This chapter begins our in-depth look at the Python *module*, the highest-level program organization unit, which packages program code and data for reuse. In concrete terms, modules usually correspond to Python program files (or extensions coded in external languages such as C, Java, or C#). Each file is a module, and modules import other modules to use the names they define. Modules are processed with two statements and one important function:

`import`
: Lets a client (importer) fetch a module as a whole

`from`
: Allows clients to fetch particular names from a module

`imp.reload`
: Provides a way to reload a module's code without stopping Python

Chapter 3 introduced module fundamentals, and we've been using them ever since. This part of the book begins by expanding on core module concepts, then moves on to explore more advanced module usage. This first chapter offers a general look at the role of modules in overall program structure. In the following chapters, we'll dig into the coding details behind the theory.

Along the way, we'll flesh out module details omitted so far: you'll learn about reloads, the `__name__` and `__all__` attributes, package imports, relative import syntax, and so on. Because modules and classes are really just glorified namespaces, we'll formalize namespace concepts here as well.

Why Use Modules?

In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as *namespaces*. All the names defined at the top level of a module file become attributes of the imported module object. As we saw in the last part of this book, imports give access to names in a module's global

scope. That is, the module file's global scope morphs into the module object's attribute namespace when it is imported. Ultimately, Python's modules allow us to link individual files into a larger program system.

More specifically, from an abstract perspective, modules have at least three roles:

Code reuse
: As discussed in Chapter 3, modules let you save code in files permanently. Unlike code you type at the Python interactive prompt, which goes away when you exit Python, code in module files is persistent—it can be reloaded and rerun as many times as needed. More to the point, modules are a place to define names, known as *attributes*, which may be referenced by multiple external clients.

System namespace partitioning
: Modules are also the highest-level program organization unit in Python. Fundamentally, they are just packages of names. Modules seal up names into self-contained packages, which helps avoid name clashes—you can never see a name in another file, unless you explicitly import that file. In fact, everything "lives" in a module—code you execute and objects you create are always implicitly enclosed in modules. Because of that, modules are natural tools for grouping system components.

Implementing shared services or data
: From an operational perspective, modules also come in handy for implementing components that are shared across a system and hence require only a single copy. For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

For you to truly understand the role of modules in a Python system, though, we need to digress for a moment and explore the general structure of a Python program.

Python Program Architecture

So far in this book, I've sugarcoated some of the complexity in my descriptions of Python programs. In practice, programs usually involve more than just one file; for all but the simplest scripts, your programs will take the form of multifile systems. And even if you can get by with coding a single file yourself, you will almost certainly wind up using external files that someone else has already written.

This section introduces the general architecture of Python programs—the way you divide a program into a collection of source files (a.k.a. modules) and link the parts into a whole. Along the way, we'll also explore the central concepts of Python modules, imports, and object attributes.

How to Structure a Program

Generally, a Python program consists of multiple text files containing Python *statements*. The program is structured as one main, *top-level* file, along with zero or more supplemental files known as *modules* in Python.

In Python, the top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application. The module files are libraries of tools used to collect components used by the top-level file (and possibly elsewhere). Top-level files use tools defined in module files, and modules use tools defined in other modules.

Module files generally don't do anything when run directly; rather, they define tools intended for use in other files. In Python, a file *imports* a module to gain access to the tools it defines, which are known as its *attributes* (i.e., variable names attached to objects such as functions). Ultimately, we import modules and access their attributes to use their tools.

Imports and Attributes

Let's make this a bit more concrete. Figure 21-1 sketches the structure of a Python program composed of three files: *a.py*, *b.py*, and *c.py*. The file *a.py* is chosen to be the top-level file; it will be a simple text file of statements, which is executed from top to bottom when launched. The files *b.py* and *c.py* are modules; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools they define.

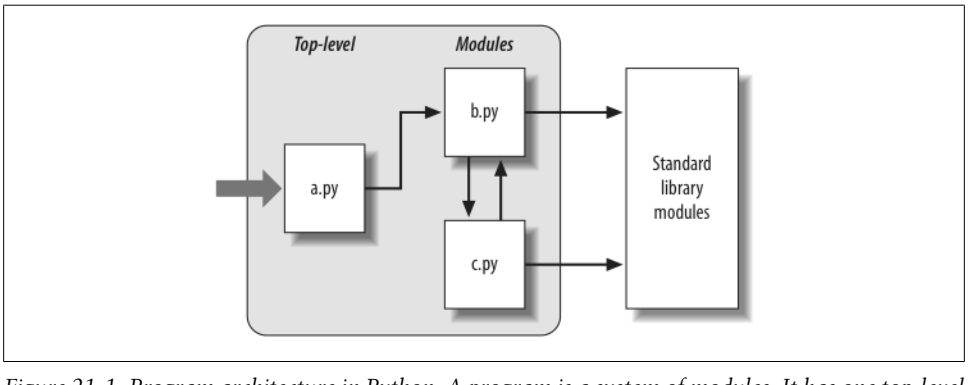

Figure 21-1. Program architecture in Python. A program is a system of modules. It has one top-level script file (launched to run the program), and multiple module files (imported libraries of tools). Scripts and modules are both text files containing Python statements, though the statements in modules usually just create objects to be used later. Python's standard library provides a collection of precoded modules.

For instance, suppose the file *b.py* in Figure 21-1 defines a function called `spam`, for external use. As we learned when studying functions in Part IV, *b.py* will contain a Python `def` statement to generate the function, which can later be run by passing zero or more values in parentheses after the function's name:

```
def spam(text):
    print(text, 'spam')
```

Now, suppose *a.py* wants to use `spam`. To this end, it might contain Python statements such as the following:

```
import b
b.spam('gumby')
```

The first of these, a Python `import` statement, gives the file *a.py* access to everything defined by top-level code in the file *b.py*. It roughly means "load the file *b.py* (unless it's already loaded), and give me access to all its attributes through the name `b`." `import` (and, as you'll see later, `from`) statements execute and load other files at runtime.

In Python, cross-file module linking is not resolved until such `import` statements are executed at runtime; their net effect is to assign module names—simple variables—to loaded module objects. In fact, the module name used in an `import` statement serves two purposes: it identifies the external file to be loaded, but it also becomes a variable assigned to the loaded module. Objects defined by a module are also created at runtime, as the `import` is executing: `import` literally runs statements in the target file one at a time to create its contents.

The second of the statements in *a.py* calls the function `spam` defined in the module `b`, using object attribute notation. The code `b.spam` means "fetch the value of the name `spam` that lives within the object `b`." This happens to be a callable function in our example, so we pass a string in parentheses (`'gumby'`). If you actually type these files, save them, and run *a.py*, the words "gumby spam" will be printed.

You'll see the `object.attribute` notation used throughout Python scripts—most objects have useful attributes that are fetched with the "." operator. Some are callable things like functions, and others are simple data values that give object properties (e.g., a person's name).

The notion of importing is also completely general throughout Python. Any file can import tools from any other file. For instance, the file *a.py* may import *b.py* to call its function, but *b.py* might also import *c.py* to leverage different tools defined there. Import chains can go as deep as you like: in this example, the module `a` can import `b`, which can import `c`, which can import `b` again, and so on.

Besides serving as the highest organizational structure, modules (and module packages, described in Chapter 23) are also the highest level of *code reuse* in Python. Coding components in module files makes them useful in your original program, and in any other programs you may write. For instance, if after coding the program in Figure 21-1 we discover that the function `b.spam` is a general-purpose tool, we can reuse

it in a completely different program; all we have to do is import the file *b.py* again from the other program's files.

Standard Library Modules

Notice the rightmost portion of Figure 21-1. Some of the modules that your programs will import are provided by Python itself and are not files you will code.

Python automatically comes with a large collection of utility modules known as the *standard library*. This collection, roughly 200 modules large at last count, contains platform-independent support for common programming tasks: operating system interfaces, object persistence, text pattern matching, network and Internet scripting, GUI construction, and much more. None of these tools are part of the Python language itself, but you can use them by importing the appropriate modules on any standard Python installation. Because they are standard library modules, you can also be reasonably sure that they will be available and will work portably on most platforms on which you will run Python.

You will see a few of the standard library modules in action in this book's examples, but for a complete look you should browse the standard Python library reference manual, available either with your Python installation (via IDLE or the Python Start button menu on Windows) or online at *http://www.python.org*.

Because there are so many modules, this is really the only way to get a feel for what tools are available. You can also find tutorials on Python library tools in commercial books that cover application-level programming, such as O'Reilly's *Programming Python*, but the manuals are free, viewable in any web browser (they ship in HTML format), and updated each time Python is rereleased.

How Imports Work

The prior section talked about importing modules without really explaining what happens when you do so. Because imports are at the heart of program structure in Python, this section goes into more detail on the import operation to make this process less abstract.

Some C programmers like to compare the Python module import operation to a `C #include`, but they really shouldn't—in Python, imports are not just textual insertions of one file into another. They are really runtime operations that perform three distinct steps the first time a program imports a given file:

1. *Find* the module's file.
2. *Compile* it to byte code (if needed).
3. *Run* the module's code to build the objects it defines.

To better understand module imports, we'll explore these steps in turn. Bear in mind that all three of these steps are carried out only the *first time* a module is imported during a program's execution; later imports of the same module bypass all of these steps and simply fetch the already loaded module object in memory. Technically, Python does this by storing loaded modules in a table named `sys.modules` and checking there at the start of an import operation. If the module is not present, a three-step process begins.

1. Find It

First, Python must locate the module file referenced by an `import` statement. Notice that the `import` statement in the prior section's example names the file without a *.py* suffix and without its directory path: it just says `import b`, instead of something like `import c:\dir1\b.py`. In fact, you can only list a simple name; path and suffix details are omitted on purpose and Python uses a standard *module search path* to locate the module file corresponding to an `import` statement.* Because this is the main part of the import operation that programmers must know about, we'll return to this topic in a moment.

2. Compile It (Maybe)

After finding a source code file that matches an `import` statement by traversing the module search path, Python next compiles it to byte code, if necessary. (We discussed byte code in Chapter 2.)

Python checks the file timestamps and, if the byte code file is older than the source file (i.e., if you've changed the source), automatically regenerates the byte code when the program is run. If, on the other hand, it finds a *.pyc* byte code file that is not older than the corresponding *.py* source file, it skips the source-to–byte code compile step. In addition, if Python finds only a byte code file on the search path and no source, it simply loads the byte code directly (this means you can ship a program as just byte code files and avoid sending source). In other words, the compile step is bypassed if possible to speed program startup.

Notice that compilation happens when a file is being imported. Because of this, you will not usually see a *.pyc* byte code file for the top-level file of your program, unless it is also imported elsewhere—only imported files leave behind *.pyc* files on your

* It's actually syntactically illegal to include path and suffix details in a standard `import`. *Package imports*, which we'll discuss in Chapter 23, allow `import` statements to include part of the directory path leading to a file as a set of period-separated names; however, package imports still rely on the normal module search path to locate the leftmost directory in a package path (i.e., they are relative to a directory in the search path). They also cannot make use of any platform-specific directory syntax in the `import` statements; such syntax only works on the search path. Also, note that module file search path issues are not as relevant when you run *frozen executables* (discussed in Chapter 2); they typically embed byte code in the binary image.

machine. The byte code of top-level files is used internally and discarded; byte code of imported files is saved in files to speed future imports.

Top-level files are often designed to be executed directly and not imported at all. Later, we'll see that it is possible to design a file that serves both as the top-level code of a program and as a module of tools to be imported. Such a file may be both executed and imported, and thus does generate a *.pyc*. To learn how this works, watch for the discussion of the special `__name__` attribute and `__main__` in Chapter 24.

3. Run It

The final step of an import operation executes the byte code of the module. All statements in the file are executed in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This execution step therefore generates all the tools that the module's code defines. For instance, `def` statements in a file are run at import time to create functions and assign attributes within the module to those functions. The functions can then be called later in the program by the file's importers.

Because this last import step actually runs the file's code, if any top-level code in a module file does real work, you'll see its results at import time. For example, top-level `print` statements in a module show output when the file is imported. Function `def` statements simply define objects for later use.

As you can see, import operations involve quite a bit of work—they search for files, possibly run a compiler, and run Python code. Because of this, any given module is imported only *once* per process by default. Future imports skip all three import steps and reuse the already loaded module in memory. If you need to import a file again after it has already been loaded (for example, to support end-user customization), you have to force the issue with an `imp.reload` call—a tool we'll meet in the next chapter.†

The Module Search Path

As mentioned earlier, the part of the import procedure that is most important to programmers is usually the first—locating the file to be imported (the "find it" part). Because you may need to tell Python where to look to find files to import, you need to know how to tap into its search path in order to extend it.

† As described earlier, Python keeps already imported modules in the built-in `sys.modules` dictionary so it can keep track of what's been loaded. In fact, if you want to see which modules are loaded, you can import `sys` and print `list(sys.modules.keys())`. More on other uses for this internal table in Chapter 24.

In many cases, you can rely on the automatic nature of the module import search path and won't need to configure this path at all. If you want to be able to import files across directory boundaries, though, you will need to know how the search path works in order to customize it. Roughly, Python's module search path is composed of the concatenation of these major components, some of which are preset for you and some of which you can tailor to tell Python where to look:

1. The home directory of the program
2. `PYTHONPATH` directories (if set)
3. Standard library directories
4. The contents of any *.pth* files (if present)

Ultimately, the concatenation of these four components becomes `sys.path`, a list of directory name strings that I'll expand upon later in this section. The first and third elements of the search path are defined automatically. Because Python searches the concatenation of these components from first to last, though, the second and fourth elements can be used to extend the path to include your own source code directories. Here is how Python uses each of these path components:

Home directory
: Python first looks for the imported file in the home directory. The meaning of this entry depends on how you are running the code. When you're running a program, this entry is the directory containing your program's top-level script file. When you're working interactively, this entry is the directory in which you are working (i.e., the current working directory).

  Because this directory is always searched first, if a program is located entirely in a single directory, all of its imports will work automatically with no path configuration required. On the other hand, because this directory is searched first, its files will also override modules of the same name in directories elsewhere on the path; be careful not to accidentally hide library modules this way if you need them in your program.

`PYTHONPATH` *directories*
: Next, Python searches all directories listed in your `PYTHONPATH` environment variable setting, from left to right (assuming you have set this at all). In brief, `PYTHONPATH` is simply set to a list of user-defined and platform-specific names of directories that contain Python code files. You can add all the directories from which you wish to be able to import, and Python will extend the module search path to include all the directories your `PYTHONPATH` lists.

  Because Python searches the home directory first, this setting is only important when importing files across directory boundaries—that is, if you need to import a file that is stored in a different directory from the file that imports it. You'll probably want to set your `PYTHONPATH` variable once you start writing substantial programs, but when you're first starting out, as long as you save all your module files in the

directory in which you're working (i.e., the home directory, described earlier) your imports will work without you needing to worry about this setting at all.

Standard library directories
: Next, Python automatically searches the directories where the standard library modules are installed on your machine. Because these are always searched, they normally do not need to be added to your `PYTHONPATH` or included in path files (discussed next).

.pth path file directories
: Finally, a lesser-used feature of Python allows users to add directories to the module search path by simply listing them, one per line, in a text file whose name ends with a *.pth* suffix (for "path"). These path configuration files are a somewhat advanced installation-related feature; we won't them cover fully here, but they provide an alternative to `PYTHONPATH` settings.

  In short, text files of directory names dropped in an appropriate directory can serve roughly the same role as the `PYTHONPATH` environment variable setting. For instance, if you're running Windows and Python 3.0, a file named *myconfig.pth* may be placed at the top level of the Python install directory (*C:\Python30*) or in the *site-packages* subdirectory of the standard library there (*C:\Python30\Lib\site-packages*) to extend the module search path. On Unix-like systems, this file might be located in *usr/local/lib/python3.0/site-packages* or */usr/local/lib/site-python* instead.

  When present, Python will add the directories listed on each line of the file, from first to last, near the end of the module search path list. In fact, Python will collect the directory names in all the path files it finds and will filter out any duplicates and nonexistent directories. Because they are files rather than shell settings, path files can apply to all users of an installation, instead of just one user or shell. Moreover, for some users text files may be simpler to code than environment settings.

  This feature is more sophisticated than I've described here. For more details consult the Python library manual, and especially its documentation for the standard library module `site`—this module allows the locations of Python libraries and path files to be configured, and its documentation describes the expected locations of path files in general. I recommend that beginners use `PYTHONPATH` or perhaps a single *.pth* file, and then only if you must import across directories. Path files are used more often by third-party libraries, which commonly install a path file in Python's *site-packages* directory so that user settings are not required (Python's `distutils` install system, described in an upcoming sidebar, automates many install steps).

Configuring the Search Path

The net effect of all of this is that both the `PYTHONPATH` and path file components of the search path allow you to tailor the places where imports look for files. The way you set environment variables and where you store path files varies per platform. For instance,

on Windows, you might use your Control Panel's System icon to set PYTHONPATH to a list of directories separated by semicolons, like this:

```
c:\pycode\utilities;d:\pycode\package1
```

Or you might instead create a text file called *C:\Python30\pydirs.pth*, which looks like this:

```
c:\pycode\utilities
d:\pycode\package1
```

These settings are analogous on other platforms, but the details can vary too widely for us to cover in this chapter. See Appendix A for pointers on extending your module search path with PYTHONPATH or *.pth* files on various platforms.

Search Path Variations

This description of the module search path is accurate, but generic; the exact configuration of the search path is prone to changing across platforms and Python releases. Depending on your platform, additional directories may automatically be added to the module search path as well.

For instance, Python may add an entry for the *current working directory*—the directory from which you launched your program—in the search path after the PYTHONPATH directories, and before the standard library entries. When you're launching from a command line, the current working directory may not be the same as the home directory of your top-level file (i.e., the directory where your program file resides). Because the current working directory can vary each time your program runs, you normally shouldn't depend on its value for import purposes. See Chapter 3 for more on launching programs from command lines.‡

To see how your Python configures the module search path on your platform, you can always inspect sys.path—the topic of the next section.

The sys.path List

If you want to see how the module search path is truly configured on your machine, you can always inspect the path as Python knows it by printing the built-in sys.path list (that is, the path attribute of the standard library module sys). This list of directory name strings is the actual search path within Python; on imports, Python searches each directory in this list from left to right.

‡ See also Chapter 23's discussion of the new *relative import syntax* in Python 3.0; this modifies the search path for from statements in files inside packages when "." characters are used (e.g., from . import string). By default, a package's own directory is not automatically searched by imports in Python 3.0, unless relative imports are used by files in the package itself.

Really, `sys.path` *is* the module search path. Python configures it at program startup, automatically merging the home directory of the top-level file (or an empty string to designate the current working directory), any `PYTHONPATH` directories, the contents of any *.pth* file paths you've created, and the standard library directories. The result is a list of directory name strings that Python searches on each import of a new file.

Python exposes this list for two good reasons. First, it provides a way to verify the search path settings you've made—if you don't see your settings somewhere in this list, you need to recheck your work. For example, here is what my module search path looks like on Windows under Python 3.0, with my `PYTHONPATH` set to `C:\users` and a *C:\Python30\mypath.py* path file that lists `C:\users\mark`. The empty string at the front means current directory and my two settings are merged in (the rest are standard library directories and files):

```
>>> import sys
>>> sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', 'c:\\Python30\\DLLs',
'c:\\Python30\\lib', 'c:\\Python30\\lib\\plat-win', 'c:\\Python30',
'C:\\Users\\Mark', 'c:\\Python30\\lib\\site-packages']
```

Second, if you know what you're doing, this list provides a way for scripts to tailor their search paths manually. As you'll see later in this part of the book, by modifying the `sys.path` list, you can modify the search path for all future imports. Such changes only last for the duration of the script, however; `PYTHONPATH` and *.pth* files offer more permanent ways to modify the path.§

Module File Selection

Keep in mind that filename suffixes (e.g., *.py*) are intentionally omitted from `import` statements. Python chooses the first file it can find on the search path that matches the imported name. For example, an `import` statement of the form `import b` might load:

- A source code file named *b.py*
- A byte code file named *b.pyc*
- A directory named *b*, for package imports (described in Chapter 23)
- A compiled extension module, usually coded in C or C++ and dynamically linked when imported (e.g., *b.so* on Linux, or *b.dll* or *b.pyd* on Cygwin and Windows)
- A compiled built-in module coded in C and statically linked into Python
- A ZIP file component that is automatically extracted when imported
- An in-memory image, for frozen executables

§ Some programs really need to change `sys.path`, though. Scripts that run on web servers, for example, often run as the user "nobody" to limit machine access. Because such scripts cannot usually depend on "nobody" to have set `PYTHONPATH` in any particular way, they often set `sys.path` manually to include required source directories, prior to running any `import` statements. A `sys.path.append(dirname)` will often suffice.

- A Java class, in the Jython version of Python
- A .NET component, in the IronPython version of Python

C extensions, Jython, and package imports all extend imports beyond simple files. To importers, though, differences in the loaded file type are completely transparent, both when importing and when fetching module attributes. Saying `import b` gets whatever module `b` is, according to your module search path, and `b.attr` fetches an item in the module, be it a Python variable or a linked-in C function. Some standard modules we will use in this book are actually coded in C, not Python; because of this transparency, their clients don't have to care.

If you have both a *b.py* and a *b.so* in different directories, Python will always load the one found in the first (leftmost) directory of your module search path during the left-to-right search of `sys.path`. But what happens if it finds both a *b.py* and a *b.so* in the *same* directory? In this case, Python follows a standard picking order, though this order is not guaranteed to stay the same over time. In general, you should not depend on which type of file Python will choose within a given directory—make your module names distinct, or configure your module search path to make your module selection preferences more obvious.

Advanced Module Selection Concepts

Normally, imports work as described in this section—they find and load files on your machine. However, it is possible to redefine much of what an import operation does in Python, using what are known as *import hooks*. These hooks can be used to make imports do various useful things, such as loading files from archives, performing decryption, and so on.

In fact, Python itself makes use of these hooks to enable files to be directly imported from ZIP archives: archived files are automatically extracted at import time when a *.zip* file is selected from the module import search path. One of the standard library directories in the earlier `sys.path` display, for example, is a *.zip* file today. For more details, see the Python standard library manual's description of the built-in `__import__` function, the customizable tool that `import` statements actually run.

Python also supports the notion of *.pyo* optimized byte code files, created and run with the `-O` Python command-line flag; because these run only slightly faster than normal *.pyc* files (typically 5 percent faster), however, they are infrequently used. The Psyco system (see Chapter 2) provides more substantial speedups.

Third-Party Software: distutils

This chapter's description of module search path settings is targeted mainly at user-defined source code that you write on your own. Third-party extensions for Python typically use the `distutils` tools in the standard library to automatically install themselves, so no path configuration is required to use their code.

Systems that use `distutils` generally come with a *setup.py* script, which is run to install them; this script imports and uses `distutils` modules to place such systems in a directory that is automatically part of the module search path (usually in the *Lib\site-packages* subdirectory of the Python install tree, wherever that resides on the target machine).

For more details on distributing and installing with `distutils`, see the Python standard manual set; its use is beyond the scope of this book (for instance, it also provides ways to automatically compile C-coded extensions on the target machine). Also check out the emerging third-party open source *eggs* system, which adds dependency checking for installed Python software.

Chapter Summary

In this chapter, we covered the basics of modules, attributes, and imports and explored the operation of `import` statements. We learned that imports find the designated file on the module search path, compile it to byte code, and execute all of its statements to generate its contents. We also learned how to configure the search path to be able to import from directories other than the home directory and the standard library directories, primarily with `PYTHONPATH` settings.

As this chapter demonstrated, the import operation and modules are at the heart of program architecture in Python. Larger programs are divided into multiple files, which are linked together at runtime by imports. Imports in turn use the module search path to locate files, and modules define attributes for external use.

Of course, the whole point of imports and modules is to provide a structure to your program, which divides its logic into self-contained software components. Code in one module is isolated from code in another; in fact, no file can ever see the names defined in another, unless explicit `import` statements are run. Because of this, modules minimize name collisions between different parts of your program.

You'll see what this all means in terms of actual statements and code in the next chapter. Before we move on, though, let's run through the chapter quiz.

Test Your Knowledge: Quiz

1. How does a module source code file become a module object?
2. Why might you have to set your `PYTHONPATH` environment variable?
3. Name the four major components of the module import search path.
4. Name four file types that Python might load in response to an import operation.
5. What is a namespace, and what does a module's namespace contain?

Test Your Knowledge: Answers

1. A module's source code file automatically becomes a module object when that module is imported. Technically, the module's source code is run during the import, one statement at a time, and all the names assigned in the process become attributes of the module object.
2. You only need to set `PYTHONPATH` to import from directories other than the one in which you are working (i.e., the current directory when working interactively, or the directory containing your top-level file).
3. The four major components of the module import search path are the top-level script's home directory (the directory containing it), all directories listed in the `PYTHONPATH` environment variable, the standard library directories, and all directories listed in *.pth* path files located in standard places. Of these, programmers can customize `PYTHONPATH` and *.pth* files.
4. Python might load a source code (*.py*) file, a byte code (*.pyc*) file, a C extension module (e.g., a *.so* file on Linux or a *.dll* or *.pyd* file on Windows), or a directory of the same name for package imports. Imports may also load more exotic things such as ZIP file components, Java classes under the Jython version of Python, .NET components under IronPython, and statically linked C extensions that have no files present at all. With import hooks, imports can load anything.
5. A namespace is a self-contained package of variables, which are known as the *attributes* of the namespace object. A module's namespace contains all the names assigned by code at the top level of the module file (i.e., not nested in `def` or `class` statements). Technically, a module's global scope morphs into the module object's attributes namespace. A module's namespace may also be altered by assignments from other files that import it, though this is frowned upon (see Chapter 17 for more on this issue).

CHAPTER 22
Module Coding Basics

Now that we've looked at the larger ideas behind modules, let's turn to a simple example of modules in action. Python modules are easy to *create*; they're just files of Python program code created with a text editor. You don't need to write special syntax to tell Python you're making a module; almost any text file will do. Because Python handles all the details of finding and loading modules, modules are also easy to *use*; clients simply import a module, or specific names a module defines, and use the objects they reference.

Module Creation

To define a module, simply use your text editor to type some Python code into a text file, and save it with a ".py" extension; any such file is automatically considered a Python module. All the names assigned at the top level of the module become its *attributes* (names associated with the module object) and are exported for clients to use.

For instance, if you type the following `def` into a file called *module1.py* and import it, you create a module object with one attribute—the name `printer`, which happens to be a reference to a function object:

```
def printer(x):                    # Module attribute
    print(x)
```

Before we go on, I should say a few more words about module filenames. You can call modules just about anything you like, but module filenames should end in a *.py* suffix if you plan to import them. The *.py* is technically optional for top-level files that will be run but not imported, but adding it in all cases makes your files' types more obvious and allows you to import any of your files in the future.

Because module names become variable names inside a Python program (without the *.py*), they should also follow the normal variable name rules outlined in Chapter 11. For instance, you can create a module file named *if.py*, but you cannot import it because `if` is a reserved word—when you try to run `import if`, you'll get a syntax error. In fact, both the names of module files and the names of directories used in

package imports (discussed in the next chapter) must conform to the rules for variable names presented in Chapter 11; they may, for instance, contain only letters, digits, and underscores. Package directories also cannot contain platform-specific syntax such as spaces in their names.

When a module is imported, Python maps the internal module name to an external filename by adding a directory path from the module search path to the front, and a *.py* or other extension at the end. For instance, a module named `M` ultimately maps to some external file *<directory>\M.<extension>* that contains the module's code.

As mentioned in the preceding chapter, it is also possible to create a Python module by writing code in an external language such as C or C++ (or Java, in the Jython implementation of the language). Such modules are called *extension modules*, and they are generally used to wrap up external libraries for use in Python scripts. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with `import` statements, and they provide functions and objects as module attributes. Extension modules are beyond the scope of this book; see Python's standard manuals or advanced texts such as *Programming Python* for more details.

Module Usage

Clients can use the simple module file we just wrote by running an `import` or `from` statement. Both statements find, compile, and run a module file's code, if it hasn't yet been loaded. The chief difference is that `import` fetches the module as a whole, so you must qualify to fetch its names; in contrast, `from` fetches (or copies) specific names out of the module.

Let's see what this means in terms of code. All of the following examples wind up calling the `printer` function defined in the prior section's *module1.py* module file, but in different ways.

The import Statement

In the first example, the name `module1` serves two different purposes—it identifies an external file to be loaded, and it becomes a variable in the script, which references the module object after the file is loaded:

```
>>> import module1                      # Get module as a whole
>>> module1.printer('Hello world!')     # Qualify to get names
Hello world!
```

Because `import` gives a name that refers to the whole module object, we must go through the module name to fetch its attributes (e.g., `module1.printer`).

The from Statement

By contrast, because `from` also copies names from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., `printer`):

```
>>> from module1 import printer          # Copy out one variable
>>> printer('Hello world!')              # No need to qualify name
Hello world!
```

This has the same effect as the prior example, but because the imported name is copied into the scope where the `from` statement appears, using that name in the script requires less typing: we can use it directly instead of naming the enclosing module.

As you'll see in more detail later, the `from` statement is really just a minor extension to the `import` statement—it imports the module file as usual, but adds an extra step that copies one or more names out of the file.

The from * Statement

Finally, the next example uses a special form of `from`: when we use a `*`, we get copies of *all* the names assigned at the top level of the referenced module. Here again, we can then use the copied name `printer` in our script without going through the module name:

```
>>> from module1 import *                # Copy out all variables
>>> printer('Hello world!')
Hello world!
```

Technically, both `import` and `from` statements invoke the same import operation; the `from *` form simply adds an extra step that copies all the names in the module into the importing scope. It essentially collapses one module's namespace into another; again, the net effect is less typing for us.

And that's it—modules really are simple to use. To give you a better understanding of what really happens when you define and use modules, though, let's move on to look at some of their properties in more detail.

In Python 3.0, the `from ...*` statement form described here can be used *only* at the top level of a module file, not within a function. Python 2.6 allows it to be used within a function, but issues a warning. It's extremely rare to see this statement used inside a function in practice; when present, it makes it impossible for Python to detect variables statically, before the function runs.

Imports Happen Only Once

One of the most common questions people seem to ask when they start using modules is, "Why won't my imports keep working?" They often report that the first import works fine, but later imports during an interactive session (or program run) seem to have no effect. In fact, they're not supposed to. This section explains why.

Modules are loaded and run on the first `import` or `from`, and only the first. This is on purpose—because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

As one consequence, because top-level code in a module file is usually executed only once, you can use it to initialize variables. Consider the file *simple.py*, for example:

```
print('hello')
spam = 1                    # Initialize variable
```

In this example, the `print` and `=` statements run the first time the module is imported, and the variable `spam` is initialized at import time:

```
% python
>>> import simple           # First import: loads and runs file's code
hello
>>> simple.spam             # Assignment makes an attribute
1
```

Second and later imports don't rerun the module's code; they just fetch the already created module object from Python's internal modules table. Thus, the variable `spam` is not reinitialized:

```
>>> simple.spam = 2         # Change attribute in module
>>> import simple           # Just fetches already loaded module
>>> simple.spam             # Code wasn't rerun: attribute unchanged
2
```

Of course, sometimes you really want a module's code to be rerun on a subsequent import. We'll see how to do this with Python's `reload` function later in this chapter.

import and from Are Assignments

Just like `def`, `import` and `from` are executable statements, not compile-time declarations. They may be nested in `if` tests, appear in function `def`s, and so on, and they are not resolved or run until Python reaches them while executing your program. In other words, imported modules and names are not available until their associated `import` or `from` statements run. Also, like `def`, `import` and `from` are implicit assignments:

- `import` assigns an entire module object to a single name.
- `from` assigns one or more names to objects of the same names in another module.

All the things we've already discussed about assignment apply to module access, too. For instance, names copied with a `from` become references to shared objects; as with function arguments, reassigning a fetched name has no effect on the module from which it was copied, but changing a fetched *mutable object* can change it in the module from which it was imported. To illustrate, consider the following file, *small.py*:

```
x = 1
y = [1, 2]
```

```
% python
>>> from small import x, y          # Copy two names out
>>> x = 42                          # Changes local x only
>>> y[0] = 42                       # Changes shared mutable in-place
```

Here, `x` is not a shared mutable object, but `y` is. The name `y` in the importer and the importee reference the same list object, so changing it from one place changes it in the other:

```
>>> import small                    # Get module name (from doesn't)
>>> small.x                         # Small's x is not my x
1
>>> small.y                         # But we share a changed mutable
[42, 2]
```

For a graphical picture of what `from` assignments do with references, flip back to Figure 18-1 (function argument passing), and mentally replace "caller" and "function" with "imported" and "importer." The effect is the same, except that here we're dealing with names in modules, not functions. Assignment works the same everywhere in Python.

Cross-File Name Changes

Recall from the preceding example that the assignment to `x` in the interactive session changed the name `x` in that scope only, not the `x` in the file—there is no link from a name copied with `from` back to the file it came from. To really change a global name in another file, you must use `import`:

```
% python
>>> from small import x, y          # Copy two names out
>>> x = 42                          # Changes my x only

>>> import small                    # Get module name
>>> small.x = 42                    # Changes x in other module
```

This phenomenon was introduced in Chapter 17. Because changing variables in other modules like this is a common source of confusion (and often a bad design choice), we'll revisit this technique again later in this part of the book. Note that the change to `y[0]` in the prior session is different; it changes an object, not a name.

import and from Equivalence

Notice in the prior example that we have to execute an `import` statement after the `from` to access the `small` module name at all. `from` only copies names from one module to another; it does not assign the module name itself. At least conceptually, a `from` statement like this one:

```
from module import name1, name2      # Copy these two names out (only)
```

is equivalent to this statement sequence:

```
import module                        # Fetch the module object
name1 = module.name1                 # Copy names out by assignment
name2 = module.name2
del module                           # Get rid of the module name
```

Like all assignments, the `from` statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the names are copied out, though, not the module itself. When we use the `from *` form of this statement (`from module import *`), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Notice that the first step of the `from` runs a normal `import` operation. Because of this, the `from` always imports the entire module into memory if it has not yet been imported, regardless of how many names it copies out of the file. There is no way to load just part of a module file (e.g., just one function), but because modules are byte code in Python instead of machine code, the performance implications are generally negligible.

Potential Pitfalls of the from Statement

Because the `from` statement makes the location of a variable more implicit and obscure (`name` is less meaningful to the reader than `module.name`), some Python users recommend using `import` instead of `from` most of the time. I'm not sure this advice is warranted, though; `from` is commonly and widely used, without too many dire consequences. In practice, in realistic programs, it's often convenient not to have to type a module's name every time you wish to use one of its tools. This is especially true for large modules that provide many attributes—the standard library's `tkinter` GUI module, for example.

It is true that the `from` statement has the potential to corrupt namespaces, at least in principle—if you use it to import variables that happen to have the same names as existing variables in your scope, your variables will be silently overwritten. This problem doesn't occur with the simple `import` statement because you must always go through a module's name to get to its contents (`module.attr` will not clash with a variable named `attr` in your scope). As long as you understand and expect that this can happen when using `from`, though, this isn't a major concern in practice, especially if you list the imported names explicitly (e.g., `from module import x, y, z`).

On the other hand, the `from` statement has more serious issues when used in conjunction with the `reload` call, as imported names might reference prior versions of objects.

Moreover, the `from module import *` form really can corrupt namespaces and make names difficult to understand, especially when applied to more than one file—in this case, there is no way to tell which module a name came from, short of searching the external source files. In effect, the `from *` form collapses one namespace into another, and so defeats the namespace partitioning feature of modules. We will explore these issues in more detail in the section "Module Gotchas" on page 599 at the end of this part of the book (see Chapter 24).

Probably the best real-world advice here is to generally prefer `import` to `from` for simple modules, to explicitly list the variables you want in most `from` statements, and to limit the `from *` form to just one import per file. That way, any undefined names can be assumed to live in the module referenced with the `from *`. Some care is required when using the `from` statement, but armed with a little knowledge, most programmers find it to be a convenient way to access modules.

When import is required

The only time you really must use `import` instead of `from` is when you must use the same name defined in two different modules. For example, if two files define the same name differently:

```
# M.py

def func():
    ...do something...

# N.py

def func():
    ...do something else...
```

and you must use both versions of the name in your program, the `from` statement will fail—you can only have one assignment to the name in your scope:

```
# O.py

from M import func
from N import func          # This overwites the one we got from M
func()                      # Calls N.func only
```

An `import` will work here, though, because including the name of the enclosing module makes the two names unique:

```
# O.py

import M, N                 # Get the whole modules, not their names
M.func()                    # We can call both names now
N.func()                    # The module names make them unique
```

This case is unusual enough that you're unlikely to encounter it very often in practice. If you do, though, `import` allows you to avoid the name collision.

Module Namespaces

Modules are probably best understood as simply packages of names—i.e., places to define names you want to make visible to the rest of a system. Technically, modules usually correspond to files, and Python creates a module object to contain all the names assigned in a module file. But in simple terms, modules are just namespaces (places where names are created), and the names that live in a module are called its *attributes*. We'll explore how all this works in this section.

Files Generate Namespaces

So, how do files morph into namespaces? The short story is that every name that is assigned a value at the top level of a module file (i.e., not nested in a function or class body) becomes an attribute of that module.

For instance, given an assignment statement such as `X = 1` at the top level of a module file *M.py*, the name `X` becomes an attribute of `M`, which we can refer to from outside the module as `M.X`. The name `X` also becomes a global variable to other code inside *M.py*, but we need to explain the notion of module loading and scopes a bit more formally to understand why:

- **Module statements run on the first import**. The first time a module is imported anywhere in a system, Python creates an empty module object and executes the statements in the module file one after another, from the top of the file to the bottom.
- **Top-level assignments create module attributes**. During an import, statements at the top level of the file not nested in a `def` or `class` that assign names (e.g., `=`, `def`) create attributes of the module object; assigned names are stored in the module's namespace.
- **Module namespaces can be accessed via the attribute `__dict__` or `dir(M)`**. Module namespaces created by imports are dictionaries; they may be accessed through the built-in `__dict__` attribute associated with module objects and may be inspected with the `dir` function. The `dir` function is roughly equivalent to the sorted keys list of an object's `__dict__` attribute, but it includes inherited names for classes, may not be complete, and is prone to changing from release to release.
- **Modules are a single scope (local is global)**. As we saw in Chapter 17, names at the top level of a module follow the same reference/assignment rules as names in a function, but the local and global scopes are the same (more formally, they follow the LEGB scope rule we met in Chapter 17, but without the L and E lookup layers). But, in modules, the module *scope* becomes an attribute dictionary of a module *object* after the module has been loaded. Unlike with functions (where the local namespace exists only while the function runs), a module file's scope becomes a module object's attribute namespace and lives on after the import.

Here's a demonstration of these ideas. Suppose we create the following module file in a text editor and call it *module2.py*:

```
print('starting to load...')
import sys
name = 42

def func(): pass

class klass: pass

print('done loading.')
```

The first time this module is imported (or run as a program), Python executes its statements from top to bottom. Some statements create names in the module's namespace as a side effect, but others do actual work while the import is going on. For instance, the two `print` statements in this file execute at import time:

```
>>> import module2
starting to load...
done loading.
```

Once the module is loaded, its scope becomes an attribute namespace in the module object we get back from `import`. We can then access attributes in this namespace by qualifying them with the name of the enclosing module:

```
>>> module2.sys
<module 'sys' (built-in)>

>>> module2.name
42

>>> module2.func
<function func at 0xO26D3BB8>

>>> module2.klass
<class 'module2.klass'>
```

Here, `sys`, `name`, `func`, and `klass` were all assigned while the module's statements were being run, so they are attributes after the import. We'll talk about classes in Part VI, but notice the `sys` attribute—`import` statements really *assign* module objects to names, and any type of assignment to a name at the top level of a file generates a module attribute.

Internally, module namespaces are stored as dictionary objects. These are just normal dictionary objects with the usual methods. We can access a module's namespace dictionary through the module's `__dict__` attribute (remember to wrap this in a `list` call in Python 3.0—it's a view object):

```
>>> list(module2.__dict__.keys())
['name', '__builtins__', '__file__', '__package__', 'sys', 'klass', 'func',
'__name__', '__doc__']
```

The names we assigned in the module file become dictionary keys internally, so most of the names here reflect top-level assignments in our file. However, Python also adds some names in the module's namespace for us; for instance, `__file__` gives the name of the file the module was loaded from, and `__name__` gives its name as known to importers (without the *.py* extension and directory path).

Attribute Name Qualification

Now that you're becoming more familiar with modules, we should look at the notion of name *qualification* (fetching attributes) in more depth. In Python, you can access the attributes of any object that has attributes using the qualification syntax *`object.attribute`*.

Qualification is really an expression that returns the value assigned to an attribute name associated with an object. For example, the expression `module2.sys` in the previous example fetches the value assigned to `sys` in `module2`. Similarly, if we have a built-in list object `L`, `L.append` returns the `append` method object associated with that list.

So, what does attribute qualification do to the scope rules we studied in Chapter 17? Nothing, really: it's an independent concept. When you use qualification to access names, you give Python an explicit object from which to fetch the specified names. The LEGB rule applies only to bare, unqualified names. Here are the rules:

Simple variables
: `X` means search for the name `X` in the current scopes (following the LEGB rule).

Qualification
: `X.Y` means find `X` in the current scopes, then search for the attribute `Y` in the object `X` (not in scopes).

Qualification paths
: `X.Y.Z` means look up the name `Y` in the object `X`, then look up `Z` in the object `X.Y`.

Generality
: Qualification works on all objects with attributes: modules, classes, C extension types, etc.

In Part VI, we'll see that qualification means a bit more for classes (it's also the place where something called *inheritance* happens), but in general, the rules outlined here apply to all names in Python.

Imports Versus Scopes

As we've learned, it is never possible to access names defined in another module file without first importing that file. That is, you never automatically get to see names in another file, regardless of the structure of imports or function calls in your program. A variable's meaning is always determined by the locations of assignments in your source code, and attributes are always requested of an object explicitly.

For example, consider the following two simple modules. The first, *moda.py*, defines a variable `X` global to code in its file only, along with a function that changes the global `X` in this file:

```
X = 88                          # My X: global to this file only
def f():
    global X                    # Change this file's X
    X = 99                      # Cannot see names in other modules
```

The second module, *modb.py*, defines its own global variable `X` and imports and calls the function in the first module:

```
X = 11                          # My X: global to this file only

import moda                     # Gain access to names in moda
moda.f()                        # Sets moda.X, not this file's X
print(X, moda.X)
```

When run, `moda.f` changes the `X` in `moda`, not the `X` in `modb`. The global scope for `moda.f` is always the file enclosing it, regardless of which module it is ultimately called from:

```
% python modb.py
11 99
```

In other words, import operations never give upward visibility to code in imported files—an imported file cannot see names in the importing file. More formally:

- Functions can never see names in other functions, unless they are physically enclosing.
- Module code can never see names in other modules, unless they are explicitly imported.

Such behavior is part of the *lexical scoping* notion—in Python, the scopes surrounding a piece of code are completely determined by the code's physical position in your file. Scopes are never influenced by function calls or module imports.*

Namespace Nesting

In some sense, although imports do not nest namespaces upward, they do nest downward. Using attribute qualification paths, it's possible to descend into arbitrarily nested modules and access their attributes. For example, consider the next three files. *mod3.py* defines a single global name and attribute by assignment:

```
X = 3
```

mod2.py in turn defines its own `X`, then imports `mod3` and uses qualification to access the imported module's attribute:

* Some languages act differently and provide for *dynamic scoping*, where scopes really may depend on runtime calls. This tends to make code trickier, though, because the meaning of a variable can differ over time.

```
X = 2
import mod3

print(X, end=' ')              # My global X
print(mod3.X)                  # mod3's X
```

mod1.py also defines its own `X`, then imports `mod2`, and fetches attributes in both the first and second files:

```
X = 1
import mod2

print(X, end=' ')              # My global X
print(mod2.X, end=' ')         # mod2's X
print(mod2.mod3.X)             # Nested mod3's X
```

Really, when `mod1` imports `mod2` here, it sets up a two-level namespace nesting. By using the path of names `mod2.mod3.X`, it can descend into `mod3`, which is nested in the imported `mod2`. The net effect is that `mod1` can see the `X`s in all three files, and hence has access to all three global scopes:

```
% python mod1.py
2 3
1 2 3
```

The reverse, however, is not true: `mod3` cannot see names in `mod2`, and `mod2` cannot see names in `mod1`. This example may be easier to grasp if you don't think in terms of namespaces and scopes, but instead focus on the objects involved. Within `mod1`, `mod2` is just a name that refers to an object with attributes, some of which may refer to other objects with attributes (`import` is an assignment). For paths like `mod2.mod3.X`, Python simply evaluates from left to right, fetching attributes from objects along the way.

Note that `mod1` can say `import mod2`, and then `mod2.mod3.X`, but it cannot say `import mod2.mod3`—this syntax invokes something called package (directory) imports, described in the next chapter. Package imports also create module namespace nesting, but their `import` statements are taken to reflect directory trees, not simple import chains.

Reloading Modules

As we've seen, a module's code is run only once per process by default. To force a module's code to be reloaded and rerun, you need to ask Python to do so explicitly by calling the `reload` built-in function. In this section, we'll explore how to use reloads to make your systems more dynamic. In a nutshell:

- Imports (via both `import` and `from` statements) load and run a module's code only the first time the module is imported in a process.
- Later imports use the already loaded module object without reloading or rerunning the file's code.

- The `reload` function forces an already loaded module's code to be reloaded and rerun. Assignments in the file's new code change the existing module object in-place.

Why all the fuss about reloading modules? The `reload` function allows parts of a program to be changed without stopping the whole program. With `reload`, therefore, the effects of changes in components can be observed immediately. Reloading doesn't help in every situation, but where it does, it makes for a much shorter development cycle. For instance, imagine a database program that must connect to a server on startup; because program changes or customizations can be tested immediately after reloads, you need to connect only once while debugging. Long-running servers can update themselves this way, too.

Because Python is interpreted (more or less), it already gets rid of the compile/link steps you need to go through to get a C program to run: modules are loaded dynamically when imported by a running program. Reloading offers a further performance advantage by allowing you to also change parts of running programs without stopping. Note that `reload` currently only works on modules written in Python; compiled extension modules coded in a language such as C can be dynamically loaded at runtime, too, but they can't be reloaded.

Version skew note: In Python 2.6, `reload` is available as a built-in function. In Python 3.0, it has been moved to the `imp` standard library module—it's known as `imp.reload` in 3.0. This simply means that an extra `import` or `from` statement is required to load this tool (in 3.0 only). Readers using 2.6 can ignore these imports in this book's examples, or use them anyhow—2.6 also has a `reload` in its `imp` module to ease migration to 3.0. Reloading works the same regardless of its packaging.

reload Basics

Unlike `import` and `from`:

- `reload` is a function in Python, not a statement.
- `reload` is passed an existing module object, not a name.
- `reload` lives in a module in Python 3.0 and must be imported itself.

Because `reload` expects an object, a module must have been previously imported successfully before you can reload it (if the import was unsuccessful, due to a syntax or other error, you may need to repeat it before you can reload the module). Furthermore, the syntax of `import` statements and `reload` calls differs: reloads require parentheses, but imports do not. Reloading looks like this:

```
import module                   # Initial import
...use module.attributes...
...                             # Now, go change the module file
...
```

```
from imp import reload              # Get reload itself (in 3.0)
reload(module)                      # Get updated exports
...use module.attributes...
```

The typical usage pattern is that you import a module, then change its source code in a text editor, and then reload it. When you call `reload`, Python rereads the module file's source code and reruns its top-level statements. Perhaps the most important thing to know about `reload` is that it changes a module object *in-place*; it does not delete and re-create the module object. Because of that, every reference to a module object anywhere in your program is automatically affected by a reload. Here are the details:

- **`reload` runs a module file's new code in the module's current namespace**. Rerunning a module file's code overwrites its existing namespace, rather than deleting and re-creating it.
- **Top-level assignments in the file replace names with new values**. For instance, rerunning a `def` statement replaces the prior version of the function in the module's namespace by reassigning the function name.
- **Reloads impact all clients that use `import` to fetch modules**. Because clients that use `import` qualify to fetch attributes, they'll find new values in the module object after a reload.
- **Reloads impact future `from` clients only**. Clients that used `from` to fetch attributes in the past won't be affected by a reload; they'll still have references to the old objects fetched before the reload.

reload Example

To demonstrate, here's a more concrete example of `reload` in action. In the following, we'll change and reload a module file without stopping the interactive Python session. Reloads are used in many other scenarios, too (see the sidebar "Why You Will Care: Module Reloads" on page 557), but we'll keep things simple for illustration here. First, in the text editor of your choice, write a module file named *changer.py* with the following contents:

```
message = "First version"
def printer():
    print(message)
```

This module creates and exports two names—one bound to a string, and another to a function. Now, start the Python interpreter, import the module, and call the function it exports. The function will print the value of the global `message` variable:

```
% python
>>> import changer
>>> changer.printer()
First version
```

Keeping the interpreter active, now edit the module file in another window:

```
...modify changer.py without stopping Python...
% vi changer.py
```

Change the global `message` variable, as well as the `printer` function body:

```
message = "After editing"
def printer():
    print('reloaded:', message)
```

Then, return to the Python window and reload the module to fetch the new code. Notice in the following interaction that importing the module again has no effect; we get the original message, even though the file's been changed. We have to call `reload` in order to get the new version:

```
...back to the Python interpreter/program...

>>> import changer
>>> changer.printer()                  # No effect: uses loaded module
First version
>>> from imp import reload
>>> reload(changer)                    # Forces new code to load/run
<module 'changer' from 'changer.py'>
>>> changer.printer()                  # Runs the new version now
reloaded: After editing
```

Notice that `reload` actually *returns* the module object for us—its result is usually ignored, but because expression results are printed at the interactive prompt, Python shows a default `<module 'name'...>` representation.

Why You Will Care: Module Reloads

Besides allowing you to reload (and hence rerun) modules at the interactive prompt, module reloads are also useful in larger systems, especially when the cost of restarting the entire application is prohibitive. For instance, systems that must connect to servers over a network on startup are prime candidates for dynamic reloads.

They're also useful in GUI work (a widget's callback action can be changed while the GUI remains active), and when Python is used as an embedded language in a C or C++ program (the enclosing program can request a reload of the Python code it runs, without having to stop). See *Programming Python* for more on reloading GUI callbacks and embedded Python code.

More generally, reloads allow programs to provide highly dynamic interfaces. For instance, Python is often used as a *customization* language for larger systems—users can customize products by coding bits of Python code onsite, without having to recompile the entire product (or even having its source code at all). In such worlds, the Python code already adds a dynamic flavor by itself.

To be even more dynamic, though, such systems can automatically reload the Python customization code periodically at runtime. That way, users' changes are picked up while the system is running; there is no need to stop and restart each time the Python code is modified. Not all systems require such a dynamic approach, but for those that do, module reloads provide an easy-to-use dynamic customization tool.

Chapter Summary

This chapter delved into the basics of module coding tools—the `import` and `from` statements, and the `reload` call. We learned how the `from` statement simply adds an extra step that copies names out of a file after it has been imported, and how `reload` forces a file to be imported again without stopping and restarting Python. We also surveyed namespace concepts, saw what happens when imports are nested, explored the way files become module namespaces, and learned about some potential pitfalls of the `from` statement.

Although we've already seen enough to handle module files in our programs, the next chapter extends our coverage of the import model by presenting *package imports*—a way for our `import` statements to specify part of the directory path leading to the desired module. As we'll see, package imports give us a hierarchy that is useful in larger systems and allow us to break conflicts between same-named modules. Before we move on, though, here's a quick quiz on the concepts presented here.

Test Your Knowledge: Quiz

1. How do you make a module?
2. How is the `from` statement related to the `import` statement?
3. How is the `reload` function related to imports?
4. When must you use `import` instead of `from`?
5. Name three potential pitfalls of the `from` statement.
6. What...is the airspeed velocity of an unladen swallow?

Test Your Knowledge: Answers

1. To create a module, you just write a text file containing Python statements; every source code file is automatically a module, and there is no syntax for declaring one. Import operations load module files into module objects in memory. You can also make a module by writing code in an external language like C or Java, but such extension modules are beyond the scope of this book.

2. The `from` statement imports an entire module, like the `import` statement, but as an extra step it also copies one or more variables from the imported module into the scope where the `from` appears. This enables you to use the imported names directly (`name`) instead of having to go through the module (`module.name`).
3. By default, a module is imported only once per process. The `reload` function forces a module to be imported again. It is mostly used to pick up new versions of a module's source code during development, and in dynamic customization scenarios.
4. You must use `import` instead of `from` only when you need to access the same name in two different modules; because you'll have to specify the names of the enclosing modules, the two names will be unique.
5. The `from` statement can obscure the meaning of a variable (which module it is defined in), can have problems with the `reload` call (names may reference prior versions of objects), and can corrupt namespaces (it might silently overwrite names you are using in your scope). The `from *` form is worse in most regards—it can seriously corrupt namespaces and obscure the meaning of variables, so it is probably best used sparingly.
6. What do you mean? An African or European swallow?

CHAPTER 23

Module Packages

So far, when we've imported modules, we've been loading files. This represents typical module usage, and it's probably the technique you'll use for most imports you'll code early on in your Python career. However, the module import story is a bit richer than I have thus far implied.

In addition to a module name, an import can name a directory path. A directory of Python code is said to be a *package*, so such imports are known as *package imports*. In effect, a package import turns a directory on your computer into another Python namespace, with attributes corresponding to the subdirectories and module files that the directory contains.

This is a somewhat advanced feature, but the hierarchy it provides turns out to be handy for organizing the files in a large system and tends to simplify module search path settings. As we'll see, package imports are also sometimes required to resolve import ambiguities when multiple program files of the same name are installed on a single machine.

Because it is relevant to code in packages only, we'll also introduce Python's recent *relative imports* model and syntax here. As we'll see, this model modifies search paths and extends the `from` statement for imports within packages.

Package Import Basics

So, how do package imports work? In the place where you have been naming a simple file in your `import` statements, you can instead list a path of names separated by periods:

```
import dir1.dir2.mod
```

The same goes for `from` statements:

```
from dir1.dir2.mod import x
```

The "dotted" path in these statements is assumed to correspond to a path through the directory hierarchy on your machine, leading to the file *mod.py* (or similar; the extension may vary). That is, the preceding statements indicate that on your machine there is a directory *dir1*, which has a subdirectory *dir2*, which contains a module file *mod.py* (or similar).

Furthermore, these imports imply that *dir1* resides within some container directory *dir0*, which is a component of the Python module search path. In other words, the two `import` statements imply a directory structure that looks something like this (shown with DOS backslash separators):

```
dir0\dir1\dir2\mod.py              # Or mod.pyc, mod.so, etc.
```

The container directory *dir0* needs to be added to your module search path (unless it's the home directory of the top-level file), exactly as if *dir1* were a simple module file.

More generally, the leftmost component in a package import path is still relative to a directory included in the `sys.path` module search path list we met in Chapter 21. From there down, though, the `import` statements in your script give the directory paths leading to the modules explicitly.

Packages and Search Path Settings

If you use this feature, keep in mind that the directory paths in your `import` statements can only be variables separated by periods. You cannot use any platform-specific path syntax in your `import` statements, such as `C:\dir1`, `My Documents.dir2` or `../dir1`—these do not work syntactically. Instead, use platform-specific syntax in your module search path settings to name the container directories.

For instance, in the prior example, *dir0*—the directory name you add to your module search path—can be an arbitrarily long and platform-specific directory path leading up to *dir1*. Instead of using an invalid statement like this:

```
import C:\mycode\dir1\dir2\mod      # Error: illegal syntax
```

add *C:\mycode* to your `PYTHONPATH` variable or a *.pth* file (assuming it is not the program's home directory, in which case this step is not necessary), and say this in your script:

```
import dir1.dir2.mod
```

In effect, entries on the module search path provide platform-specific directory path prefixes, which lead to the leftmost names in `import` statements. `import` statements provide directory path tails in a platform-neutral fashion.*

* The dot path syntax was chosen partly for platform neutrality, but also because paths in `import` statements become real nested object paths. This syntax also means that you get odd error messages if you forget to omit the *.py* in your `import` statements. For example, `import mod.py` is assumed to be a directory path import—it loads *mod.py*, then tries to load a *mod\py.py*, and ultimately issues a potentially confusing "No module named py" error message.

Package __init__.py Files

If you choose to use package imports, there is one more constraint you must follow: each directory named within the path of a package import statement must contain a file named *__init__.py*, or your package imports will fail. That is, in the example we've been using, both *dir1* and *dir2* must contain a file called *__init__.py*; the container directory *dir0* does not require such a file because it's not listed in the `import` statement itself. More formally, for a directory structure such as this:

```
dir0\dir1\dir2\mod.py
```

and an `import` statement of the form:

```
import dir1.dir2.mod
```

the following rules apply:

- *dir1* and *dir2* both must contain an *__init__.py* file.
- *dir0*, the container, does not require an *__init__.py* file; this file will simply be ignored if present.
- *dir0*, not *dir0\dir1*, must be listed on the module search path (i.e., it must be the home directory, or be listed in your `PYTHONPATH`, etc.).

The net effect is that this example's directory structure should be as follows, with indentation designating directory nesting:

```
dir0\                              # Container on module search path
    dir1\
        __init__.py
        dir2\
            __init__.py
            mod.py
```

The *__init__.py* files can contain Python code, just like normal module files. They are partly present as a declaration to Python, however, and can be completely empty. As declarations, these files serve to prevent directories with common names from unintentionally hiding true modules that appear later on the module search path. Without this safeguard, Python might pick a directory that has nothing to do with your code, just because it appears in an earlier directory on the search path.

More generally, the *__init__.py* file serves as a hook for package-initialization-time actions, generates a module namespace for a directory, and implements the behavior of `from *` (i.e., `from .. import *`) statements when used with directory imports:

Package initialization
: The first time Python imports through a directory, it automatically runs all the code in the directory's *__init__.py* file. Because of that, these files are a natural place to put code to initialize the state required by files in a package. For instance, a package might use its initialization file to create required data files, open connections to

databases, and so on. Typically, *__init__.py* files are not meant to be useful if executed directly; they are run automatically when a package is first accessed.

Module namespace initialization
: In the package import model, the directory paths in your script become real nested object paths after an import. For instance, in the preceding example, after the import the expression `dir1.dir2` works and returns a module object whose namespace contains all the names assigned by *dir2*'s *__init__.py* file. Such files provide a namespace for module objects created for directories, which have no real associated module files.

`from *` *statement behavior*
: As an advanced feature, you can use `__all__` lists in *__init__.py* files to define what is exported when a directory is imported with the `from *` statement form. In an *__init__.py* file, the `__all__` list is taken to be the list of submodule names that should be imported when `from *` is used on the package (directory) name. If `__all__` is not set, the `from *` statement does not automatically load submodules nested in the directory; instead, it loads just names defined by assignments in the directory's *__init__.py* file, including any submodules explicitly imported by code in this file. For instance, the statement `from submodule import X` in a directory's *__init__.py* makes the name `X` available in that directory's namespace. (We'll see additional roles for `__all__` in Chapter 24.)

You can also simply leave these files empty, if their roles are beyond your needs (and frankly, they are often empty in practice). They must exist, though, for your directory imports to work at all.

Don't confuse package *__init__.py* files with the class `__init__` constructor methods we'll meet in the next part of the book. The former are files of code run when `import`s first step through a package directory, while the latter are called when an instance is created. Both have initialization roles, but they are otherwise very different.

Package Import Example

Let's actually code the example we've been talking about to show how initialization files and paths come into play. The following three files are coded in a directory *dir1* and its subdirectory *dir2*—comments give the path names of these files:

```
# dir1\__init__.py
print('dir1 init')
x = 1

# dir1\dir2\__init__.py
print('dir2 init')
y = 2
```

```
# dir1\dir2\mod.py
print('in mod.py')
z = 3
```

Here, *dir1* will be either a subdirectory of the one we're working in (i.e., the home directory), or a subdirectory of a directory that is listed on the module search path (technically, on `sys.path`). Either way, *dir1*'s container does not need an *__init__.py* file.

`import` statements run each directory's initialization file the first time that directory is traversed, as Python descends the path; `print` statements are included here to trace their execution. As with module files, an already imported directory may be passed to `reload` to force reexecution of that single item. As shown here, `reload` accepts a dotted pathname to reload nested directories and files:

```
% python
>>> import dir1.dir2.mod       # First imports run init files
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod       # Later imports do not
>>>
>>> from imp import reload     # Needed in 3.0
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

Once imported, the path in your `import` statement becomes a *nested object path* in your script. Here, `mod` is an object nested in the object `dir2`, which in turn is nested in the object `dir1`:

```
>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

In fact, each directory name in the path becomes a variable assigned to a module object whose namespace is initialized by all the assignments in that directory's *__init__.py* file. `dir1.x` refers to the variable `x` assigned in *dir1__init__.py*, much as `mod.z` refers to the variable `z` assigned in *mod.py*:

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

from Versus import with Packages

`import` statements can be somewhat inconvenient to use with packages, because you may have to retype the paths frequently in your program. In the prior section's example, for instance, you must retype and rerun the full path from `dir1` each time you want to reach `z`. If you try to access `dir2` or `mod` directly, you'll get an error:

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

It's often more convenient, therefore, to use the `from` statement with packages to avoid retyping the paths at each access. Perhaps more importantly, if you ever restructure your directory tree, the `from` statement requires just one path update in your code, whereas `import`s may require many. The `import as` extension, discussed formally in the next chapter, can also help here by providing a shorter synonym for the full path:

```
% python
>>> from dir1.dir2 import mod       # Code path here only
dir1 init
dir2 init
in mod.py
>>> mod.z                           # Don't repeat path
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod     # Use shorter name (see Chapter 24)
>>> mod.z
3
```

Why Use Package Imports?

If you're new to Python, make sure that you've mastered simple modules before stepping up to packages, as they are a somewhat advanced feature. They do serve useful roles, though, especially in larger programs: they make imports more informative, serve as an organizational tool, simplify your module search path, and can resolve ambiguities.

First of all, because package imports give some directory information in program files, they both make it easier to locate your files and serve as an organizational tool. Without package paths, you must often resort to consulting the module search path to find files. Moreover, if you organize your files into subdirectories for functional areas, package imports make it more obvious what role a module plays, and so make your code more readable. For example, a normal import of a file in a directory somewhere on the module search path, like this:

```
import utilities
```

offers much less information than an import that includes the path:

```
import database.client.utilities
```

Package imports can also greatly simplify your `PYTHONPATH` and *.pth* file search path settings. In fact, if you use explicit package imports for all your cross-directory imports, and you make those package imports relative to a common root directory where all your Python code is stored, you really only need a single entry on your search path: the common root. Finally, package imports serve to resolve ambiguities by making explicit exactly which files you want to import. The next section explores this role in more detail.

A Tale of Three Systems

The only time package imports are actually required is to resolve ambiguities that may arise when multiple programs with same-named files are installed on a single machine. This is something of an install issue, but it can also become a concern in general practice. Let's turn to a hypothetical scenario to illustrate.

Suppose that a programmer develops a Python program that contains a file called *utilities.py* for common utility code and a top-level file named *main.py* that users launch to start the program. All over this program, its files say `import utilities` to load and use the common code. When the program is shipped, it arrives as a single *.tar* or *.zip* file containing all the program's files, and when it is installed, it unpacks all its files into a single directory named *system1* on the target machine:

```
system1\
    utilities.py        # Common utility functions, classes
    main.py             # Launch this to start the program
    other.py            # Import utilities to load my tools
```

Now, suppose that a second programmer develops a different program with files also called *utilities.py* and *main.py*, and again uses `import utilities` throughout the program to load the common code file. When this second system is fetched and installed on the same computer as the first system, its files will unpack into a new directory called *system2* somewhere on the receiving machine (ensuring that they do not overwrite same-named files from the first system):

```
system2\
    utilities.py        # Common utilities
    main.py             # Launch this to run
    other.py            # Imports utilities
```

So far, there's no problem: both systems can coexist and run on the same machine. In fact, you won't even need to configure the module search path to use these programs on your computer—because Python always searches the home directory first (that is, the directory containing the top-level file), imports in either system's files will automatically see all the files in that system's directory. For instance, if you click on *system1\main.py*, all imports will search *system1* first. Similarly, if you launch

system2\main.py, *system2* will be searched first instead. Remember, module search path settings are only needed to import across directory boundaries.

However, suppose that after you've installed these two programs on your machine, you decide that you'd like to use some of the code in each of the *utilities.py* files in a system of your own. It's common utility code, after all, and Python code by nature wants to be reused. In this case, you want to be able to say the following from code that you're writing in a third directory to load one of the two files:

```
import utilities
utilities.func('spam')
```

Now the problem starts to materialize. To make this work at all, you'll have to set the module search path to include the directories containing the *utilities.py* files. But which directory do you put first in the path—*system1* or *system2*?

The problem is the *linear* nature of the search path. It is always scanned from left to right, so no matter how long you ponder this dilemma, you will always get *utilities.py* from the directory listed first (leftmost) on the search path. As is, you'll never be able to import it from the other directory at all. You could try changing `sys.path` within your script before each import operation, but that's both extra work and highly error prone. By default, you're stuck.

This is the issue that packages actually fix. Rather than installing programs as flat lists of files in standalone directories, you can package and install them as *subdirectories* under a common root. For instance, you might organize all the code in this example as an install hierarchy that looks like this:

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
    system3\                    # Here or elsewhere
        __init__.py             # Your new code here
        myfile.py
```

Now, add just the common root directory to your search path. If your code's imports are all relative to this common root, you can import *either* system's utility file with a package import—the enclosing directory name makes the path (and hence, the module reference) unique. In fact, you can import *both* utility files in the same module, as long as you use an `import` statement and repeat the full path each time you reference the utility modules:

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

The names of the enclosing directories here make the module references unique.

Note that you have to use `import` instead of `from` with packages only if you need to access the same attribute in two or more paths. If the name of the called function here was different in each path, `from` statements could be used to avoid repeating the full package path whenever you call one of the functions, as described earlier.

Also, notice in the install hierarchy shown earlier that *__init__.py* files were added to the *system1* and *system2* directories to make this work, but not to the *root* directory. Only directories listed within `import` statements in your code require these files; as you'll recall, they are run automatically the first time the Python process imports through a package directory.

Technically, in this case the *system3* directory doesn't have to be under *root*—just the packages of code from which you will import. However, because you never know when your own modules might be useful in other programs, you might as well place them under the common *root* directory as well to avoid similar name-collision problems in the future.

Finally, notice that both of the two original systems' imports will keep working unchanged. Because their *home* directories are searched first, the addition of the common root on the search path is irrelevant to code in *system1* and *system2*; they can keep saying just `import utilities` and expect to find their own files. Moreover, if you're careful to unpack all your Python systems under a common root like this, path configuration becomes simple: you'll only need to add the common root directory, once.

Package Relative Imports

The coverage of package imports so far has focused mostly on importing package files from *outside* the package. Within the package itself, imports of package files can use the same path syntax as outside imports, but they can also make use of special intra-package search rules to simplify `import` statements. That is, rather than listing package import paths, imports within the package can be relative to the package.

The way this works is version-dependent today: Python 2.6 implicitly searches package directories first on imports, while 3.0 requires explicit relative import syntax. This 3.0 change can enhance code readability, by making same-package imports more obvious. If you're starting out in Python with version 3.0, your focus in this section will likely be on its new import syntax. If you've used other Python packages in the past, though, you'll probably also be interested in how the 3.0 model differs.

Changes in Python 3.0

The way import operations in packages work has changed slightly in Python 3.0. This change applies only to imports within files located in the package directories we've been studying in this chapter; imports in other files work as before. For imports in packages, though, Python 3.0 introduces two changes:

- It modifies the module import search path semantics to skip the package's own directory by default. Imports check only other components of the search path. These are known as "absolute" imports.
- It extends the syntax of `from` statements to allow them to explicitly request that imports search the package's directory only. This is known as "relative" import syntax.

These changes are fully present in Python 3.0. The new `from` statement relative syntax is also available in Python 2.6, but the default search path change must be enabled as an option. It's currently scheduled to be added in the 2.7 release†—this change is being phased in this way because the search path portion is not backward compatible with earlier Pythons.

The impact of this change is that in 3.0 (and optionally in 2.6), you must generally use special `from` syntax to import modules located in the same package as the importer, unless you spell out a complete path from a package root. Without this syntax, your package is not automatically searched.

Relative Import Basics

In Python 3.0 and 2.6, `from` statements can now use leading dots (".") to specify that they require modules located within the same package (known as *package relative imports*), instead of modules located elsewhere on the module import search path (called *absolute imports*). That is:

- In both Python 3.0 and 2.6, you can use leading dots in `from` statements to indicate that imports should be *relative* to the containing package—such imports will search for modules inside the package only and will not look for same-named modules located elsewhere on the import search path (`sys.path`). The net effect is that package modules override outside modules.
- In Python 2.6, normal imports in a package's code (without leading dots) currently default to a relative-then-absolute search path order—that is, they search the package's own directory first. However, in Python 3.0, imports within a package are absolute by default—in the absence of any special dot syntax, imports skip the containing package itself and look elsewhere on the `sys.path` search path.

† Yes, there will be a 2.7 release, and possibly 2.8 and later releases, in parallel with new releases in the 3.X line. As described in the Preface, both the Python 2 and Python 3 lines are expected to be fully supported for years to come, to accommodate the large existing Python 2 user and code bases.

For example, in both Python 3.0 and 2.6, a statement of the form:

```
from . import spam                      # Relative to this package
```

instructs Python to import a module named `spam` located in the same package directory as the file in which this statement appears. Similarly, this statement:

```
from .spam import name
```

means "from a module named `spam` located in the same package as the file that contains this statement, import the variable `name`."

The behavior of a statement *without* the leading dot depends on which version of Python you use. In 2.6, such an import will still default to the current relative-then-absolute search path order (i.e., searching the package's directory first), unless a statement of the following form is included in the importing file:

```
from __future__ import  absolute_import    # Required until 2.7?
```

If present, this statement enables the Python 3.0 absolute-by-default default search path change, described in the next paragraph.

In 3.0, an import without a leading dot always causes Python to skip the relative components of the module import search path and look instead in the absolute directories that `sys.path` contains. For instance, in 3.0's model, a statement of the following form will always find a `string` module somewhere on `sys.path`, instead of a module of the same name in the package:

```
import string                           # Skip this package's version
```

Without the `from __future__` statement in 2.6, if there's a `string` module in the package, it will be imported instead. To get the same behavior in 3.0 and in 2.6 when the absolute import change is enabled, run a statement of the following form to force a relative import:

```
from . import string                    # Searches this package only
```

This works in both Python 2.6 and 3.0 today. The only difference in the 3.0 model is that it is *required* in order to load a module that is located in the same package directory as the file in which this appears, when the module is given with a simple name.

Note that leading dots can be used to force relative imports only with the `from` statement, not with the `import` statement. In Python 3.0, the `import modname` statement is always absolute, skipping the containing package's directory. In 2.6, this statement form still performs relative imports today (i.e., the package's directory is searched first), but these will become absolute in Python 2.7, too. `from` statements without leading dots behave the same as `import` statements—absolute in 3.0 (skipping the package directory), and relative-then-absolute in 2.6 (searching the package directory first).

Other dot-based relative reference patterns are possible, too. Within a module file located in a package directory named *mypkg*, the following alternative import forms work as described:

```
from .string import name1, name2          # Imports names from mypkg.string
from . import string                      # Imports mypkg.string
from .. import string                     # Imports string sibling of mypkg
```

To understand these latter forms better, we need to understand the rationale behind this change.

Why Relative Imports?

This feature is designed to allow scripts to resolve ambiguities that can arise when a same-named file appears in multiple places on the module search path. Consider the following package directory:

```
mypkg\
    __init__.py
    main.py
    string.py
```

This defines a package named `mypkg` containing modules named `mypkg.main` and `mypkg.string`. Now, suppose that the `main` module tries to import a module named `string`. In Python 2.6 and earlier, Python will first look in the *mypkg* directory to perform a *relative* import. It will find and import the *string.py* file located there, assigning it to the name `string` in the `mypkg.main` module's namespace.

It could be, though, that the intent of this import was to load the Python standard library's `string` module instead. Unfortunately, in these versions of Python, there's no straightforward way to ignore `mypkg.string` and look for the standard library's `string` module located on the module search path. Moreover, we cannot resolve this with package import paths, because we cannot depend on any extra package directory structure above the standard library being present on every machine.

In other words, imports in packages can be ambiguous—within a package, it's not clear whether an `import spam` statement refers to a module within or outside the package. More accurately, a local module or package can hide another hanging directly off of `sys.path`, whether intentionally or not.

In practice, Python users can avoid reusing the names of standard library modules they need for modules of their own (if you need the standard `string`, don't name a new module `string`!). But this doesn't help if a package accidentally hides a standard module; moreover, Python might add a new standard library module in the future that has the same name as a module of your own. Code that relies on relative imports is also less easy to understand, because the reader may be confused about which module is intended to be used. It's better if the resolution can be made explicit in code.

The relative imports solution in 3.0

To address this dilemma, imports run within packages have changed in Python 3.0 (and as an option in 2.6) to be absolute. Under this model, an `import` statement of the

following form in our example file *mypkg/main.py* will always find a `string` outside the package, via an absolute import search of `sys.path`:

```
import string                    # Imports string outside package
```

A `from` import without leading-dot syntax is considered absolute as well:

```
from string import name          # Imports name from string outside package
```

If you really want to import a module from your package without giving its full path from the package root, though, relative imports are still possible by using the dot syntax in the `from` statement:

```
from . import string             # Imports mypkg.string (relative)
```

This form imports the `string` module relative to the current package only and is the relative equivalent to the prior `import` example's absolute form; when this special relative syntax is used, the package's directory is the only directory searched.

We can also copy specific names from a module with relative syntax:

```
from .string import name1, name2     # Imports names from mypkg.string
```

This statement again refers to the `string` module relative to the current package. If this code appears in our `mypkg.main` module, for example, it will import `name1` and `name2` from `mypkg.string`.

In effect, the "." in a relative import is taken to stand for the package directory *containing* the file in which the import appears. An additional leading dot performs the relative import starting from the *parent* of the current package. For example, this statement:

```
from .. import spam              # Imports a sibling of mypkg
```

will load a sibling of `mypkg`—i.e., the `spam` module located in the package's own container directory, next to `mypkg`. More generally, code located in some module `A.B.C` can do any of these:

```
from . import D                  # Imports A.B.D    (. means A.B)
from .. import E                 # Imports A.E      (.. means A)

from .D import X                 # Imports A.B.D.X  (. means A.B)
from ..E import X                # Imports A.E.X    (.. means A)
```

Relative imports versus absolute package paths

Alternatively, a file can sometimes name its own package explicitly in an absolute import statement. For example, in the following, `mypkg` will be found in an absolute directory on `sys.path`:

```
from mypkg import string             # Imports mypkg.string (absolute)
```

However, this relies on both the configuration and the order of the module search path settings, while relative import dot syntax does not. In fact, this form requires that the directory immediately containing `mypkg` be included in the module search path. In

general, absolute import statements must list all the directories below the package's root entry in `sys.path` when naming packages explicitly like this:

```
from system.section.mypkg import string      # system container on sys.path only
```

In large or deep packages, that could be much more work than a dot:

```
from . import string                         # Relative import syntax
```

With this latter form, the containing package is searched automatically, regardless of the search path settings.

The Scope of Relative Imports

Relative imports can seem a bit perplexing on first encounter, but it helps if you remember a few key points about them:

- **Relative imports apply to imports within packages only**. Keep in mind that this feature's module search path change applies only to `import` statements within module files located in a package. Normal imports coded outside package files still work exactly as described earlier, automatically searching the directory containing the top-level script first.
- **Relative imports apply to the from statement only**. Also remember that this feature's new syntax applies only to `from` statements, not `import` statements. It's detected by the fact that the module name in a `from` begins with one or more dots (periods). Module names that contain dots but don't have a leading dot are package imports, not relative imports.
- **The terminology is ambiguous**. Frankly, the terminology used to describe this feature is probably more confusing than it needs to be. Really, all imports are relative to something. Outside a package, imports are still relative to directories listed on the `sys.path` module search path. As we learned in Chapter 21, this path includes the program's container directory, `PYTHONPATH` settings, path file settings, and standard libraries. When working interactively, the program container directory is simply the current working directory.

  For imports made inside packages, 2.6 augments this behavior by searching the package itself first. In the 3.0 model, all that really changes is that normal "absolute" import syntax skips the package directory, but special "relative" import syntax causes it to be searched first and only. When we talk about 3.0 imports as being "absolute," what we really mean is that they are relative to the directories on `sys.path`, but not the package itself. Conversely, when we speak of "relative" imports, we mean they are relative to the package directory only. Some `sys.path` entries could, of course, be absolute or relative paths too. (And I could probably make up something more confusing, but it would be a stretch!)

In other words, "package relative imports" in 3.0 really just boil down to a removal of 2.6's special search path behavior for packages, along with the addition of special `from` syntax to explicitly request relative behavior. If you wrote your package imports in the past to not depend on 2.6's special implicit relative lookup (e.g., by always spelling out full paths from a package root), this change is largely a moot point. If you didn't, you'll need to update your package files to use the new `from` syntax for local package files.

Module Lookup Rules Summary

With packages and relative imports, the module search story in Python 3.0 in its entirety can be summarized as follows:

- Simple module names (e.g., `A`) are looked up by searching each directory on the `sys.path` list, from left to right. This list is constructed from both system defaults and user-configurable settings.
- Packages are simply directories of Python modules with a special *__init__.py* file, which enables `A.B.C` directory path syntax in imports. In an import of `A.B.C`, for example, the directory named `A` is located relative to the normal module import search of `sys.path`, `B` is another package subdirectory within `A`, and `C` is a module or other importable item within `B`.
- Within a package's files, normal `import` statements use the same `sys.path` search rule as imports elsewhere. Imports in packages using `from` statements and leading dots, however, are relative to the package; that is, only the package directory is checked, and the normal `sys.path` lookup is not used. In `from . import A`, for example, the module search is restricted to the directory containing the file in which this statement appears.

Relative Imports in Action

But enough theory: let's run some quick tests to demonstrate the concepts behind relative imports.

Imports outside packages

First of all, as mentioned previously, this feature does not impact imports outside a package. Thus, the following finds the standard library `string` module as expected:

```
C:\test> c:\Python30\python
>>> import string
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

But if we add a module of the same name in the directory we're working in, it is selected instead, because the first entry on the module search path is the current working directory (CWD):

```
# test\string.py
print('string' * 8)

C:\test> c:\Python30\python
>>> import string
stringstringstringstringstringstringstringstring
>>> string
<module 'string' from 'string.py'>
```

In other words, normal imports are still relative to the "home" directory (the top-level script's container, or the directory you're working in). In fact, relative import syntax is not even allowed in code that is not in a file being used as part of a package:

```
>>> from . import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

In this and all examples in this section, code entered at the interactive prompt behaves the same as it would if run in a top-level script, because the first entry on `sys.path` is either the interactive working directory or the directory containing the top-level file. The only difference is that the start of `sys.path` is an absolute directory, not an empty string:

```
# test\main.py
import string
print(string)

C:\test> C:\python30\python main.py                    # Same results in 2.6
stringstringstringstringstringstringstringstring
<module 'string' from 'C:\test\string.py'>
```

Imports within packages

Now, let's get rid of the local `string` module we coded in the CWD and build a package directory there with two modules, including the required but empty *test\pkg__init__.py* file (which I'll omit here):

```
C:\test> del string*
C:\test> mkdir pkg

# test\pkg\spam.py
import eggs                    # <== Works in 2.6 but not 3.0!
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)
```

The first file in this package tries to import the second with a normal `import` statement. Because this is taken to be relative in 2.6 but absolute in 3.0, it fails in the latter. That is, 2.6 searches the containing package first, but 3.0 does not. This is the noncompatible behavior you have to be aware of in 3.0:

```
C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pkg\spam.py", line 1, in <module>
    import eggs
ImportError: No module named eggs
```

To make this work in both 2.6 and 3.0, change the first file to use the special relative import syntax, so that its import searches the package directory in 3.0, too:

```
# test\pkg\spam.py
from . import eggs              # <== Use package relative import in 2.6 or 3.0
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)

C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>
99999
```

Imports are still relative to the CWD

Notice in the preceding example that the package modules still have access to standard library modules like `string`. Really, their imports are still relative to the entries on the module search path, even if those entries are relative themselves. If you add a `string` module to the CWD again, imports in a package will find it there instead of in the standard library. Although you can skip the package directory with an absolute import in 3.0, you still can't skip the home directory of the program that imports the package:

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import eggs
```

```
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string                   # <== Gets string in CWD, not Python lib!
print(string)

C:\test> c:\Python30\python     # Same result in 2.6
>>> import pkg.spam
stringstringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>
99999
```

Selecting modules with relative and absolute imports

To show how this applies to imports of standard library modules, reset the package one more time. Get rid of the local `string` module, and define a new one inside the package itself:

```
C:\test> del string*

# test\pkg\spam.py
import string                   # <== Relative in 2.6, absolute in 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

Now, which version of the `string` module you get depends on which Python you use. As before, 3.0 interprets the import in the first file as absolute and skips the package, but 2.6 does not:

```
C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Using relative import syntax in 3.0 forces the package to be searched again, as it is in 2.6—by using absolute or relative import syntax in 3.0, you can either skip or select the package directory explicitly. In fact, this is the use case that the 3.0 model addresses:

```
# test\pkg\spam.py
from . import string            # <== Relative in both 2.6 and 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
```

```
<module 'pkg.string' from 'pkg\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

It's important to note that relative import syntax is really a binding declaration, not just a preference. If we delete the *string.py* file in this example, the relative import in *spam.py fails* in both 3.0 and 2.6, instead of falling back on the standard library's version of this module (or any other):

```
# test\pkg\spam.py
from . import string            # <== Fails if no string.py here!

C:\test> C:\python30\python
>>> import pkg.spam
...text omitted...
ImportError: cannot import name string
```

Modules referenced by relative imports must exist in the package directory.

Imports are still relative to the CWD (again)

Although absolute imports let you skip package modules, they still rely on other components of `sys.path`. For one last test, let's define two `string` modules of our own. In the following, there is one module by that name in the CWD, one in the package, and another in the standard library:

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import string            # <== Relative in both 2.6 and 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

When we import the `string` module with relative import syntax, we get the version in the package, as desired:

```
C:\test> c:\Python30\python     # Same result in 2.6
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

When absolute syntax is used, though, the module we get varies per version again. 2.6 interprets this as relative to the package, but 3.0 makes it "absolute," which in this case really just means it skips the package and loads the version relative to the CWD (*not* the version the standard library):

```
# test\string.py
print('string' * 8)
```

```
# test\pkg\spam.py
import string                   # <== Relative in 2.6, "absolute" in 3.0: CWD!
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.pyc'>
```

As you can see, although packages can explicitly request modules within their own directories, their imports are otherwise still relative to the rest of the normal module search path. In this case, a file in the program using the package hides the standard library module the package may want. All that the change in 3.0 really accomplishes is allowing package code to select files either inside or outside the package (i.e., relatively or absolutely). Because import resolution can depend on an enclosing context that may not be foreseen, absolute imports in 3.0 are not a guarantee of finding a module in the standard library.

Experiment with these examples on your own for more insight. In practice, this is not usually as ad-hoc as it might seem: you can generally structure your imports, search paths, and module names to work the way you wish during development. You should keep in mind, though, that imports in larger systems may depend upon context of use, and the module import protocol is part of a successful library's design.

Now that you've learned about package-relative imports, also keep in mind that they may not always be your best option. Absolute package imports, relative to a directory on `sys.path`, are still sometimes preferred over both implicit package-relative imports in Python 2, and explicit package-relative import syntax in both Python 2 and 3.

Package-relative import syntax and Python 3.0's new absolute import search rules at least require relative imports from a package to be made explicit, and thus easier to understand and maintain. Files that use imports with dots, though, are implicitly bound to a package directory and cannot be used elsewhere without code changes.

Naturally, the extent to which this may impact your modules can vary per package; absolute imports may also require changes when directories are reorganized.

Why You Will Care: Module Packages

Now that packages are a standard part of Python, it's common to see larger third-party extensions shipped as sets of package directories, rather than flat lists of modules. The *win32all* Windows extensions package for Python, for instance, was one of the first to jump on the package bandwagon. Many of its utility modules reside in packages imported with paths. For instance, to load client-side COM tools, you use a statement like this:

```
from win32com.client import constants, Dispatch
```

This line fetches names from the `client` module of the `win32com` package (an install subdirectory).

Package imports are also pervasive in code run under the Jython Java-based implementation of Python, because Java libraries are organized into hierarchies as well. In recent Python releases, the email and XML tools are likewise organized into package subdirectories in the standard library, and Python 3.0 groups even more related modules into packages (including tkinter GUI tools, HTTP networking tools, and more). The following imports access various standard library tools in 3.0:

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

Whether you create package directories or not, you will probably import from them eventually.

Chapter Summary

This chapter introduced Python's package import model—an optional but useful way to explicitly list part of the directory path leading up to your modules. Package imports are still relative to a directory on your module import search path, but rather than relying on Python to traverse the search path manually, your script gives the rest of the path to the module explicitly.

As we've seen, packages not only make imports more meaningful in larger systems, but also simplify import search path settings (if all cross-directory imports are relative to a common root directory) and resolve ambiguities when there is more than one module of the same name (including the name of the enclosing directory in a package import helps distinguish between them).

Because it's relevant only to code in packages, we also explored the newer relative import model here—a way for imports in package files to select modules in the same package using leading dots in a `from`, instead of relying on an older implicit package search rule.

In the next chapter, we will survey a handful of more advanced module-related topics, such as relative import syntax and the `__name__` usage mode variable. As usual, though, we'll close out this chapter with a short quiz to test what you've learned here.

Test Your Knowledge: Quiz

1. What is the purpose of an *__init__.py* file in a module package directory?
2. How can you avoid repeating the full package path every time you reference a package's content?
3. Which directories require *__init__.py* files?
4. When must you use `import` instead of `from` with packages?
5. What is the difference between `from mypkg import spam` and `from . import spam`?

Test Your Knowledge: Answers

1. The *__init__.py* file serves to declare and initialize a module package; Python automatically runs its code the first time you import through a directory in a process. Its assigned variables become the attributes of the module object created in memory to correspond to that directory. It is also not optional—you can't import through a directory with package syntax unless it contains this file.
2. Use the `from` statement with a package to copy names out of the package directly, or use the `as` extension with the `import` statement to rename the path to a shorter synonym. In both cases, the path is listed in only one place, in the `from` or `import` statement.
3. Each directory listed in an `import` or `from` statement must contain an *__init__.py* file. Other directories, including the directory containing the leftmost component of a package path, do not need to include this file.
4. You must use `import` instead of `from` with packages only if you need to access the same name defined in more than one path. With `import`, the path makes the references unique, but `from` allows only one version of any given name.
5. `from mypkg import spam` is an *absolute* import—the search for `mypkg` skips the package directory and the module is located in an absolute directory in `sys.path`. A statement `from . import spam`, on the other hand, is a *relative* import—`spam` is looked up relative to the package in which this statement is contained before `sys.path` is searched.

CHAPTER 24

Advanced Module Topics

This chapter concludes this part of the book with a collection of more advanced module-related topics—data hiding, the `__future__` module, the `__name__` variable, `sys.path` changes, listing tools, running modules by name string, transitive reloads, and so on—along with the standard set of gotchas and exercises related to what we've covered in this part of the book.

Along the way, we'll build some larger and more useful tools than we have so far, that combine functions and modules. Like functions, modules are more effective when their interfaces are well defined, so this chapter also briefly reviews module design concepts, some of which we have explored in prior chapters.

Despite the word "advanced" in this chapter's title, this is also something of a grab bag of additional module topics. Because some of the topics discussed here are widely used (especially the `__name__` trick), be sure to take a look before moving on to classes in the next part of the book.

Data Hiding in Modules

As we've seen, a Python module exports all the names assigned at the top level of its file. There is no notion of declaring which names should and shouldn't be visible outside the module. In fact, there's no way to prevent a client from changing names inside a module if it wants to.

In Python, data hiding in modules is a convention, not a syntactical constraint. If you want to break a module by trashing its names, you can, but fortunately, I've yet to meet a programmer who would. Some purists object to this liberal attitude toward data hiding, claiming that it means Python can't implement encapsulation. However, encapsulation in Python is more about packaging than about restricting.

Minimizing from * Damage: _X and __all__

As a special case, you can prefix names with a single underscore (e.g., `_X`) to prevent them from being copied out when a client imports a module's names with a `from *` statement. This really is intended only to minimize namespace pollution; because `from *` copies out all names, the importer may get more than it's bargained for (including names that overwrite names in the importer). Underscores aren't "private" declarations: you can still see and change such names with other import forms, such as the `import` statement.

Alternatively, you can achieve a hiding effect similar to the `_X` naming convention by assigning a list of variable name strings to the variable `__all__` at the top level of the module. For example:

```
__all__ = ["Error", "encode", "decode"]      # Export these only
```

When this feature is used, the `from *` statement will copy out only those names listed in the `__all__` list. In effect, this is the converse of the `_X` convention: `__all__` identifies names to be copied, while `_X` identifies names not to be copied. Python looks for an `__all__` list in the module first; if one is not defined, `from *` copies all names without a single leading underscore.

Like the `_X` convention, the `__all__` list has meaning only to the `from *` statement form and does not amount to a privacy declaration. Module writers can use either trick to implement modules that are well behaved when used with `from *`. (See also the discussion of `__all__` lists in package *__init__.py* files in Chapter 23; there, these lists declare submodules to be loaded for a `from *`.)

Enabling Future Language Features

Changes to the language that may potentially break existing code are introduced gradually. Initially, they appear as optional extensions, which are disabled by default. To turn on such extensions, use a special `import` statement of this form:

```
from __future__ import featurename
```

This statement should generally appear at the top of a module file (possibly after a docstring), because it enables special compilation of code on a per-module basis. It's also possible to submit this statement at the interactive prompt to experiment with upcoming language changes; the feature will then be available for the rest of the interactive session.

For example, in prior editions of this book, we had to use this statement form to demonstrate generator functions, which required a keyword that was not yet enabled by default (they use a *`featurename`* of `generators`). We also used this statement to activate 3.0 true division in Chapter 5, 3.0 `print` calls in Chapter 11, and 3.0 absolute imports for packages in Chapter 23.

All of these changes have the potential to break existing code in Python 2.6, so they are being phased in gradually as optional features enabled with this special import.

Mixed Usage Modes: __name__ and __main__

Here's another module-related trick that lets you both import a file as a module and run it as a standalone program. Each module has a built-in attribute called `__name__`, which Python sets automatically as follows:

- If the file is being run as a top-level program file, `__name__` is set to the string `"__main__"` when it starts.
- If the file is being imported instead, `__name__` is set to the module's name as known by its clients.

The upshot is that a module can test its own `__name__` to determine whether it's being run or imported. For example, suppose we create the following module file, named *runme.py*, to export a single function called `tester`:

```
def tester():
    print("It's Christmas in Heaven...")

if __name__ == '__main__':            # Only when run
    tester()                          # Not when imported
```

This module defines a function for clients to import and use as usual:

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven...
```

But, the module also includes code at the bottom that is set up to call the function when this file is run as a program:

```
% python runme.py
It's Christmas in Heaven...
```

In effect, a module's `__name__` variable serves as a *usage mode flag*, allowing its code to be leveraged as both an importable library and a top-level script. Though simple, you'll see this hook used in nearly every realistic Python program file you are likely to encounter.

Perhaps the most common way you'll see the `__name__` test applied is for *self-test* code. In short, you can package code that tests a module's exports in the module itself by wrapping it in a `__name__` test at the bottom of the file. This way, you can use the file in clients by importing it, but also test its logic by running it from the system shell or via another launching scheme. In practice, self-test code at the bottom of a file under the `__name__` test is probably the most common and simplest unit-testing protocol in Python. (Chapter 35 will discuss other commonly used options for testing Python

code—as you'll see, the `unittest` and `doctest` standard library modules provide more advanced testing tools.)

The `__name__` trick is also commonly used when writing files that can be used both as command-line utilities and as tool libraries. For instance, suppose you write a file-finder script in Python. You can get more mileage out of your code if you package it in functions and add a `__name__` test in the file to automatically call those functions when the file is run standalone. That way, the script's code becomes reusable in other programs.

Unit Tests with `__name__`

In fact, we've already seen a prime example in this book of an instance where the `__name__` check could be useful. In the section on arguments in Chapter 18, we coded a script that computed the minimum value from the set of arguments sent in:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))        # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

This script includes self-test code at the bottom, so we can test it without having to retype everything at the interactive command line each time we run it. The problem with the way it is currently coded, however, is that the output of the self-test call will appear every time this file is imported from another file to be used as a tool—not exactly a user-friendly feature! To improve it, we can wrap up the self-test call in a `__name__` check, so that it will be launched only when the file is run as a top-level script, not when it is imported:

```
print('I am:', __name__)

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))    # Self-test code
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

We're also printing the value of `__name__` at the top here to trace its value. Python creates and assigns this usage-mode variable as soon as it starts loading a file. When we run this file as a top-level script, its name is set to `__main__`, so its self-test code kicks in automatically:

```
% python min.py
I am: __main__
1
6
```

But, if we import the file, its name is not `__main__`, so we must explicitly call the function to make it run:

```
>>> import min
I am: min
>>> min.minmax(min.lessthan, 's', 'p', 'a', 'm')
'a'
```

Again, regardless of whether this is used for testing, the net effect is that we get to use our code in two different roles—as a library module of tools, or as an executable program.

Using Command-Line Arguments with __name__

Here's a more substantial module example that demonstrates another way that the `__name__` trick is commonly employed. The following module, *formats.py*, defines string formatting utilities for importers, but also checks its name to see if it is being run as a top-level script; if so, it tests and uses arguments listed on the system command line to run a canned or passed-in test. In Python, the `sys.argv` list contains *command-line arguments*—it is a list of strings reflecting words typed on the command line, where the first item is always the name of the script being run:

```
"""
Various specialized string display formatting utilities.
Test me with canned self-test or command-line arguments.
"""

def commas(N):
    """
    format positive integer-like N for display with
    commas between digit groupings: xxx,yyy,zzz
    """
    digits = str(N)
    assert(digits.isdigit())
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = (last3 + ',' + result) if result else last3
    return result

def money(N, width=0):
    """
```

```
    format number N for display with commas, 2 decimal digits,
    leading $ and sign, and optional padding: $  -xxx,yyy.zz
    """
    sign   = '-' if N < 0 else ''
    N      = abs(N)
    whole  = commas(int(N))
    fract  = ('%.2f' % N)[-2:]
    format = '%s%s.%s' % (sign, whole, fract)
    return '$%*s' % (width, format)

if __name__ == '__main__':
    def selftest():
        tests  = 0, 1        # fails: -1, 1.23
        tests += 12, 123, 1234, 12345, 123456, 1234567
        tests += 2 ** 32, 2 ** 100
        for test in tests:
            print(commas(test))

        print('')
        tests  = 0, 1, -1, 1.23, 1., 1.2, 3.14159
        tests += 12.34, 12.344, 12.345, 12.346
        tests += 2 ** 32, (2 ** 32 + .2345)
        tests += 1.2345, 1.2, 0.2345
        tests += -1.2345, -1.2, -0.2345
        tests += -(2 ** 32), -(2**32 + .2345)
        tests += (2 ** 100), -(2 ** 100)
        for test in tests:
            print('%s [%s]' % (money(test, 17), test))

    import sys
    if len(sys.argv) == 1:
        selftest()
    else:
        print(money(float(sys.argv[1]), int(sys.argv[2])))
```

This file works the same in Python 2.6 and 3.0. When run directly, it tests itself as before, but it uses options on the command line to control the test behavior. Run this file directly with no command-line arguments on your own to see what its self-test code prints. To test specific strings, pass them in on the command line along with a minimum field width:

```
C:\misc> python formats.py 999999999 0
$999,999,999.00

C:\misc> python formats.py -999999999 0
$-999,999,999.00

C:\misc> python formats.py 123456789012345 0
$123,456,789,012,345.00

C:\misc> python formats.py -123456789012345 25
$  -123,456,789,012,345.00

C:\misc> python formats.py 123.456 0
$123.46
```

```
C:\misc> python formats.py -123.454 0
$-123.45

C:\misc> python formats.py
...canned tests: try this yourself...
```

As before, because this code is instrumented for dual-mode usage, we can also import its tools normally in other contexts as library components:

```
>>> from formats import money, commas
>>> money(123.456)
'$123.46'
>>> money(-9999999.99, 15)
'$  -9,999,999.99'
>>> X = 99999999999999999999
>>> '%s (%s)' % (commas(X), X)
'99,999,999,999,999,999,999 (99999999999999999999)'
```

Because this file uses the *docstring* feature introduced in Chapter 15, we can use the `help` function to explore its tools as well—it serves as a general-purpose tool:

```
>>> import formats
>>> help(formats)
Help on module formats:

NAME
    formats

FILE
    c:\misc\formats.py

DESCRIPTION
    Various specialized string display formatting utilities.
    Test me with canned self-test or command-line arguments.

FUNCTIONS
    commas(N)
        format positive integer-like N for display with
        commas between digit groupings: xxx,yyy,zzz

    money(N, width=0)
        format number N for display with commas, 2 decimal digits,
        leading $ and sign, and optional padding: $  -xxx,yyy.zz
```

You can use command-line arguments in similar ways to provide general inputs to scripts that may also package their code as functions and classes for reuse by importers. For more advanced command-line processing, be sure to see the `getopt` and `optparse` modules in Python's standard library and manuals. In some scenarios, you might also use the built-in `input` function introduced in Chapter 3 and used in Chapter 10 to prompt the shell user for test inputs instead of pulling them from the command line.

Also see Chapter 7's discussion of the new {,d} string format method syntax that will be available in Python 3.1 and later; this formatting extension separates thousands groups with commas much like the code here. The module listed here, though, adds money formatting and serves as a manual alternative for comma insertion for Python versions before 3.1.

Changing the Module Search Path

In Chapter 21, we learned that the module search path is a list of directories that can be customized via the environment variable PYTHONPATH, and possibly via *.pth* files. What I haven't shown you until now is how a Python program itself can actually change the search path by changing a built-in list called sys.path (the path attribute in the built-in sys module). sys.path is initialized on startup, but thereafter you can delete, append, and reset its components however you like:

```
>>> import sys
>>> sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', ...more deleted...]

>>> sys.path.append('C:\\sourcedir')          # Extend module search path
>>> import string                             # All imports search the new dir last
```

Once you've made such a change, it will impact future imports anywhere in the Python program, as all imports and all files share the single sys.path list. In fact, this list may be changed arbitrarily:

```
>>> sys.path = [r'd:\temp']                   # Change module search path
>>> sys.path.append('c:\\lp4e\\examples')     # For this process only
>>> sys.path
['d:\\temp', 'c:\\lp4e\\examples']

>>> import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named string
```

Thus, you can use this technique to dynamically configure a search path inside a Python program. Be careful, though: if you delete a critical directory from the path, you may lose access to critical utilities. In the prior example, for instance, we no longer have access to the string module because we deleted the Python source library's directory from the path.

Also, remember that such sys.path settings endure for only as long as the Python session or program (technically, process) that made them runs; they are not retained after Python exits. PYTHONPATH and *.pth* file path configurations live in the operating system instead of a running Python program, and so are more global: they are picked up by every program on your machine and live on after a program completes.

The as Extension for import and from

Both the `import` and `from` statements have been extended to allow an imported name to be given a different name in your script. The following `import` statement:

```
import modulename as name
```

is equivalent to:

```
import modulename
name = modulename
del modulename                          # Don't keep original name
```

After such an `import`, you can (and in fact must) use the name listed after the `as` to refer to the module. This works in a `from` statement, too, to assign a name imported from a file to a different name in your script:

```
from modulename import attrname as name
```

This extension is commonly used to provide short synonyms for longer names, and to avoid name clashes when you are already using a name in your script that would otherwise be overwritten by a normal `import` statement:

```
import reallylongmodulename as name     # Use shorter nickname
name.func()

from module1 import utility as util1    # Can have only 1 "utility"
from module2 import utility as util2
util1(); util2()
```

It also comes in handy for providing a short, simple name for an entire directory path when using the package import feature described in Chapter 23:

```
import dir1.dir2.mod as mod             # Only list full path once
mod.func()
```

Modules Are Objects: Metaprograms

Because modules expose most of their interesting properties as built-in attributes, it's easy to write programs that manage other programs. We usually call such manager programs *metaprograms* because they work on top of other systems. This is also referred to as *introspection*, because programs can see and process object internals. Introspection is an advanced feature, but it can be useful for building programming tools.

For instance, to get to an attribute called `name` in a module called `M`, we can use qualification or index the module's attribute dictionary, exposed in the built-in `__dict__` attribute we met briefly in Chapter 22. Python also exports the list of all loaded modules as the `sys.modules` dictionary (that is, the `modules` attribute of the `sys` module) and provides a built-in called `getattr` that lets us fetch attributes from their string names (it's like saying `object.attr`, but `attr` is an expression that yields a string at runtime). Because of that, all the following expressions reach the same attribute and object:

```
M.name                                  # Qualify object
M.__dict__['name']                      # Index namespace dictionary manually
sys.modules['M'].name                   # Index loaded-modules table manually
getattr(M, 'name')                      # Call built-in fetch function
```

By exposing module internals like this, Python helps you build programs about programs.* For example, here is a module named *mydir.py* that puts these ideas to work to implement a customized version of the built-in `dir` function. It defines and exports a function called `listing`, which takes a module object as an argument and prints a formatted listing of the module's namespace:

```
"""
mydir.py: a module that lists the namespaces of other modules
"""

seplen = 60
sepchr = '-'

def listing(module, verbose=True):
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print('name:', module.__name__, 'file:', module.__file__)
        print(sepline)

    count = 0
    for attr in module.__dict__:                # Scan namespace keys
        print('%02d) %s' % (count, attr), end = ' ')
        if attr.startswith('__'):
            print('<built-in name>')            # Skip __file__, etc.
        else:
            print(getattr(module, attr))        # Same as .__dict__[attr]
        count += 1

    if verbose:
        print(sepline)
        print(module.__name__, 'has %d names' % count)
        print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir)                              # Self-test code: list myself
```

Notice the docstring at the top; as in the prior *formats.py* example, because we may want to use this as a general tool, a docstring is coded to provide functional information accessible via `__doc__` attributes or the `help` function (see Chapter 15 for details):

* As we saw in Chapter 17, because a function can access its enclosing module by going through the `sys.modules` table like this, it's possible to emulate the effect of the `global` statement. For instance, the effect of `global X; X=0` can be simulated (albeit with much more typing!) by saying this inside a function: `import sys; glob=sys.modules[__name__]; glob.X=0`. Remember, each module gets a `__name__` attribute for free; it's visible as a global name inside the functions within the module. This trick provides another way to change both local and global variables of the same name inside a function.

```
>>> import mydir
>>> help(mydir)
Help on module mydir:

NAME
    mydir - mydir.py: a module that lists the namespaces of other modules

FILE
    c:\users\veramark\mark\mydir.py

FUNCTIONS
    listing(module, verbose=True)

DATA
    sepchr = '-'
    seplen = 60
```

I've also provided *self-test* logic at the bottom of this module, which narcissistically imports and lists itself. Here's the sort of output produced in Python 3.0 (to use this in 2.6, enable 3.0 `print` calls with the `__future__` import described in Chapter 11—the `end` keyword is 3.0-only):

```
C:\Users\veramark\Mark> c:\Python30\python mydir.py
------------------------------------------------------------
name: mydir file: C:\Users\veramark\Mark\mydir.py
------------------------------------------------------------
00) seplen 60
01) __builtins__ <built-in name>
02) __file__ <built-in name>
03) __package__ <built-in name>
04) listing <function listing at 0x026D3B70>
05) __name__ <built-in name>
06) sepchr -
07) __doc__ <built-in name>
------------------------------------------------------------
mydir has 8 names
------------------------------------------------------------
```

To use this as a tool for listing other modules, simply pass the modules in as objects to this file's function. Here it is listing attributes in the `tkinter` GUI module in the standard library (a.k.a. `Tkinter` in Python 2.6):

```
>>> import mydir
>>> import tkinter
>>> mydir.listing(tkinter)
------------------------------------------------------------
name: tkinter file: c:\PYTHON30\lib\tkinter\__init__.py
------------------------------------------------------------
00) getdouble <class 'float'>
01) MULTIPLE multiple
02) mainloop <function mainloop at 0x02913B70>
03) Canvas <class 'tkinter.Canvas'>
04) AtSelLast <function AtSelLast at 0x028FA7C8>
...many more name omitted...
151) StringVar <class 'tkinter.StringVar'>
```

```
152) ARC arc
153) At <function At at 0x028FA738>
154) NSEW nsew
155) SCROLL scroll
------------------------------------------------------------
tkinter has 156 names
------------------------------------------------------------
```

We'll meet `getattr` and its relatives again later. The point to notice here is that `mydir` is a program that lets you browse other programs. Because Python exposes its internals, you can process objects generically.†

Importing Modules by Name String

The module name in an `import` or `from` statement is a hardcoded variable name. Sometimes, though, your program will get the name of a module to be imported as a string at runtime (e.g., if a user selects a module name from within a GUI). Unfortunately, you can't use `import` statements directly to load a module given its name as a string—Python expects a variable name, not a string. For instance:

```
>>> import "string"
  File "<stdin>", line 1
    import "string"
                ^
SyntaxError: invalid syntax
```

It also won't work to simply assign the string to a variable name:

```
x = "string"
import x
```

Here, Python will try to import a file *x.py*, not the `string` module—the name in an `import` statement both becomes a variable assigned to the loaded module and identifies the external file literally.

To get around this, you need to use special tools to load a module dynamically from a string that is generated at runtime. The most general approach is to construct an `import` statement as a string of Python code and pass it to the `exec` built-in function to run (`exec` is a statement in Python 2.6, but it can be used exactly as shown here—the parentheses are simply ignored):

```
>>> modname = "string"
>>> exec("import " + modname)       # Run a string of code
>>> string                          # Imported in this namespace
<module 'string' from 'c:\Python30\lib\string.py'>
```

† Tools such as `mydir.listing` can be preloaded into the interactive namespace by importing them in the file referenced by the `PYTHONSTARTUP` environment variable. Because code in the startup file runs in the interactive namespace (module `__main__`), importing common tools in the startup file can save you some typing. See Appendix A for more details.

The exec function (and its cousin for expressions, eval) compiles a string of code and passes it to the Python interpreter to be executed. In Python, the byte code compiler is available at runtime, so you can write programs that construct and run other programs like this. By default, exec runs the code in the current scope, but you can get more specific by passing in optional namespace dictionaries.

The only real drawback to exec is that it must compile the import statement each time it runs; if it runs many times, your code may run quicker if it uses the built-in __import__ function to load from a name string instead. The effect is similar, but __import__ returns the module object, so assign it to a name here to keep it:

```
>>> modname = "string"
>>> string = __import__(modname)
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

Transitive Module Reloads

We studied module reloads in Chapter 22, as a way to pick up changes in code without stopping and restarting a program. When you reload a module, though, Python only reloads that particular module's file; it doesn't automatically reload modules that the file being reloaded happens to import.

For example, if you reload some module A, and A imports modules B and C, the reload applies only to A, not to B and C. The statements inside A that import B and C are rerun during the reload, but they just fetch the already loaded B and C module objects (assuming they've been imported before). In actual code, here's the file *A.py*:

```
import B                    # Not reloaded when A is
import C                    # Just an import of an already loaded module
```

```
% python
>>> . . .
>>> from imp import reload
>>> reload(A)
```

By default, this means that you cannot depend on reloads picking up changes in all the modules in your program transitively—instead, you must use multiple reload calls to update the subcomponents independently. This can require substantial work for large systems you're testing interactively. You can design your systems to reload their subcomponents automatically by adding reload calls in parent modules like A, but this complicates the modules' code.

A better approach is to write a general tool to do transitive reloads automatically by scanning modules' __dict__ attributes and checking each item's type to find nested modules to reload. Such a utility function could call itself *recursively* to navigate arbitrarily shaped import dependency chains. Module __dict__ attributes were introduced earlier in, the section "Modules Are Objects: Metaprograms" on page 591, and the type call was presented in Chapter 9; we just need to combine the two tools.

For example, the module *reloadall.py* listed next has a `reload_all` function that automatically reloads a module, every module that the module imports, and so on, all the way to the bottom of each import chain. It uses a dictionary to keep track of already reloaded modules, recursion to walk the import chains, and the standard library's `types` module, which simply predefines `type` results for built-in types. The `visited` dictionary technique works to avoid cycles here when imports are recursive or redundant, because module objects can be dictionary keys (as we learned in Chapter 5, a set would offer similar functionality if we use `visited.add(module)` to insert):

```
"""
reloadall.py: transitively reload nested modules
"""

import types
from imp import reload                              # from required in 3.0

def status(module):
    print('reloading ' + module.__name__)

def transitive_reload(module, visited):
    if not module in visited:                       # Trap cycles, duplicates
        status(module)                              # Reload this module
        reload(module)                              # And visit children
        visited[module] = None
        for attrobj in module.__dict__.values():    # For all attrs
            if type(attrobj) == types.ModuleType:   # Recur if module
                transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {}
    for arg in args:
        if type(arg) == types.ModuleType:
            transitive_reload(arg, visited)

if __name__ == '__main__':
    import reloadall                                # Test code: reload myself
    reload_all(reloadall)                           # Should reload this, types
```

To use this utility, import its `reload_all` function and pass it the name of an already loaded module (like you would the built-in `reload` function). When the file runs standalone, its self-test code will test itself—it has to import itself because its own name is not defined in the file without an import (this code works in both 3.0 and 2.6 and prints identical output, because we've used `+` instead of a comma in the `print`):

```
C:\misc> c:\Python30\python reloadall.py
reloading reloadall
reloading types
```

Here is this module at work in 3.0 on some standard library modules. Notice how `os` is imported by `tkinter`, but `tkinter` reaches `sys` before `os` can (if you want to test this on Python 2.6, substitute `Tkinter` for `tkinter`):

```
>>> from reloadall import reload_all
>>> import os, tkinter

>>> reload_all(os)
reloading os
reloading copyreg
reloading ntpath
reloading genericpath
reloading stat
reloading sys
reloading errno

>>> reload_all(tkinter)
reloading tkinter
reloading _tkinter
reloading tkinter._fix
reloading sys
reloading ctypes
reloading os
reloading copyreg
reloading ntpath
reloading genericpath
reloading stat
reloading errno
reloading ctypes._endian
reloading tkinter.constants
```

And here is a session that shows the effect of normal versus transitive reloads—changes made to the two nested files are not picked up by reloads, unless the transitive utility is used:

```
import b                        # a.py
X = 1

import c                        # b.py
Y = 2

Z = 3                           # c.py

C:\misc> C:\Python30\python
>>> import a
>>> a.X, a.b.Y, a.b.c.Z
(1, 2, 3)

# Change all three files' assignment values and save

>>> from imp import reload
>>> reload(a)                   # Normal reload is top level only
<module 'a' from 'a.py'>
>>> a.X, a.b.Y, a.b.c.Z
(111, 2, 3)

>>> from reloadall import reload_all
>>> reload_all(a)
reloading a
```

```
reloading b
reloading c
>>> a.X, a.b.Y, a.b.c.Z          # Reloads all nested modules too
(111, 222, 333)
```

For more insight, study and experiment with this example on your own; it's another importable tool you might want to add to your own source code library.

Module Design Concepts

Like functions, modules present design tradeoffs: you have to think about which functions go in which modules, module communication mechanisms, and so on. All of this will become clearer when you start writing bigger Python systems, but here are a few general ideas to keep in mind:

- **You're always in a module in Python**. There's no way to write code that doesn't live in some module. In fact, code typed at the interactive prompt really goes in a built-in module called `__main__`; the only unique things about the interactive prompt are that code runs and is discarded immediately, and expression results are printed automatically.
- **Minimize module coupling: global variables**. Like functions, modules work best if they're written to be closed boxes. As a rule of thumb, they should be as independent of global variables used within other modules as possible, except for functions and classes imported from them.
- **Maximize module cohesion: unified purpose**. You can minimize a module's couplings by maximizing its cohesion; if all the components of a module share a general purpose, you're less likely to depend on external names.
- **Modules should rarely change other modules' variables**. We illustrated this with code in Chapter 17, but it's worth repeating here: it's perfectly OK to use globals defined in another module (that's how clients import services, after all), but changing globals in another module is often a symptom of a design problem. There are exceptions, of course, but you should try to communicate results through devices such as function arguments and return values, not cross-module changes. Otherwise, your globals' values become dependent on the order of arbitrarily remote assignments in other files, and your modules become harder to understand and reuse.

As a summary, Figure 24-1 sketches the environment in which modules operate. Modules contain variables, functions, classes, and other modules (if imported). Functions have local variables of their own, as do classes—i.e., objects that live within modules, which we'll meet next in Chapter 25.

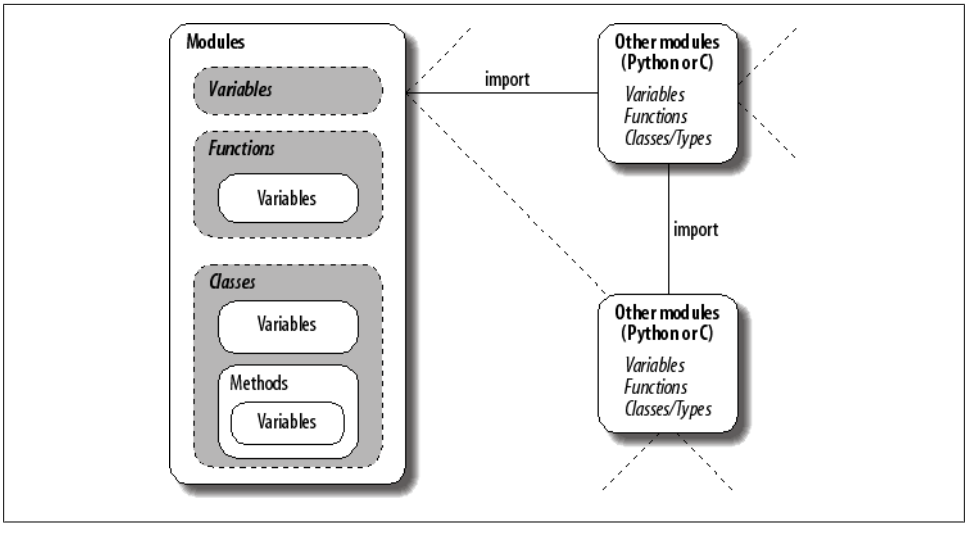

Figure 24-1. Module execution environment. Modules are imported, but modules also import and use other modules, which may be coded in Python or another language such as C. Modules in turn contain variables, functions, and classes to do their work, and their functions and classes may contain variables and other items of their own. At the top, though, programs are just sets of modules.

Module Gotchas

In this section, we'll take a look at the usual collection of boundary cases that make life interesting for Python beginners. Some are so obscure that it was hard to come up with examples, but most illustrate something important about the language.

Statement Order Matters in Top-Level Code

When a module is first imported (or reloaded), Python executes its statements one by one, from the top of the file to the bottom. This has a few subtle implications regarding forward references that are worth underscoring here:

- Code at the top level of a module file (not nested in a function) runs as soon as Python reaches it during an import; because of that, it can't reference names assigned lower in the file.
- Code inside a function body doesn't run until the function is called; because names in a function aren't resolved until the function actually runs, they can usually reference names anywhere in the file.

Generally, forward references are only a concern in top-level module code that executes immediately; functions can reference names arbitrarily. Here's an example that illustrates forward reference:

```
func1()                          # Error: "func1" not yet assigned

def func1():
    print(func2())               # Okay: "func2" looked up later

func1()                          # Error: "func2" not yet assigned

def func2():
    return "Hello"

func1()                          # Okay: "func1" and "func2" assigned
```

When this file is imported (or run as a standalone program), Python executes its statements from top to bottom. The first call to `func1` fails because the `func1 def` hasn't run yet. The call to `func2` inside `func1` works as long as `func2`'s `def` has been reached by the time `func1` is called (it hasn't when the second top-level `func1` call is run). The last call to `func1` at the bottom of the file works because `func1` and `func2` have both been assigned.

Mixing `def`s with top-level code is not only hard to read, it's dependent on statement ordering. As a rule of thumb, if you need to mix immediate code with `def`s, put your `def`s at the top of the file and your top-level code at the bottom. That way, your functions are guaranteed to be defined and assigned by the time code that uses them runs.

from Copies Names but Doesn't Link

Although it's commonly used, the `from` statement is the source of a variety of potential gotchas in Python. The `from` statement is really an assignment to names in the importer's scope—a name-copy operation, not a name aliasing. The implications of this are the same as for all assignments in Python, but they're subtle, especially given that the code that shares the objects lives in different files. For instance, suppose we define the following module, *nested1.py*:

```
# nested1.py
X = 99
def printer(): print(X)
```

If we import its two names using `from` in another module, *nested2.py*, we get copies of those names, not links to them. Changing a name in the importer resets only the binding of the local version of that name, not the name in *nested1.py*:

```
# nested2.py
from nested1 import X, printer     # Copy names out
X = 88                             # Changes my "X" only!
printer()                          # nested1's X is still 99

% python nested2.py
99
```

If we use `import` to get the whole module and then assign to a qualified name, however, we change the name in *nested1.py*. Qualification directs Python to a name in the module object, rather than a name in the importer, *nested3.py*:

```
# nested3.py
import nested1                  # Get module as a whole
nested1.X = 88                  # OK: change nested1's X
nested1.printer()

% python nested3.py
88
```

from * Can Obscure the Meaning of Variables

I mentioned this earlier but saved the details for here. Because you don't list the variables you want when using the `from module import *` statement form, it can accidentally overwrite names you're already using in your scope. Worse, it can make it difficult to determine where a variable comes from. This is especially true if the `from *` form is used on more than one imported file.

For example, if you use `from *` on three modules, you'll have no way of knowing what a raw function call really means, short of searching all three external module files (all of which may be in other directories):

```
>>> from module1 import *       # Bad: may overwrite my names silently
>>> from module2 import *       # Worse: no way to tell what we get!
>>> from module3 import *
>>> . . .

>>> func()                      # Huh???
```

The solution again is not to do this: try to explicitly list the attributes you want in your `from` statements, and restrict the `from *` form to at most one imported module per file. That way, any undefined names must by deduction be in the module named in the single `from *`. You can avoid the issue altogether if you always use `import` instead of `from`, but that advice is too harsh; like much else in programming, `from` is a convenient tool if used wisely. Even this example isn't an absolute evil—it's OK for a program to use this technique to collect names in a single space for convenience, as long as it's well known.

reload May Not Impact from Imports

Here's another `from`-related gotcha: as discussed previously, because `from` copies (assigns) names when run, there's no link back to the modules where the names came from. Names imported with `from` simply become references to objects, which happen to have been referenced by the same names in the importee when the `from` ran.

Because of this behavior, reloading the importee has no effect on clients that import its names using `from`. That is, the client's names will still reference the original objects fetched with `from`, even if the names in the original module are later reset:

```
from module import X          # X may not reflect any module reloads!
 . . .
from imp import reload
reload(module)                # Changes module, but not my names
X                             # Still references old object
```

To make reloads more effective, use `import` and name qualification instead of `from`. Because qualifications always go back to the module, they will find the new bindings of module names after reloading:

```
import module                 # Get module, not names
 . . .
from imp import reload
reload(module)                # Changes module in-place
module.X                      # Get current X: reflects module reloads
```

reload, from, and Interactive Testing

In fact, the prior gotcha is even more subtle than it appears. Chapter 3 warned that it's usually better not to launch programs with imports and reloads because of the complexities involved. Things get even worse when `from` is brought into the mix. Python beginners often stumble onto its issues in scenarios like the one outlined next. Say that after opening a module file in a text edit window, you launch an interactive session to load and test your module with `from`:

```
from module import function
function(1, 2, 3)
```

Finding a bug, you jump back to the edit window, make a change, and try to reload the module this way:

```
from imp import reload
reload(module)
```

This doesn't work, because the `from` statement assigned the name `function`, not `module`. To refer to the module in a `reload`, you have to first load it with an `import` statement at least once:

```
from imp import reload
import module
reload(module)
function(1, 2, 3)
```

However, this doesn't quite work either—`reload` updates the module object, but as discussed in the preceding section, names like `function` that were copied out of the module in the past still refer to the *old objects* (in this instance, the original version of the function). To really get the new function, you must refer to it as `module.function` after the `reload`, or rerun the `from`:

```
from imp import reload
import module
reload(module)
from module import function          # Or give up and use module.function()
function(1, 2, 3)
```

Now, the new version of the function will finally run.

As you can see, there are problems inherent in using `reload` with `from`: not only do you have to remember to reload after imports, but you also have to remember to rerun your `from` statements after reloads. This is complex enough to trip up even an expert once in a while. (In fact, the situation has gotten even worse in Python 3.0, because you must also remember to import `reload` itself!)

The short story is that you should not expect `reload` and `from` to play together nicely. The best policy is not to combine them at all—use `reload` with `import`, or launch your programs other ways, as suggested in Chapter 3: using the Run→Run Module menu option in IDLE, file icon clicks, system command lines, or the `exec` built-in function.

Recursive from Imports May Not Work

I saved the most bizarre (and, thankfully, obscure) gotcha for last. Because imports execute a file's statements from top to bottom, you need to be careful when using modules that import each other (known as *recursive imports*). Because the statements in a module may not all have been run when it imports another module, some of its names may not yet exist.

If you use `import` to fetch the module as a whole, this may or may not matter; the module's names won't be accessed until you later use qualification to fetch their values. But if you use `from` to fetch specific names, you must bear in mind that you will only have access to names in that module that have already been assigned.

For instance, take the following modules, `recur1` and `recur2`. `recur1` assigns a name `X`, and then imports `recur2` before assigning the name `Y`. At this point, `recur2` can fetch `recur1` as a whole with an `import` (it already exists in Python's internal modules table), but if it uses `from`, it will be able to see only the name `X`; the name `Y`, which is assigned below the `import` in `recur1`, doesn't yet exist, so you get an error:

```
# recur1.py
X = 1
import recur2                        # Run recur2 now if it doesn't exist
Y = 2

# recur2.py
from recur1 import X                 # OK: "X" already assigned
from recur1 import Y                 # Error: "Y" not yet assigned

C:\misc> C:\Python30\python
>>> import recur1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "recur1.py", line 2, in <module>
    import recur2
  File "recur2.py", line 2, in <module>
    from recur1 import Y
ImportError: cannot import name Y
```

Python avoids rerunning `recur1`'s statements when they are imported recursively from `recur2` (otherwise the imports would send the script into an infinite loop), but `recur1`'s namespace is incomplete when it's imported by `recur2`.

The solution? Don't use `from` in recursive imports (no, really!). Python won't get stuck in a cycle if you do, but your programs will once again be dependent on the order of the statements in the modules.

There are two ways out of this gotcha:

- You can usually eliminate import cycles like this by careful design—maximizing cohesion and minimizing coupling are good first steps.
- If you can't break the cycles completely, postpone module name accesses by using `import` and qualification (instead of `from`), or by running your `from`s either inside functions (instead of at the top level of the module), or near the bottom of your file to defer their execution.

Chapter Summary

This chapter surveyed some more advanced module-related concepts. We studied data hiding techniques, enabling new language features with the `__future__` module, the `__name__` usage mode variable, transitive reloads, importing by name strings, and more. We also explored and summarized module design issues and looked at common mistakes related to modules to help you avoid them in your code.

The next chapter begins our look at Python's object-oriented programming tool, the class. Much of what we've covered in the last few chapters will apply there, too—classes live in modules and are namespaces as well, but they add an extra component to attribute lookup called *inheritance search*. As this is the last chapter in this part of the book, however, before we dive into that topic, be sure to work through this part's set of lab exercises. And before that, here is this chapter's quiz to review the topics covered here.

Test Your Knowledge: Quiz

1. What is significant about variables at the top level of a module whose names begin with a single underscore?
2. What does it mean when a module's `__name__` variable is the string `"__main__"`?

3. If the user interactively types the name of a module to test, how can you import it?
4. How is changing `sys.path` different from setting `PYTHONPATH` to modify the module search path?
5. If the module `__future__` allows us to import from the future, can we also import from the past?

Test Your Knowledge: Answers

1. Variables at the top level of a module whose names begin with a single underscore are not copied out to the importing scope when the `from *` statement form is used. They can still be accessed by an `import` or the normal `from` statement form, though.
2. If a module's `__name__` variable is the string `"__main__"`, it means that the file is being executed as a top-level script instead of being imported from another file in the program. That is, the file is being used as a program, not a library.
3. User input usually comes into a script as a string; to import the referenced module given its string name, you can build and run an `import` statement with `exec`, or pass the string name in a call to the `__import__` function.
4. Changing `sys.path` only affects one running program, and is temporary—the change goes away when the program ends. `PYTHONPATH` settings live in the operating system—they are picked up globally by all programs on a machine, and changes to these settings endure after programs exit.
5. No, we can't import from the past in Python. We can install (or stubbornly use) an older version of the language, but the latest Python is generally the best Python.

Test Your Knowledge: Part V Exercises

See "Part V, Modules" on page 1119 in Appendix B for the solutions.

1. *Import basics*. Write a program that counts the lines and characters in a file (similar in spirit to *wc* on Unix). With your text editor, code a Python module called *mymod.py* that exports three top-level names:
   - A `countLines(name)` function that reads an input file and counts the number of lines in it (hint: `file.readlines` does most of the work for you, and `len` does the rest).
   - A `countChars(name)` function that reads an input file and counts the number of characters in it (hint: `file.read` returns a single string).
   - A `test(name)` function that calls both counting functions with a given input filename. Such a filename generally might be passed in, hardcoded, input with the `input` built-in function, or pulled from a command line via the `sys.argv` list shown in this chapter's *formats.py* example; for now, you can assume it's a passed-in function argument.

All three `mymod` functions should expect a filename string to be passed in. If you type more than two or three lines per function, you're working much too hard—use the hints I just gave!

Next, test your module interactively, using `import` and attribute references to fetch your exports. Does your `PYTHONPATH` need to include the directory where you created *mymod.py*? Try running your module on itself: e.g., `test("mymod.py")`. Note that `test` opens the file twice; if you're feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: `file.seek(0)` is a file rewind).

2. `from`/`from *`. Test your `mymod` module from exercise 1 interactively by using `from` to load the exports directly, first by name, then using the `from *` variant to fetch everything.
3. *__main__*. Add a line in your `mymod` module that calls the `test` function automatically only when the module is run as a script, not when it is imported. The line you add will probably test the value of `__name__` for the string `"__main__"`, as shown in this chapter. Try running your module from the system command line; then, import the module and test its functions interactively. Does it still work in both modes?
4. *Nested imports*. Write a second module, *myclient.py*, that imports `mymod` and tests its functions; then run `myclient` from the system command line. If `myclient` uses `from` to fetch from `mymod`, will `mymod`'s functions be accessible from the top level of `myclient`? What if it imports with `import` instead? Try coding both variations in `myclient` and test interactively by importing `myclient` and inspecting its `__dict__` attribute.
5. *Package imports*. Import your file from a package. Create a subdirectory called *mypkg* nested in a directory on your module import search path, move the *mymod.py* module file you created in exercise 1 or 3 into the new directory, and try to import it with a package import of the form `import mypkg.mymod`.

   You'll need to add an *__init__.py* file in the directory your module was moved to make this go, but it should work on all major Python platforms (that's part of the reason Python uses "." as a path separator). The package directory you create can be simply a subdirectory of the one you're working in; if it is, it will be found via the home directory component of the search path, and you won't have to configure your path. Add some code to your *__init__.py*, and see if it runs on each import.
6. *Reloads*. Experiment with module reloads: perform the tests in Chapter 22's *changer.py* example, changing the called function's message and/or behavior repeatedly, without stopping the Python interpreter. Depending on your system, you might be able to edit `changer` in another window, or suspend the Python interpreter and edit in the same window (on Unix, a Ctrl-Z key combination usually suspends the current process, and an `fg` command later resumes it).

7. *Circular imports*.‡ In the section on recursive import gotchas, importing `recur1` raised an error. But if you restart Python and import `recur2` interactively, the error doesn't occur—test this and see for yourself. Why do you think it works to import `recur2`, but not `recur1`? (Hint: Python stores new modules in the built-in `sys.modules` table—a dictionary—before running their code; later imports fetch the module from this table first, whether the module is "complete" yet or not.) Now, try running `recur1` as a top-level script file: **`python recur1.py`**. Do you get the same error that occurs when `recur1` is imported interactively? Why? (Hint: when modules are run as programs, they aren't imported, so this case has the same effect as importing `recur2` interactively; `recur2` is the first module imported.) What happens when you run `recur2` as a script?

‡ Note that circular imports are extremely rare in practice. On the other hand, if you can understand why they are a potential problem, you know a lot about Python's import semantics.

PART VI

Classes and OOP

CHAPTER 25
OOP: The Big Picture

So far in this book, we've been using the term "object" generically. Really, the code written up to this point has been *object-based*—we've passed objects around our scripts, used them in expressions, called their methods, and so on. For our code to qualify as being truly *object-oriented* (OO), though, our objects will generally need to also participate in something called an *inheritance hierarchy*.

This chapter begins our exploration of the Python *class*—a device used to implement new kinds of objects in Python that support inheritance. Classes are Python's main object-oriented programming (OOP) tool, so we'll also look at OOP basics along the way in this part of the book. OOP offers a different and often more effective way of looking at programming, in which we factor code to minimize redundancy, and write new programs by *customizing* existing code instead of changing it in-place.

In Python, classes are created with a new statement: the `class` statement. As you'll see, the objects defined with classes can look a lot like the built-in types we studied earlier in the book. In fact, classes really just apply and extend the ideas we've already covered; roughly, they are packages of functions that use and process built-in object types. Classes, though, are designed to create and manage new objects, and they also support *inheritance*—a mechanism of code customization and reuse above and beyond anything we've seen so far.

One note up front: in Python, OOP is entirely optional, and you don't need to use classes just to get started. In fact, you can get plenty of work done with simpler constructs such as functions, or even simple top-level script code. Because using classes well requires some up-front planning, they tend to be of more interest to people who work in *strategic* mode (doing long-term product development) than to people who work in *tactical* mode (where time is in very short supply).

Still, as you'll see in this part of the book, classes turn out to be one of the most useful tools Python provides. When used well, classes can actually cut development time radically. They're also employed in popular Python tools like the tkinter GUI API, so most Python programmers will usually find at least a working knowledge of class basics helpful.

Why Use Classes?

Remember when I told you that programs "do things with stuff"? In simple terms, classes are just a way to define new sorts of stuff, reflecting real objects in a program's domain. For instance, suppose we decide to implement that hypothetical pizza-making robot we used as an example in Chapter 16. If we implement it using classes, we can model more of its real-world structure and relationships. Two aspects of OOP prove useful here:

Inheritance
: Pizza-making robots are kinds of robots, so they possess the usual robot-y properties. In OOP terms, we say they "inherit" properties from the general category of all robots. These common properties need to be implemented only once for the general case and can be reused by all types of robots we may build in the future.

Composition
: Pizza-making robots are really collections of components that work together as a team. For instance, for our robot to be successful, it might need arms to roll dough, motors to maneuver to the oven, and so on. In OOP parlance, our robot is an example of composition; it contains other objects that it activates to do its bidding. Each component might be coded as a class, which defines its own behavior and relationships.

General OOP ideas like inheritance and composition apply to any application that can be decomposed into a set of objects. For example, in typical GUI systems, interfaces are written as collections of widgets—buttons, labels, and so on—which are all drawn when their container is drawn (*composition*). Moreover, we may be able to write our own custom widgets—buttons with unique fonts, labels with new color schemes, and the like—which are specialized versions of more general interface devices (*inheritance*).

From a more concrete programming perspective, classes are Python program units, just like functions and modules: they are another compartment for packaging logic and data. In fact, classes also define new namespaces, much like modules. But, compared to other program units we've already seen, classes have three critical distinctions that make them more useful when it comes to building new objects:

Multiple instances
: Classes are essentially factories for generating one or more objects. Every time we call a class, we generate a new object with a distinct namespace. Each object generated from a class has access to the class's attributes *and* gets a namespace of its own for data that varies per object.

Customization via inheritance
: Classes also support the OOP notion of inheritance; we can extend a class by redefining its attributes outside the class itself. More generally, classes can build up namespace hierarchies, which define names to be used by objects created from classes in the hierarchy.

Operator overloading
: By providing special protocol methods, classes can define objects that respond to the sorts of operations we saw at work on built-in types. For instance, objects made with classes can be sliced, concatenated, indexed, and so on. Python provides hooks that classes can use to intercept and implement any built-in type operation.

OOP from 30,000 Feet

Before we see what this all means in terms of code, I'd like to say a few words about the general ideas behind OOP. If you've never done anything object-oriented in your life before now, some of the terminology in this chapter may seem a bit perplexing on the first pass. Moreover, the motivation for these terms may be elusive until you've had a chance to study the ways that programmers apply them in larger systems. OOP is as much an experience as a technology.

Attribute Inheritance Search

The good news is that OOP is much simpler to understand and use in Python than in other languages, such as C++ or Java. As a dynamically typed scripting language, Python removes much of the syntactic clutter and complexity that clouds OOP in other tools. In fact, most of the OOP story in Python boils down to this expression:

```
object.attribute
```

We've been using this expression throughout the book to access module attributes, call methods of objects, and so on. When we say this to an object that is derived from a `class` statement, however, the expression kicks off a *search* in Python—it searches a tree of linked objects, looking for the first appearance of `attribute` that it can find. When classes are involved, the preceding Python expression effectively translates to the following in natural language:

> Find the first occurrence of `attribute` by looking in `object`, then in all classes above it, from bottom to top and left to right.

In other words, attribute fetches are simply tree searches. The term *inheritance* is applied because objects lower in a tree inherit attributes attached to objects higher in that tree. As the search proceeds from the bottom up, in a sense, the objects linked into a tree are the union of all the attributes defined in all their tree parents, all the way up the tree.

In Python, this is all very literal: we really do build up trees of linked objects with code, and Python really does climb this tree at runtime searching for attributes every time we use the `object.attribute` expression. To make this more concrete, Figure 25-1 sketches an example of one of these trees.

In this figure, there is a tree of five objects labeled with variables, all of which have attached attributes, ready to be searched. More specifically, this tree links together three

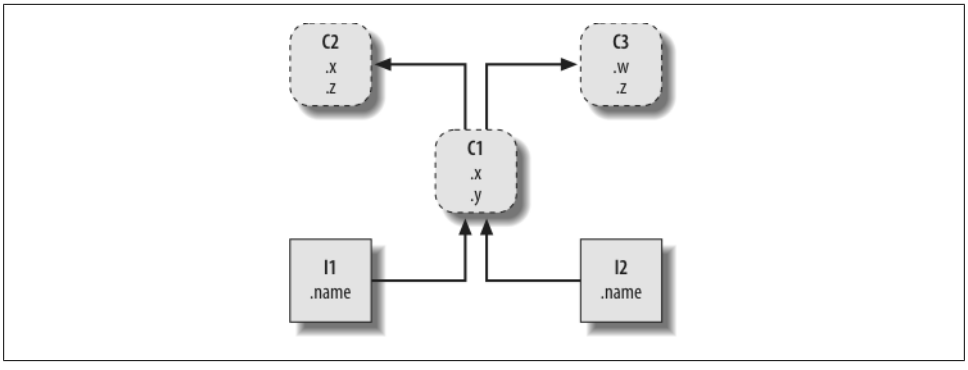

Figure 25-1. A class tree, with two instances at the bottom (I1 and I2), a class above them (C1), and two superclasses at the top (C2 and C3). All of these objects are namespaces (packages of variables), and the inheritance search is simply a search of the tree from bottom to top looking for the lowest occurrence of an attribute name. Code implies the shape of such trees.

class objects (the ovals `C1`, `C2`, and `C3`) and two *instance objects* (the rectangles `I1` and `I2`) into an inheritance search tree. Notice that in the Python object model, classes and the instances you generate from them are two distinct object types:

Classes
: Serve as instance factories. Their attributes provide behavior—data and functions—that is inherited by all the instances generated from them (e.g., a function to compute an employee's salary from pay and hours).

Instances
: Represent the concrete items in a program's domain. Their attributes record data that varies per specific object (e.g., an employee's Social Security number).

In terms of search trees, an instance inherits attributes from its class, and a class inherits attributes from all classes above it in the tree.

In Figure 25-1, we can further categorize the ovals by their relative positions in the tree. We usually call classes higher in the tree (like `C2` and `C3`) *superclasses*; classes lower in the tree (like `C1`) are known as *subclasses*.* These terms refer to relative tree positions and roles. Superclasses provide behavior shared by all their subclasses, but because the search proceeds from the bottom up, subclasses may override behavior defined in their superclasses by redefining superclass names lower in the tree.

As these last few words are really the crux of the matter of software customization in OOP, let's expand on this concept. Suppose we build up the tree in Figure 25-1, and then say this:

```
I2.w
```

* In other literature, you may also occasionally see the terms *base classes* and *derived classes* used to describe superclasses and subclasses, respectively.

Right away, this code invokes inheritance. Because this is an *`object.attribute`* expression, it triggers a search of the tree in Figure 25-1—Python will search for the attribute `w` by looking in `I2` and above. Specifically, it will search the linked objects in this order:

```
I2, C1, C2, C3
```

and stop at the first attached `w` it finds (or raise an error if `w` isn't found at all). In this case, `w` won't be found until `C3` is searched because it appears only in that object. In other words, `I2.w` resolves to `C3.w` by virtue of the automatic search. In OOP terminology, `I2` "inherits" the attribute `w` from `C3`.

Ultimately, the two instances inherit four attributes from their classes: `w`, `x`, `y`, and `z`. Other attribute references will wind up following different paths in the tree. For example:

- `I1.x` and `I2.x` both find `x` in `C1` and stop because `C1` is lower than `C2`.
- `I1.y` and `I2.y` both find `y` in `C1` because that's the only place `y` appears.
- `I1.z` and `I2.z` both find `z` in `C2` because `C2` is further to the left than `C3`.
- `I2.name` finds `name` in `I2` without climbing the tree at all.

Trace these searches through the tree in Figure 25-1 to get a feel for how inheritance searches work in Python.

The first item in the preceding list is perhaps the most important to notice—because `C1` redefines the attribute `x` lower in the tree, it effectively *replaces* the version above it in `C2`. As you'll see in a moment, such redefinitions are at the heart of software customization in OOP—by redefining and replacing the attribute, `C1` effectively customizes what it inherits from its superclasses.

Classes and Instances

Although they are technically two separate object types in the Python model, the classes and instances we put in these trees are almost identical—each type's main purpose is to serve as another kind of *namespace*—a package of variables, and a place where we can attach attributes. If classes and instances therefore sound like modules, they should; however, the objects in class trees also have automatically searched links to other namespace objects, and classes correspond to statements, not entire files.

The primary difference between classes and instances is that classes are a kind of *factory* for generating instances. For example, in a realistic application, we might have an `Employee` class that defines what it means to be an employee; from that class, we generate actual `Employee` instances. This is another difference between classes and modules: we only ever have one instance of a given module in memory (that's why we have to reload a module to get its new code), but with classes, we can make as many instances as we need.

Operationally, classes will usually have functions attached to them (e.g., `computeSalary`), and the instances will have more basic data items used by the class' functions (e.g., `hoursWorked`). In fact, the object-oriented model is not that different from the classic data-processing model of *programs* plus *records*; in OOP, instances are like records with "data," and classes are the "programs" for processing those records. In OOP, though, we also have the notion of an inheritance hierarchy, which supports software customization better than earlier models.

Class Method Calls

In the prior section, we saw how the attribute reference `I2.w` in our example class tree was translated to `C3.w` by the inheritance search procedure in Python. Perhaps just as important to understand as the inheritance of attributes, though, is what happens when we try to call methods (i.e., functions attached to classes as attributes).

If this `I2.w` reference is a *function* call, what it really means is "call the `C3.w` function to process `I2`." That is, Python will automatically map the call `I2.w()` into the call `C3.w(I2)`, passing in the instance as the first argument to the inherited function.

In fact, whenever we call a function attached to a class in this fashion, an instance of the class is always implied. This implied subject or context is part of the reason we refer to this as an *object-oriented* model—there is always a subject object when an operation is run. In a more realistic example, we might invoke a method called `giveRaise` attached as an attribute to an `Employee` class; such a call has no meaning unless qualified with the employee to whom the raise should be given.

As we'll see later, Python passes in the implied instance to a special first argument in the method, called `self` by convention. As we'll also learn, methods can be called through either an instance (e.g., `bob.giveRaise()`) or a class (e.g., `Employee.giveRaise(bob)`), and both forms serve purposes in our scripts. To see how methods receive their subjects, though, we need to move on to some code.

Coding Class Trees

Although we are speaking in the abstract here, there is tangible code behind all these ideas. We construct trees, and their objects with `class` statements and class calls, which we'll meet in more detail later. In short:

- Each `class` statement generates a new class object.
- Each time a class is called, it generates a new instance object.
- Instances are automatically linked to the classes from which they are created.
- Classes are linked to their superclasses by listing them in parentheses in a `class` header line; the left-to-right order there gives the order in the tree.

To build the tree in Figure 25-1, for example, we would run Python code of this form (I've omitted the guts of the `class` statements here):

```
class C2: ...                       # Make class objects (ovals)
class C3: ...
class C1(C2, C3): ...               # Linked to superclasses

I1 = C1()                           # Make instance objects (rectangles)
I2 = C1()                           # Linked to their classes
```

Here, we build the three class objects by running three `class` statements, and make the two instance objects by calling the class `C1` twice, as though it were a function. The instances remember the class they were made from, and the class `C1` remembers its listed superclasses.

Technically, this example is using something called *multiple inheritance*, which simply means that a class has more than one superclass above it in the class tree. In Python, if there is more than one superclass listed in parentheses in a `class` statement (like `C1`'s here), their left-to-right order gives the order in which those superclasses will be searched for attributes.

Because of the way inheritance searches proceed, the object to which you attach an attribute turns out to be crucial—it determines the name's scope. Attributes attached to instances pertain only to those single instances, but attributes attached to classes are shared by all their subclasses and instances. Later, we'll study the code that hangs attributes on these objects in depth. As we'll find:

- Attributes are usually attached to classes by assignments made within `class` statements, and not nested inside function `def` statements.
- Attributes are usually attached to instances by assignments to a special argument passed to functions inside classes, called `self`.

For example, classes provide behavior for their instances with functions created by coding `def` statements inside `class` statements. Because such nested `def`s assign names within the class, they wind up attaching attributes to the class object that will be inherited by all instances and subclasses:

```
class C1(C2, C3):                   # Make and link class C1
    def setname(self, who):         # Assign name: C1.setname
        self.name = who             # Self is either I1 or I2

I1 = C1()                           # Make two instances
I2 = C1()
I1.setname('bob')                   # Sets I1.name to 'bob'
I2.setname('mel')                   # Sets I2.name to 'mel'
print(I1.name)                      # Prints 'bob'
```

There's nothing syntactically unique about `def` in this context. Operationally, when a `def` appears inside a `class` like this, it is usually known as a *method*, and it automatically receives a special first argument—called `self` by convention—that provides a handle back to the instance to be processed.†

Because classes are factories for multiple instances, their methods usually go through this automatically passed-in `self` argument whenever they need to fetch or set attributes of the particular instance being processed by a method call. In the preceding code, `self` is used to store a name in one of two instances.

Like simple variables, attributes of classes and instances are not declared ahead of time, but spring into existence the first time they are assigned values. When a method assigns to a `self` attribute, it creates or changes an attribute in an instance at the bottom of the class tree (i.e., one of the rectangles) because `self` automatically refers to the instance being processed.

In fact, because all the objects in class trees are just namespace objects, we can fetch or set any of their attributes by going through the appropriate names. Saying `C1.setname` is as valid as saying `I1.setname`, as long as the names `C1` and `I1` are in your code's scopes.

As currently coded, our `C1` class doesn't attach a `name` attribute to an instance until the `setname` method is called. In fact, referencing `I1.name` before calling `I1.setname` would produce an undefined name error. If a class wants to guarantee that an attribute like `name` is always set in its instances, it more typically will fill out the attribute at construction time, like this:

```
class C1(C2, C3):
    def __init__(self, who):      # Set name when constructed
        self.name = who           # Self is either I1 or I2

I1 = C1('bob')                    # Sets I1.name to 'bob'
I2 = C1('mel')                    # Sets I2.name to 'mel'
print(I1.name)                    # Prints 'bob'
```

If it's coded and inherited, Python automatically calls a method named `__init__` each time an instance is generated from a class. The new instance is passed in to the `self` argument of `__init__` as usual, and any values listed in parentheses in the class call go to arguments two and beyond. The effect here is to initialize instances when they are made, without requiring extra method calls.

The `__init__` method is known as the *constructor* because of when it is run. It's the most commonly used representative of a larger class of methods called *operator overloading methods*, which we'll discuss in more detail in the chapters that follow. Such methods are inherited in class trees as usual and have double underscores at the start and end of their names to make them distinct. Python runs them automatically when instances that support them appear in the corresponding operations, and they are

† If you've ever used C++ or Java, you'll recognize that Python's `self` is the same as the `this` pointer, but `self` is always explicit in Python to make attribute accesses more obvious.

mostly an alternative to using simple method calls. They're also optional: if omitted, the operations are not supported.

For example, to implement set intersection, a class might either provide a method named `intersect`, or overload the `&` expression operator to dispatch to the required logic by coding a method named `__and__`. Because the operator scheme makes instances look and feel more like built-in types, it allows some classes to provide a consistent and natural interface, and be compatible with code that expects a built-in type.

OOP Is About Code Reuse

And that, along with a few syntax details, is most of the OOP story in Python. Of course, there's a bit more to it than just inheritance. For example, operator overloading is much more general than I've described so far—classes may also provide their own implementations of operations such as indexing, fetching attributes, printing, and more. By and large, though, OOP is about looking up attributes in trees.

So why would we be interested in building and searching trees of objects? Although it takes some experience to see how, when used well, classes support code *reuse* in ways that other Python program components cannot. With classes, we code by customizing existing software, instead of either changing existing code in-place or starting from scratch for each new project.

At a fundamental level, classes are really just packages of functions and other names, much like modules. However, the automatic attribute inheritance search that we get with classes supports customization of software above and beyond what we can do with modules and functions. Moreover, classes provide a natural structure for code that localizes logic and names, and so aids in debugging.

For instance, because methods are simply functions with a special first argument, we can mimic some of their behavior by manually passing objects to be processed to simple functions. The participation of methods in class inheritance, though, allows us to naturally customize existing software by coding subclasses with new method definitions, rather than changing existing code in-place. There is really no such concept with modules and functions.

As an example, suppose you're assigned the task of implementing an employee database application. As a Python OOP programmer, you might begin by coding a general superclass that defines default behavior common to all the kinds of employees in your organization:

```
class Employee:                         # General superclass
    def computeSalary(self): ...        # Common or default behavior
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Once you've coded this general behavior, you can specialize it for each specific kind of employee to reflect how the various types differ from the norm. That is, you can code subclasses that customize just the bits of behavior that differ per employee type; the rest of the employee types' behavior will be inherited from the more general class. For example, if engineers have a unique salary computation rule (i.e., not hours times rate), you can replace just that one method in a subclass:

```
class Engineer(Employee):            # Specialized subclass
     def computeSalary(self): ...    # Something custom here
```

Because the `computeSalary` version here appears lower in the class tree, it will replace (override) the general version in `Employee`. You then create instances of the kinds of employee classes that the real employees belong to, to get the correct behavior:

```
bob = Employee()                     # Default behavior
mel = Engineer()                     # Custom salary calculator
```

Notice that you can make instances of any class in a tree, not just the ones at the bottom—the class you make an instance from determines the level at which the attribute search will begin. Ultimately, these two instance objects might wind up embedded in a larger container object (e.g., a list, or an instance of another class) that represents a department or company using the composition idea mentioned at the start of this chapter.

When you later ask for these employees' salaries, they will be computed according to the classes from which the objects were made, due to the principles of the inheritance search:‡

```
company = [bob, mel]                 # A composite object
for emp in company:
    print(emp.computeSalary())       # Run this object's version
```

This is yet another instance of the idea of *polymorphism* introduced in Chapter 4 and revisited in Chapter 16. Recall that polymorphism means that the meaning of an operation depends on the object being operated on. Here, the method `computeSalary` is located by inheritance search in each object before it is called. In other applications, polymorphism might also be used to hide (i.e., *encapsulate*) interface differences. For example, a program that processes data streams might be coded to expect objects with input and output methods, without caring what those methods actually do:

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
```

‡ Note that the `company` list in this example could be stored in a file with Python object pickling, introduced in Chapter 9 when we met files, to yield a persistent employee database. Python also comes with a module named `shelve`, which would allow you to store the pickled representation of the class instances in an access-by-key filesystem; the third-party open source ZODB system does the same but has better support for production-quality object-oriented databases.

```
        data = converter(data)
        writer.write(data)
```

By passing in instances of subclasses that specialize the required `read` and `write` method interfaces for various data sources, we can reuse the `processor` function for any data source we need to use, both now and in the future:

```
class Reader:
    def read(self): ...               # Default behavior and tools
    def other(self): ...
class FileReader(Reader):
    def read(self): ...               # Read from a local file
class SocketReader(Reader):
    def read(self): ...               # Read from a network socket
...
processor(FileReader(...),   Converter,  FileWriter(...))
processor(SocketReader(...), Converter,  TapeWriter(...))
processor(FtpReader(...),    Converter,  XmlWriter(...))
```

Moreover, because the internal implementations of those `read` and `write` methods have been factored into single locations, they can be changed without impacting code such as this that uses them. In fact, the `processor` function might itself be a class to allow the conversion logic of `converter` to be filled in by inheritance, and to allow readers and writers to be embedded by composition (we'll see how this works later in this part of the book).

Once you get used to programming this way (by software customization), you'll find that when it's time to write a new program, much of your work may already be done—your task largely becomes one of mixing together existing superclasses that already implement the behavior required by your program. For example, someone else might have written the `Employee`, `Reader`, and `Writer` classes in this example for use in a completely different program. If so, you get all of that person's code "for free."

In fact, in many application domains, you can fetch or purchase collections of superclasses, known as *frameworks*, that implement common programming tasks as classes, ready to be mixed into your applications. These frameworks might provide database interfaces, testing protocols, GUI toolkits, and so on. With frameworks, you often simply code a subclass that fills in an expected method or two; the framework classes higher in the tree do most of the work for you. Programming in such an OOP world is just a matter of combining and specializing already debugged code by writing subclasses of your own.

Of course, it takes a while to learn how to leverage classes to achieve such OOP utopia. In practice, object-oriented work also entails substantial design work to fully realize the code reuse benefits of classes—to this end, programmers have begun cataloging common OOP structures, known as *design patterns*, to help with design issues. The actual code you write to do OOP in Python, though, is so simple that it will not in itself pose an additional obstacle to your OOP quest. To see why, you'll have to move on to Chapter 26.

Chapter Summary

We took an abstract look at classes and OOP in this chapter, taking in the big picture before we dive into syntax details. As we've seen, OOP is mostly about looking up attributes in trees of linked objects; we call this lookup an inheritance search. Objects at the bottom of the tree inherit attributes from objects higher up in the tree—a feature that enables us to program by customizing code, rather than changing it, or starting from scratch. When used well, this model of programming can cut development time radically.

The next chapter will begin to fill in the coding details behind the picture painted here. As we get deeper into Python classes, though, keep in mind that the OOP model in Python is very simple; as I've already stated, it's really just about looking up attributes in object trees. Before we move on, here's a quick quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is the main point of OOP in Python?
2. Where does an inheritance search look for an attribute?
3. What is the difference between a class object and an instance object?
4. Why is the first argument in a class method function special?
5. What is the `__init__` method used for?
6. How do you create a class instance?
7. How do you create a class?
8. How do you specify a class's superclasses?

Test Your Knowledge: Answers

1. OOP is about code reuse—you factor code to minimize redundancy and program by customizing what already exists instead of changing code in-place or starting from scratch.
2. An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher superclasses, progressing from the bottom to the top of the object tree, and from left to right (by default). The search stops at the first place the attribute is found. Because the lowest version of a name found along the way wins, class hierarchies naturally support customization by extension.

3. Both class and instance objects are namespaces (packages of variables that appear as attributes). The main difference between them is that classes are a kind of factory for creating multiple instances. Classes also support operator overloading methods, which instances inherit, and treat any functions nested within them as special methods for processing instances.
4. The first argument in a class method function is special because it always receives the instance object that is the implied subject of the method call. It's usually called `self` by convention. Because method functions always have this implied subject object context by default, we say they are "object-oriented"—i.e., designed to process or change objects.
5. If the `__init__` method is coded or inherited in a class, Python calls it automatically each time an instance of that class is created. It's known as the constructor method; it is passed the new instance implicitly, as well as any arguments passed explicitly to the class name. It's also the most commonly used operator overloading method. If no `__init__` method is present, instances simply begin life as empty namespaces.
6. You create a class instance by calling the class name as though it were a function; any arguments passed into the class name show up as arguments two and beyond in the `__init__` constructor method. The new instance remembers the class it was created from for inheritance purposes.
7. You create a class by running a `class` statement; like function definitions, these statements normally run when the enclosing module file is imported (more on this in the next chapter).
8. You specify a class's superclasses by listing them in parentheses in the `class` statement, after the new class's name. The left-to-right order in which the classes are listed in the parentheses gives the left-to-right inheritance search order in the class tree.

CHAPTER 26

Class Coding Basics

Now that we've talked about OOP in the abstract, it's time to see how this translates to actual code. This chapter begins to fill in the syntax details behind the class model in Python.

If you've never been exposed to OOP in the past, classes can seem somewhat complicated if taken in a single dose. To make class coding easier to absorb, we'll begin our detailed exploration of OOP by taking a first look at some basic classes in action in this chapter. We'll expand on the details introduced here in later chapters of this part of the book, but in their basic form, Python classes are easy to understand.

In fact, classes have just three primary distinctions. At a base level, they are mostly just namespaces, much like the modules we studied in Part V. Unlike modules, though, classes also have support for generating multiple objects, for namespace inheritance, and for operator overloading. Let's begin our `class` statement tour by exploring each of these three distinctions in turn.

Classes Generate Multiple Instance Objects

To understand how the multiple objects idea works, you have to first understand that there are two kinds of objects in Python's OOP model: *class* objects and *instance* objects. Class objects provide default behavior and serve as factories for instance objects. Instance objects are the real objects your programs process—each is a namespace in its own right, but inherits (i.e., has automatic access to) names in the class from which it was created. Class objects come from statements, and instances come from calls; each time you call a class, you get a new instance of that class.

This object-generation concept is very different from any of the other program constructs we've seen so far in this book. In effect, classes are essentially *factories* for generating multiple instances. By contrast, only one copy of each module is ever imported into a single program (in fact, one reason that we have to call `imp.reload` is to update the single module object so that changes are reflected once they've been made).

The following is a quick summary of the bare essentials of Python OOP. As you'll see, Python classes are in some ways similar to both `def`s and modules, but they may be quite different from what you're used to in other languages.

Class Objects Provide Default Behavior

When we run a `class` statement, we get a class object. Here's a rundown of the main properties of Python classes:

- **The `class` statement creates a class object and assigns it a name**. Just like the function `def` statement, the Python `class` statement is an *executable* statement. When reached and run, it generates a new class object and assigns it to the name in the `class` header. Also, like `def`s, `class` statements typically run when the files they are coded in are first imported.
- **Assignments inside `class` statements make class attributes**. Just like in module files, top-level assignments within a `class` statement (not nested in a `def`) generate attributes in a class object. Technically, the `class` statement scope *morphs* into the attribute namespace of the class object, just like a module's global scope. After running a `class` statement, class attributes are accessed by name qualification: *`object.name`*.
- **Class attributes provide object state and behavior**. Attributes of a class object record state information and behavior to be shared by all instances created from the class; function `def` statements nested inside a `class` generate *methods*, which process instances.

Instance Objects Are Concrete Items

When we call a class object, we get an instance object. Here's an overview of the key points behind class instances:

- **Calling a class object like a function makes a new instance object**. Each time a class is called, it creates and returns a new instance object. Instances represent concrete items in your program's domain.
- **Each instance object inherits class attributes and gets its own namespace**. Instance objects created from classes are new namespaces; they start out empty but inherit attributes that live in the class objects from which they were generated.

- **Assignments to attributes of `self` in methods make per-instance attributes**. Inside class method functions, the first argument (called `self` by convention) references the instance object being processed; assignments to attributes of `self` create or change data in the instance, not the class.

A First Example

Let's turn to a real example to show how these ideas work in practice. To begin, let's define a class named `FirstClass` by running a Python `class` statement interactively:

```
>>> class FirstClass:                  # Define a class object
...     def setdata(self, value):      # Define class methods
...         self.data = value          # self is the instance
...     def display(self):
...         print(self.data)           # self.data: per instance
...
```

We're working interactively here, but typically, such a statement would be run when the module file it is coded in is imported. Like functions created with `def`s, this class won't even exist until Python reaches and runs this statement.

Like all compound statements, the `class` starts with a header line that lists the class name, followed by a body of one or more nested and (usually) indented statements. Here, the nested statements are `def`s; they define functions that implement the behavior the class means to export.

As we learned in Part IV, `def` is really an assignment. Here, it assigns function objects to the names `setdata` and `display` in the `class` statement's scope, and so generates attributes attached to the class: `FirstClass.setdata` and `FirstClass.display`. In fact, any name assigned at the top level of the class's nested block becomes an attribute of the class.

Functions inside a class are usually called *methods*. They're coded with normal `def`s, and they support everything we've learned about functions already (they can have defaults, return values, and so on). But in a method function, the first argument automatically receives an implied instance object when called—the subject of the call. We need to create a couple of instances to see how this works:

```
>>> x = FirstClass()                   # Make two instances
>>> y = FirstClass()                   # Each is a new namespace
```

By *calling* the class this way (notice the parentheses), we generate instance objects, which are just namespaces that have access to their classes' attributes. Properly speaking, at this point, we have three objects: two instances and a class. Really, we have three linked namespaces, as sketched in Figure 26-1. In OOP terms, we say that `x` "is a" `FirstClass`, as is `y`.

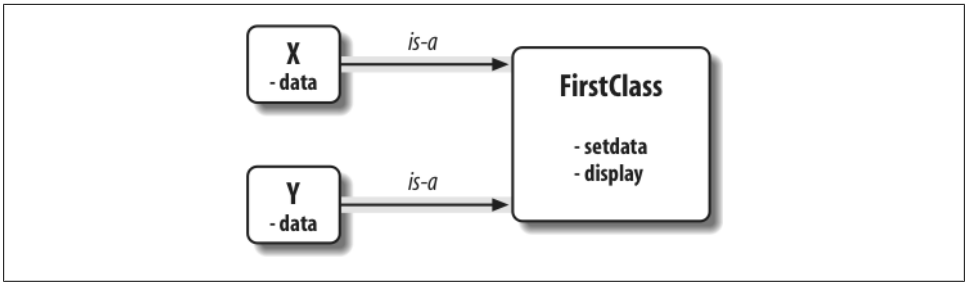

Figure 26-1. Classes and instances are linked namespace objects in a class tree that is searched by inheritance. Here, the "data" attribute is found in instances, but "setdata" and "display" are in the class above them.

The two instances start out empty but have links back to the class from which they were generated. If we qualify an instance with the name of an attribute that lives in the class object, Python fetches the name from the class by inheritance search (unless it also lives in the instance):

```
>>> x.setdata("King Arthur")        # Call methods: self is x
>>> y.setdata(3.14159)              # Runs: FirstClass.setdata(y, 3.14159)
```

Neither `x` nor `y` has a `setdata` attribute of its own, so to find it, Python follows the link from instance to class. And that's about all there is to inheritance in Python: it happens at attribute qualification time, and it just involves looking up names in linked objects (e.g., by following the is-a links in Figure 26-1).

In the `setdata` function inside `FirstClass`, the value passed in is assigned to `self.data`. Within a method, `self`—the name given to the leftmost argument by convention—automatically refers to the instance being processed (`x` or `y`), so the assignments store values in the instances' namespaces, not the class's (that's how the `data` names in Figure 26-1 are created).

Because classes can generate multiple instances, methods must go through the `self` argument to get to the instance to be processed. When we call the class's `display` method to print `self.data`, we see that it's different in each instance; on the other hand, the name `display` itself is the same in `x` and `y`, as it comes (is inherited) from the class:

```
>>> x.display()                     # self.data differs in each instance
King Arthur
>>> y.display()
3.14159
```

Notice that we stored different object types in the `data` member in each instance (a string, and a floating point). As with everything else in Python, there are no declarations for instance attributes (sometimes called *members*); they spring into existence the first time they are assigned values, just like simple variables. In fact, if we were to call `display` on one of our instances before calling `setdata`, we would trigger an undefined name error—the attribute named `data` doesn't even exist in memory until it is assigned within the `setdata` method.

As another way to appreciate how dynamic this model is, consider that we can change instance attributes in the class itself, by assigning to `self` in methods, or outside the class, by assigning to an explicit instance object:

```
>>> x.data = "New value"          # Can get/set attributes
>>> x.display()                   # Outside the class too
New value
```

Although less common, we could even generate a brand new attribute in the instance's namespace by assigning to its name outside the class's method functions:

```
>>> x.anothername = "spam"        # Can set new attributes here too!
```

This would attach a new attribute called `anothername`, which may or may not be used by any of the class's methods, to the instance object `x`. Classes usually create all of the instance's attributes by assignment to the `self` argument, but they don't have to; programs can fetch, change, or create attributes on any objects to which they have references.

Classes Are Customized by Inheritance

Besides serving as factories for generating multiple instance objects, classes also allow us to make changes by introducing new components (called *subclasses*), instead of changing existing components in-place. Instance objects generated from a class inherit the class's attributes. Python also allows classes to inherit from other classes, opening the door to coding *hierarchies* of classes that specialize behavior—by redefining attributes in subclasses that appear lower in the hierarchy, we override the more general definitions of those attributes higher in the tree. In effect, the further down the hierarchy we go, the more specific the software becomes. Here, too, there is no parallel with modules: their attributes live in a single, flat namespace that is not as amenable to customization.

In Python, instances inherit from classes, and classes inherit from superclasses. Here are the key ideas behind the machinery of attribute inheritance:

- **Superclasses are listed in parentheses in a `class` header**. To inherit attributes from another class, just list the class in parentheses in a `class` statement's header. The class that inherits is usually called a *subclass*, and the class that is inherited from is its *superclass*.
- **Classes inherit attributes from their superclasses**. Just as instances inherit the attribute names defined in their classes, classes inherit all the attribute names defined in their superclasses; Python finds them automatically when they're accessed, if they don't exist in the subclasses.
- **Instances inherit attributes from all accessible classes**. Each instance gets names from the class it's generated from, as well as all of that class's superclasses. When looking for a name, Python checks the instance, then its class, then all superclasses.

- **Each *`object.attribute`* reference invokes a new, independent search**. Python performs an independent search of the class tree for each attribute fetch expression. This includes references to instances and classes made outside `class` statements (e.g., `X.attr`), as well as references to attributes of the `self` instance argument in class method functions. Each `self.attr` expression in a method invokes a new search for `attr` in `self` and above.
- **Logic changes are made by subclassing, not by changing superclasses**. By redefining superclass names in subclasses lower in the hierarchy (class tree), subclasses replace and thus customize inherited behavior.

The net effect, and the main purpose of all this searching, is that classes support factoring and customization of code better than any other language tool we've seen so far. On the one hand, they allow us to minimize code redundancy (and so reduce maintenance costs) by factoring operations into a single, shared implementation; on the other, they allow us to program by customizing what already exists, rather than changing it in-place or starting from scratch.

A Second Example

To illustrate the role of inheritance, this next example builds on the previous one. First, we'll define a new class, `SecondClass`, that inherits all of `FirstClass`'s names and provides one of its own:

```
>>> class SecondClass(FirstClass):                  # Inherits setdata
...     def display(self):                          # Changes display
...         print('Current value = "%s"' % self.data)
...
```

`SecondClass` defines the `display` method to print with a different format. By defining an attribute with the same name as an attribute in `FirstClass`, `SecondClass` effectively replaces the `display` attribute in its superclass.

Recall that inheritance searches proceed upward from instances, to subclasses, to superclasses, stopping at the first appearance of the attribute name that it finds. In this case, since the `display` name in `SecondClass` will be found before the one in `FirstClass`, we say that `SecondClass` *overrides* `FirstClass`'s `display`. Sometimes we call this act of replacing attributes by redefining them lower in the tree *overloading*.

The net effect here is that `SecondClass` specializes `FirstClass` by changing the behavior of the `display` method. On the other hand, `SecondClass` (and any instances created from it) still inherits the `setdata` method in `FirstClass` verbatim. Let's make an instance to demonstrate:

```
>>> z = SecondClass()
>>> z.setdata(42)           # Finds setdata in FirstClass
>>> z.display()             # Finds overridden method in SecondClass
Current value = "42"
```

As before, we make a `SecondClass` instance object by calling it. The `setdata` call still runs the version in `FirstClass`, but this time the `display` attribute comes from `SecondClass` and prints a custom message. Figure 26-2 sketches the namespaces involved.

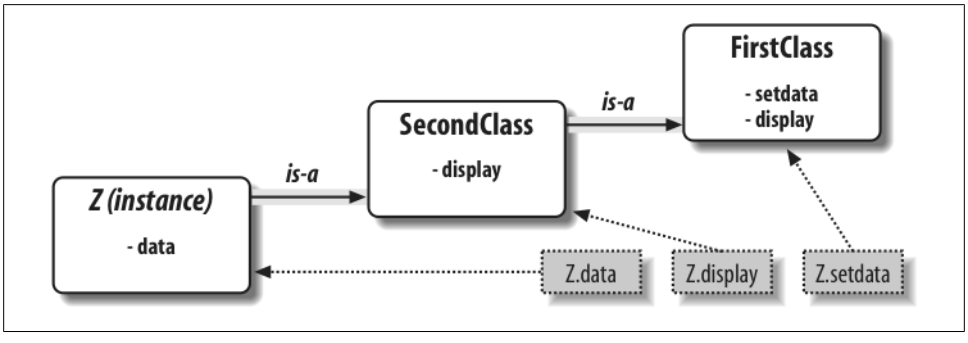

Figure 26-2. Specialization by overriding inherited names by redefining them in extensions lower in the class tree. Here, SecondClass redefines and so customizes the "display" method for its instances.

Now, here's a very important thing to notice about OOP: the specialization introduced in `SecondClass` is completely *external* to `FirstClass`. That is, it doesn't affect existing or future `FirstClass` objects, like the `x` from the prior example:

```
>>> x.display()             # x is still a FirstClass instance (old message)
New value
```

Rather than *changing* `FirstClass`, we *customized* it. Naturally, this is an artificial example, but as a rule, because inheritance allows us to make changes like this in external components (i.e., in subclasses), classes often support extension and reuse better than functions or modules can.

Classes Are Attributes in Modules

Before we move on, remember that there's nothing magic about a class name. It's just a variable assigned to an object when the `class` statement runs, and the object can be referenced with any normal expression. For instance, if our `FirstClass` was coded in a module file instead of being typed interactively, we could import it and use its name normally in a `class` header line:

```
from modulename import FirstClass          # Copy name into my scope
class SecondClass(FirstClass):             # Use class name directly
    def display(self): ...
```

Or, equivalently:

```
import modulename                           # Access the whole module
class SecondClass(modulename.FirstClass):  # Qualify to reference
    def display(self): ...
```

Like everything else, class names always live within a module, so they must follow all the rules we studied in Part V. For example, more than one class can be coded in a single module file—like other statements in a module, `class` statements are run during imports to define names, and these names become distinct module attributes. More generally, each module may arbitrarily mix any number of variables, functions, and classes, and all names in a module behave the same way. The file *food.py* demonstrates:

```
# food.py
var = 1                                 # food.var
def func():                             # food.func
    ...
class spam:                             # food.spam
    ...
class ham:                              # food.ham
    ...
class eggs:                             # food.eggs
    ...
```

This holds true even if the module and class happen to have the same name. For example, given the following file, *person.py*:

```
class person:
   ...
```

we need to go through the module to fetch the class as usual:

```
import person                           # Import module
x = person.person()                     # Class within module
```

Although this path may look redundant, it's required: `person.person` refers to the `person` class inside the `person` module. Saying just `person` gets the module, not the class, unless the `from` statement is used:

```
from person import person               # Get class from module
x = person()                            # Use class name
```

As with any other variable, we can never see a class in a file without first importing and somehow fetching it from its enclosing file. If this seems confusing, don't use the same name for a module and a class within it. In fact, common convention in Python dictates that class names should begin with an uppercase letter, to help make them more distinct:

```
import person                           # Lowercase for modules
x = person.Person()                     # Uppercase for classes
```

Also, keep in mind that although classes and modules are both namespaces for attaching attributes, they correspond to very different source code structures: a module reflects an entire *file*, but a class is a *statement* within a file. We'll say more about such distinctions later in this part of the book.

Classes Can Intercept Python Operators

Let's move on to the third major difference between classes and modules: operator overloading. In simple terms, *operator overloading* lets objects coded with classes intercept and respond to operations that work on built-in types: addition, slicing, printing, qualification, and so on. It's mostly just an automatic dispatch mechanism—expressions and other built-in operations route control to implementations in classes. Here, too, there is nothing similar in modules: modules can implement function calls, but not the behavior of expressions.

Although we could implement all class behavior as method functions, operator overloading lets objects be more tightly integrated with Python's object model. Moreover, because operator overloading makes our own objects act like built-ins, it tends to foster object interfaces that are more consistent and easier to learn, and it allows class-based objects to be processed by code written to expect a built-in type's interface. Here is a quick rundown of the main ideas behind overloading operators:

- **Methods named with double underscores (`__X__`) are special hooks**. Python operator overloading is implemented by providing specially named methods to intercept operations. The Python language defines a fixed and unchangeable mapping from each of these operations to a specially named method.
- **Such methods are called automatically when instances appear in built-in operations**. For instance, if an instance object inherits an `__add__` method, that method is called whenever the object appears in a `+` expression. The method's return value becomes the result of the corresponding expression.
- **Classes may override most built-in type operations**. There are dozens of special operator overloading method names for intercepting and implementing nearly every operation available for built-in types. This includes expressions, but also basic operations like printing and object creation.
- **There are no defaults for operator overloading methods, and none are required**. If a class does not define or inherit an operator overloading method, it just means that the corresponding operation is not supported for the class's instances. If there is no `__add__`, for example, `+` expressions raise exceptions.
- **Operators allow classes to integrate with Python's object model**. By overloading type operations, user-defined objects implemented with classes can act just like built-ins, and so provide consistency as well as compatibility with expected interfaces.

Operator overloading is an optional feature; it's used primarily by people developing tools for other Python programmers, not by application developers. And, candidly, you probably shouldn't try to use it just because it seems "cool." Unless a class needs to mimic built-in type interfaces, it should usually stick to simpler named methods. Why would an employee database application support expressions like * and +, for example? Named methods like `giveRaise` and `promote` would usually make more sense.

Because of this, we won't go into details on every operator overloading method available in Python in this book. Still, there is one operator overloading method you are likely to see in almost every realistic Python class: the `__init__` method, which is known as the *constructor* method and is used to initialize objects' state. You should pay special attention to this method, because `__init__`, along with the `self` argument, turns out to be a key requirement to understanding most OOP code in Python.

A Third Example

On to another example. This time, we'll define a subclass of `SecondClass` that implements three specially named attributes that Python will call automatically:

- `__init__` is run when a new instance object is created (`self` is the new `ThirdClass` object).*
- `__add__` is run when a `ThirdClass` instance appears in a `+` expression.
- `__str__` is run when an object is printed (technically, when it's converted to its print string by the `str` built-in function or its Python internals equivalent).

Our new subclass also defines a normally named method named `mul`, which changes the instance object in-place. Here's the new subclass:

```
>>> class ThirdClass(SecondClass):                  # Inherit from SecondClass
...     def __init__(self, value):                  # On "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                   # On "self + other"
...         return ThirdClass(self.data + other)
...     def __str__(self):                          # On "print(self)", "str()"
...         return '[ThirdClass: %s]' % self.data
...     def mul(self, other):                       # In-place change: named
...         self.data *= other
...
>>> a = ThirdClass('abc')           # __init__ called
>>> a.display()                     # Inherited method called
Current value = "abc"
>>> print(a)                        # __str__: returns display string
[ThirdClass: abc]

>>> b = a + 'xyz'                   # __add__: makes a new instance
>>> b.display()                     # b has all ThirdClass methods
Current value = "abcxyz"
>>> print(b)                        # __str__: returns display string
[ThirdClass: abcxyz]

>>> a.mul(3)                        # mul: changes instance in-place
>>> print(a)
[ThirdClass: abcabcabc]
```

* Not to be confused with the *__init__.py* files in module packages! See Chapter 23 for more details.

`ThirdClass` "is a" `SecondClass`, so its instances inherit the customized `display` method from `SecondClass`. This time, though, `ThirdClass` creation calls pass an argument (e.g., "abc"). This argument is passed to the `value` argument in the `__init__` constructor and assigned to `self.data` there. The net effect is that `ThirdClass` arranges to set the `data` attribute automatically at construction time, instead of requiring `setdata` calls after the fact.

Further, `ThirdClass` objects can now show up in `+` expressions and `print` calls. For `+`, Python passes the instance object on the left to the `self` argument in `__add__` and the value on the right to `other`, as illustrated in Figure 26-3; whatever `__add__` returns becomes the result of the `+` expression. For `print`, Python passes the object being printed to `self` in `__str__`; whatever string this method returns is taken to be the print string for the object. With `__str__` we can use a normal `print` to display objects of this class, instead of calling the special `display` method.

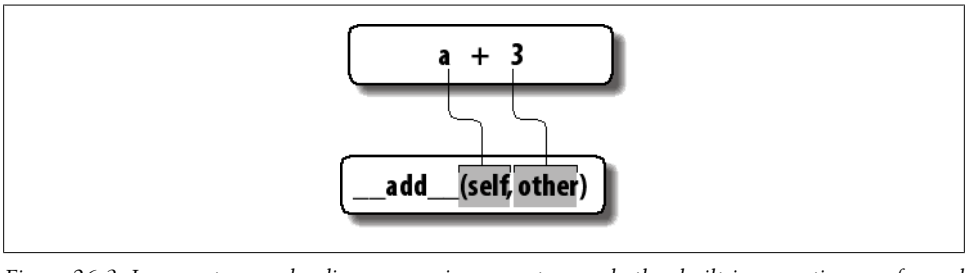

Figure 26-3. In operator overloading, expression operators and other built-in operations performed on class instances are mapped back to specially named methods in the class. These special methods are optional and may be inherited as usual. Here, a + expression triggers the __add__ method.

Specially named methods such as `__init__`, `__add__`, and `__str__` are inherited by subclasses and instances, just like any other names assigned in a `class`. If they're not coded in a class, Python looks for such names in all its superclasses, as usual. Operator overloading method names are also not built-in or reserved words; they are just attributes that Python looks for when objects appear in various contexts. Python usually calls them automatically, but they may occasionally be called by your code as well; the `__init__` method, for example, is often called manually to trigger superclass constructors (more on this later).

Notice that the `__add__` method makes and returns a *new* instance object of its class, by calling `ThirdClass` with the result value. By contrast, `mul` *changes* the current instance object in-place, by reassigning the `self` attribute. We could overload the * expression to do the latter, but this would be too different from the behavior of * for built-in types such as numbers and strings, for which it always makes new objects. Common practice dictates that overloaded operators should work the same way that built-in operator implementations do. Because operator overloading is really just an expression-to-method dispatch mechanism, though, you can interpret operators any way you like in your own class objects.

Why Use Operator Overloading?

As a class designer, you can choose to use operator overloading or not. Your choice simply depends on how much you want your object to look and feel like built-in types. As mentioned earlier, if you omit an operator overloading method and do not inherit it from a superclass, the corresponding operation will not be supported for your instances; if it's attempted, an exception will be thrown (or a standard default will be used).

Frankly, many operator overloading methods tend to be used only when implementing objects that are mathematical in nature; a vector or matrix class may overload the addition operator, for example, but an employee class likely would not. For simpler classes, you might not use overloading at all, and would rely instead on explicit method calls to implement your objects' behavior.

On the other hand, you might decide to use operator overloading if you need to pass a user-defined object to a function that was coded to expect the operators available on a built-in type like a list or a dictionary. Implementing the same operator set in your class will ensure that your objects support the same expected object interface and so are compatible with the function. Although we won't cover every operator overloading method in this book, we'll see some additional operator overloading techniques in action in Chapter 29.

One overloading method we will explore here is the `__init__` constructor method, which seems to show up in almost every realistic class. Because it allows classes to fill out the attributes in their newly created instances immediately, the constructor is useful for almost every kind of class you might code. In fact, even though instance attributes are not declared in Python, you can usually find out which attributes an instance will have by inspecting its class's `__init__` method.

The World's Simplest Python Class

We've begun studying `class` statement syntax in detail in this chapter, but I'd again like to remind you that the basic inheritance model that classes produce is very simple—all it really involves is searching for attributes in trees of linked objects. In fact, we can create a class with nothing in it at all. The following statement makes a class with no attributes attached (an empty namespace object):

```
>>> class rec: pass                # Empty namespace object
```

We need the no-operation `pass` statement (discussed in Chapter 13) here because we don't have any methods to code. After we make the class by running this statement interactively, we can start attaching attributes to the class by assigning names to it completely outside of the original `class` statement:

```
>>> rec.name = 'Bob'               # Just objects with attributes
>>> rec.age  = 40
```

And, after we've created these attributes by assignment, we can fetch them with the usual syntax. When used this way, a class is roughly similar to a "struct" in C, or a "record" in Pascal. It's basically an object with field names attached to it (we can do similar work with dictionary keys, but it requires extra characters):

```
>>> print(rec.name)                # Like a C struct or a record
Bob
```

Notice that this works even though there are no instances of the class yet; classes are objects in their own right, even without instances. In fact, they are just self-contained namespaces, so as long as we have a reference to a class, we can set or change its attributes anytime we wish. Watch what happens when we do create two instances, though:

```
>>> x = rec()                      # Instances inherit class names
>>> y = rec()
```

These instances begin their lives as completely empty namespace objects. Because they remember the class from which they were made, though, they will obtain the attributes we attached to the class by inheritance:

```
>>> x.name, y.name                 # name is stored on the class only
('Bob', 'Bob')
```

Really, these instances have no attributes of their own; they simply fetch the `name` attribute from the class object where it is stored. If we do assign an attribute to an instance, though, it creates (or changes) the attribute in that object, and no other—attribute references kick off inheritance searches, but attribute assignments affect only the objects in which the assignments are made. Here, `x` gets its own `name`, but `y` still inherits the `name` attached to the class above it:

```
>>> x.name = 'Sue'                 # But assignment changes x only
>>> rec.name, x.name, y.name
('Bob', 'Sue', 'Bob')
```

In fact, as we'll explore in more detail in Chapter 28, the attributes of a namespace object are usually implemented as dictionaries, and class inheritance trees are (generally speaking) just dictionaries with links to other dictionaries. If you know where to look, you can see this explicitly.

For example, the `__dict__` attribute is the namespace dictionary for most class-based objects (some classes may also define attributes in `__slots__`, an advanced and seldom-used feature that we'll study in Chapters 30 and 31). The following was run in Python 3.0; the order of names and set of `__X__` internal names present can vary from release to release, but the names we assigned are present in all:

```
>>> rec.__dict__.keys()
['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__']

>>> list(x.__dict__.keys())
['name']
```

```
>>> list(y.__dict__.keys())          # list() not required in Python 2.6
[]
```

Here, the class's namespace dictionary shows the `name` and `age` attributes we assigned to it, `x` has its own `name`, and `y` is still empty. Each instance has a link to its class for inheritance, though—it's called `__class__`, if you want to inspect it:

```
>>> x.__class__
<class '__main__.rec'>
```

Classes also have a `__bases__` attribute, which is a tuple of their superclasses:

```
>>> rec.__bases__                    # () empty tuple in Python 2.6
(<class 'object'>,)
```

These two attributes are how class trees are literally represented in memory by Python.

The main point to take away from this look under the hood is that Python's class model is extremely dynamic. Classes and instances are just namespace objects, with attributes created on the fly by assignment. Those assignments usually happen within the `class` statements you code, but they can occur anywhere you have a reference to one of the objects in the tree.

Even methods, normally created by a `def` nested in a `class`, can be created completely independently of any class object. The following, for example, defines a simple function outside of any class that takes one argument:

```
>>> def upperName(self):
...     return self.name.upper()     # Still needs a self
```

There is nothing about a class here yet—it's a simple function, and it can be called as such at this point, provided we pass in an object with a `name` attribute (the name `self` does not make this special in any way):

```
>>> upperName(x)                     # Call as a simple function
'SUE'
```

If we assign this simple function to an attribute of our class, though, it becomes a method, callable through any instance (as well as through the class name itself, as long as we pass in an instance manually):†

```
>>> rec.method = upperName

>>> x.method()                       # Run  method to process x
'SUE'

>>> y.method()                       # Same, but pass y to self
'BOB'
```

† In fact, this is one of the reasons the `self` argument must always be explicit in Python methods—because methods can be created as simple functions independent of a class, they need to make the implied instance argument explicit. They can be called as either functions or methods, and Python can neither guess nor assume that a simple function might eventually become a class method. The main reason for the explicit `self` argument, though, is to make the meanings of names more obvious: names not referenced through `self` are simple variables, while names referenced through `self` are obviously instance attributes.

```
>>> rec.method(x)                    # Can call through instance or class
'SUE'
```

Normally, classes are filled out by `class` statements, and instance attributes are created by assignments to `self` attributes in method functions. The point again, though, is that they don't have to be; OOP in Python really is mostly about looking up attributes in linked namespace objects.

Classes Versus Dictionaries

Although the simple classes of the prior section are meant to illustrate class model basics, the techniques they employ can also be used for real work. For example, Chapter 8 showed how to use dictionaries to record properties of entities in our programs. It turns out that classes can serve this role, too—they package information like dictionaries, but can also bundle processing logic in the form of methods. For reference, here is the example for dictionary-based records we used earlier in the book:

```
>>> rec = {}
>>> rec['name'] = 'mel'              # Dictionary-based record
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print(rec['name'])
mel
```

This code emulates tools like records in other languages. As we just saw, though, there are also multiple ways to do the same with classes. Perhaps the simplest is this—trading keys for attributes:

```
>>> class rec: pass
...
>>> rec.name = 'mel'                 # Class-based record
>>> rec.age  = 45
>>> rec.job  = 'trainer/writer'
>>>
>>> print(rec.age)
40
```

This code has substantially less syntax than the dictionary equivalent. It uses an empty `class` statement to generate an empty namespace object. Once we make the empty class, we fill it out by assigning class attributes over time, as before.

This works, but a new `class` statement will be required for each distinct record we will need. Perhaps more typically, we can instead generate *instances* of an empty class to represent each distinct entity:

```
>>> class rec: pass
...
>>> pers1 = rec()                    # Instance-based records
>>> pers1.name = 'mel'
>>> pers1.job  = 'trainer'
```

```
>>> pers1.age   = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'vls'
>>> pers2.job  = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'vls')
```

Here, we make two records from the same class. Instances start out life empty, just like classes. We then fill in the records by assigning to attributes. This time, though, there are two separate objects, and hence two separate `name` attributes. In fact, instances of the same class don't even have to have the same set of attribute names; in this example, one has a unique `age` name. Instances really are distinct namespaces, so each has a distinct attribute dictionary. Although they are normally filled out consistently by class methods, they are more flexible than you might expect.

Finally, we might instead code a more full-blown class to implement the record and its processing:

```
>>> class Person:
...     def __init__(self, name, job):       # Class = Data + Logic
...         self.name = name
...         self.job  = job
...     def info(self):
...         return (self.name, self.job)
...
>>> rec1 = Person('mel', 'trainer')
>>> rec2 = Person('vls', 'developer')
>>>
>>> rec1.job, rec2.info()
('trainer', ('vls', 'developer'))
```

This scheme also makes multiple instances, but the class is not empty this time: we've added *logic* (methods) to initialize instances at construction time and collect attributes into a tuple. The constructor imposes some consistency on instances here by always setting the `name` and `job` attributes. Together, the class's methods and instance attributes create a *package*, which combines both data and logic.

We could further extend this code by adding logic to compute salaries, parse names, and so on. Ultimately, we might link the class into a larger hierarchy to inherit an existing set of methods via the automatic attribute search of classes, or perhaps even store instances of the class in a file with Python object pickling to make them persistent. In fact, we will—in the next chapter, we'll expand on this analogy between classes and records with a more realistic running example that demonstrates class basics in action.

In the end, although types like dictionaries are flexible, classes allow us to add behavior to objects in ways that built-in types and simple functions do not directly support. Although we can store functions in dictionaries, too, using them to process implied instances is nowhere near as natural as it is in classes.

Chapter Summary

This chapter introduced the basics of coding classes in Python. We studied the syntax of the `class` statement, and we saw how to use it to build up a class inheritance tree. We also studied how Python automatically fills in the first argument in method functions, how attributes are attached to objects in a class tree by simple assignment, and how specially named operator overloading methods intercept and implement built-in operations for our instances (e.g., expressions and printing).

Now that we've learned all about the mechanics of coding classes in Python, the next chapter turns to a larger and more realistic example that ties together much of what we've learned about OOP so far. After that, we'll continue our look at class coding, taking a second pass over the model to fill in some of the details that were omitted here to keep things simple. First, though, let's work through a quiz to review the basics we've covered so far.

Test Your Knowledge: Quiz

1. How are classes related to modules?
2. How are instances and classes created?
3. Where and how are class attributes created?
4. Where and how are instance attributes created?
5. What does `self` mean in a Python class?
6. How is operator overloading coded in a Python class?
7. When might you want to support operator overloading in your classes?
8. Which operator overloading method is most commonly used?
9. What are the two key concepts required to understand Python OOP code?

Test Your Knowledge: Answers

1. Classes are always nested inside a module; they are attributes of a module object. Classes and modules are both namespaces, but classes correspond to statements (not entire files) and support the OOP notions of multiple instances, inheritance, and operator overloading. In a sense, a module is like a single-instance class, without inheritance, which corresponds to an entire file of code.
2. Classes are made by running `class` statements; instances are created by calling a class as though it were a function.

3. Class attributes are created by assigning attributes to a class object. They are normally generated by top-level assignments nested in a `class` statement—each name assigned in the `class` statement block becomes an attribute of the class object (technically, the `class` statement scope morphs into the class object's attribute namespace). Class attributes can also be created, though, by assigning attributes to the class anywhere a reference to the class object exists—i.e., even outside the `class` statement.
4. Instance attributes are created by assigning attributes to an instance object. They are normally created within class method functions inside the `class` statement by assigning attributes to the `self` argument (which is always the implied instance). Again, though, they may be created by assignment anywhere a reference to the instance appears, even outside the `class` statement. Normally, all instance attributes are initialized in the `__init__` constructor method; that way, later method calls can assume the attributes already exist.
5. `self` is the name commonly given to the first (leftmost) argument in a class method function; Python automatically fills it in with the instance object that is the implied subject of the method call. This argument need not be called `self` (though this is a very strong convention); its position is what is significant. (Ex-C++ or Java programmers might prefer to call it `this` because in those languages that name reflects the same idea; in Python, though, this argument must always be explicit.)
6. Operator overloading is coded in a Python class with specially named methods; they all begin and end with double underscores to make them unique. These are not built-in or reserved names; Python just runs them automatically when an instance appears in the corresponding operation. Python itself defines the mappings from operations to special method names.
7. Operator overloading is useful to implement objects that resemble built-in types (e.g., sequences or numeric objects such as matrixes), and to mimic the built-in type interface expected by a piece of code. Mimicking built-in type interfaces enables you to pass in class instances that also have state information—i.e., attributes that remember data between operation calls. You shouldn't use operator overloading when a simple named method will suffice, though.
8. The `__init__` constructor method is the most commonly used; almost every class uses this method to set initial values for instance attributes and perform other startup tasks.
9. The special `self` argument in method functions and the `__init__` constructor method are the two cornerstones of OOP code in Python.

CHAPTER 27
A More Realistic Example

We'll dig into more class syntax details in the next chapter. Before we do, though, I'd like to show you a more realistic example of classes in action that's more practical than what we've seen so far. In this chapter, we're going to build a set of classes that do something more concrete—recording and processing information about people. As you'll see, what we call *instances* and *classes* in Python programming can often serve the same roles as *records* and *programs* in more traditional terms.

Specifically, in this chapter we're going to code two classes:

- `Person`—a class that creates and processes information about people
- `Manager`—a customization of Person that modifies inherited behavior

Along the way, we'll make instances of both classes and test out their functionality. When we're done, I'll show you a nice example use case for classes—we'll store our instances in a *shelve* object-oriented database, to make them permanent. That way, you can use this code as a template for fleshing out a full-blown personal database written entirely in Python.

Besides actual utility, though, our aim here is also *educational*: this chapter provides a tutorial on object-oriented programming in Python. Often, people grasp the last chapter's class syntax on paper, but have trouble seeing how to get started when confronted with having to code a new class from scratch. Toward this end, we'll take it one step at a time here, to help you learn the basics; we'll build up the classes gradually, so you can see how their features come together in complete programs.

In the end, our classes will still be relatively small in terms of code, but they will demonstrate *all* of the main ideas in Python's OOP model. Despite its syntax details, Python's class system really is largely just a matter of searching for an attribute in a tree of objects, along with a special first argument for functions.

Step 1: Making Instances

OK, so much for the design phase—let's move on to implementation. Our first task is to start coding the main class, `Person`. In your favorite text editor, open a new file for the code we'll be writing. It's a fairly strong convention in Python to begin module names with a lowercase letter and class names with an uppercase letter; like the name of `self` arguments in methods, this is not required by the language, but it's so common that deviating might be confusing to people who later read your code. To conform, we'll call our new module file *person.py* and our class within it `Person`, like this:

```
# File person.py (start)

class Person:
```

All our work will be done in this file until later in this chapter. We can code any number of functions and classes in a single module file in Python, and this one's *person.py* name might not make much sense if we add unrelated components to it later. For now, we'll assume everything in it will be `Person`-related. It probably should be anyhow—as we've learned, modules tend to work best when they have a single, *cohesive* purpose.

Coding Constructors

Now, the first thing we want to do with our `Person` class is record basic information about people—to fill out record fields, if you will. Of course, these are known as instance object *attributes* in Python-speak, and they generally are created by assignment to `self` attributes in class method functions. The normal way to give instance attributes their first values is to assign them to `self` in the `__init__` *constructor method*, which contains code run automatically by Python each time an instance is created. Let's add one to our class:

```
# Add record field initialization

class Person:
    def __init__(self, name, job, pay):       # Constructor takes 3 arguments
        self.name = name                      # Fill out fields when created
        self.job  = job                       # self is the new instance object
        self.pay  = pay
```

This is a very common coding pattern: we pass in the data to be attached to an instance as arguments to the constructor method and assign them to `self` to retain them permanently. In OO terms, `self` is the newly created instance object, and `name`, `job`, and `pay` become *state information*—descriptive data saved on an object for later use. Although other techniques (such as enclosing scope references) can save details, too, instance attributes make this very explicit and easy to understand.

Notice that the argument names appear *twice* here. This code might seem a bit redundant at first, but it's not. The `job` argument, for example, is a local variable in the scope of the `__init__` function, but `self.job` is an attribute of the instance that's the implied

subject of the method call. They are two different variables, which happen to have the same name. By assigning the `job` local to the `self.job` attribute with `self.job=job`, we save the passed-in `job` on the instance for later use. As usual in Python, where a name is assigned (or what object it is assigned to) determines what it means.

Speaking of arguments, there's really nothing magical about `__init__`, apart from the fact that it's called automatically when an instance is made and has a special first argument. Despite its weird name, it's a normal function and supports all the features of functions we've already covered. We can, for example, provide *defaults* for some of its arguments, so they need not be provided in cases where their values aren't available or useful.

To demonstrate, let's make the `job` argument optional—it will default to `None`, meaning the person being created is not (currently) employed. If `job` defaults to `None`, we'll probably want to default `pay` to `0`, too, for consistency (unless some of the people you know manage to get paid without having jobs!). In fact, we have to specify a default for `pay` because according to Python's syntax rules, any arguments in a function's header after the first default must all have defaults, too:

```
# Add defaults for constructor arguments

class Person:
    def __init__(self, name, job=None, pay=0):          # Normal function args
        self.name = name
        self.job  = job
        self.pay  = pay
```

What this code means is that we'll need to pass in a name when making `Person`s, but `job` and `pay` are now optional; they'll default to `None` and `0` if omitted. The `self` argument, as usual, is filled in by Python automatically to refer to the instance object—assigning values to attributes of `self` attaches them to the new instance.

Testing As You Go

This class doesn't do much yet—it essentially just fills out the fields of a new record—but it's a real working class. At this point we could add more code to it for more features, but we won't do that yet. As you've probably begun to appreciate already, programming in Python is really a matter of *incremental prototyping*—you write some code, test it, write more code, test again, and so on. Because Python provides both an interactive session and nearly immediate turnaround after code changes, it's more natural to test as you go than to write a huge amount of code to test all at once.

Before adding more features, then, let's test what we've got so far by making a few instances of our class and displaying their attributes as created by the constructor. We could do this interactively, but as you've also probably surmised by now, interactive testing has its limits—it gets tedious to have to reimport modules and retype test cases each time you start a new testing session. More commonly, Python programmers use

the interactive prompt for simple one-off tests but do more substantial testing by writing code at the bottom of the file that contains the objects to be tested, like this:

```
# Add incremental self-test code

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

bob = Person('Bob Smith')                          # Test the class
sue = Person('Sue Jones', job='dev', pay=100000)   # Runs __init__ automatically
print(bob.name, bob.pay)                           # Fetch attached attributes
print(sue.name, sue.pay)                           # sue's and bob's attrs differ
```

Notice here that the `bob` object accepts the defaults for `job` and `pay`, but `sue` provides values explicitly. Also note how we use *keyword arguments* when making `sue`; we could pass by position instead, but the keywords may help remind us later what the data is (and they allow us to pass the arguments in any left-to-right order we like). Again, despite its unusual name, `__init__` is a normal function, supporting everything you already know about functions—including both defaults and pass-by-name keyword arguments.

When this file runs as a script, the test code at the bottom makes two instances of our class and prints two attributes of each (`name` and `pay`):

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000
```

You can also type this file's test code at Python's interactive prompt (assuming you import the `Person` class there first), but coding canned tests inside the module file like this makes it much easier to rerun them in the future.

Although this is fairly simple code, it's already demonstrating something important. Notice that `bob`'s `name` is not `sue`'s, and `sue`'s `pay` is not `bob`'s. Each is an independent record of information. Technically, `bob` and `sue` are both *namespace objects*—like all class instances, they each have their own independent copy of the state information created by the class. Because each instance of a class has its own set of `self` attributes, classes are a natural for recording information for multiple objects this way; just like built-in types, classes serve as a sort of *object factory*. Other Python program structures, such as functions and modules, have no such concept.

Using Code Two Ways

As is, the test code at the bottom of the file works, but there's a big catch—its top-level `print` statements run both when the file is run as a script and when it is imported as a module. This means if we ever decide to import the class in this file in order to use it somewhere else (and we will later in this chapter), we'll see the output of its test code

every time the file is imported. That's not very good software citizenship, though: client programs probably don't care about our internal tests and won't want to see our output mixed in with their own.

Although we could split the test code off into a separate file, it's often more convenient to code tests in the same file as the items to be tested. It would be better to arrange to run the test statements at the bottom *only* when the file is run for testing, not when the file is imported. That's exactly what the module `__name__` check is designed for, as you learned in the preceding part of this book. Here's what this addition looks like:

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':                  # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Now, we get exactly the behavior we're after—running the file as a top-level script tests it because its `__name__` is `__main__`, but importing it as a library of classes later does not:

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000

c:\misc> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) ...
>>> import person
>>>
```

When imported, the file now defines the class, but does not use it. When run directly, this file creates two instances of our class as before, and prints two attributes of each; again, because each instance is an independent namespace object, the values of their attributes differ.

Version Portability Note

I'm running all the code in this chapter under Python 3.0, and using the 3.0 `print` function call syntax. If you run under 2.6 the code will work as-is, but you'll notice parentheses around some output lines because the extra parentheses in `print`s turn multiple items into a tuple:

```
c:\misc> c:\python26\python person.py
('Bob Smith', 0)
('Sue Jones', 100000)
```

If this difference is the sort of detail that might keep you awake at nights, simply remove the parentheses to use 2.6 `print` statements. You can also avoid the extra parentheses portably by using formatting to yield a single object to print. Either of the following works in both 2.6 and 3.0, though the method form is newer:

```
print('{0} {1}'.format(bob.name, bob.pay))      # New format method
print('%s %s' % (bob.name, bob.pay))            # Format expression
```

Step 2: Adding Behavior Methods

Everything looks good so far—at this point, our class is essentially a record *factory*; it creates and fills out fields of records (attributes of instances, in more Pythonic terms). Even as limited as it is, though, we can still run some operations on its objects. Although classes add an extra layer of structure, they ultimately do most of their work by embedding and processing basic *core data types* like lists and strings. In other words, if you already know how to use Python's simple core types, you already know much of the Python class story; classes are really just a minor structural extension.

For example, the `name` field of our objects is a simple string, so we can extract last names from our objects by splitting on spaces and indexing. These are all core data type operations, which work whether their subjects are embedded in class instances or not:

```
>>> name = 'Bob Smith'       # Simple string, outside class
>>> name.split()             # Extract last name
['Bob', 'Smith']
>>> name.split()[-1]         # Or [1], if always just two parts
'Smith'
```

Similarly, we can give an object a pay raise by updating its `pay` field—that is, by changing its state information in-place with an assignment. This task also involves basic operations that work on Python's core objects, regardless of whether they are standalone or embedded in a class structure:

```
>>> pay = 100000             # Simple variable, outside class
>>> pay *= 1.10              # Give a 10% raise
>>> print(pay)               # Or: pay = pay * 1.10, if you like to type
110000.0                     # Or: pay = pay + (pay * .10), if you _really_ do!
```

To apply these operations to the `Person` objects created by our script, simply do to `bob.name` and `sue.pay` what we just did to `name` and `pay`. The operations are the same, but the subject objects are attached to attributes in our class structure:

```
# Process embedded built-in types: strings, mutability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
```

```
bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.pay)
print(sue.name, sue.pay)
print(bob.name.split()[-1])            # Extract object's last name
sue.pay *= 1.10                        # Give this object a raise
print(sue.pay)
```

We've added the last two lines here; when they're run, we extract `bob`'s last name by using basic string and list operations and give `sue` a pay raise by modifying her `pay` attribute in-place with basic number operations. In a sense, `sue` is also a *mutable* object—her state changes in-place just like a list after an `append` call:

```
Bob Smith 0
Sue Jones 100000
Smith
110000.0
```

The preceding code works as planned, but if you show it to a veteran software developer he'll probably tell you that its general approach is not a great idea in practice. Hardcoding operations like these *outside* of the class can lead to maintenance problems in the future.

For example, what if you've hardcoded the last-name-extraction formula at many different places in your program? If you ever need to change the way it works (to support a new name structure, for instance), you'll need to hunt down and update *every* occurrence. Similarly, if the pay-raise code ever changes (e.g., to require approval or database updates), you may have multiple copies to modify. Just finding all the appearances of such code may be problematic in larger programs—they may be scattered across many files, split into individual steps, and so on.

Coding Methods

What we really want to do here is employ a software design concept known as *encapsulation*. The idea with encapsulation is to wrap up operation logic behind interfaces, such that each operation is coded only once in our program. That way, if our needs change in the future, there is just one copy to update. Moreover, we're free to change the single copy's internals almost arbitrarily, without breaking the code that uses it.

In Python terms, we want to code operations on objects in class *methods*, instead of littering them throughout our program. In fact, this is one of the things that classes are very good at—*factoring* code to remove redundancy and thus optimize maintainability. As an added bonus, turning operations into methods enables them to be applied to any instance of the class, not just those that they've been hardcoded to process.

This is all simpler in code than it may sound in theory. The following achieves encapsulation by moving the two operations from code outside the class into class methods. While we're at it, let's change our self-test code at the bottom to use the new methods we're creating, instead of hardcoding operations:

```
# Add methods to encapsulate operations for maintainability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                          # Behavior methods
        return self.name.split()[-1]             # self is implied subject
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))     # Must change here only

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())        # Use the new methods
    sue.giveRaise(.10)                           # instead of hardcoding
    print(sue.pay)
```

As we've learned, *methods* are simply normal functions that are attached to classes and designed to process instances of those classes. The instance is the subject of the method call and is passed to the method's `self` argument automatically.

The transformation to the methods in this version is straightforward. The new `lastName` method, for example, simply does to `self` what the previous version hardcoded for `bob`, because `self` is the implied subject when the method is called. `lastName` also returns the result, because this operation is a called function now; it computes a value for its caller to use, even if it is just to be printed. Similarly, the new `giveRaise` method just does to `self` what we did to `sue` before.

When run now, our file's output is similar to before—we've mostly just *refactored* the code to allow for easier changes in the future, not altered its behavior:

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

A few coding details are worth pointing out here. First, notice that `sue`'s pay is now still an *integer* after a pay raise—we convert the math result back to an integer by calling the `int` built-in within the method. Changing the value to either `int` or `float` is probably not a significant concern for most purposes (integer and floating-point objects have the same interfaces and can be mixed within expressions), but we may need to address rounding issues in a real system (money probably matters to `Persons`!).

As we learned in Chapter 5, we might handle this by using the `round(N, 2)` built-in to round and retain cents, using the `decimal` type to fix precision, or storing monetary values as full floating-point numbers and displaying them with a `%.2f` or `{0:.2f}` formatting string to show cents. For this example, we'll simply truncate any cents with

`int`. (For another idea, also see the `money` function in the *formats.py* module of Chapter 24; you can import this tool to show pay with commas, cents, and dollar signs.)

Second, notice that we're also printing `sue`'s last name this time—because the last-name logic has been encapsulated in a method, we get to use it on *any instance* of the class. As we've seen, Python tells a method which instance to process by automatically passing it in to the first argument, usually called `self`. Specifically:

- In the first call, `bob.lastName()`, `bob` is the implied subject passed to `self`.
- In the second call, `sue.lastName()`, `sue` goes to `self` instead.

Trace through these calls to see how the instance winds up in `self`. The net effect is that the method fetches the name of the implied subject each time. The same happens for `giveRaise`. We could, for example, give `bob` a raise by calling `giveRaise` for both instances this way, too; but unfortunately, `bob`'s zero pay will prevent him from getting a raise as the program is currently coded (something we may want to address in a future 2.0 release of our software).

Finally, notice that the `giveRaise` method assumes that `percent` is passed in as a floating-point number between zero and one. That may be too radical an assumption in the real world (a 1000% raise would probably be a bug for most of us!); we'll let it pass for this prototype, but we might want to test or at least document this in a future iteration of this code. Stay tuned for a rehash of this idea in a later chapter in this book, where we'll code something called *function decorators* and explore Python's `assert` statement—alternatives that can do the validity test for us automatically during development.

Step 3: Operator Overloading

At this point, we have a fairly full-featured class that generates and initializes instances, along with two new bits of behavior for processing instances (in the form of methods). So far, so good.

As it stands, though, testing is still a bit less convenient than it needs to be—to trace our objects, we have to manually fetch and print *individual attributes* (e.g., `bob.name`, `sue.pay`). It would be nice if displaying an instance all at once actually gave us some useful information. Unfortunately, the default display format for an instance object isn't very good—it displays the object's class name, and its address in memory (which is essentially useless in Python, except as a unique identifier).

To see this, change the last line in the script to `print(sue)` so it displays the object as a whole. Here's what you'll get (the output says that `sue` is an "object" in 3.0 and an "instance" in 2.6):

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x02614430>
```

Providing Print Displays

Fortunately, it's easy to do better by employing *operator overloading*—coding methods in a class that intercept and process built-in operations when run on the class's instances. Specifically, we can make use of what is probably the second most commonly used operator overloading method in Python, after `__init__`: the `__str__` method introduced in the preceding chapter. `__str__` is run automatically every time an instance is converted to its print string. Because that's what printing an object does, the net transitive effect is that printing an object displays whatever is returned by the object's `__str__` method, if it either defines one itself or inherits one from a superclass (double-underscored names are inherited just like any other).

Technically speaking, the `__init__` constructor method we've already coded is operator overloading too—it is run automatically at construction time to initialize a newly created instance. Constructors are so common, though, that they almost seem like a special case. More focused methods like `__str__` allow us to tap into specific operations and provide *specialized behavior* when our objects are used in those contexts.

Let's put this into code. The following extends our class to give a custom display that lists attributes when our class's instances are displayed as a whole, instead of relying on the less useful default display:

```
# Add __str__ overload method for printing objects

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):                                  # Added method
        return '[Person: %s, %s]' % (self.name, self.pay)      # String to print

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

Notice that we're doing string `%` formatting to build the display string in `__str__` here; at the bottom, classes use built-in type objects and operations like these to get their work done. Again, everything you've already learned about both built-in types and functions applies to class-based code. Classes largely just add an additional layer of *structure* that packages functions and data together and supports extensions.

We've also changed our self-test code to print objects directly, instead of printing individual attributes. When run, the output is more coherent and meaningful now; the "[...]" lines are returned by our new `__str__`, run automatically by print operations:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

Here's a subtle point: as we'll learn in the next chapter, a related overloading method, `__repr__`, provides an as-code low-level display of an object when present. Sometimes classes provide both a `__str__` for user-friendly displays and a `__repr__` with extra details for developers to view. Because printing runs `__str__` and the interactive prompt echoes results with `__repr__`, this can provide both target audiences with an appropriate display. Since we're not interested in displaying an as-code format, `__str__` is sufficient for our class.

Step 4: Customizing Behavior by Subclassing

At this point, our class captures much of the OOP machinery in Python: it makes instances, provides behavior in methods, and even does a bit of operator overloading now to intercept print operations in `__str__`. It effectively packages our data and logic together into a single, self-contained *software component*, making it easy to locate code and straightforward to change it in the future. By allowing us to encapsulate behavior, it also allows us to factor that code to avoid redundancy and its associated maintenance headaches.

The only major OOP concept it does not yet capture is *customization by inheritance*. In some sense, we're already doing inheritance, because instances inherit methods from their classes. To demonstrate the real power of OOP, though, we need to define a superclass/subclass relationship that allows us to extend our software and replace bits of inherited behavior. That's the main idea behind OOP, after all; by fostering a coding model based upon customization of work already done, it can dramatically cut development time.

Coding Subclasses

As a next step, then, let's put OOP's methodology to use and customize our `Person` class by extending our software hierarchy. For the purpose of this tutorial, we'll define a subclass of `Person` called `Manager` that replaces the inherited `giveRaise` method with a more specialized version. Our new class begins as follows:

```
class Manager(Person):                    # Define a subclass of Person
```

This code means that we're defining a new class named `Manager`, which inherits from and may add customizations to the superclass `Person`. In plain terms, a `Manager` is almost

like a Person (admittedly, a very long journey for a very small joke...), but Manager has a custom way to give raises.

For the sake of argument, let's assume that when a Manager gets a raise, it receives the passed-in percentage as usual, but also gets an extra bonus that defaults to 10%. For instance, if a Manager's raise is specified as 10%, it will really get 20%. (Any relation to Persons living or dead is, of course, strictly coincidental.) Our new method begins as follows; because this redefinition of giveRaise will be closer in the class tree to Manager instances than the original version in Person, it effectively replaces, and thereby customizes, the operation. Recall that according to the inheritance search rules, the *lowest* version of the name wins:

```
class Manager(Person):                          # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):    # Redefine to customize
```

Augmenting Methods: The Bad Way

Now, there are two ways we might code this Manager customization: a good way and a bad way. Let's start with the *bad way*, since it might be a bit easier to understand. The bad way is to cut and paste the code of giveRaise in Person and modify it for Manager, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut-and-paste
```

This works as advertised—when we later call the giveRaise method of a Manager instance, it will run this custom version, which tacks on the extra bonus. So what's wrong with something that runs correctly?

The problem here is a very general one: any time you copy code with cut and paste, you essentially *double* your maintenance effort in the future. Think about it: because we copied the original version, if we ever have to change the way raises are given (and we probably will), we'll have to change the code in *two* places, not one. Although this is a small and artificial example, it's also representative of a universal issue—any time you're tempted to program by copying code this way, you probably want to look for a better approach.

Augmenting Methods: The Good Way

What we really want to do here is somehow *augment* the original giveRaise, instead of replacing it altogether. The *good way* to do that in Python is by calling to the original version directly, with augmented arguments, like this:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)             # Good: augment original
```

This code leverages the fact that a class method can always be called either through an *instance* (the usual way, where Python sends the instance to the `self` argument automatically) or through the *class* (the less common scheme, where you must pass the instance manually). In more symbolic terms, recall that a normal method call of this form:

```
instance.method(args...)
```

is automatically translated by Python into this equivalent form:

```
class.method(instance, args...)
```

where the class containing the method to be run is determined by the inheritance search rule applied to the method's name. You can code *either* form in your script, but there is a slight asymmetry between the two—you must remember to pass along the instance manually if you call through the class directly. The method always needs a subject instance one way or another, and Python provides it automatically only for calls made through an instance. For calls through the class name, you need to send an instance to `self` yourself; for code inside a method like `giveRaise`, `self` already *is* the subject of the call, and hence the instance to pass along.

Calling through the class directly effectively subverts inheritance and kicks the call higher up the class tree to run a specific version. In our case, we can use this technique to invoke the default `giveRaise` in `Person`, even though it's been redefined at the `Manager` level. In some sense, we *must* call through `Person` this way, because a `self.giveRaise()` inside `Manager`'s `giveRaise` code would loop—since `self` already is a `Manager`, `self.giveRaise()` would resolve again to `Manager.giveRaise`, and so on and so forth until available memory is exhausted.

This "good" version may seem like a small difference in code, but it can make a huge difference for future *code maintenance*—because the `giveRaise` logic lives in just one place now (`Person`'s method), we have only one version to change in the future as needs evolve. And really, this form captures our intent more directly anyhow—we want to perform the standard `giveRaise` operation, but simply tack on an extra bonus. Here's our entire module file with this step applied:

```
# Add customization of one behavior in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
```

```
    def giveRaise(self, percent, bonus=.10):          # Redefine at this level
        Person.giveRaise(self, percent + bonus)       # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)          # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                        # Runs inherited __str__
```

To test our `Manager` subclass customization, we've also added self-test code that makes a `Manager`, calls its methods, and prints it. Here's the new version's output:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

Everything looks good here: `bob` and `sue` are as before, and when `tom` the `Manager` is given a 10% raise, he really gets 20% (his pay goes from $50K to $60K), because the customized `giveRaise` in `Manager` is run for him only. Also notice how printing `tom` as a whole at the end of the test code displays the nice format defined in `Person`'s `__str__`: `Manager` objects get this, `lastName`, and the `__init__` constructor method's code "for free" from `Person`, by inheritance.

Polymorphism in Action

To make this acquisition of inherited behavior even more striking, we can add the following code at the end of our file:

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for object in (bob, sue, tom):                    # Process objects generically
        object.giveRaise(.10)                         # Run this object's giveRaise
        print(object)                                 # Run the common __str__
```

Here's the resulting output:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

In the added code, `object` is *either* a `Person` or a `Manager`, and Python runs the appropriate `giveRaise` automatically—our original version in `Person` for `bob` and `sue`, and our customized version in `Manager` for `tom`. Trace the method calls yourself to see how Python selects the right `giveRaise` method for each object.

This is just Python's notion of *polymorphism*, which we met earlier in the book, at work again—what `giveRaise` does depends on what you do it to. Here, it's made all the more obvious when it selects from code we've written ourselves in classes. The practical effect in this code is that `sue` gets another 10% but `tom` gets another 20%, because `giveRaise` is dispatched based upon the object's type. As we've learned, polymorphism is at the heart of Python's flexibility. Passing any of our three objects to a function that calls a `giveRaise` method, for example, would have the same effect: the appropriate version would be run automatically, depending on which type of object was passed.

On the other hand, printing runs the *same* `__str__` for all three objects, because it's coded just once in `Person`. `Manager` both specializes and applies the code we originally wrote in `Person`. Although this example is small, it's already leveraging OOP's talent for code customization and reuse; with classes, this almost seems automatic at times.

Inherit, Customize, and Extend

In fact, classes can be even more flexible than our example implies. In general, classes can *inherit*, *customize*, or *extend* existing code in superclasses. For example, although we're focused on customization here, we can also add unique methods to `Manager` that are not present in `Person`, if `Manager`s require something completely different (Python namesake reference intended). The following snippet illustrates. Here, `giveRaise` redefines a superclass method to customize it, but `someThingElse` defines something new to extend:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __str__(self): ...

class Manager(Person):                      # Inherit
    def giveRaise(self, ...): ...           # Customize
    def someThingElse(self, ...): ...       # Extend

tom = Manager()
tom.lastName()                              # Inherited verbatim
tom.giveRaise()                             # Customized version
tom.someThingElse()                         # Extension here
print(tom)                                  # Inherited overload method
```

Extra methods like this code's `someThingElse` *extend* the existing software and are available on `Manager` objects only, not on `Person`s. For the purposes of this tutorial, however,

we'll limit our scope to customizing some of `Person`'s behavior by redefining it, not adding to it.

OOP: The Big Idea

As is, our code may be small, but it's fairly functional. And really, it already illustrates the main point behind OOP in general: in OOP, we program by *customizing* what has already been done, rather than copying or changing existing code. This isn't always an obvious win to newcomers at first glance, especially given the extra coding requirements of classes. But overall, the programming style implied by classes can cut development time radically compared to other approaches.

For instance, in our example we could theoretically have implemented a custom `giveRaise` operation without subclassing, but none of the other options yield code as optimal as ours:

- Although we could have simply coded `Manager` *from scratch* as new, independent code, we would have had to reimplement all the behaviors in `Person` that are the same for `Manager`s.
- Although we could have simply *changed* the existing `Person` class in-place for the requirements of `Manager`'s `giveRaise`, doing so would probably break the places where we still need the original `Person` behavior.
- Although we could have simply *copied* the `Person` class in its entirety, renamed the copy to `Manager`, and changed its `giveRaise`, doing so would introduce code redundancy that would double our work in the future—changes made to `Person` in the future would not be picked up automatically, but would have to be manually propagated to `Manager`'s code. As usual, the cut-and-paste approach may seem quick now, but it doubles your work in the future.

The *customizable hierarchies* we can build with classes provide a much better solution for software that will evolve over time. No other tools in Python support this development mode. Because we can tailor and extend our prior work by coding new subclasses, we can leverage what we've already done, rather than starting from scratch each time, breaking what already works, or introducing multiple copies of code that may all have to be updated in the future. When done right, OOP is a powerful programmer's ally.

Step 5: Customizing Constructors, Too

Our code works as it is, but if you study the current version closely, you may be struck by something a bit odd—it seems pointless to have to provide a `mgr` job name for `Manager` objects when we create them: this is already implied by the class itself. It would be better if we could somehow fill in this value automatically when a `Manager` is made.

The trick we need to improve on this turns out to be the *same* as the one we employed in the prior section: we want to customize the constructor logic for `Manager`s in such a

way as to provide a job name automatically. In terms of code, we want to redefine an `__init__` method in `Manager` that provides the `mgr` string for us. And like with the `giveRaise` customization, we also want to run the original `__init__` in `Person` by calling through the class name, so it still initializes our objects' state information attributes.

The following extension will do the job—we've coded the new `Manager` constructor and changed the call that creates `tom` to not pass in the `mgr` job name:

```
# Add customization of constructor in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):                    # Redefine constructor
        Person.__init__(self, name, 'mgr', pay)       # Run original with 'mgr'
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)                 # Job name not needed:
    tom.giveRaise(.10)                                # Implied/set by class
    print(tom.lastName())
    print(tom)
```

Again, we're using the same technique to augment the `__init__` constructor here that we used for `giveRaise` earlier—running the superclass version by calling through the class name directly and passing the `self` instance along explicitly. Although the constructor has a strange name, the effect is identical. Because we need `Person`'s construction logic to run too (to initialize instance attributes), we really have to call it this way; otherwise, instances would not have any attributes attached.

Calling superclass constructors from redefinitions this way turns out to be a very common coding pattern in Python. By itself, Python uses inheritance to look for and call only *one* `__init__` method at construction time—the *lowest* one in the class tree. If you need higher `__init__` methods to be run at construction time (and you usually do),

you must call them manually through the superclass's name. The upside to this is that you can be explicit about which argument to pass up to the superclass's constructor and can choose to not call it at all: not calling the superclass constructor allows you to replace its logic altogether, rather than augmenting it.

The output of this file's self-test code is the same as before—we haven't changed what it does, we've simply restructured to get rid of some logical redundancy:

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

OOP Is Simpler Than You May Think

In this complete form, despite their sizes, our classes capture nearly all the important concepts in Python's OOP machinery:

- Instance creation—filling out instance attributes
- Behavior methods—encapsulating logic in class methods
- Operator overloading—providing behavior for built-in operations like printing
- Customizing behavior—redefining methods in subclasses to specialize them
- Customizing constructors—adding initialization logic to superclass steps

Most of these concepts are based upon just three simple ideas: the inheritance search for attributes in object trees, the special `self` argument in methods, and operator overloading's automatic dispatch to methods.

Along the way, we've also made our code easy to change in the future, by harnessing the class's propensity for factoring code to reduce *redundancy*. For example, we wrapped up logic in methods and called back to superclass methods from extensions to avoid having multiple copies of the same code. Most of these steps were a natural outgrowth of the structuring power of classes.

By and large, that's all there is to OOP in Python. Classes certainly can become larger than this, and there are some more advanced class concepts, such as decorators and metaclasses, which we will meet in later chapters. In terms of the basics, though, our classes already do it all. In fact, if you've grasped the workings of the classes we've written, most OOP Python code should now be within your reach.

Other Ways to Combine Classes

Having said that, I should also tell you that although the basic mechanics of OOP are simple in Python, some of the art in larger programs lies in the way that classes are put together. We're focusing on *inheritance* in this tutorial because that's the mechanism

the Python language provides, but programmers sometimes combine classes in other ways, too. For example, a common coding pattern involves nesting objects inside each other to build up *composites*. We'll explore this pattern in more detail in Chapter 30, which is really more about design than about Python; as a quick example, though, we could use this composition idea to code our `Manager` extension by *embedding* a `Person`, instead of inheriting from it.

The following alternative does so by using the `__getattr__` operator overloading method we will meet in Chapter 29 to intercept undefined attribute fetches and delegate them to the embedded object with the `getattr` built-in. The `giveRaise` method here still achieves customization, by changing the argument passed along to the embedded object. In effect, `Manager` becomes a controller layer that passes calls *down* to the embedded object, rather than *up* to superclass methods:

```
# Embedding-based Manager alternative

class Person:
    ...same...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)      # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)      # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)           # Delegate all other attrs
    def __str__(self):
        return str(self.person)                     # Must overload again (in 3.0)

if __name__ == '__main__':
    ...same...
```

In fact, this `Manager` alternative is representative of a general coding pattern usually known as *delegation*—a composite-based structure that manages a wrapped object and propagates method calls to it. This pattern works in our example, but it requires about twice as much code and is less well suited than inheritance to the kinds of direct customizations we meant to express (in fact, no reasonable Python programmer would code this example this way in practice, except perhaps those writing general tutorials). `Manager` isn't really a `Person` here, so we need extra code to manually dispatch method calls to the embedded object; operator overloading methods like `__str__` must be redefined (in 3.0, at least, as noted in the upcoming sidebar "Catching Built-in Attributes in 3.0" on page 662), and adding new `Manager` behavior is less straightforward since state information is one level removed.

Still, *object embedding*, and design patterns based upon it, can be a very good fit when embedded objects require more limited interaction with the container than direct customization implies. A controller layer like this alternative `Manager`, for example, might come in handy if we want to trace or validate calls to another object's methods (indeed, we will use a nearly identical coding pattern when we study *class decorators* later in the

book). Moreover, a hypothetical `Department` class like the following could *aggregate* other objects in order to treat them as a set. Add this to the bottom of the *person.py* file to try this on your own:

```
# Aggregate embedded objects into a composite

...
bob = Person(...)
sue = Person(...)
tom = Manager(...)

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

development = Department(bob, sue)          # Embed objects in a composite
development.addMember(tom)
development.giveRaises(.10)                 # Runs embedded objects' giveRaise
development.showAll()                       # Runs embedded objects' __str__s
```

Interestingly, this code uses both inheritance *and* composition—`Department` is a composite that embeds and controls other objects to aggregate, but the embedded `Person` and `Manager` objects themselves use inheritance to customize. As another example, a GUI might similarly use *inheritance* to customize the behavior or appearance of labels and buttons, but also *composition* to build up larger packages of embedded widgets, such as input forms, calculators, and text editors. The class structure to use depends on the objects you are trying to model.

Design issues like composition are explored in Chapter 30, so we'll postpone further investigations for now. But again, in terms of the basic mechanics of OOP in Python, our `Person` and `Manager` classes already tell the entire story. Having mastered the basics of OOP, though, developing general tools for applying it more easily in your scripts is often a natural next step—and the topic of the next section.

Catching Built-in Attributes in 3.0

In Python 3.0 (and 2.6 if new-style classes are used), the alternative delegation-based `Manager` class we just coded will not be able to intercept and delegate operator overloading method attributes like `__str__` without redefining them. Although we know that `__str__` is the only such name used in our specific example, this a general issue for delegation-based classes.

Recall that built-in operations like printing and indexing implicitly invoke operator overloading methods such as `__str__` and `__getitem__`. In 3.0, built-in operations like

these do not route their implicit attribute fetches through generic attribute managers: neither `__getattr__` (run for undefined attributes) nor its cousin `__getattribute__` (run for all attributes) is invoked. This is why we have to redefine `__str__` redundantly in the alternative `Manager`, in order to ensure that printing is routed to the embedded `Person` object when run in Python 3.0.

Technically, this happens because classic classes normally look up operator overloading names in instances at runtime, but new-style classes do not—they skip the instance entirely and look up such methods in classes. In 2.6 classic classes, built-ins *do* route attributes generically—printing, for example, routes `__str__` through `__getattr__`. New-style classes also inherit a default for `__str__` that would foil `__getattr__`, but `__getattribute__` doesn't intercept the name in 3.0 either.

This is a change, but isn't a show-stopper—delegation-based classes can generally redefine operator overloading methods to delegate them to wrapped objects in 3.0, either manually or via tools or superclasses. This topic is too advanced to explore further in this tutorial, though, so don't sweat the details too much here. Watch for it to be revisited in the attribute management coverage of Chapter 37, and again in the context of `Private` class decorators in Chapter 38.

Step 6: Using Introspection Tools

Let's make one final tweak before we throw our objects onto a database. As they are, our classes are complete and demonstrate most of the basics of OOP in Python. They still have two remaining issues we probably should iron out, though, before we go live with them:

- First, if you look at the display of the objects as they are right now, you'll notice that when you print `tom` the `Manager` labels him as a `Person`. That's not technically incorrect, since `Manager` is a kind of customized and specialized `Person`. Still, it would be more accurate to display objects with the most specific (that is, *lowest*) classes possible.
- Second, and perhaps more importantly, the current display format shows *only* the attributes we include in our `__str__`, and that might not account for future goals. For example, we can't yet verify that `tom`'s job name has been set to `mgr` correctly by `Manager`'s constructor, because the `__str__` we coded for `Person` does not print this field. Worse, if we ever expand or otherwise change the set of attributes assigned to our objects in `__init__`, we'll have to remember to also update `__str__` for new names to be displayed, or it will become out of sync over time.

The last point means that, yet again, we've made potential extra work for ourselves in the future by introducing *redundancy* in our code. Because any disparity in `__str__` will be reflected in the program's output, this redundancy may be more obvious than the other forms we addressed earlier; still, avoiding extra work in the future is generally *a good thing*.

Special Class Attributes

We can address both issues with Python's *introspection tools*—special attributes and functions that give us access to some of the internals of objects' implementations. These tools are somewhat advanced and generally used more by people writing tools for other programmers to use than by programmers developing applications. Even so, a basic knowledge of some of these tools is useful because they allow us to write code that processes classes in generic ways. In our code, for example, there are two hooks that can help us out, both of which were introduced near the end of the preceding chapter:

- The built-in *`instance`*`.__class__` attribute provides a link from an instance to the class from which it was created. Classes in turn have a `__name__`, just like modules, and a `__bases__` sequence that provides access to superclasses. We can use these here to print the name of the class from which an instance is made rather than one we've hardcoded.
- The built-in *`object`*`.__dict__` attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). Because it is a dictionary, we can fetch its keys list, index by key, iterate over its keys, and so on, to process all attributes generically. We can use this here to print every attribute in any instance, not just those we hardcode in custom displays.

Here's what these tools look like in action at Python's interactive prompt. Notice how we load `Person` at the interactive prompt with a `from` statement here—class names live in and are imported from modules, exactly like function names and other variables:

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> print(bob)                                  # Show bob's __str__
[Person: Bob Smith, 0]

>>> bob.__class__                               # Show bob's class and its name
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())                   # Attributes are really dict keys
['pay', 'job', 'name']                          # Use list to force list in 3.0

>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key])     # Index manually

pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key))     # obj.attr, but attr is a var

pay => 0
```

```
job => None
name => Bob Smith
```

As noted briefly in the prior chapter, some attributes accessible from an instance might not be stored in the `__dict__` dictionary if the instance's class defines `__slots__`, an optional and relatively obscure feature of new-style classes (and all classes in Python 3.0) that stores attributes in an array and that we'll discuss in Chapters 30 and 31. Since slots really belong to classes instead of instances, and since they are very rarely used in any event, we can safely ignore them here and focus on the normal `__dict__`.

A Generic Display Tool

We can put these interfaces to work in a superclass that displays accurate class names and formats all attributes of an instance of any class. Open a new file in your text editor to code the following—it's a new, independent module named *classtools.py* that implements just such a class. Because its `__str__` print overload uses generic introspection tools, it will work on *any instance*, regardless of its attributes set. And because this is a class, it automatically becomes a general formatting tool: thanks to inheritance, it can be mixed into *any class* that wishes to use its display format. As an added bonus, if we ever want to change how instances are displayed we need only change this class, as every class that inherits its `__str__` will automatically pick up the new format when it's next run:

```
# File classtools.py (new)
"Assorted class utilities and tools"

class AttrDisplay:
    """
    Provides an inheritable print overload method that displays
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __str__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())


if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
```

```
class SubTest(TopTest):
    pass

X, Y = TopTest(), SubTest()
print(X)                          # Show all instance attrs
print(Y)                          # Show lowest class name
```

Notice the docstrings here—as a general-purpose tool, we want to add some functional documentation for potential users to read. As we saw in Chapter 15, docstrings can be placed at the top of simple functions and modules, and also at the start of classes and their methods; the `help` function and the PyDoc tool extracts and displays these automatically (we'll look at docstrings again in Chapter 28).

When run directly, this module's self-test makes two instances and prints them; the `__str__` defined here shows the instance's class, and all its attributes names and values, in sorted attribute name order:

```
C:\misc> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]
```

Instance Versus Class Attributes

If you study the `classtools` module's self-test code long enough, you'll notice that its class displays only *instance attributes*, attached to the `self` object at the bottom of the inheritance tree; that's what `self`'s `__dict__` contains. As an intended consequence, we don't see attributes inherited by the instance from classes above it in the tree (e.g., `count` in this file's self-test code). Inherited class attributes are attached to the class only, not copied down to instances.

If you ever do wish to include inherited attributes too, you can climb the `__class__` link to the instance's class, use the `__dict__` there to fetch class attributes, and then iterate through the class's `__bases__` attribute to climb to even higher superclasses (repeating as necessary). If you're a fan of simple code, running a built-in `dir` call on the instance instead of using `__dict__` and climbing would have much the same effect, since `dir` results include inherited names in the sorted results list:

```
>>> from person import Person
>>> bob = Person('Bob Smith')

# In Python 2.6:

>>> bob.__dict__.keys()                 # Instance attrs only
['pay', 'job', 'name']

>>> dir(bob)                            # + inherited attrs in classes
['__doc__', '__init__', '__module__', '__str__', 'giveRaise', 'job',
'lastName', 'name', 'pay']

# In Python 3.0:
```

```
>>> list(bob.__dict__.keys())          # 3.0 keys is a view, not a list
['pay', 'job', 'name']

>>> dir(bob)                           # 3.0 includes class type methods
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
...more lines omitted...
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'giveRaise', 'job', 'lastName', 'name', 'pay']
```

The output here varies between Python 2.6 and 3.0, because 3.0's `dict.keys` is not a list, and 3.0's `dir` returns extra class-type implementation attributes. Technically, `dir` returns more in 3.0 because classes are all "new style" and inherit a large set of operator overloading names from the class type. In fact, you'll probably want to filter out most of the `__X__` names in the 3.0 `dir` result, since they are internal implementation details and not something you'd normally want to display.

In the interest of space, we'll leave optional display of inherited class attributes with either tree climbs or `dir` as suggested experiments for now. For more hints on this front, though, watch for the *classtree.py* inheritance tree climber we will write in Chapter 28, and the *lister.py* attribute listers and climbers we'll code in Chapter 30.

Name Considerations in Tool Classes

One last subtlety here: because our `AttrDisplay` class in the `classtools` module is a general tool designed to be mixed into other arbitrary classes, we have to be aware of the potential for unintended *name collisions* with client classes. As is, I've assumed that client subclasses may want to use both its `__str__` and `gatherAttrs`, but the latter of these may be more than a subclass expects—if a subclass innocently defines a `gatherAttrs` name of its own, it will likely break our class, because the lower version in the subclass will be used instead of ours.

To see this for yourself, add a `gatherAttrs` to `TopTest` in the file's self-test code; unless the new method is identical, or intentionally customizes the original, our tool class will no longer work as planned:

```
class TopTest(AttrDisplay):
    ....
    def gatherAttrs(self):          # Replaces method in AttrDisplay!
        return 'Spam'
```

This isn't necessarily bad—sometimes we want other methods to be available to subclasses, either for direct calls or for customization. If we really meant to provide a `__str__` only, though, this is less than ideal.

To minimize the chances of name collisions like this, Python programmers often prefix methods not meant for external use with a *single underscore*: `_gatherAttrs` in our case. This isn't foolproof (what if another class defines `_gatherAttrs`, too?), but it's usually sufficient, and it's a common Python naming convention for methods internal to a class.

A better and less commonly used solution would be to use *two underscores* at the front of the method name only: `__gatherAttrs` for us. Python automatically expands such names to include the enclosing class's name, which makes them truly unique. This is a feature usually called *pseudoprivate class attributes*, which we'll expand on in Chapter 30. For now, we'll make both our methods available.

Our Classes' Final Form

Now, to use this generic tool in our classes, all we need to do is import it from its module, mix it in by inheritance in our top-level class, and get rid of the more specific `__str__` we coded before. The new `print` overload method will be inherited by instances of `Person`, as well as `Manager`; `Manager` gets `__str__` from `Person`, which now obtains it from the `AttrDisplay` coded in another module. Here is the final version of our *person.py* file with these changes applied:

```
# File person.py (final)

from classtools import AttrDisplay                  # Use generic display tool

class Person(AttrDisplay):
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay
    def lastName(self):                             # Assumes last is last
        return self.name.split()[-1]
    def giveRaise(self, percent):                   # Percent must be 0..1
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(.10)
```

```
print(tom.lastName())
print(tom)
```

As this is the final revision, we've added a few *comments* here to document our work—docstrings for functional descriptions and # for smaller notes, per best-practice conventions. When we run this code now, we see all the attributes of our objects, not just the ones we hardcoded in the original `__str__`. And our final issue is resolved: because `AttrDisplay` takes class names off the `self` instance directly, each object is shown with the name of its closest (lowest) class—`tom` displays as a `Manager` now, not a `Person`, and we can finally verify that his job name has been correctly filled in by the `Manager` constructor:

```
C:\misc> person.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Tom Jones, pay=60000]
```

This is the more useful display we were after. From a larger perspective, though, our attribute display class has become a *general tool*, which we can mix into any class by inheritance to leverage the display format it defines. Further, all its clients will automatically pick up future changes in our tool. Later in the book, we'll meet even more powerful class tool concepts, such as decorators and metaclasses; along with Python's introspection tools, they allow us to write code that augments and manages classes in structured and maintainable ways.

Step 7 (Final): Storing Objects in a Database

At this point, our work is almost complete. We now have a *two-module system* that not only implements our original design goals for representing people, but also provides a general attribute display tool we can use in other programs in the future. By coding functions and classes in module files, we've ensured that they naturally support reuse. And by coding our software as classes, we've ensured that it naturally supports extension.

Although our classes work as planned, though, the objects they create are not real database records. That is, if we kill Python, our instances will disappear—they're transient objects in memory and are not stored in a more permanent medium like a file, so they won't be available in future program runs. It turns out that it's easy to make instance objects more permanent, with a Python feature called *object persistence*—making objects live on after the program that creates them exits. As a final step in this tutorial, let's make our objects permanent.

Pickles and Shelves

Object persistence is implemented by three standard library modules, available in every Python:

`pickle`
: Serializes arbitrary Python objects to and from a string of bytes

`dbm` *(named* `anydbm` *in Python 2.6)*
: Implements an access-by-key filesystem for storing strings

`shelve`
: Uses the other two modules to store Python objects on a file by key

We met these modules very briefly in Chapter 9 when we studied file basics. They provide powerful data storage options. Although we can't do them complete justice in this tutorial or book, they are simple enough that a brief introduction is enough to get you started.

The `pickle` module is a sort of super-general object formatting and deformatting tool: given a nearly arbitrary Python object in memory, it's clever enough to convert the object to a string of bytes, which it can use later to reconstruct the original object in memory. The `pickle` module can handle almost any object you can create—lists, dictionaries, nested combinations thereof, and class instances. The latter are especially useful things to pickle, because they provide both data (attributes) and behavior (methods); in fact, the combination is roughly equivalent to "records" and "programs." Because `pickle` is so general, it can replace extra code you might otherwise write to create and parse custom text file representations for your objects. By storing an object's pickle string on a file, you effectively make it permanent and persistent: simply load and unpickle it later to re-create the original object.

Although it's easy to use `pickle` by itself to store objects in simple flat files and load them from there later, the `shelve` module provides an extra layer of structure that allows you to store pickled objects by *key*. `shelve` translates an object to its pickled string with `pickle` and stores that string under a key in a `dbm` file; when later loading, `shelve` fetches the pickled string by key and re-creates the original object in memory with `pickle`. This is all quite a trick, but to your script a shelve* of pickled objects looks just like a *dictionary*—you index by key to fetch, assign to keys to store, and use dictionary tools such as `len`, `in`, and `dict.keys` to get information. Shelves automatically map dictionary operations to objects stored in a file.

In fact, to your script the only coding difference between a shelve and a normal dictionary is that you must *open* shelves initially and must *close* them after making changes. The net effect is that a shelve provides a simple database for storing and fetching native Python objects by keys, and thus makes them persistent across program runs. It does

* Yes, we use "shelve" as a noun in Python, much to the chagrin of a variety of editors I've worked with over the years, both electronic and human.

not support query tools such as SQL, and it lacks some advanced features found in enterprise-level databases (such as true transaction processing), but native Python objects stored on a shelve may be processed with the full power of the Python language once they are fetched back by key.

Storing Objects on a Shelve Database

Pickling and shelves are somewhat advanced topics, and we won't go into all their details here; you can read more about them in the standard library manuals, as well as application-focused books such as *Programming Python*. This is all simpler in Python than in English, though, so let's jump into some code.

Let's write a new script that throws objects of our classes onto a shelve. In your text editor, open a new file we'll call *makedb.py*. Since this is a new file, we'll need to import our classes in order to create a few instances to store. We used `from` to load a class at the interactive prompt earlier, but really, as with functions and other variables, there are two ways to load a class from a file (class names are variables like any other, and not at all magic in this context):

```
import person                          # Load class with import
bob = person.Person(...)               # Go through module name

from person import Person              # Load class with from
bob = Person(...)                      # Use name directly
```

We'll use `from` to load in our script, just because it's a bit less to type. Copy or retype this code to make instances of our classes in the new script, so we have something to store (this is a simple demo, so we won't worry about the test-code redundancy here). Once we have some instances, it's almost trivial to store them on a shelve. We simply import the `shelve` module, open a new shelve with an external filename, assign the objects to keys in the shelve, and close the shelve when we're done because we've made changes:

```
# File makedb.py: store Person objects on a shelve database

from person import Person, Manager             # Load our classes
bob = Person('Bob Smith')                      # Re-create objects to be stored
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')                   # Filename where objects are stored
for object in (bob, sue, tom):                 # Use object's name attr as key
    db[object.name] = object                   # Store object on shelve by key
db.close()                                     # Close after making changes
```

Notice how we assign objects to the shelve using their own names as keys. This is just for convenience; in a shelve, the *key* can be any string, including one we might create to be unique using tools such as process IDs and timestamps (available in the `os` and `time` standard library modules). The only rule is that the keys must be strings and should

be unique, since we can store just one object per key (though that object can be a list or dictionary containing many objects). The *values* we store under keys, though, can be Python objects of almost any sort: built-in types like strings, lists, and dictionaries, as well as user-defined class instances, and nested combinations of all of these.

That's all there is to it—if this script has no output when run, it means it probably worked; we're not printing anything, just creating and storing objects:

```
C:\misc> makedb.py
```

Exploring Shelves Interactively

At this point, there are one or more real files in the current directory whose names all start with "persondb". The actual files created can vary per platform, and just like in the built-in `open` function, the filename in `shelve.open()` is relative to the current working directory unless it includes a directory path. Wherever they are stored, these files implement a keyed-access file that contains the pickled representation of our three Python objects. Don't delete these files—they are your database, and are what you'll need to copy or transfer when you back up or move your storage.

You can look at the shelve's files if you want to, either from Windows Explorer or the Python shell, but they are binary hash files, and most of their content makes little sense outside the context of the `shelve` module. With Python 3.0 and no extra software installed, our database is stored in three files (in 2.6, it's just one file, *persondb*, because the `bsddb` extension module is preinstalled with Python for shelves; in 3.0, `bsddb` is a third-party open source add-on):

```
# Directory listing module: verify files are present

>>> import glob
>>> glob.glob('person*')
['person.py', 'person.pyc', 'persondb.bak', 'persondb.dat', 'persondb.dir']

# Type the file: text mode for string, binary mode for bytes

>>> print(open('persondb.dir').read())
'Tom Jones', (1024, 91)
...more omitted...

>>> print(open('persondb.dat', 'rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00payq\x03K...
...more omitted...
```

This content isn't impossible to decipher, but it can vary on different platforms and doesn't exactly qualify as a user-friendly database interface! To verify our work better, we can write another script, or poke around our shelve at the interactive prompt. Because shelves are Python objects containing Python objects, we can process them with normal Python syntax and development modes. Here, the interactive prompt effectively becomes a *database client*:

```
>>> import shelve
>>> db = shelve.open('persondb')                # Reopen the shelve

>>> len(db)                                     # Three 'records' stored
3
>>> list(db.keys())                             # keys is the index
['Tom Jones', 'Sue Jones', 'Bob Smith']         # list to make a list in 3.0

>>> bob = db['Bob Smith']                       # Fetch bob by key
>>> print(bob)                                  # Runs __str__ from AttrDisplay
[Person: job=None, name=Bob Smith, pay=0]

>>> bob.lastName()                              # Runs lastName from Person
'Smith'

>>> for key in db:                              # Iterate, fetch, print
        print(key, '=>', db[key])

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

>>> for key in sorted(db):
        print(key, '=>', db[key])                # Iterate by sorted keys

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Notice that we don't have to import our `Person` or `Manager` classes here in order to load or use our stored objects. For example, we can call `bob`'s `lastName` method freely, and get his custom print display format automatically, even though we don't have his `Person` class in our scope here. This works because when Python pickles a class instance, it records its `self` instance attributes, along with the name of the class it was created from and the module where the class lives. When `bob` is later fetched from the shelve and unpickled, Python will automatically reimport the class and link `bob` to it.

The upshot of this scheme is that class instances automatically acquire all their class behavior when they are loaded in the future. We have to import our classes only to make new instances, not to process existing ones. Although a deliberate feature, this scheme has somewhat mixed consequences:

- The *downside* is that classes and their module's files must be importable when an instance is later loaded. More formally, pickleable classes must be coded at the top level of a module file accessible from a directory listed on the `sys.path` module search path (and shouldn't live in the most script files' module `__main__` unless they're always in that module when used). Because of this external module file requirement, some applications choose to pickle simpler objects such as dictionaries or lists, especially if they are to be transferred across the Internet.

- The *upside* is that changes in a class's source code file are automatically picked up when instances of the class are loaded again; there is often no need to update stored objects themselves, since updating their class's code changes their behavior.

Shelves also have well-known limitations (the database suggestions at the end of this chapter mention a few of these). For simple object storage, though, shelves and pickles are remarkably easy-to-use tools.

Updating Objects on a Shelve

Now for one last script: let's write a program that updates an instance (record) each time it runs, to prove the point that our objects really are *persistent* (i.e., that their current values are available every time a Python program runs). The following file, *updatedb.py*, prints the database and gives a raise to one of our stored objects each time. If you trace through what's going on here, you'll notice that we're getting a lot of utility "for free"—printing our objects automatically employs the general `__str__` overloading method, and we give raises by calling the `giveRaise` method we wrote earlier. This all "just works" for objects based on OOP's inheritance model, even when they live in a file:

```
# File updatedb.py: update Person object on database

import shelve
db = shelve.open('persondb')                # Reopen shelve with same filename

for key in sorted(db):                      # Iterate to display database objects
    print(key, '\t=>', db[key])             # Prints with custom format

sue = db['Sue Jones']                       # Index by key to fetch
sue.giveRaise(.10)                          # Update in memory using class method
db['Sue Jones'] = sue                       # Assign to key to update in shelve
db.close()                                  # Close after making changes
```

Because this script prints the database when it starts up, we have to run it a few times to see our objects change. Here it is in action, displaying all records and increasing `sue`'s pay each time it's run (it's a pretty good script for `sue`...):

```
c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=121000]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
```

```
Bob Smith       => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones       => [Person: job=dev, name=Sue Jones, pay=133100]
Tom Jones       => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

Again, what we see here is a product of the `shelve` and `pickle` tools we get from Python, and of the behavior we coded in our classes ourselves. And once again, we can verify our script's work at the interactive prompt (the shelve's equivalent of a database client):

```
c:\misc> python
>>> import shelve
>>> db = shelve.open('persondb')              # Reopen database
>>> rec = db['Sue Jones']                     # Fetch object by key
>>> print(rec)
[Person: job=dev, name=Sue Jones, pay=146410]
>>> rec.lastName()
'Jones'
>>> rec.pay
146410
```

For another example of object persistence in this book, see the sidebar in Chapter 30 titled "Why You Will Care: Classes and Persistence" on page 744. It stores a somewhat larger composite object in a flat file with `pickle` instead of `shelve`, but the effect is similar. For more details on both pickles and shelves, see other books or Python's manuals.

Future Directions

And that's a wrap for this tutorial. At this point, you've seen all the basics of Python's OOP machinery in action, and you've learned ways to avoid redundancy and its associated maintenance issues in your code. You've built full-featured classes that do real work. As an added bonus, you've made them real database records by storing them in a Python shelve, so their information lives on persistently.

There is much more we could explore here, of course. For example, we could extend our classes to make them more realistic, add new kinds of behavior to them, and so on. Giving a raise, for instance, should in practice verify that pay increase rates are between zero and one—an extension we'll add when we meet decorators later in this book. You might also mutate this example into a personal contacts database, by changing the state information stored on objects, as well as the class methods used to process it. We'll leave this a suggested exercise open to your imagination.

We could also expand our scope to use tools that either come with Python or are freely available in the open source world:

GUIs
: As is, we can only process our database with the interactive prompt's command-based interface, and scripts. We could also work on expanding our object database's usability by adding a graphical user interface for browsing and updating its records. GUIs can be built portably with either Python's `tkinter` (`Tkinter` in 2.6)

standard library support, or third-party toolkits such as WxPython and PyQt. `tkinter` ships with Python, lets you build simple GUIs quickly, and is ideal for learning GUI programming techniques; WxPython and PyQt tend to be more complex to use but often produce higher-grade GUIs in the end.

Websites
: Although GUIs are convenient and fast, the Web is hard to beat in terms of accessibility. We might also implement a website for browsing and updating records, instead of or in addition to GUIs and the interactive prompt. Websites can be constructed with either basic CGI scripting tools that come with Python, or full-featured third-party web frameworks such as Django, TurboGears, Pylons, web2Py, Zope, or Google's App Engine. On the Web, your data can still be stored in a shelve, pickle file, or other Python-based medium; the scripts that process it are simply run automatically on a server in response to requests from web browsers and other clients, and they produce HTML to interact with a user, either directly or by interfacing with Framework APIs.

Web services
: Although web clients can often parse information in the replies from websites (a technique colorfully known as "screen scraping"), we might go further and provide a more direct way to fetch records on the Web via a web services interface such as SOAP or XML-RPC calls—APIs supported by either Python itself or the third-party open source domain. Such APIs return data in a more direct form, rather than embedded in the HTML of a reply page.

Databases
: If our database becomes higher-volume or critical, we might eventually move it from shelves to a more full-featured storage mechanism such as the open source ZODB object-oriented database system (OODB), or a more traditional SQL-based relational database system such as MySQL, Oracle, PostgreSQL, or SQLite. Python itself comes with the in-process SQLite database system built-in, but other open source options are freely available on the Web. ZODB, for example, is similar to Python's `shelve` but addresses many of its limitations, supporting larger databases, concurrent updates, transaction processing, and automatic write-through on in-memory changes. SQL-based systems like MySQL offer enterprise-level tools for database storage and may be directly used from a within a Python script.

ORMs
: If we do migrate to a relational database system for storage, we don't have to sacrifice Python's OOP tools. Object-relational mappers (ORMs) like SQLObject and SQLAlchemy can automatically map relational tables and rows to and from Python classes and instances, such that we can process the stored data using normal Python class syntax. This approach provides an alternative to OODBs like `shelve` and ZODB and leverages the power of both relational databases and Python's class model.

While I hope this introduction whets your appetite for future exploration, all of these topics are of course far beyond the scope of this tutorial and this book at large. If you want to explore any of them on your own, see the Web, Python's standard library manuals, and application-focused books such as *Programming Python*. In the latter I pick up this example where we've stopped here, showing how to add both a GUI and a website on top of the database to allow for browsing and updating instance records. I hope to see you there eventually, but first, let's return to class fundamentals and finish up the rest of the core Python language story.

Chapter Summary

In this chapter, we explored all the fundamentals of Python classes and OOP in action, by building upon a simple but real example, step by step. We added constructors, methods, operator overloading, customization with subclasses, and introspection tools, and we met other concepts (such as composition, delegation, and polymorphism) along the way.

In the end, we took objects created by our classes and made them persistent by storing them on a shelve object database—an easy-to-use system for saving and retrieving native Python objects by key. While exploring class basics, we also encountered multiple ways to factor our code to reduce redundancy and minimize future maintenance costs. Finally, we briefly previewed ways to extend our code with application-programming tools such as GUIs and databases, covered in follow-up books.

In the next chapters of this part of the book we'll return to our study of the details behind Python's class model and investigate its application to some of the design concepts used to combine classes in larger programs. Before we move ahead, though, let's work through this chapter's quiz to review what we covered here. Since we've already done a lot of hands-on work in this chapter, we'll close with a set of mostly theory-oriented questions designed to make you trace through some of the code and ponder some of the bigger ideas behind it.

Test Your Knowledge: Quiz

1. When we fetch a `Manager` object from the shelve and print it, where does the display format logic come from?
2. When we fetch a `Person` object from a shelve without importing its module, how does the object know that it has a `giveRaise` method that we can call?
3. Why is it so important to move processing into methods, instead of hardcoding it outside the class?

4. Why is it better to customize by subclassing rather than copying the original and modifying?
5. Why is it better to call back to a superclass method to run default actions, instead of copying and modifying its code in a subclass?
6. Why is it better to use tools like `__dict__` that allow objects to be processed generically than to write more custom code for each type of class?
7. In general terms, when might you choose to use object embedding and composition instead of inheritance?
8. How might you modify the classes in this chapter to implement a personal contacts database in Python?

Test Your Knowledge: Answers

1. In the final version of our classes, `Manager` ultimately inherits its `__str__` printing method from `AttrDisplay` in the separate `classtools` module. `Manager` doesn't have one itself, so the inheritance search climbs to its `Person` superclass; because there is no `__str__` there either, the search climbs higher and finds it in `AttrDisplay`. The class names listed in parentheses in a `class` statement's header line provide the links to higher superclasses.
2. Shelves (really, the `pickle` module they use) automatically relink an instance to the class it was created from when that instance is later loaded back into memory. Python reimports the class from its module internally, creates an instance with its stored attributes, and sets the instance's `__class__` link to point to its original class. This way, loaded instances automatically obtain all their original methods (like `lastName`, `giveRaise`, and `__str__`), even if we have not imported the instance's class into our scope.
3. It's important to move processing into methods so that there is only one copy to change in the future, and so that the methods can be run on any instance. This is Python's notion of *encapsulation*—wrapping up logic behind interfaces, to better support future code maintenance. If you don't do so, you create code redundancy that can multiply your work effort as the code evolves in the future.
4. Customizing with subclasses reduces development effort. In OOP, we code by *customizing* what has already been done, rather than copying or changing existing code. This is the real "big idea" in OOP—because we can easily extend our prior work by coding new subclasses, we can leverage what we've already done. This is much better than either starting from scratch each time, or introducing multiple redundant copies of code that may all have to be updated in the future.

5. Copying and modifying code *doubles* your potential work effort in the future, regardless of the context. If a subclass needs to perform default actions coded in a superclass method, it's much better to call back to the original through the superclass's name than to copy its code. This also holds true for superclass constructors. Again, copying code creates redundancy, which is a major issue as code evolves.
6. Generic tools can avoid hardcoded solutions that must be kept in sync with the rest of the class as it evolves over time. A generic `__str__` print method, for example, need not be updated each time a new attribute is added to instances in an `__init__` constructor. In addition, a generic `print` method inherited by all classes only appears, and need only be modified, in one place—changes in the generic version are picked up by all classes that inherit from the generic class. Again, eliminating code *redundancy* cuts future development effort; that's one of the primary assets classes bring to the table.
7. Inheritance is best at coding extensions based on direct customization (like our `Manager` specialization of `Person`). Composition is well suited to scenarios where multiple objects are aggregated into a whole and directed by a controller layer class. Inheritance passes calls up to reuse, and composition passes down to delegate. Inheritance and composition are not mutually exclusive; often, the objects embedded in a controller are themselves customizations based upon inheritance.
8. The classes in this chapter could be used as boilerplate "template" code to implement a variety of types of databases. Essentially, you can repurpose them by modifying the constructors to record different attributes and providing whatever methods are appropriate for the target application. For instance, you might use attributes such as `name`, `address`, `birthday`, `phone`, `email`, and so on for a contacts database, and methods appropriate for this purpose. A method named `sendmail`, for example, might use Python's standard library `smptlib` module to send an email to one of the contacts automatically when called (see Python's manuals or application-level books for more details on such tools). The `AttrDisplay` tool we wrote here could be used verbatim to print your objects, because it is intentionally generic. Most of the shelve database code here can be used to store your objects, too, with minor changes.

CHAPTER 28

Class Coding Details

If you haven't quite gotten all of Python OOP yet, don't worry; now that we've had a quick tour, we're going to dig a bit deeper and study the concepts introduced earlier in further detail. In this and the following chapter, we'll take another look at class mechanics. Here, we're going to study classes, methods, and inheritance, formalizing and expanding on some of the coding ideas introduced in Chapter 26. Because the class is our last namespace tool, we'll summarize Python's namespace concepts here as well.

The next chapter continues this in-depth second pass over class mechanics by covering one specific aspect: operator overloading. Besides presenting the details, this chapter and the next also give us an opportunity to explore some larger classes than those we have studied so far.

The class Statement

Although the Python `class` statement may seem similar to tools in other OOP languages on the surface, on closer inspection, it is quite different from what some programmers are used to. For example, as in C++, the `class` statement is Python's main OOP tool, but unlike in C++, Python's `class` is not a declaration. Like a `def`, a `class` statement is an object builder, and an implicit assignment—when run, it generates a class object and stores a reference to it in the name used in the header. Also like a `def`, a `class` statement is true executable code—your class doesn't exist until Python reaches and runs the `class` statement that defines it (typically while importing the module it is coded in, but not before).

General Form

`class` is a compound statement, with a body of indented statements typically appearing under the header. In the header, superclasses are listed in parentheses after the class name, separated by commas. Listing more than one superclass leads to multiple inheritance (which we'll discuss more formally in Chapter 30). Here is the statement's general form:

```
class <name>(superclass,...):          # Assign to name
    data = value                       # Shared class data
    def method(self,...):              # Methods
        self.member = value            # Per-instance data
```

Within the `class` statement, any assignments generate class attributes, and specially named methods overload operators; for instance, a function called `__init__` is called at instance object construction time, if defined.

Example

As we've seen, classes are mostly just namespaces—that is, tools for defining names (i.e., attributes) that export data and logic to clients. So, how do you get from the `class` statement to a namespace?

Here's how. Just like in a module file, the statements nested in a `class` statement body create its attributes. When Python executes a `class` statement (not a call to a class), it runs all the statements in its body, from top to bottom. Assignments that happen during this process create names in the class's local scope, which become attributes in the associated class object. Because of this, classes resemble both modules and functions:

- Like functions, `class` statements are local scopes where names created by nested assignments live.
- Like names in a module, names assigned in a `class` statement become attributes in a class object.

The main distinction for classes is that their namespaces are also the basis of inheritance in Python; reference attributes that are not found in a class or instance object are fetched from other classes.

Because `class` is a compound statement, any sort of statement can be nested inside its body—`print`, `=`, `if`, `def`, and so on. All the statements inside the `class` statement run when the `class` statement itself runs (not when the class is later called to make an instance). Assigning names inside the `class` statement makes class attributes, and nested `def`s make class methods, but other assignments make attributes, too.

For example, assignments of simple nonfunction objects to class attributes produce *data attributes*, shared by all instances:

```
>>> class SharedData:
...     spam = 42          # Generates a class data attribute
...
>>> x = SharedData()       # Make two instances
>>> y = SharedData()
>>> x.spam, y.spam         # They inherit and share 'spam'
(42, 42)
```

Here, because the name `spam` is assigned at the top level of a `class` statement, it is attached to the class and so will be shared by all instances. We can change it by going through the class name, and we can refer to it through either instances or the class.*

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

Such class attributes can be used to manage information that spans all the instances—a counter of the number of instances generated, for example (we'll expand on this idea by example in Chapter 31). Now, watch what happens if we assign the name `spam` through an instance instead of the class:

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

Assignments to instance attributes create or change the names in the instance, rather than in the shared class. More generally, inheritance searches occur only on attribute *references*, not on assignment: assigning to an object's attribute always changes that object, and no other.† For example, `y.spam` is looked up in the class by inheritance, but the assignment to `x.spam` attaches a name to `x` itself.

Here's a more comprehensive example of this behavior that stores the same name in two places. Suppose we run the following class:

```
class MixedNames:                          # Define class
    data = 'spam'                          # Assign class attr
    def __init__(self, value):             # Assign method name
        self.data = value                  # Assign instance attr
    def display(self):
        print(self.data, MixedNames.data)  # Instance attr, class attr
```

This class contains two `def`s, which bind class attributes to method functions. It also contains an `=` assignment statement; because this assignment assigns the name `data` inside the `class`, it lives in the class's local scope and becomes an attribute of the class object. Like all class attributes, this `data` is inherited and shared by all instances of the class that don't have `data` attributes of their own.

When we make instances of this class, the name `data` is attached to those instances by the assignment to `self.data` in the constructor method:

```
>>> x = MixedNames(1)          # Make two instance objects
>>> y = MixedNames(2)          # Each has its own data
```

* If you've used C++ you may recognize this as similar to the notion of C++'s "static" data members—members that are stored in the class, independent of instances. In Python, it's nothing special: all class attributes are just names assigned in the `class` statement, whether they happen to reference functions (C++'s "methods") or something else (C++'s "members"). In Chapter 31, we'll also meet Python static methods (akin to those in C++), which are just self-less functions that usually process class attributes.

† Unless the class has redefined the attribute assignment operation to do something unique with the `__setattr__` operator overloading method (discussed in Chapter 29).

```
>>> x.display(); y.display()    # self.data differs, MixedNames.data is the same
1 spam
2 spam
```

The net result is that `data` lives in two places: in the instance objects (created by the `self.data` assignment in `__init__`), and in the class from which they inherit names (created by the `data` assignment in the `class`). The class's `display` method prints both versions, by first qualifying the `self` instance, and then the class.

By using these techniques to store attributes in different objects, we determine their scope of visibility. When attached to classes, names are shared; in instances, names record per-instance data, not shared behavior or data. Although inheritance searches look up names for us, we can always get to an attribute anywhere in a tree by accessing the desired object directly.

In the preceding example, for instance, specifying `x.data` or `self.data` will return an instance name, which normally hides the same name in the class; however, `MixedNames.data` grabs the class name explicitly. We'll see various roles for such coding patterns later; the next section describes one of the most common.

Methods

Because you already know about functions, you also know about methods in classes. Methods are just function objects created by `def` statements nested in a `class` statement's body. From an abstract perspective, methods provide behavior for instance objects to inherit. From a programming perspective, methods work in exactly the same way as simple functions, with one crucial exception: a method's first argument always receives the instance object that is the implied subject of the method call.

In other words, Python automatically maps instance method calls to class method functions as follows. Method calls made through an instance, like this:

```
instance.method(args...)
```

are automatically translated to class method function calls of this form:

```
class.method(instance, args...)
```

where the class is determined by locating the method name using Python's inheritance search procedure. In fact, both call forms are valid in Python.

Besides the normal inheritance of method attribute names, the special first argument is the only real magic behind method calls. In a class method, the first argument is usually called `self` by convention (technically, only its position is significant, not its name). This argument provides methods with a hook back to the instance that is the subject of the call—because classes generate many instance objects, they need to use this argument to manage data that varies per instance.

C++ programmers may recognize Python's `self` argument as being similar to C++'s `this` pointer. In Python, though, `self` is always explicit in your code: methods must always go through `self` to fetch or change attributes of the instance being processed by the current method call. This explicit nature of `self` is by design—the presence of this name makes it obvious that you are using instance attribute names in your script, not names in the local or global scope.

Method Example

To clarify these concepts, let's turn to an example. Suppose we define the following class:

```
class NextClass:                        # Define class
    def printer(self, text):            # Define method
        self.message = text             # Change instance
        print(self.message)             # Access instance
```

The name `printer` references a function object; because it's assigned in the `class` statement's scope, it becomes a class object attribute and is inherited by every instance made from the class. Normally, because methods like `printer` are designed to process instances, we call them through instances:

```
>>> x = NextClass()                     # Make instance

>>> x.printer('instance call')          # Call its method
instance call

>>> x.message                           # Instance changed
'instance call'
```

When we call the method by qualifying an instance like this, `printer` is first located by inheritance, and then its `self` argument is automatically assigned the instance object (`x`); the `text` argument gets the string passed at the call (`'instance call'`). Notice that because Python automatically passes the first argument to `self` for us, we only actually have to pass in one argument. Inside `printer`, the name `self` is used to access or set per-instance data because it refers back to the instance currently being processed.

Methods may be called in one of two ways—through an instance, or through the class itself. For example, we can also call `printer` by going through the class name, provided we pass an instance to the `self` argument explicitly:

```
>>> NextClass.printer(x, 'class call')  # Direct class call
class call

>>> x.message                           # Instance changed again
'class call'
```

Calls routed through the instance and the class have the exact same effect, as long as we pass the same instance object ourselves in the class form. By default, in fact, you get an error message if you try to call a method without any instance:

```
>>> NextClass.printer('bad call')
TypeError: unbound method printer() must be called with NextClass instance...
```

Calling Superclass Constructors

Methods are normally called through instances. Calls to methods through a class, though, do show up in a variety of special roles. One common scenario involves the constructor method. The `__init__` method, like all attributes, is looked up by inheritance. This means that at construction time, Python locates and calls just one `__init__`. If subclass constructors need to guarantee that superclass construction-time logic runs, too, they generally must call the superclass's `__init__` method explicitly through the class:

```
class Super:
    def __init__(self, x):
        ...default code...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)             # Run superclass __init__
        ...custom code...                   # Do my init actions

I = Sub(1, 2)
```

This is one of the few contexts in which your code is likely to call an operator overloading method directly. Naturally, you should only call the superclass constructor this way if you really want it to run—without the call, the subclass replaces it completely. For a more realistic illustration of this technique in action, see the `Manager` class example in the prior chapter's tutorial.‡

Other Method Call Possibilities

This pattern of calling methods through a class is the general basis of extending (instead of completely replacing) inherited method behavior. In Chapter 31, we'll also meet a new option added in Python 2.2, *static methods*, that allow you to code methods that do not expect instance objects in their first arguments. Such methods can act like simple instanceless functions, with names that are local to the classes in which they are coded, and may be used to manage class data. A related concept, the *class method*, receives a class when called instead of an instance and can be used to manage per-class data. These are advanced and optional extensions, though; normally, you must always pass an instance to a method, whether it is called through an instance or a class.

‡ On a somewhat related note, you can also code multiple `__init__` methods within the same class, but only the last definition will be used; see Chapter 30 for more details on multiple method definitions.

Inheritance

The whole point of a namespace tool like the `class` statement is to support name inheritance. This section expands on some of the mechanisms and roles of attribute inheritance in Python.

In Python, inheritance happens when an object is qualified, and it involves searching an attribute definition tree (one or more namespaces). Every time you use an expression of the form *`object.attr`* (where *`object`* is an instance or class object), Python searches the namespace tree from bottom to top, beginning with *`object`*, looking for the first *`attr`* it can find. This includes references to `self` attributes in your methods. Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.

Attribute Tree Construction

Figure 28-1 summarizes the way namespace trees are constructed and populated with names. Generally:

- Instance attributes are generated by assignments to `self` attributes in methods.
- Class attributes are created by statements (assignments) in `class` statements.
- Superclass links are made by listing classes in parentheses in a `class` statement header.

The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the `class` header. Python searches upward in this tree, from instances to superclasses, each time you use qualification to fetch an attribute name from an instance object.§

Specializing Inherited Methods

The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in subclasses before it checks superclasses, subclasses can replace default behavior by redefining their superclasses' attributes. In fact, you can build entire systems as hierarchies of classes, which are extended by adding new external subclasses rather than changing existing logic in-place.

§ This description isn't 100% complete, because we can also create instance and class attributes by assigning to objects outside `class` statements—but that's a much less common and sometimes more error-prone approach (changes aren't isolated to `class` statements). In Python, all attributes are always accessible by default. We'll talk more about attribute name privacy in Chapter 29 when we study `__setattr__`, in Chapter 30 when we meet `__X` names, and again in Chapter 38, where we'll implement it with a class decorator.

The idea of redefining inherited names leads to a variety of specialization techniques. For instance, subclasses may *replace* inherited attributes completely, *provide* attributes that a superclass expects to find, and *extend* superclass methods by calling back to the superclass from an overridden method. We've already seen replacement in action. Here's an example that shows how extension works:

```
>>> class Super:
...     def method(self):
...         print('in Super.method')
...
>>> class Sub(Super):
...     def method(self):                    # Override method
...         print('starting Sub.method')     # Add actions here
...         Super.method(self)               # Run default action
...         print('ending Sub.method')
...
```

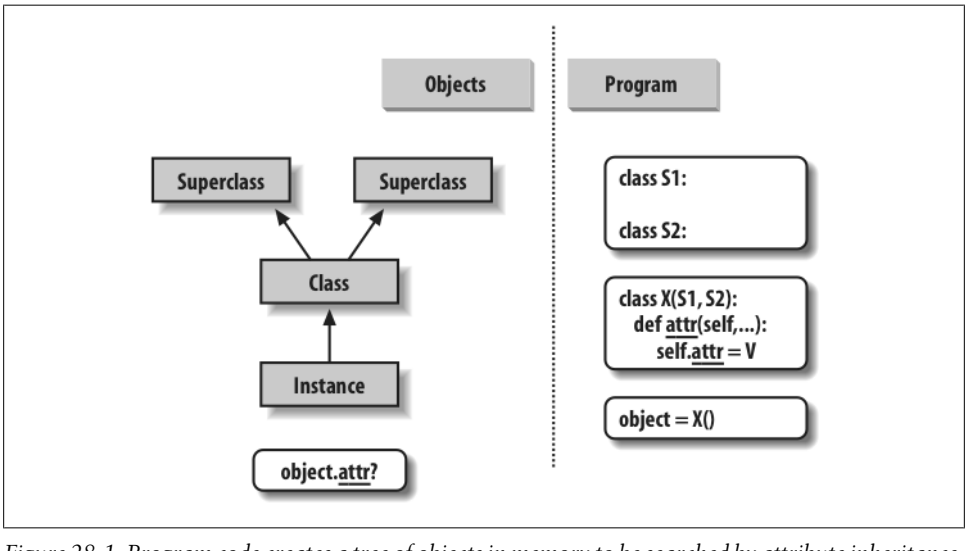

Figure 28-1. Program code creates a tree of objects in memory to be searched by attribute inheritance. Calling a class creates a new instance that remembers its class, running a class statement creates a new class, and superclasses are listed in parentheses in the class statement header. Each attribute reference triggers a new bottom-up tree search—even references to self attributes within a class's methods.

Direct superclass method calls are the crux of the matter here. The `Sub` class replaces `Super`'s `method` function with its own specialized version, but within the replacement, `Sub` calls back to the version exported by `Super` to carry out the default behavior. In other words, `Sub.method` just extends `Super.method`'s behavior, rather than replacing it completely:

```
>>> x = Super()                    # Make a Super instance
>>> x.method()                     # Runs Super.method
in Super.method

>>> x = Sub()                      # Make a Sub instance
>>> x.method()                     # Runs Sub.method, calls Super.method
starting Sub.method
in Super.method
ending Sub.method
```

This extension coding pattern is also commonly used with constructors; see the section "Methods" on page 684 for an example.

Class Interface Techniques

Extension is only one way to interface with a superclass. The file shown in this section, *specialize.py*, defines multiple classes that illustrate a variety of common techniques:

`Super`
: Defines a `method` function and a `delegate` that expects an `action` in a subclass.

`Inheritor`
: Doesn't provide any new names, so it gets everything defined in `Super`.

`Replacer`
: Overrides `Super`'s `method` with a version of its own.

`Extender`
: Customizes `Super`'s `method` by overriding and calling back to run the default.

`Provider`
: Implements the `action` method expected by `Super`'s `delegate` method.

Study each of these subclasses to get a feel for the various ways they customize their common superclass. Here's the file:

```
class Super:
    def method(self):
        print('in Super.method')            # Default behavior
    def delegate(self):
        self.action()                       # Expected to be defined

class Inheritor(Super):                     # Inherit method verbatim
    pass

class Replacer(Super):                      # Replace method completely
    def method(self):
        print('in Replacer.method')

class Extender(Super):                      # Extend method behavior
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')
```

```
class Provider(Super):                          # Fill in a required method
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

A few things are worth pointing out here. First, the self-test code at the end of this example creates instances of three different classes in a `for` loop. Because classes are objects, you can put them in a tuple and create instances generically (more on this idea later). Classes also have the special `__name__` attribute, like modules; it's preset to a string containing the name in the class header. Here's what happens when we run the file:

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Abstract Superclasses

Notice how the `Provider` class in the prior example works. When we call the `delegate` method through a `Provider` instance, *two* independent inheritance searches occur:

1. On the initial `x.delegate` call, Python finds the `delegate` method in `Super` by searching the `Provider` instance and above. The instance `x` is passed into the method's `self` argument as usual.
2. Inside the `Super.delegate` method, `self.action` invokes a new, independent inheritance search of `self` and above. Because `self` references a `Provider` instance, the `action` method is located in the `Provider` subclass.

This "filling in the blanks" sort of coding structure is typical of OOP frameworks. At least in terms of the `delegate` method, the superclass in this example is what is sometimes called an *abstract superclass*—a class that expects parts of its behavior to be

provided by its subclasses. If an expected method is not defined in a subclass, Python raises an undefined name exception when the inheritance search fails.

Class coders sometimes make such subclass requirements more obvious with `assert` statements, or by raising the built-in `NotImplementedError` exception with `raise` statements (we'll study statements that may trigger exceptions in depth in the next part of this book). As a quick preview, here's the `assert` scheme in action:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'       # If this version is called

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

We'll meet `assert` in Chapters 32 and 33; in short, if its first expression evaluates to false, it raises an exception with the provided error message. Here, the expression is always false so as to trigger an error message if a method is not redefined, and inheritance locates the version here. Alternatively, some classes simply raise a `NotImplementedError` exception directly in such method stubs to signal the mistake:

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')

>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

For instances of subclasses, we still get the exception unless the subclass provides the expected method to replace the default in the superclass:

```
>>> class Sub(Super): pass
...
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam
```

For a somewhat more realistic example of this section's concepts in action, see the "Zoo animal hierarchy" exercise (exercise 8) at the end of Chapter 31, and its solution in "Part VI, Classes and OOP" on page 1122 in Appendix B. Such taxonomies are a

traditional way to introduce OOP, but they're a bit removed from most developers' job descriptions.

Python 2.6 and 3.0 Abstract Superclasses

As of Python 2.6 and 3.0, the prior section's abstract superclasses (a.k.a. "abstract base classes"), which require methods to be filled in by subclasses, may also be implemented with special class syntax. The way we code this varies slightly depending on the version. In Python 3.0, we use a keyword argument in a `class` header, along with special `@` decorator syntax, both of which we'll study in detail later in this book:

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):
        pass
```

But in Python 2.6, we use a class attribute instead:

```
class Super:
    __metaclass__ = ABCMeta
    @abstractmethod
    def method(self, ...):
        pass
```

Either way, the effect is the same—we can't make an instance unless the method is defined lower in the class tree. In 3.0, for example, here is the special syntax equivalent of the prior section's example:

```
>>> from abc import ABCMeta, abstractmethod
>>>
>>> class Super(metaclass=ABCMeta):
...     def delegate(self):
...         self.action()
...     @abstractmethod
...     def action(self):
...         pass
...
>>> X = Super()
TypeError: Can't instantiate abstract class Super with abstract methods action

>>> class Sub(Super): pass
...
>>> X = Sub()
TypeError: Can't instantiate abstract class Sub with abstract methods action

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam
```

Coded this way, a class with an abstract method cannot be instantiated (that is, we cannot create an instance by calling it) unless all of its abstract methods have been defined in subclasses. Although this requires more code, the advantage of this approach is that errors for missing methods are issued when we attempt to make an instance of the class, not later when we try to call a missing method. This feature may also be used to define an expected interface, automatically verified in client classes.

Unfortunately, this scheme also relies on two advanced language tools we have not met yet—*function decorators*, introduced in Chapter 31 and covered in depth in Chapter 38, as well as *metaclass declarations*, mentioned in Chapter 31 and covered in Chapter 39—so we will finesse other facets of this option here. See Python's standard manuals for more on this, as well as precoded abstract superclasses Python provides.

Namespaces: The Whole Story

Now that we've examined class and instance objects, the Python namespace story is complete. For reference, I'll quickly summarize all the rules used to resolve names here. The first things you need to remember are that qualified and unqualified names are treated differently, and that some scopes serve to initialize object namespaces:

- Unqualified names (e.g., `X`) deal with scopes.
- Qualified attribute names (e.g., `object.X`) use object namespaces.
- Some scopes initialize object namespaces (for modules and classes).

Simple Names: Global Unless Assigned

Unqualified simple names follow the LEGB lexical scoping rule outlined for functions in Chapter 17:

Assignment (`X = value`)
: Makes names local: creates or changes the name `X` in the current local scope, unless declared global.

Reference (`X`)
: Looks for the name `X` in the current local scope, then any and all enclosing functions, then the current global scope, then the built-in scope.

Attribute Names: Object Namespaces

Qualified attribute names refer to attributes of specific objects and obey the rules for modules and classes. For class and instance objects, the reference rules are augmented to include the inheritance search procedure:

Assignment (`object.X = value`)
: Creates or alters the attribute name `X` in the namespace of the *`object`* being qualified, and none other. Inheritance-tree climbing happens only on attribute reference, not on attribute assignment.

Reference (`object.X`)
: For class-based objects, searches for the attribute name `X` in *`object`*, then in all accessible classes above it, using the inheritance search procedure. For nonclass objects such as modules, fetches `X` from *`object`* directly.

The "Zen" of Python Namespaces: Assignments Classify Names

With distinct search procedures for qualified and unqualified names, and multiple lookup layers for both, it can sometimes be difficult to tell where a name will wind up going. In Python, the place where you *assign* a name is crucial—it fully determines the scope or object in which a name will reside. The file *manynames.py* illustrates how this principle translates to code and summarizes the namespace ideas we have seen throughout this book:

```
# manynames.py

X = 11                          # Global (module) name/attribute (X, or manynames.X)

def f():
    print(X)                    # Access global X (11)

def g():
    X = 22                      # Local (function) variable (X, hides module X)
    print(X)

class C:
    X = 33                      # Class attribute (C.X)
    def m(self):
        X = 44                  # Local variable in method (X)
        self.X = 55             # Instance attribute (instance.X)
```

This file assigns the same name, `X`, five times. Because this name is assigned in five different locations, though, all five `X`s in this program are completely different variables. From top to bottom, the assignments to `X` here generate: a module attribute (`11`), a local variable in a function (`22`), a class attribute (`33`), a local variable in a method (`44`), and an instance attribute (`55`). Although all five are named `X`, the fact that they are all assigned at different places in the source code or to different objects makes all of these unique variables.

You should take the time to study this example carefully because it collects ideas we've been exploring throughout the last few parts of this book. When it makes sense to you, you will have achieved a sort of Python namespace nirvana. Of course, an alternative route to nirvana is to simply run the program and see what happens. Here's the remainder of this source file, which makes an instance and prints all the `X`s that it can fetch:

```
# manynames.py, continued

if __name__ == '__main__':
    print(X)                    # 11: module (a.k.a. manynames.X outside file)
    f()                         # 11: global
    g()                         # 22: local
    print(X)                    # 11: module name unchanged

    obj = C()                   # Make instance
    print(obj.X)                # 33: class name inherited by instance

    obj.m()                     # Attach attribute name X to instance now
    print(obj.X)                # 55: instance
    print(C.X)                  # 33: class (a.k.a. obj.X if no X in instance)

    #print(C.m.X)               # FAILS: only visible in method
    #print(g.X)                 # FAILS: only visible in function
```

The outputs that are printed when the file is run are noted in the comments in the code; trace through them to see which variable named `X` is being accessed each time. Notice in particular that we can go through the class to fetch its attribute (`C.X`), but we can never fetch local variables in functions or methods from outside their `def` statements. Locals are visible only to other code within the `def`, and in fact only live in memory while a call to the function or method is executing.

Some of the names defined by this file are visible *outside the file* to other modules, but recall that we must always import before we can access names in another file—that is the main point of modules, after all:

```
# otherfile.py

import manynames

X = 66
print(X)                        # 66: the global here
print(manynames.X)              # 11: globals become attributes after imports

manynames.f()                   # 11: manynames's X, not the one here!
manynames.g()                   # 22: local in other file's function

print(manynames.C.X)            # 33: attribute of class in other module
I = manynames.C()
print(I.X)                      # 33: still from class here
I.m()
print(I.X)                      # 55: now from instance!
```

Notice here how `manynames.f()` prints the `X` in `manynames`, not the `X` assigned in this file—scopes are always determined by the position of assignments in your source code (i.e., lexically) and are never influenced by what imports what or who imports whom. Also, notice that the instance's own `X` is not created until we call `I.m()`—attributes, like all variables, spring into existence when assigned, and not before. Normally we create instance attributes by assigning them in class `__init__` constructor methods, but this isn't the only option.

Finally, as we learned in Chapter 17, it's also possible for a function to *change* names outside itself, with `global` and (in Python 3.0) `nonlocal` statements—these statements provide write access, but also modify assignment's namespace binding rules:

```
X = 11                          # Global in module

def g1():
    print(X)                    # Reference global in module

def g2():
    global X
    X = 22                      # Change global in module

def h1():
    X = 33                      # Local in function
    def nested():
        print(X)                # Reference local in enclosing scope

def h2():
    X = 33                      # Local in function
    def nested():
        nonlocal X              # Python 3.0 statement
        X = 44                  # Change local in enclosing scope
```

Of course, you generally shouldn't use the same name for every variable in your script—but as this example demonstrates, even if you do, Python's namespaces will work to keep names used in one context from accidentally clashing with those used in another.

Namespace Dictionaries

In Chapter 22, we learned that module namespaces are actually implemented as dictionaries and exposed with the built-in `__dict__` attribute. The same holds for class and instance objects: attribute qualification is really a dictionary indexing operation internally, and attribute inheritance is just a matter of searching linked dictionaries. In fact, instance and class objects are mostly just dictionaries with links inside Python. Python exposes these dictionaries, as well as the links between them, for use in advanced roles (e.g., for coding tools).

To help you understand how attributes work internally, let's work through an interactive session that traces the way namespace dictionaries grow when classes are involved. We saw a simpler version of this type of code in Chapter 26, but now that we know more about methods and superclasses, let's embellish it here. First, let's define a superclass and a subclass with methods that will store data in their instances:

```
>>> class super:
...     def hello(self):
...         self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
```

```
...         self.data2 = 'eggs'
...
```

When we make an instance of the subclass, the instance starts out with an empty namespace dictionary, but it has links back to the class for the inheritance search to follow. In fact, the inheritance tree is explicitly available in special attributes, which you can inspect. Instances have a `__class__` attribute that links to their class, and classes have a `__bases__` attribute that is a tuple containing links to higher superclasses (I'm running this on Python 3.0; name formats and some internal attributes vary slightly in 2.6):

```
>>> X = sub()
>>> X.__dict__                          # Instance namespace dict
{}

>>> X.__class__                         # Class of instance
<class '__main__.sub'>

>>> sub.__bases__                       # Superclasses of class
(<class '__main__.super'>,)

>>> super.__bases__                     # () empty tuple in Python 2.6
(<class 'object'>,)
```

As classes assign to `self` attributes, they populate the instance objects—that is, attributes wind up in the instances' attribute namespace dictionaries, not in the classes'. An instance object's namespace records data that can vary from instance to instance, and `self` is a hook into that namespace:

```
>>> Y = sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}

>>> sub.__dict__.keys()
['__module__', '__doc__', 'hola']

>>> super.__dict__.keys()
['__dict__', '__module__', '__weakref__', 'hello', '__doc__']

>>> Y.__dict__
{}
```

Notice the extra underscore names in the class dictionaries; Python sets these automatically. Most are not used in typical programs, but there are tools that use some of them (e.g., `__doc__` holds the docstrings discussed in Chapter 15).

Also, observe that `Y`, a second instance made at the start of this series, still has an empty namespace dictionary at the end, even though `X`'s dictionary has been populated by assignments in methods. Again, each instance has an independent namespace dictionary, which starts out empty and can record completely different attributes than those recorded by the namespace dictionaries of other instances of the same class.

Because attributes are actually dictionary keys inside Python, there are really two ways to fetch and assign their values—by qualification, or by key indexing:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')

>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

This equivalence applies only to attributes actually attached to the instance, though. Because attribute fetch qualification also performs an inheritance search, it can access attributes that namespace dictionary indexing cannot. The inherited attribute `X.hello`, for instance, cannot be accessed by `X.__dict__['hello']`.

Finally, here is the built-in `dir` function we met in Chapters 4 and 15 at work on class and instance objects. This function works on anything with attributes: `dir(`*`object`*`)` is similar to an *`object`*`.__dict__.keys()` call. Notice, though, that `dir` sorts its list and includes some system attributes. As of Python 2.2, `dir` also collects inherited attributes automatically, and in 3.0 it includes names inherited from the `object` class that is an implied superclass of all classes:‖

```
>>> X.__dict__, Y.__dict__
({'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}, {})
>>> list(X.__dict__.keys())                              # Need list in 3.0
['data1', 'data3', 'data2']

# In Python 2.6:

>>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']
```

‖ As you can see, the contents of attribute dictionaries and `dir` call results may change over time. For example, because Python now allows built-in types to be subclassed like classes, the contents of `dir` results for built-in types have expanded to include operator overloading methods, just like our `dir` results here for user-defined classes under Python 3.0. In general, attribute names with leading and trailing double underscores are interpreter-specific. Type subclasses will be discussed further in Chapter 31.

```
# In Python 3.0:

>>> dir(X)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'data1', 'data2', 'data3', 'hello', 'hola']

>>> dir(sub)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello', 'hola']

>>> dir(super)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello'
]
```

Experiment with these special attributes on your own to get a better feel for how namespaces actually do their attribute business. Even if you will never use these in the kinds of programs you write, seeing that they are just normal dictionaries will help demystify the notion of namespaces in general.

Namespace Links

The prior section introduced the special `__class__` and `__bases__` instance and class attributes, without really explaining why you might care about them. In short, these attributes allow you to inspect inheritance hierarchies within your own code. For example, they can be used to display a class tree, as in the following example:

```
# classtree.py

"""
Climb inheritance trees using namespace links,
displaying higher superclasses with indentation
"""

def classtree(cls, indent):
    print('.' * indent + cls.__name__)     # Print class name here
    for supercls in cls.__bases__:         # Recur to all superclasses
        classtree(supercls, indent+3)      # May visit super > once

def instancetree(inst):
    print('Tree of %s' % inst)             # Show instance
    classtree(inst.__class__, 3)           # Climb to its class

def selftest():
    class A:      pass
    class B(A):   pass
    class C(A):   pass
    class D(B,C): pass
    class E:      pass
    class F(D,E): pass
```

```
        instancetree(B())
        instancetree(F())

if __name__ == '__main__': selftest()
```

The `classtree` function in this script is *recursive*—it prints a class's name using `__name__`, then climbs up to the superclasses by calling itself. This allows the function to traverse arbitrarily shaped class trees; the recursion climbs to the top, and stops at root superclasses that have empty `__bases__` attributes. When using recursion, each active level of a function gets its own copy of the local scope; here, this means that `cls` and `indent` are different at each `classtree` level.

Most of this file is self-test code. When run standalone in Python 3.0, it builds an empty class tree, makes two instances from it, and prints their class tree structures:

```
C:\misc> c:\python26\python classtree.py
Tree of <__main__.B instance at 0x02557328>
...B
......A
Tree of <__main__.F instance at 0x02557328>
...F
......D
.........B
............A
.........C
............A
......E
```

When run under Python 3.0, the tree includes the implied `object` superclasses that are automatically added above standalone classes, because all classes are "new style" in 3.0 (more on this change in Chapter 31):

```
C:\misc> c:\python30\python classtree.py
Tree of <__main__.B object at 0x02810650>
...B
......A
.........object
Tree of <__main__.F object at 0x02810650>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

Here, indentation marked by periods is used to denote class tree height. Of course, we could improve on this output format, and perhaps even sketch it in a GUI display. Even as is, though, we can import these functions anywhere we want a quick class tree display:

```
C:\misc> c:\python30\python
>>> class Emp: pass
...
>>> class Person(Emp): pass
>>> bob = Person()

>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person object at 0x028203B0>
...Person
......Emp
.........object
```

Regardless of whether you will ever code or use such tools, this example demonstrates one of the many ways that you can make use of special attributes that expose interpreter internals. You'll see another when we code the *lister.py* general-purpose class display tools in the section "Multiple Inheritance: "Mix-in" Classes" on page 756—there, we will extend this technique to also display attributes in each object in a class tree. And in the last part of this book, we'll revisit such tools in the context of Python tool building at large, to code tools that implement attribute privacy, argument validation, and more. While not for every Python programmer, access to internals enables powerful development tools.

Documentation Strings Revisited

The last section's example includes a docstring for its module, but remember that docstrings can be used for class components as well. Docstrings, which we covered in detail in Chapter 15, are string literals that show up at the top of various structures and are automatically saved by Python in the corresponding objects' `__doc__` attributes. This works for module files, function `def`s, and classes and methods.

Now that we know more about classes and methods, the following file, *docstr.py*, provides a quick but comprehensive example that summarizes the places where docstrings can show up in your code. All of these can be triple-quoted blocks:

```
"I am: docstr.__doc__"

def func(args):
    "I am: docstr.func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"
    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass
```

The main advantage of documentation strings is that they stick around at runtime. Thus, if it's been coded as a docstring, you can qualify an object with its `__doc__` attribute to fetch its documentation:

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'

>>> docstr.func.__doc__
'I am: docstr.func.__doc__'

>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__'

>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'
```

A discussion of the *PyDoc* tool, which knows how to format all these strings in reports, appears in Chapter 15. Here it is running on our code under Python 2.6 (Python 3.0 shows additional attributes inherited from the implied `object` superclass in the new-style class model—run this on your own to see the 3.0 extras, and watch for more about this difference in Chapter 31):

```
>>> help(docstr)
Help on module docstr:

NAME
    docstr - I am: docstr.__doc__

FILE
    c:\misc\docstr.py

CLASSES
    spam

    class spam
     |  I am: spam.__doc__ or docstr.spam.__doc__
     |
     |  Methods defined here:
     |
     |  method(self, arg)
     |      I am: spam.method.__doc__ or self.method.__doc__

FUNCTIONS
    func(args)
        I am: docstr.func.__doc__
```

Documentation strings are available at runtime, but they are less flexible syntactically than # comments (which can appear anywhere in a program). Both forms are useful tools, and any program documentation is good (as long as it's accurate, of course!). As a best-practice rule of thumb, use docstrings for functional documentation (what your objects do) and hash-mark comments for more micro-level documentation (how arcane expressions work).

Classes Versus Modules

Let's wrap up this chapter by briefly comparing the topics of this book's last two parts: modules and classes. Because they're both about namespaces, the distinction can be confusing. In short:

- Modules
  - —Are data/logic packages
  - —Are created by writing Python files or C extensions
  - —Are used by being imported
- Classes
  - —Implement new objects
  - —Are created by `class` statements
  - —Are used by being called
  - —Always live within a module

Classes also support extra features that modules don't, such as operator overloading, multiple instance generation, and inheritance. Although both classes and modules are namespaces, you should be able to tell by now that they are very different things.

Chapter Summary

This chapter took us on a second, more in-depth tour of the OOP mechanisms of the Python language. We learned more about classes, methods, and inheritance, and we wrapped up the namespace story in Python by extending it to cover its application to classes. Along the way, we looked at some more advanced concepts, such as abstract superclasses, class data attributes, namespace dictionaries and links, and manual calls to superclass methods and constructors.

Now that we've learned all about the mechanics of coding classes in Python, Chapter 29 turns to a specific facet of those mechanics: operator overloading. After that we'll explore common design patterns, looking at some of the ways that classes are commonly used and combined to optimize code reuse. Before you read ahead, though, be sure to work though the usual chapter quiz to review what we've covered here.

Test Your Knowledge: Quiz

1. What is an abstract superclass?
2. What happens when a simple assignment statement appears at the top level of a `class` statement?

3. Why might a class need to manually call the `__init__` method in a superclass?
4. How can you augment, instead of completely replacing, an inherited method?
5. What...was the capital of Assyria?

Test Your Knowledge: Answers

1. An abstract superclass is a class that calls a method, but does not inherit or define it—it expects the method to be filled in by a subclass. This is often used as a way to generalize classes when behavior cannot be predicted until a more specific subclass is coded. OOP frameworks also use this as a way to dispatch to client-defined, customizable operations.
2. When a simple assignment statement (`X = Y`) appears at the top level of a `class` statement, it attaches a data attribute to the class (*`Class`*`.X`). Like all class attributes, this will be shared by all instances; data attributes are not callable method functions, though.
3. A class must manually call the `__init__` method in a superclass if it defines an `__init__` constructor of its own, but it also must still kick off the superclass's construction code. Python itself automatically runs just one constructor—the lowest one in the tree. Superclass constructors are called through the class name, passing in the `self` instance manually: *`Superclass`*`.__init__(self, ...)`.
4. To augment instead of completely replacing an inherited method, redefine it in a subclass, but call back to the superclass's version of the method manually from the new version of the method in the subclass. That is, pass the `self` instance to the superclass's version of the method manually: *`Superclass.method`*`(self, ...)`.
5. Ashur (or Qalat Sherqat), Calah (or Nimrud), the short-lived Dur Sharrukin (or Khorsabad), and finally Nineveh.

CHAPTER 29

Operator Overloading

This chapter continues our in-depth survey of class mechanics by focusing on operator overloading. We looked briefly at operator overloading in prior chapters; here, we'll fill in more details and look at a handful of commonly used overloading methods. Although we won't demonstrate each of the many operator overloading methods available, those we will code here are a representative sample large enough to uncover the possibilities of this Python class feature.

The Basics

Really "operator overloading" simply means *intercepting* built-in operations in class methods—Python automatically invokes your methods when instances of the class appear in built-in operations, and your method's return value becomes the result of the corresponding operation. Here's a review of the key ideas behind overloading:

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python expression operators.
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc.
- Overloading makes class instances act more like built-in types.
- Overloading is implemented by providing specially named class methods.

In other words, when certain specially named methods are provided in a class, Python automatically calls them when instances of the class appear in their associated expressions. As we've learned, operator overloading methods are never required and generally don't have defaults; if you don't code or inherit one, it just means that your class does not support the corresponding operation. When used, though, these methods allow classes to emulate the interfaces of built-in objects, and so appear more consistent.

Constructors and Expressions: __init__ and __sub__

Consider the following simple example: its `Number` class, coded in the file *number.py*, provides a method to intercept instance construction (`__init__`), as well as one for catching subtraction expressions (`__sub__`). Special methods such as these are the hooks that let you tie into built-in operations:

```
class Number:
    def __init__(self, start):                  # On Number(start)
        self.data = start
    def __sub__(self, other):                   # On instance - other
        return Number(self.data - other)        # Result is a new instance

>>> from number import Number                   # Fetch class from module
>>> X = Number(5)                               # Number.__init__(X, 5)
>>> Y = X - 2                                   # Number.__sub__(X, 2)
>>> Y.data                                      # Y is new Number instance
3
```

As discussed previously, the `__init__` constructor method seen in this code is the most commonly used operator overloading method in Python; it's present in most classes. In this chapter, we will tour some of the other tools available in this domain and look at example code that applies them in common use cases.

Common Operator Overloading Methods

Just about everything you can do to built-in objects such as integers and lists has a corresponding specially named method for overloading in classes. Table 29-1 lists a few of the most common; there are many more. In fact, many overloading methods come in multiple versions (e.g., `__add__`, `__radd__`, and `__iadd__` for addition), which is one reason there are so many. See other Python books, or the Python language reference manual, for an exhaustive list of the special method names available.

Table 29-1. Common operator overloading methods

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of `X`
`__add__`	Operator +	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X(*args, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`

Method	Implements	Called for
`__getitem__`	Indexing, slicing, iteration	`X[key]`, `X[i:j]`, `for` loops and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[key] = value`, `X[i:j] = sequence`
`__delitem__`	Index and slice deletion	`del X[key]`, `del X[i:j]`
`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests (named `__nonzero__` in 2.6)
`__lt__, __gt__, __le__, __ge__, __eq__, __ne__`	Comparisons	`X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y` (or else `__cmp__` in 2.6 only)
`__radd__`	Right-side operators	`Other + X`
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__, __next__`	Iteration contexts	`I=iter(X)`; `next(I)`; `for` loops, `in` if no `__contains__`, all comprehensions, `map(F,X)`, others (`__next__` is named `next` in 2.6)
`__contains__`	Membership test	`item in X` (any iterable)
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (replaces Python 2 `__oct__`, `__hex__`)
`__enter__, __exit__`	Context manager (Chapter 33)	`with obj as var:`
`__get__, __set__, __delete__`	Descriptor attributes (Chapter 37)	`X.attr, X.attr = value, del X.attr`
`__new__`	Creation (Chapter 39)	Object creation, before `__init__`

All overloading methods have names that start and end with two underscores to keep them distinct from other names you define in your classes. The mappings from special method names to expressions or operations are predefined by the Python language (and documented in the standard language manual). For example, the name `__add__` always maps to `+` expressions by Python language definition, regardless of what an `__add__` method's code actually does.

Operator overloading methods may be inherited from superclasses if not defined, just like any other methods. Operator overloading methods are also all optional—if you don't code or inherit one, that operation is simply unsupported by your class, and attempting it will raise an exception. Some built-in operations, like printing, have defaults (inherited for the implied `object` class in Python 3.0), but most built-ins fail for class instances if no corresponding operator overloading method is present.

Most overloading methods are used only in advanced programs that require objects to behave like built-ins; the `__init__` constructor tends to appear in most classes, however, so pay special attention to it. We've already met the `__init__` initialization-time constructor method, and a few of the others in Table 29-1. Let's explore some of the additional methods in the table by example.

Indexing and Slicing: __getitem__ and __setitem__

If defined in a class (or inherited by it), the `__getitem__` method is called automatically for instance-indexing operations. When an instance `X` appears in an indexing expression like `X[i]`, Python calls the `__getitem__` method inherited by the instance, passing `X` to the first argument and the index in brackets to the second argument. For example, the following class returns the square of an index value:

```
>>> class Indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = Indexer()
>>> X[2]                                # X[i] calls X.__getitem__(i)
4

>>> for i in range(5):
...     print(X[i], end=' ')            # Runs __getitem__(X, i) each time
...
0 1 4 9 16
```

Intercepting Slices

Interestingly, in addition to indexing, `__getitem__` is also called for *slice expressions*. Formally speaking, built-in types handle slicing the same way. Here, for example, is slicing at work on a built-in list, using upper and lower bounds and a stride (see Chapter 7 if you need a refresher on slicing):

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                              # Slice with slice syntax
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

Really, though, slicing bounds are bundled up into a *slice object* and passed to the list's implementation of indexing. In fact, you can always pass a slice object manually—slice syntax is mostly syntactic sugar for indexing with a slice object:

```
>>> L[slice(2, 4)]                      # Slice with slice objects
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

This matters in classes with a __getitem__ method—the method will be called both for basic indexing (with an index) and for slicing (with a slice object). Our previous class won't handle slicing because its math assumes integer indexes are passed, but the following class will. When called for indexing, the argument is an integer as before:

```
>>> class Indexer:
...     data = [5, 6, 7, 8, 9]
...     def __getitem__(self, index):    # Called for index or slice
...         print('getitem:', index)
...         return self.data[index]      # Perform index or slice
...
>>> X = Indexer()
>>> X[0]                                 # Indexing sends __getitem__ an integer
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

When called for slicing, though, the method receives a slice object, which is simply passed along to the embedded list indexer in a new index expression:

```
>>> X[2:4]                               # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

If used, the __setitem__ index assignment method similarly intercepts both index and slice assignments—it receives a slice object for the latter, which may be passed along in another index assignment in the same way:

```
def __setitem__(self, index, value):     # Intercept index or slice assignment
    ...
    self.data[index] = value              # Assign index or slice
```

In fact, __getitem__ may be called automatically in even more contexts than indexing and slicing, as the next section explains.

Slicing and Indexing in Python 2.6

Prior to Python 3.0, classes could also define __getslice__ and __setslice__ methods to intercept slice fetches and assignments specifically; they were passed the bounds of the slice expression and were preferred over __getitem__ and __setitem__ for slices.

These slice-specific methods have been removed in 3.0, so you should use `__getitem__` and `__setitem__` instead and allow for both indexes and slice objects as arguments. In most classes, this works without any special code, because indexing methods can manually pass along the slice object in the square brackets of another index expression (as in our example). See the section "Membership: __contains__, __iter__, and __getitem__" on page 716 for another example of slice interception at work.

Also, don't confuse the (arguably unfortunately named) `__index__` method in Python 3.0 for index interception; this method returns an integer value for an instance when needed and is used by built-ins that convert to digit strings:

```
>>> class C:
...     def __index__(self):
...         return 255
...
>>> X = C()
>>> hex(X)                  # Integer value
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Although this method does not intercept instance indexing like `__getitem__`, it is also used in contexts that require an integer—including indexing:

```
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]          # As index (not X[i])
'C'
>>> ('C' * 256)[X:]         # As index (not X[i:])
'C'
```

This method works the same way in Python 2.6, except that it is not called for the `hex` and `oct` built-in functions (use `__hex__` and `__oct__` in 2.6 instead to intercept these calls).

Index Iteration: __getitem__

Here's a trick that isn't always obvious to beginners, but turns out to be surprisingly useful. The `for` statement works by repeatedly indexing a sequence from zero to higher indexes, until an out-of-bounds exception is detected. Because of that, `__getitem__` also turns out to be one way to overload iteration in Python—if this method is defined, `for` loops call the class's `__getitem__` each time through, with successively higher offsets. It's a case of "buy one, get one free"—any built-in or user-defined object that responds to indexing also responds to iteration:

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
```

```
>>> X = stepper()                    # X is a stepper object
>>> X.data = "Spam"
>>>
>>> X[1]                             # Indexing calls __getitem__
'p'
>>> for item in X:                   # for loops call __getitem__
...     print(item, end=' ')         # for indexes items 0..N
...
S p a m
```

In fact, it's really a case of "buy one, get a bunch free." Any class that supports `for` loops automatically supports all iteration contexts in Python, many of which we've seen in earlier chapters (iteration contexts were presented in Chapter 14). For example, the `in` membership test, list comprehensions, the `map` built-in, list and tuple assignments, and type constructors will also call `__getitem__` automatically, if it's defined:

```
>>> 'p' in X                         # All call __getitem__ too
True

>>> [c for c in X]                   # List comprehension
['S', 'p', 'a', 'm']

>>> list(map(str.upper, X))          # map calls (use list() in 3.0)
['S', 'P', 'A', 'M']

>>> (a, b, c, d) = X                 # Sequence assignments
>>> a, c, d
('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X
<__main__.stepper object at 0x00A8D5D0>
```

In practice, this technique can be used to create objects that provide a sequence interface and to add logic to built-in sequence type operations; we'll revisit this idea when extending built-in types in Chapter 31.

Iterator Objects: __iter__ and __next__

Although the `__getitem__` technique of the prior section works, it's really just a fallback for iteration. Today, all iteration contexts in Python will try the `__iter__` method first, before trying `__getitem__`. That is, they prefer the iteration protocol we learned about in Chapter 14 to repeatedly indexing an object; only if the object does not support the iteration protocol is indexing attempted instead. Generally speaking, you should prefer `__iter__` too—it supports general iteration contexts better than `__getitem__` can.

Technically, iteration contexts work by calling the `iter` built-in function to try to find an `__iter__` method, which is expected to return an iterator object. If it's provided, Python then repeatedly calls this iterator object's `__next__` method to produce items

until a `StopIteration` exception is raised. If no such `__iter__` method is found, Python falls back on the `__getitem__` scheme and repeatedly indexes by offsets as before, until an `IndexError` exception is raised. A `next` built-in function is also available as a convenience for manual iterations: `next(I)` is the same as `I.__next__()`.

Version skew note: As described in Chapter 14, if you are using Python 2.6, the `I.__next__()` method just described is named `I.next()` in your Python, and the `next(I)` built-in is present for portability: it calls `I.next()` in 2.6 and `I.__next__()` in 3.0. Iteration works the same in 2.6 in all other respects.

User-Defined Iterators

In the `__iter__` scheme, classes implement user-defined iterators by simply implementing the iteration protocol introduced in Chapters 14 and 20 (refer back to those chapters for more background details on iterators). For example, the following file, *iters.py*, defines a user-defined iterator class that generates squares:

```
class Squares:
    def __init__(self, start, stop):     # Save state when created
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):                  # Get iterator object on iter
        return self
    def __next__(self):                  # Return a square on each iteration
        if self.value == self.stop:      # Also called by next built-in
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):              # for calls iter, which calls __iter__
...     print(i, end=' ')                # Each iteration calls __next__
...
1 4 9 16 25
```

Here, the iterator object is simply the instance `self`, because the `__next__` method is part of this class. In more complex scenarios, the iterator object may be defined as a separate class and object with its own state information to support multiple active iterations over the same data (we'll see an example of this in a moment). The end of the iteration is signaled with a Python `raise` statement (more on raising exceptions in the next part of this book). Manual iterations work as for built-in types as well:

```
>>> X = Squares(1, 5)                   # Iterate manually: what loops do
>>> I = iter(X)                         # iter calls __iter__
>>> next(I)                             # next calls __next__
1
>>> next(I)
4
```

```
...more omitted...
>>> next(I)
25
>>> next(I)                          # Can catch this in try statement
StopIteration
```

An equivalent coding of this iterator with `__getitem__` might be less natural, because the `for` would then iterate through all offsets zero and higher; the offsets passed in would be only indirectly related to the range of values produced (`0..N` would need to map to `start..stop`). Because `__iter__` objects retain explicitly managed state between `next` calls, they can be more general than `__getitem__`.

On the other hand, using iterators based on `__iter__` can sometimes be more complex and less convenient than using `__getitem__`. They are really designed for iteration, not random indexing—in fact, they don't overload the indexing expression at all:

```
>>> X = Squares(1, 5)
>>> X[1]
AttributeError: Squares instance has no attribute '__getitem__'
```

The `__iter__` scheme is also the implementation for all the other iteration contexts we saw in action for `__getitem__` (membership tests, type constructors, sequence assignment, and so on). However, unlike our prior `__getitem__` example, we also need to be aware that a class's `__iter__` may be designed for a single traversal, not many. For example, the `Squares` class is a one-shot iteration; once you've iterated over an instance of that class, it's empty. You need to make a new iterator object for each new iteration:

```
>>> X = Squares(1, 5)
>>> [n for n in X]                   # Exhausts items
[1, 4, 9, 16, 25]
>>> [n for n in X]                   # Now it's empty
[]
>>> [n for n in Squares(1, 5)]       # Make a new iterator object
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))
[1, 4, 9]
```

Notice that this example would probably be simpler if it were coded with generator functions (topics or expressions introduced in Chapter 20 and related to iterators):

```
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):         # or: (x ** 2 for x in range(1, 5))
...     print(i, end=' ')
...
1 4 9 16 25
```

Unlike the class, the function automatically saves its state between iterations. Of course, for this artificial example, you could in fact skip both techniques and simply use a `for` loop, `map`, or a list comprehension to build the list all at once. The best and fastest way to accomplish a task in Python is often also the simplest:

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

However, classes may be better at modeling more complex iterations, especially when they can benefit from state information and inheritance hierarchies. The next section explores one such use case.

Multiple Iterators on One Object

Earlier, I mentioned that the iterator object may be defined as a separate class with its own state information to support multiple active iterations over the same data. Consider what happens when we step across a built-in type like a string:

```
>>> S = 'ace'
>>> for x in S:
...     for y in S:
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

Here, the outer loop grabs an iterator from the string by calling `iter`, and each nested loop does the same to get an independent iterator. Because each active iterator has its own state information, each loop can maintain its own position in the string, regardless of any other active loops.

We saw related examples earlier, in Chapters 14 and 20. For instance, generator functions and expressions, as well as built-ins like `map` and `zip`, proved to be single-iterator objects; by contrast, the `range` built-in and other built-in types, like lists, support multiple active iterators with independent positions.

When we code user-defined iterators with classes, it's up to us to decide whether we will support a single active iteration or many. To achieve the multiple-iterator effect, `__iter__` simply needs to define a new stateful object for the iterator, instead of returning `self`.

The following, for example, defines an iterator class that skips every other item on iterations. Because the iterator object is created anew for each iteration, it supports multiple active loops:

```
class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped                    # Iterator state information
        self.offset  = 0
    def __next__(self):
        if self.offset >= len(self.wrapped):      # Terminate iterations
            raise StopIteration
        else:
            item = self.wrapped[self.offset]      # else return and skip
            self.offset += 2
            return item

class SkipObject:
```

```
    def __init__(self, wrapped):                    # Save item to be used
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped)           # New iterator each time

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)                     # Make container object
    I = iter(skipper)                               # Make an iterator on it
    print(next(I), next(I), next(I))                # Visit offsets 0, 2, 4

    for x in skipper:                # for calls __iter__ automatically
        for y in skipper:            # Nested fors call __iter__ again each time
            print(x + y, end=' ')    # Each iterator has its own state, offset
```

When run, this example works like the nested loops with built-in strings. Each active loop has its own position in the string because each obtains an independent iterator object that records its own state information:

```
% python skipper.py
a c e
aa ac ae ca cc ce ea ec ee
```

By contrast, our earlier `Squares` example supports just one active iteration, unless we call `Squares` again in nested loops to obtain new objects. Here, there is just one `SkipObject`, with multiple iterator objects created from it.

As before, we could achieve similar results with built-in tools—for example, slicing with a third bound to skip items:

```
>>> S = 'abcdef'
>>> for x in S[::2]:
...     for y in S[::2]:            # New objects on each iteration
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

This isn't quite the same, though, for two reasons. First, each slice expression here will physically store the result list all at once in memory; iterators, on the other hand, produce just one value at a time, which can save substantial space for large result lists. Second, slices produce new objects, so we're not really iterating over the same object in multiple places here. To be closer to the class, we would need to make a single object to step across by slicing ahead of time:

```
>>> S = 'abcdef'
>>> S = S[::2]
>>> S
'ace'
>>> for x in S:
...     for y in S:                 # Same object, new iterators
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

This is more similar to our class-based solution, but it still stores the slice result in memory all at once (there is no generator form of built-in slicing today), and it's only equivalent for this particular case of skipping every other item.

Because iterators can do anything a class can do, they are much more general than this example may imply. Regardless of whether our applications require such generality, user-defined iterators are a powerful tool—they allow us to make arbitrary objects look and feel like the other sequences and iterables we have met in this book. We could use this technique with a database object, for example, to support iterations over database fetches, with multiple cursors into the same query result.

Membership: __contains__, __iter__, and __getitem__

The iteration story is even richer than we've seen thus far. Operator overloading is often *layered*: classes may provide specific methods, or more general alternatives used as fallback options. For example:

- Comparisons in Python 2.6 use specific methods such as `__lt__` for less than if present, or else the general `__cmp__`. Python 3.0 uses only specific methods, not `__cmp__`, as discussed later in this chapter.
- Boolean tests similarly try a specific `__bool__` first (to give an explicit `True`/`False` result), and if it's absent fall back on the more general `__len__` (a nonzero length means `True`). As we'll also see later in this chapter, Python 2.6 works the same but uses the name `__nonzero__` instead of `__bool__`.

In the iterations domain, classes normally implement the `in` membership operator as an iteration, using either the `__iter__` method or the `__getitem__` method. To support more specific membership, though, classes may code a `__contains__` method—when present, this method is preferred over `__iter__`, which is preferred over `__getitem__`. The `__contains__` method should define membership as applying to keys for a *mapping* (and can use quick lookups), and as a search for *sequences*.

Consider the following class, which codes all three methods and tests membership and various iteration contexts applied to an instance. Its methods print trace messages when called:

```
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):                # Fallback for iteration
        print('get[%s]:' % i, end='')        # Also for index, slice
        return self.data[i]
    def __iter__(self):                      # Preferred for iteration
        print('iter=> ', end='')             # Allows only 1 active iterator
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
```

```
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):                # Preferred for 'in'
        print('contains: ', end='')
        return x in self.data


X = Iters([1, 2, 3, 4, 5])                    # Make instance
print(3 in X)                                 # Membership
for i in X:                                   # For loops
    print(i, end=' | ')

print()
print([i ** 2 for i in X])                    # Other iteration contexts
print( list(map(bin, X)) )

I = iter(X)                                   # Manual iteration (what other contexts do)
while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

When run as it is, this script's output is as follows—the specific `__contains__` intercepts membership, the general `__iter__` catches other iteration contexts such that `__next__` is called repeatedly, and `__getitem__` is never called:

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Watch what happens to this code's output if we comment out its `__contains__` method, though—membership is now routed to the general `__iter__` instead:

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

And finally, here is the output if both `__contains__` and `__iter__` are commented out—the indexing `__getitem__` fallback is called with successively higher indexes for membership and other iteration contexts:

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100','0b101']
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

As we've seen, the `__getitem__` method is even more general: besides iterations, it also intercepts explicit indexing as well as slicing. Slice expressions trigger `__getitem__` with a slice object containing bounds, both for built-in types and user-defined classes, so slicing is automatic in our class:

```
>>> X = Iters('spam')                  # Indexing
>>> X[0]                               # __getitem__(0)
get[0]:'s'

>>> 'spam'[1:]                         # Slice syntax
'pam'
>>> 'spam'[slice(1, None)]             # Slice object
'pam'

>>> X[1:]                              # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[:-1]
get[slice(None, -1, None)]:'spa'
```

In more realistic iteration use cases that are not sequence-oriented, though, the `__iter__` method may be easier to write since it must not manage an integer index, and `__contains__` allows for membership optimization as a special case.

Attribute Reference: `__getattr__` and `__setattr__`

The `__getattr__` method intercepts attribute qualifications. More specifically, it's called with the attribute name as a string whenever you try to qualify an instance with an *undefined* (nonexistent) attribute name. It is not called if Python can find the attribute using its inheritance tree search procedure. Because of its behavior, `__getattr__` is useful as a hook for responding to attribute requests in a generic fashion. For example:

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
40
>>> X.name
...error text omitted...
AttributeError: name
```

Here, the `empty` class and its instance `X` have no real attributes of their own, so the access to `X.age` gets routed to the `__getattr__` method; `self` is assigned the instance (`X`), and `attrname` is assigned the undefined attribute name string (`"age"`). The class makes `age` look like a real attribute by returning a real value as the result of the `X.age` qualification expression (`40`). In effect, `age` becomes a *dynamically computed* attribute.

For attributes that the class doesn't know how to handle, `__getattr__` raises the built-in `AttributeError` exception to tell Python that these are bona fide undefined names; asking for `X.name` triggers the error. You'll see `__getattr__` again when we see delegation and properties at work in the next two chapters, and I'll say more about exceptions in Part VII.

A related overloading method, `__setattr__`, intercepts *all* attribute assignments. If this method is defined, `self.attr = value` becomes `self.__setattr__('attr', value)`. This is a bit trickier to use because assigning to any `self` attributes within `__setattr__` calls `__setattr__` again, causing an infinite recursion loop (and eventually, a stack overflow exception!). If you want to use this method, be sure that it assigns any instance attributes by indexing the attribute dictionary, discussed in the next section. That is, use `self.__dict__['name'] = x`, not `self.name = x`:

```
>>> class accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value
...         else:
...             raise AttributeError, attr + ' not allowed'
...
>>> X = accesscontrol()
>>> X.age = 40                                  # Calls __setattr__
>>> X.age
40
>>> X.name = 'mel'
...text omitted...
AttributeError: name not allowed
```

These two attribute-access overloading methods allow you to control or specialize access to attributes in your objects. They tend to play highly specialized roles, some of which we'll explore later in this book.

Other Attribute Management Tools

For future reference, also note that there are other ways to manage attribute access in Python:

- The `__getattribute__` method intercepts all attribute fetches, not just those that are undefined, but when using it you must be more cautious than with `__getattr__` to avoid loops.
- The `property` built-in function allows us to associate methods with fetch and set operations on a specific class attribute.
- *Descriptors* provide a protocol for associating `__get__` and `__set__` methods of a class with accesses to a specific class attribute.

Because these are somewhat advanced tools not of interest to every Python programmer, we'll defer a look at properties until Chapter 31 and detailed coverage of all the attribute management techniques until Chapter 37.

Emulating Privacy for Instance Attributes: Part 1

The following code generalizes the previous example, to allow each subclass to have its own list of private names that cannot be assigned to its instances:

```
class PrivateExc(Exception): pass                    # More on exceptions later

class Privacy:
    def __setattr__(self, attrname, value):         # On self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value          # self.attrname = value loops!

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'

x = Test1()
y = Test2()

x.name = 'Bob'
y.name = 'Sue'                                       # Fails

y.age  = 30
x.age  = 40                                          # Fails
```

In fact, this is a first-cut solution for an implementation of *attribute privacy* in Python (i.e., disallowing changes to attribute names outside a class). Although Python doesn't support private declarations per se, techniques like this can emulate much of their purpose. This is a partial solution, though; to make it more effective, it must be augmented to allow subclasses to set private attributes more naturally, too, and to use `__getattr__` and a wrapper (sometimes called a proxy) class to check for private attribute fetches.

We'll postpone a more complete solution to attribute privacy until Chapter 38, where we'll use *class decorators* to intercept and validate attributes more generally. Even though privacy can be emulated this way, though, it almost never is in practice. Python programmers are able to write large OOP frameworks and applications without private declarations—an interesting finding about access controls in general that is beyond the scope of our purposes here.

Catching attribute references and assignments is generally a useful technique; it supports *delegation*, a design technique that allows controller objects to wrap up embedded objects, add new behaviors, and route other operations back to the wrapped objects (more on delegation and wrapper classes in Chapter 30).

String Representation: __repr__ and __str__

The next example exercises the `__init__` constructor and the `__add__` overload method, both of which we've already seen, as well as defining a `__repr__` method that returns a string representation for instances. String formatting is used to convert the managed `self.data` object to a string. If defined, `__repr__` (or its sibling, `__str__`) is called automatically when class instances are printed or converted to strings. These methods allow you to define a better display format for your objects than the default instance display.

The default display of instance objects is neither useful nor pretty:

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                   # Initialize data
...     def __add__(self, other):
...         self.data += other                  # Add other in-place (bad!)
...
>>> x = adder()                                 # Default displays
>>> print(x)
<__main__.adder object at 0x025D66B0>
>>> x
<__main__.adder object at 0x025D66B0>
```

But coding or inheriting string representation methods allows us to customize the display:

```
>>> class addrepr(adder):                       # Inherit __init__, __add__
...     def __repr__(self):                     # Add string representation
...         return 'addrepr(%s)' % self.data    # Convert to as-code string
...
>>> x = addrepr(2)                              # Runs __init__
>>> x + 1                                       # Runs __add__
>>> x                                           # Runs __repr__
addrepr(3)
>>> print(x)                                    # Runs __repr__
addrepr(3)
>>> str(x), repr(x)                             # Runs __repr__ for both
('addrepr(3)', 'addrepr(3)')
```

So why two display methods? Mostly, to support different audiences. In full detail:

- `__str__` is tried first for the `print` operation and the `str` built-in function (the internal equivalent of which `print` runs). It generally should return a user-friendly display.
- `__repr__` is used in all other contexts: for interactive echoes, the `repr` function, and nested appearances, as well as by `print` and `str` if no `__str__` is present. It should generally return an as-code string that could be used to re-create the object, or a detailed display for developers.

In a nutshell, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. Note, however, that while printing falls back on `__repr__` if no `__str__` is

defined, the inverse is not true—other contexts, such as interactive echoes, use `__repr__` only and don't try `__str__` at all:

```
>>> class addstr(adder):
...     def __str__(self):                      # __str__ but no __repr__
...         return '[Value: %s]' % self.data    # Convert to nice string
...
>>> x = addstr(3)
>>> x + 1
>>> x                                           # Default __repr__
<__main__.addstr object at 0x00B35EF0>
>>> print(x)                                    # Runs __str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object at 0x00B35EF0>')
```

Because of this, `__repr__` may be best if you want a *single* display for all contexts. By defining both methods, though, you can support different displays in different contexts—for example, an end-user display with `__str__`, and a low-level display for programmers to use during development with `__repr__`. In effect, `__str__` simply overrides `__repr__` for user-friendly display contexts:

```
>>> class addboth(adder):
...     def __str__(self):
...         return '[Value: %s]' % self.data    # User-friendly string
...     def __repr__(self):
...         return 'addboth(%s)' % self.data    # As-code string
...
>>> x = addboth(4)
>>> x + 1
>>> x                                           # Runs __repr__
addboth(5)
>>> print(x)                                    # Runs __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

I should mention two usage notes here. First, keep in mind that `__str__` and `__repr__` must both return strings; other result types are not converted and raise errors, so be sure to run them through a converter if needed. Second, depending on a container's string-conversion logic, the user-friendly display of `__str__` might only apply when objects appear at the top level of a print operation; objects nested in larger objects might still print with their `__repr__` or its default. The following illustrates both of these points:

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __str__(self):                  # Used for instance itself
...         return str(self.val)            # Convert to a string result
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # __str__ run when instance printed
...                                         # But not when instance in a list!
```

```
2
3
>>> print(objs)
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
>>> objs
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
```

To ensure that a custom display is run in all contexts regardless of the container, code `__repr__`, not `__str__`; the former is run in all cases if the latter doesn't apply:

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):                 # __repr__ used by print if no __str__
...         return str(self.val)            # __repr__ used if echoed or nested
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)                 # No __str__: runs __repr__
...
2
3
>>> print(objs)                             # Runs __repr__, not ___str__
[2, 3]
>>> objs
[2, 3]
```

In practice, `__str__` (or its low-level relative, `__repr__`) seems to be the second most commonly used operator overloading method in Python scripts, behind `__init__`. Any time you can print an object and see a custom display, one of these two tools is probably in use.

Right-Side and In-Place Addition: __radd__ and __iadd__

Technically, the `__add__` method that appeared in the prior example does not support the use of instance objects on the right side of the `+` operator. To implement such expressions, and hence support *commutative*-style operators, code the `__radd__` method as well. Python calls `__radd__` only when the object on the right side of the `+` is your class instance, but the object on the left is not an instance of your class. The `__add__` method for the object on the left is called instead in all other cases:

```
>>> class Commuter:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         print('add', self.val, other)
...         return self.val + other
...     def __radd__(self, other):
...         print('radd', self.val, other)
...         return other + self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
```

```
>>> x + 1                       # __add__: instance + noninstance
add 88 1
89
>>> 1 + y                       # __radd__: noninstance + instance
radd 99 1
100
>>> x + y                       # __add__: instance + instance, triggers __radd__
add 88 <__main__.Commuter object at 0x02630910>
radd 99 88
187
```

Notice how the order is reversed in `__radd__`: `self` is really on the right of the `+`, and `other` is on the left. Also note that `x` and `y` are instances of the same class here; when instances of different classes appear mixed in an expression, Python prefers the class of the one on the left. When we add the two instances together, Python runs `__add__`, which in turn triggers `__radd__` by simplifying the left operand.

In more realistic classes where the class type may need to be propagated in results, things can become trickier: type testing may be required to tell whether it's safe to convert and thus avoid nesting. For instance, without the `isinstance` test in the following, we could wind up with a `Commuter` whose `val` is another `Commuter` when two instances are added and `__add__` triggers `__radd__`:

```
>>> class Commuter:                 # Propagate class type in results
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         if isinstance(other, Commuter): other = other.val
...         return Commuter(self.val + other)
...     def __radd__(self, other):
...         return Commuter(other + self.val)
...     def __str__(self):
...         return '<Commuter: %s>' % self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)                   # Result is another Commuter instance
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>

>>> z = x + y                       # Not nested: doesn't recur to __radd__
>>> print(z)
<Commuter: 187>
>>> print(z + 10)
<Commuter: 197>
>>> print(z + z)
<Commuter: 374>
```

In-Place Addition

To also implement `+=` in-place augmented addition, code either an `__iadd__` or an `__add__`. The latter is used if the former is absent. In fact, the prior section's `Commuter` class supports `+=` already for this reason, but `__iadd__` allows for more efficient in-place changes:

```
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __iadd__(self, other):              # __iadd__ explicit: x += y
...         self.val += other                   # Usually returns self
...         return self
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):               # __add__ fallback: x = (x + y)
...         return Number(self.val + other)     # Propagates class type
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

Every binary operator has similar right-side and in-place overloading methods that work the same (e.g., `__mul__`, `__rmul__`, and `__imul__`). Right-side methods are an advanced topic and tend to be fairly rarely used in practice; you only code them when you need operators to be commutative, and then only if you need to support such operators at all. For instance, a `Vector` class may use these tools, but an `Employee` or `Button` class probably would not.

Call Expressions: __call__

The `__call__` method is called when your instance is called. No, this isn't a circular definition—if defined, Python runs a `__call__` method for function call expressions applied to your instances, passing along whatever positional or keyword arguments were sent:

```
>>> class Callee:
...     def __call__(self, *pargs, **kargs):        # Intercept instance calls
...         print('Called:', pargs, kargs)          # Accept arbitrary arguments
...
>>> C = Callee()
>>> C(1, 2, 3)                                      # C is a callable object
```

```
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

More formally, all the argument-passing modes we explored in Chapter 18 are supported by the `__call__` method—whatever is passed to the instance is passed to this method, along with the usual implied instance argument. For example, the method definitions:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...          # Normals and defaults

class C:
    def __call__(self, *pargs, **kargs): ...         # Collect arbitrary arguments

class C:
    def __call__(self, *pargs, d=6, **kargs): ...    # 3.0 keyword-only argument
```

all match all the following instance calls:

```
X = C()
X(1, 2)                                  # Omit defaults
X(1, 2, 3, 4)                            # Positionals
X(a=1, b=2, d=4)                         # Keywords
X(*[1, 2], **dict(c=3, d=4))             # Unpack arbitrary arguments
X(1, *(2,), c=3, **dict(d=4))            # Mixed modes
```

The net effect is that classes and instances with a `__call__` support the exact same argument syntax and semantics as normal functions and methods.

Intercepting call expression like this allows class instances to emulate the look and feel of things like functions, but also retain state information for use during calls (we saw a similar example while exploring scopes in Chapter 17, but you should be more familiar with operator overloading here):

```
>>> class Prod:
...     def __init__(self, value):                # Accept just one argument
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)                                   # "Remembers" 2 in state
>>> x(3)                                          # 3 (passed) * 2 (state)
6
>>> x(4)
8
```

In this example, the `__call__` may seem a bit gratuitous at first glance. A simple method can provide similar utility:

```
>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def comp(self, other):
...         return self.value * other
...
```

```
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

However, `__call__` can become more useful when interfacing with APIs that expect functions—it allows us to code objects that conform to an expected function call interface, but also retain state information. In fact, it's probably the third most commonly used operator overloading method, behind the `__init__` constructor and the `__str__` and `__repr__` display-format alternatives.

Function Interfaces and Callback-Based Code

As an example, the `tkinter` GUI toolkit (named `Tkinter` in Python 2.6) allows you to register functions as event handlers (a.k.a. callbacks); when events occur, `tkinter` calls the registered objects. If you want an event handler to retain state between events, you can register either a class's bound method or an instance that conforms to the expected interface with `__call__`. In this section's code, both `x.comp` from the second example and `x` from the first can pass as function-like objects this way.

I'll have more to say about `bound` methods in the next chapter, but for now, here's a hypothetical example of `__call__` applied to the GUI domain. The following class defines an object that supports a function-call interface, but also has state information that remembers the color a button should change to when it is later pressed:

```
class Callback:
    def __init__(self, color):                # Function + state information
        self.color = color
    def __call__(self):                       # Support calls with no arguments
        print('turn', self.color)
```

Now, in the context of a GUI, we can register instances of this class as event handlers for buttons, even though the GUI expects to be able to invoke event handlers as simple functions with no arguments:

```
cb1 = Callback('blue')                        # Remember blue
cb2 = Callback('green')

B1 = Button(command=cb1)                      # Register handlers
B2 = Button(command=cb2)                      # Register handlers
```

When the button is later pressed, the instance object is called as a simple function, exactly like in the following calls. Because it retains state as instance attributes, though, it remembers what to do:

```
cb1()                                         # On events: prints 'blue'
cb2()                                         # Prints 'green'
```

In fact, this is probably the best way to retain state information in the Python language—better than the techniques discussed earlier for functions (global variables,

enclosing function scope references, and default mutable arguments). With OOP, the state remembered is made explicit with attribute assignments.

Before we move on, there are two other ways that Python programmers sometimes tie information to a callback function like this. One option is to use default arguments in `lambda` functions:

```
cb3 = (lambda color='red': 'turn ' + color)  # Or: defaults
print(cb3())
```

The other is to use *bound methods* of a class. A bound method object is a kind of object that remembers the `self` instance and the referenced function. A bound method may therefore be called as a simple function without an instance later:

```
class Callback:
    def __init__(self, color):                # Class with state information
        self.color = color
    def changeColor(self):                    # A normal named method
        print('turn', self.color)

cb1 = Callback('blue')
cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor)          # Reference, but don't call
B2 = Button(command=cb2.changeColor)          # Remembers function+self
```

In this case, when this button is later pressed it's as if the GUI does this, which invokes the `changeColor` method to process the object's state information:

```
object = Callback('blue')
cb = object.changeColor                       # Registered event handler
cb()                                          # On event prints 'blue'
```

This technique is simpler, but less general than overloading calls with `__call__`; again, watch for more about bound methods in the next chapter.

You'll also see another `__call__` example in Chapter 31, where we will use it to implement something known as a *function decorator*—a callable object often used to add a layer of logic on top of an embedded function. Because `__call__` allows us to attach state information to a callable object, it's a natural implementation technique for a function that must remember and call another function.

Comparisons: `__lt__`, `__gt__`, and Others

As suggested in Table 29-1, classes can define methods to catch all six comparison operators: `<`, `>`, `<=`, `>=`, `==`, and `!=`. These methods are generally straightforward to use, but keep the following qualifications in mind:

- Unlike the `__add__`/`__radd__` pairings discussed earlier, there are no right-side variants of comparison methods. Instead, reflective methods are used when only one operand supports comparison (e.g., `__lt__` and `__gt__` are each other's reflection).
- There are no implicit relationships among the comparison operators. The truth of `==` does not imply that `!=` is false, for example, so both `__eq__` and `__ne__` should be defined to ensure that both operators behave correctly.
- In Python 2.6, a `__cmp__` method is used by all comparisons if no more specific comparison methods are defined; it returns a number that is less than, equal to, or greater than zero, to signal less than, equal, and greater than results for the comparison of its two arguments (`self` and another operand). This method often uses the `cmp(x, y)` built-in to compute its result. Both the `__cmp__` method and the `cmp` built-in function are removed in Python 3.0: use the more specific methods instead.

We don't have space for an in-depth exploration of comparison methods, but as a quick introduction, consider the following class and test code:

```
class C:
    data = 'spam'
    def __gt__(self, other):                # 3.0 and 2.6 version
        return self.data > other
    def __lt__(self, other):
        return self.data < other

X = C()
print(X > 'ham')                            # True  (runs __gt__)
print(X < 'ham')                            # False (runs __lt__)
```

When run under Python 3.0 or 2.6, the prints at the end display the expected results noted in their comments, because the class's methods intercept and implement comparison expressions.

The 2.6 `__cmp__` Method (Removed in 3.0)

In Python 2.6, the `__cmp__` method is used as a fallback if more specific methods are not defined: its integer result is used to evaluate the operator being run. The following produces the same result under 2.6, for example, but fails in 3.0 because `__cmp__` is no longer used:

```
class C:
    data = 'spam'                           # 2.6 only
    def __cmp__(self, other):               # __cmp__ not used in 3.0
        return cmp(self.data, other)        # cmp not defined in 3.0

X = C()
print(X > 'ham')                            # True  (runs __cmp__)
print(X < 'ham')                            # False (runs __cmp__)
```

Notice that this fails in 3.0 because `__cmp__` is no longer special, not because the `cmp` built-in function is no longer present. If we change the prior class to the following to try to simulate the `cmp` call, the code still works in 2.6 but fails in 3.0:

```
class C:
    data = 'spam'
    def __cmp__(self, other):
        return (self.data > other) - (self.data < other)
```

So why, you might be asking, did I just show you a comparison method that is no longer supported in 3.0? While it would be easier to erase history entirely, this book is designed to support both 2.6 and 3.0 readers. Because `__cmp__` may appear in code 2.6 readers must reuse or maintain, it's fair game in this book. Moreover, `__cmp__` was removed more abruptly than the `__getslice__` method described earlier, and so may endure longer. If you use 3.0, though, or care about running your code under 3.0 in the future, don't use `__cmp__` anymore: use the more specific comparison methods instead.

Boolean Tests: __bool__ and __len__

As mentioned earlier, classes may also define methods that give the Boolean nature of their instances—in Boolean contexts, Python first tries `__bool__` to obtain a direct Boolean value and then, if that's missing, tries `__len__` to determine a truth value from the object length. The first of these generally uses object state or other information to produce a Boolean result:

```
>>> class Truth:
...     def __bool__(self): return True
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!

>>> class Truth:
...     def __bool__(self): return False
...
>>> X = Truth()
>>> bool(X)
False
```

If this method is missing, Python falls back on length because a nonempty object is considered true (i.e., a nonzero length is taken to mean the object is true, and a zero length means it is false):

```
>>> class Truth:
...     def __len__(self): return 0
...
>>> X = Truth()
>>> if not X: print('no!')
```

```
...
no!
```

If both methods are present Python prefers `__bool__` over `__len__`, because it is more specific:

```
>>> class Truth:
...     def __bool__(self): return True          # 3.0 tries __bool__ first
...     def __len__(self): return 0              # 2.6 tries __len__ first
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!
```

If neither truth method is defined, the object is vacuously considered true (which has potential implications for metaphysically inclined readers!):

```
>>> class Truth:
...     pass
...
>>> X = Truth()
>>> bool(X)
True
```

And now that we've managed to cross over into the realm of philosophy, let's move on to look at one last overloading context: object demise.

Booleans in Python 2.6

Python 2.6 users should use `__nonzero__` instead of `__bool__` in all of the code in the section "Boolean Tests: __bool__ and __len__" on page 730. Python 3.0 renamed the 2.6 `__nonzero__` method to `__bool__`, but Boolean tests work the same otherwise (both 3.0 and 2.6 use `__len__` as a fallback).

If you don't use the 2.6 name, the very first test in this section will work the same for you anyhow, but only because `__bool__` is not recognized as a special method name in 2.6, and objects are considered true by default!

To witness this version difference live, you need to return `False`:

```
C:\misc> c:\python30\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
in bool
False
>>> if X: print(99)
...
in bool
```

This works as advertised in 3.0. In 2.6, though, `__bool__` is ignored and the object is always considered true:

```
C:\misc> c:\python26\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
True
>>> if X: print(99)
...
99
```

In 2.6, use `__nonzero__` for Boolean values (or return `0` from the `__len__` fallback method to designate false):

```
C:\misc> c:\python26\python
>>> class C:
...     def __nonzero__(self):
...         print('in nonzero')
...         return False
...
>>> X = C()
>>> bool(X)
in nonzero
False
>>> if X: print(99)
...
in nonzero
```

But keep in mind that `__nonzero__` works in 2.6 only; if used in 3.0 it will be silently ignored and the object will be classified as true by default—just like using `__bool__` in 2.6!

Object Destruction: `__del__`

We've seen how the `__init__` constructor is called whenever an instance is generated. Its counterpart, the destructor method `__del__`, is run automatically when an instance's space is being reclaimed (i.e., at "garbage collection" time):

```
>>> class Life:
...     def __init__(self, name='unknown'):
...         print('Hello', name)
...         self.name = name
...     def __del__(self):
...         print('Goodbye', self.name)
...
>>> brian = Life('Brian')
Hello Brian
>>> brian = 'loretta'
Goodbye Brian
```

Here, when `brian` is assigned a string, we lose the last reference to the `Life` instance and so trigger its destructor method. This works, and it may be useful for implementing some cleanup activities (such as terminating server connections). However, destructors are not as commonly used in Python as in some OOP languages, for a number of reasons.

For one thing, because Python automatically reclaims all space held by an instance when the instance is reclaimed, destructors are not necessary for space management.* For another, because you cannot always easily predict when an instance will be reclaimed, it's often better to code termination activities in an explicitly called method (or `try`/`finally` statement, described in the next part of the book); in some cases, there may be lingering references to your objects in system tables that prevent destructors from running.

In fact, `__del__` can be tricky to use for even more subtle reasons. Exceptions raised within it, for example, simply print a warning message to `sys.stderr` (the standard error stream) rather than triggering an exception event, because of the unpredictable context under which it is run by the garbage collector. In addition, cyclic (a.k.a. circular) references among objects may prevent garbage collection from happening when you expect it to; an optional cycle detector, enabled by default, can automatically collect such objects eventually, but only if they do not have `__del__` methods. Since this is relatively obscure, we'll ignore further details here; see Python's standard manuals' coverage of both `__del__` and the `gc` garbage collector module for more information.

Chapter Summary

That's as many overloading examples as we have space for here. Most of the other operator overloading methods work similarly to the ones we've explored, and all are just hooks for intercepting built-in type operations; some overloading methods, for example, have unique argument lists or return values. We'll see a few others in action later in the book:

- Chapter 33 uses the `__enter__` and `__exit__` `with` statement context manager methods.
- Chapter 37 uses the `__get__` and `__set__` class descriptor fetch/set methods.
- Chapter 39 uses the `__new__` object creation method in the context of metaclasses.

* In the current C implementation of Python, you also don't need to close file objects held by the instance in destructors because they are automatically closed when reclaimed. However, as mentioned in Chapter 9, it's better to explicitly call file close methods because auto-close-on-reclaim is a feature of the implementation, not of the language itself (this behavior can vary under Jython, for instance).

In addition, some of the methods we've studied here, such as `__call__` and `__str__`, will be employed by later examples in this book. For complete coverage, though, I'll defer to other documentation sources—see Python's standard language manual or reference books for details on additional overloading methods.

In the next chapter, we leave the realm of class mechanics behind to explore common design patterns—the ways that classes are commonly used and combined to optimize code reuse. Before you read on, though, take a moment to work though the chapter quiz below to review the concepts we've covered.

Test Your Knowledge: Quiz

1. What two operator overloading methods can you use to support iteration in your classes?
2. What two operator overloading methods handle printing, and in what contexts?
3. How can you intercept slice operations in a class?
4. How can you catch in-place addition in a class?
5. When should you provide operator overloading?

Test Your Knowledge: Answers

1. Classes can support iteration by defining (or inheriting) `__getitem__` or `__iter__`. In all iteration contexts, Python tries to use `__iter__` (which returns an object that supports the iteration protocol with a `__next__` method) first: if no `__iter__` is found by inheritance search, Python falls back on the `__getitem__` indexing method (which is called repeatedly, with successively higher indexes).
2. The `__str__` and `__repr__` methods implement object print displays. The former is called by the `print` and `str` built-in functions; the latter is called by `print` and `str` if there is no `__str__`, and always by the `repr` built-in, interactive echoes, and nested appearances. That is, `__repr__` is used everywhere, except by `print` and `str` when a `__str__` is defined. A `__str__` is usually used for user-friendly displays; `__repr__` gives extra details or the object's as-code form.
3. Slicing is caught by the `__getitem__` indexing method: it is called with a slice object, instead of a simple index. In Python 2.6, `__getslice__` (defunct in 3.0) may be used as well.
4. In-place addition tries `__iadd__` first, and `__add__` with an assignment second. The same pattern holds true for all binary operators. The `__radd__` method is also available for right-side addition.

5. When a class naturally matches, or needs to emulate, a built-in type's interfaces. For example, collections might imitate sequence or mapping interfaces. You generally shouldn't implement expression operators if they don't naturally map to your objects, though—use normally named methods instead.

CHAPTER 30
Designing with Classes

So far in this part of the book, we've concentrated on using Python's OOP tool, the class. But OOP is also about *design issues*—i.e., how to use classes to model useful objects. This chapter will touch on a few core OOP ideas and present some additional examples that are more realistic than those shown so far.

Along the way, we'll code some common OOP design patterns in Python, such as inheritance, composition, delegation, and factories. We'll also investigate some design-focused class concepts, such as pseudoprivate attributes, multiple inheritance, and bound methods. Many of the design terms mentioned here require more explanation than I can provide in this book; if this material sparks your curiosity, I suggest exploring a text on OOP design or design patterns as a next step.

Python and OOP

Let's begin with a review—Python's implementation of OOP can be summarized by three ideas:

Inheritance
: Inheritance is based on attribute lookup in Python (in `X.name` expressions).

Polymorphism
: In `X.method`, the meaning of `method` depends on the type (class) of `X`.

Encapsulation
: Methods and operators implement behavior; data hiding is a convention by default.

By now, you should have a good feel for what inheritance is all about in Python. We've also talked about Python's polymorphism a few times already; it flows from Python's lack of type declarations. Because attributes are always resolved at runtime, objects that implement the same interfaces are interchangeable; clients don't need to know what sorts of objects are implementing the methods they call.

Encapsulation means packaging in Python—that is, hiding implementation details behind an object's interface. It does not mean enforced privacy, though that can be implemented with code, as we'll see in Chapter 38. Encapsulation allows the implementation of an object's interface to be changed without impacting the users of that object.

Overloading by Call Signatures (or Not)

Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their arguments. But because there are no type declarations in Python, this concept doesn't really apply; polymorphism in Python is based on object *interfaces*, not types.

You can try to overload methods by their argument lists, like this:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

This code will run, but because the `def` simply assigns an object to a name in the class's scope, the last definition of the method function is the only one that will be retained (it's just as if you say `X = 1` and then `X = 2`; `X` will be `2`).

Type-based selections can always be coded using the type-testing ideas we met in Chapters 4 and 9, or the argument list tools introduced in Chapter 18:

```
class C:
    def meth(self, *args):
        if len(args) == 1:
            ...
        elif type(arg[0]) == int:
            ...
```

You normally shouldn't do this, though—as described in Chapter 16, you should write your code to expect an object interface, not a specific data type. That way, it will be useful for a broader category of types and applications, both now and in the future:

```
class C:
    def meth(self, x):
        x.operation()                    # Assume x does the right thing
```

It's also generally considered better to use distinct method names for distinct operations, rather than relying on call signatures (no matter what language you code in).

Although Python's object model is straightforward, much of the art in OOP is in the way we combine classes to achieve a program's goals. The next section begins a tour of some of the ways larger programs use classes to their advantage.

OOP and Inheritance: "Is-a" Relationships

We've explored the mechanics of inheritance in depth already, but I'd like to show you an example of how it can be used to model real-world relationships. From a programmer's point of view, inheritance is kicked off by attribute qualifications, which trigger searches for names in instances, their classes, and then any superclasses. From a designer's point of view, inheritance is a way to specify set membership: a class defines a set of properties that may be inherited and customized by more specific sets (i.e., subclasses).

To illustrate, let's put that pizza-making robot we talked about at the start of this part of the book to work. Suppose we've decided to explore alternative career paths and open a pizza restaurant. One of the first things we'll need to do is hire employees to serve customers, prepare the food, and so on. Being engineers at heart, we've decided to build a robot to make the pizzas; but being politically and cybernetically correct, we've also decided to make our robot a full-fledged employee with a salary.

Our pizza shop team can be defined by the four classes in the example file, *employees.py*. The most general class, `Employee`, provides common behavior such as bumping up salaries (`giveRaise`) and printing (`__repr__`). There are two kinds of employees, and so two subclasses of `Employee`: `Chef` and `Server`. Both override the inherited `work` method to print more specific messages. Finally, our pizza robot is modeled by an even more specific class: `PizzaRobot` is a kind of `Chef`, which is a kind of `Employee`. In OOP terms, we call these relationships "is-a" links: a robot is a chef, which is a(n) employee. Here's the *employees.py* file:

```
class Employee:
    def __init__(self, name, salary=0):
        self.name   = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "does stuff")
    def __repr__(self):
        return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food")

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer")

class PizzaRobot(Chef):
```

```
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print(self.name, "makes pizza")

if __name__ == "__main__":
    bob = PizzaRobot('bob')         # Make a robot named bob
    print(bob)                      # Run inherited __repr__
    bob.work()                      # Run type-specific action
    bob.giveRaise(0.20)             # Give bob a 20% raise
    print(bob); print()

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

When we run the self-test code included in this module, we create a pizza-making robot named `bob`, which inherits names from three classes: `PizzaRobot`, `Chef`, and `Employee`. For instance, printing `bob` runs the `Employee.__repr__` method, and giving `bob` a raise invokes `Employee.giveRaise` because that's where the inheritance search finds that method:

```
C:\python\examples> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

In a class hierarchy like this, you can usually make instances of any of the classes, not just the ones at the bottom. For instance, the `for` loop in this module's self-test code creates instances of all four classes; each responds differently when asked to work because the `work` method is different in each. Really, these classes just simulate real-world objects; `work` prints a message for the time being, but it could be expanded to do real work later.

OOP and Composition: "Has-a" Relationships

The notion of composition was introduced in Chapter 25. From a programmer's point of view, composition involves embedding other objects in a container object, and activating them to implement container methods. To a designer, composition is another way to represent relationships in a problem domain. But, rather than set membership, composition has to do with components—parts of a whole.

Composition also reflects the relationships between parts, called a "has-a" relationships. Some OOP design texts refer to composition as *aggregation* (or distinguish between the two terms by using aggregation to describe a weaker dependency between

container and contained); in this text, a "composition" simply refers to a collection of embedded objects. The composite class generally provides an interface all its own and implements it by directing the embedded objects.

Now that we've implemented our employees, let's put them in the pizza shop and let them get busy. Our pizza shop is a composite object: it has an oven, and it has employees like servers and chefs. When a customer enters and places an order, the components of the shop spring into action—the server takes the order, the chef makes the pizza, and so on. The following example (the file *pizzashop.py*) simulates all the objects and relationships in this scenario:

```
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)
    def pay(self, server):
        print(self.name, "pays for item to", server)

class Oven:
    def bake(self):
        print("oven bakes")

class PizzaShop:
    def __init__(self):
        self.server = Server('Pat')          # Embed other objects
        self.chef   = PizzaRobot('Bob')      # A robot named bob
        self.oven   = Oven()

    def order(self, name):
        customer = Customer(name)            # Activate other objects
        customer.order(self.server)          # Customer orders from server
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == "__main__":
    scene = PizzaShop()                      # Make the composite
    scene.order('Homer')                     # Simulate Homer's order
    print('...')
    scene.order('Shaggy')                    # Simulate Shaggy's order
```

The `PizzaShop` class is a container and controller; its constructor makes and embeds instances of the employee classes we wrote in the last section, as well as an `Oven` class defined here. When this module's self-test code calls the `PizzaShop order` method, the embedded objects are asked to carry out their actions in turn. Notice that we make a new `Customer` object for each order, and we pass on the embedded `Server` object to `Customer` methods; customers come and go, but the server is part of the pizza shop composite. Also notice that employees are still involved in an inheritance relationship; composition and inheritance are complementary tools.

When we run this module, our pizza shop handles two orders—one from Homer, and then one from Shaggy:

```
C:\python\examples> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

Again, this is mostly just a toy simulation, but the objects and interactions are representative of composites at work. As a rule of thumb, classes can represent just about any objects and relationships you can express in a sentence; just replace *nouns* with classes, and *verbs* with methods, and you'll have a first cut at a design.

Stream Processors Revisited

For a more realistic composition example, recall the generic data stream processor function we partially coded in the introduction to OOP in Chapter 25:

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Rather than using a simple function here, we might code this as a class that uses composition to do its work to provide more structure and support inheritance. The following file, *streams.py*, demonstrates one way to code the class:

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while 1:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert False, 'converter must be defined'       # Or raise exception
```

This class defines a `converter` method that it expects subclasses to fill in; it's an example of the *abstract superclass* model we outlined in Chapter 28 (more on `assert` in Part VII). Coded this way, `reader` and `writer` objects are embedded within the class instance (*composition*), and we supply the conversion logic in a subclass rather than passing in a converter function (*inheritance*). The file *converters.py* shows how:

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('spam.txt'), sys.stdout)
    obj.process()
```

Here, the `Uppercase` class inherits the stream-processing loop logic (and anything else that may be coded in its superclasses). It needs to define only what is unique about it—the data conversion logic. When this file is run, it makes and runs an instance that reads from the file *spam.txt* and writes the uppercase equivalent of that file to the `stdout` stream:

```
C:\lp4e> type spam.txt
spam
Spam
SPAM!

C:\lp4e> python converters.py
SPAM
SPAM
SPAM!
```

To process different sorts of streams, pass in different sorts of objects to the class construction call. Here, we use an output file instead of a stream:

```
C:\lp4e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp4e> type spamup.txt
SPAM
SPAM
SPAM!
```

But, as suggested earlier, we could also pass in arbitrary objects wrapped up in classes that define the required input and output method interfaces. Here's a simple example that passes in a writer class that wraps up the text inside HTML tags:

```
C:\lp4e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print('<PRE>%s</PRE>' % line.rstrip())
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>SPAM</PRE>
<PRE>SPAM</PRE>
<PRE>SPAM!</PRE>
```

If you trace through this example's control flow, you'll see that we get both uppercase conversion (by inheritance) and HTML formatting (by composition), even though the core processing logic in the original `Processor` superclass knows nothing about either step. The processing code only cares that writers have a `write` method and that a method named `convert` is defined; it doesn't care what those methods do when they are called. Such polymorphism and encapsulation of logic is behind much of the power of classes.

As is, the `Processor` superclass only provides a file-scanning loop. In more realistic work, we might extend it to support additional programming tools for its subclasses, and, in the process, turn it into a full-blown framework. Coding such a tool once in a superclass enables you to reuse it in all of your programs. Even in this simple example, because so much is packaged and inherited with classes, all we had to code was the HTML formatting step; the rest was free.

For another example of composition at work, see exercise 9 at the end of Chapter 31 and its solution in Appendix B; it's similar to the pizza shop example. We've focused on inheritance in this book because that is the main tool that the Python language itself provides for OOP. But, in practice, composition is used as much as inheritance as a way to structure classes, especially in larger systems. As we've seen, inheritance and composition are often complementary (and sometimes alternative) techniques. Because composition is a design issue outside the scope of the Python language and this book, though, I'll defer to other resources for more on this topic.

Why You Will Care: Classes and Persistence

I've mentioned Python's `pickle` and `shelve` object persistence support a few times in this part of the book because it works especially well with class instances. In fact, these tools are often compelling enough to motivate the use of classes in general—by picking or shelving a class instance, we get data storage that contains both data and logic combined.

For example, besides allowing us to simulate real-world interactions, the pizza shop classes developed in this chapter could also be used as the basis of a persistent restaurant database. Instances of classes can be stored away on disk in a single step using Python's `pickle` or `shelve` modules. We used shelves to store instances of classes in the OOP tutorial in Chapter 27, but the object pickling interface is remarkably easy to use as well:

```
import pickle
object = someClass()
file   = open(filename, 'wb')      # Create external file
pickle.dump(object, file)          # Save object in file

import pickle
file   = open(filename, 'rb')
object = pickle.load(file)         # Fetch it back later
```

Pickling converts in-memory objects to serialized byte streams (really, strings), which may be stored in files, sent across a network, and so on; unpickling converts back from byte streams to identical in-memory objects. Shelves are similar, but they automatically pickle objects to an access-by-key database, which exports a dictionary-like interface:

```
import shelve
object = someClass()
dbase  = shelve.open('filename')
dbase['key'] = object             # Save under key

import shelve
dbase  = shelve.open('filename')
object = dbase['key']             # Fetch it back later
```

In our pizza shop example, using classes to model employees means we can get a simple database of employees and shops with little extra work—pickling such instance objects to a file makes them persistent across Python program executions:

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.dat', 'wb'))
```

This stores an entire composite `shop` object in a file all at once. To bring it back later in another session or program, a single step suffices as well. In fact, objects restored this way retain both state and behavior:

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.dat', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> obj.order('Sue')
Sue orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Sue pays for item to <Employee: name=Pat, salary=40000>
```

See the standard library manual and later examples for more on pickles and shelves.

OOP and Delegation: "Wrapper" Objects

Beside inheritance and composition, object-oriented programmers often also talk about something called *delegation*, which usually implies controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as keeping track of accesses and so on. In Python, delegation is often implemented with the `__getattr__` method hook; because it intercepts accesses to nonexistent attributes, a *wrapper* class (sometimes called a *proxy* class) can use `__getattr__` to route arbitrary accesses to a wrapped object. The wrapper class retains the interface of the wrapped object and may add additional operations of its own.

Consider the file *trace.py*, for instance:

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)                # Trace fetch
        return getattr(self.wrapped, attrname)   # Delegate fetch
```

Recall from Chapter 29 that `__getattr__` gets the attribute name as a string. This code makes use of the `getattr` built-in function to fetch an attribute from the wrapped object by name string—`getattr(X,N)` is like `X.N`, except that `N` is an expression that evaluates to a string at runtime, not a variable. In fact, `getattr(X,N)` is similar to `X.__dict__[N]`, but the former also performs an inheritance search, like `X.N`, while the latter does not (see "Namespace Dictionaries" on page 696 for more on the `__dict__` attribute).

You can use the approach of this module's wrapper class to manage access to any object with attributes—lists, dictionaries, and even classes and instances. Here, the `wrapper` class simply prints a trace message on each attribute access and delegates the attribute request to the embedded `wrapped` object:

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])                  # Wrap a list
>>> x.append(4)                           # Delegate to list method
Trace: append
>>> x.wrapped                             # Print my member
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})         # Wrap a dictionary
>>> x.keys()                              # Delegate to dictionary method
Trace: keys
['a', 'b']
```

The net effect is to augment the entire interface of the `wrapped` object, with additional code in the `wrapper` class. We can use this to log our method calls, route method calls to extra or custom logic, and so on.

We'll revive the notions of wrapped objects and delegated operations as one way to extend built-in types in Chapter 31. If you are interested in the delegation design pattern, also watch for the discussions in Chapters 31 and 38 of *function decorators*, a strongly related concept designed to augment a specific function or method call rather than the entire interface of an object, and *class decorators*, which serve as a way to automatically add such delegation-based wrappers to all instances of a class.

Version skew note: In Python 2.6, operator overloading methods run by built-in operations are routed through generic attribute interception methods like `__getattr__`. Printing a wrapped object directly, for example, calls this method for `__repr__` or `__str__`, which then passes the call on to the wrapped object. In Python 3.0, this no longer happens: printing does not trigger `__getattr__`, and a default display is used instead. In 3.0, new-style classes look up operator overloading methods in classes and skip the normal instance lookup entirely. We'll return to this issue in Chapter 37, in the context of managed attributes; for now, keep in mind that you may need to redefine operator overloading methods in wrapper classes (either by hand, by tools, or by superclasses) if you want them to be intercepted in 3.0.

Pseudoprivate Class Attributes

Besides larger structuring goals, class designs often must address name usage too. In Part V, we learned that every name assigned at the top level of a module file is exported. By default, the same holds for classes—data hiding is a convention, and clients may fetch or change any class or instance attribute they like. In fact, attributes are all "public" and "virtual," in C++ terms; they're all accessible everywhere and are looked up dynamically at runtime.*

That said, Python today does support the notion of name "mangling" (i.e., expansion) to localize some names in classes. Mangled names are sometimes misleadingly called "private attributes," but really this is just a way to *localize* a name to the class that created it—name mangling does not prevent access by code outside the class. This feature is mostly intended to avoid namespace collisions in instances, not to restrict access to names in general; mangled names are therefore better called "pseudoprivate" than "private."

Pseudoprivate names are an advanced and entirely optional feature, and you probably won't find them very useful until you start writing general tools or larger class hierarchies for use in multiprogrammer projects. In fact, they are not always used even when they probably should be—more commonly, Python programmers code internal names with a single underscore (e.g., `_X`), which is just an informal convention to let you know that a name shouldn't be changed (it means nothing to Python itself).

Because you may see this feature in other people's code, though, you need to be somewhat aware of it, even if you don't use it yourself.

* This tends to scare people with a C++ background unnecessarily. In Python, it's even possible to change or completely delete a class method at runtime. On the other hand, almost nobody ever does this in practical programs. As a scripting language, Python is more about enabling than restricting. Also, recall from our discussion of operator overloading in Chapter 29 that `__getattr__` and `__setattr__` can be used to emulate privacy, but are generally not used for this purpose in practice. More on this when we code a more realistic privacy decorator Chapter 38.

Name Mangling Overview

Here's how name mangling works: names inside a `class` statement that start with two underscores but don't end with two underscores are automatically expanded to include the name of the enclosing class. For instance, a name like `__X` within a class named `Spam` is changed to `_Spam__X` automatically: the original name is prefixed with a single underscore and the enclosing class's name. Because the modified name contains the name of the enclosing class, it's somewhat unique; it won't clash with similar names created by other classes in a hierarchy.

Name mangling happens only in `class` statements, and only for names that begin with two leading underscores. However, it happens for *every* name preceded with double underscores—both class attributes (like method names) and instance attribute names assigned to `self` attributes. For example, in a class named `Spam`, a method named `__meth` is mangled to `_Spam__meth`, and an instance attribute reference `self.__X` is transformed to `self._Spam__X`. Because more than one class may add attributes to an instance, this mangling helps avoid clashes—but we need to move on to an example to see how.

Why Use Pseudoprivate Attributes?

One of the main problems that the pseudoprivate attribute feature is meant to alleviate has to do with the way instance attributes are stored. In Python, all instance attributes wind up in the single instance object at the bottom of the class tree. This is different from the C++ model, where each class gets its own space for data members it defines.

Within a class method in Python, whenever a method assigns to a `self` attribute (e.g., `self.`*`attr`* `=` *`value`*), it changes or creates an attribute in the instance (inheritance searches happen only on reference, not on assignment). Because this is true even if multiple classes in a hierarchy assign to the same attribute, collisions are possible.

For example, suppose that when a programmer codes a class, she assumes that she owns the attribute name `X` in the instance. In this class's methods, the name is set, and later fetched:

```
class C1:
    def meth1(self): self.X = 88          # I assume X is mine
    def meth2(self): print(self.X)
```

Suppose further that another programmer, working in isolation, makes the same assumption in a class that he codes:

```
class C2:
    def metha(self): self.X = 99          # Me too
    def methb(self): print(self.X)
```

Both of these classes work by themselves. The problem arises if the two classes are ever mixed together in the same class tree:

```
class C3(C1, C2): ...
I = C3()                                # Only 1 X in I!
```

Now, the value that each class gets back when it says `self.X` will depend on which class assigned it last. Because all assignments to `self.X` refer to the same single instance, there is only one `X` attribute—`I.X`—no matter how many classes use that attribute name.

To guarantee that an attribute belongs to the class that uses it, prefix the name with double underscores everywhere it is used in the class, as in this file, *private.py*:

```
class C1:
    def meth1(self): self.__X = 88      # Now X is mine
    def meth2(self): print(self.__X)    # Becomes _C1__X in I
class C2:
    def metha(self): self.__X = 99      # Me too
    def methb(self): print(self.__X)    # Becomes _C2__X in I

class C3(C1, C2): pass
I = C3()                                # Two X names in I

I.meth1(); I.metha()
print(I.__dict__)
I.meth2(); I.methb()
```

When thus prefixed, the `X` attributes will be expanded to include the names of their classes before being added to the instance. If you run a `dir` call on `I` or inspect its namespace dictionary after the attributes have been assigned, you'll see the expanded names, `_C1__X` and `_C2__X`, but not `X`. Because the expansion makes the names unique within the instance, the class coders can safely assume that they truly own any names that they prefix with two underscores:

```
% python private.py
{'_C2__X': 99, '_C1__X': 88}
88
99
```

This trick can avoid potential name collisions in the instance, but note that it does not amount to true privacy. If you know the name of the enclosing class, you can still access either of these attributes anywhere you have a reference to the instance by using the fully expanded name (e.g., `I._C1__X = 77`). On the other hand, this feature makes it less likely that you will *accidentally* step on a class's names.

Pseudoprivate attributes are also useful in larger frameworks or tools, both to avoid introducing new method names that might accidentally hide definitions elsewhere in the class tree and to reduce the chance of internal methods being replaced by names defined lower in the tree. If a method is intended for use only within a class that may be mixed into other classes, the double underscore prefix ensures that the method won't interfere with other names in the tree, especially in multiple-inheritance scenarios:

```
class Super:
    def method(self): ...                   # A real application method

class Tool:
```