CHAPTER 4

# Introducing Python Object Types

This chapter begins our tour of the Python language. In an informal sense, in Python, we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations. In this part of the book, our focus is on that stuff, and the things our programs can do with it.

Somewhat more formally, in Python, data takes the form of *objects*—either built-in objects that Python provides, or objects we create using Python or external language tools such as C extension libraries. Although we'll firm up this definition later, objects are essentially just pieces of memory, with values and sets of associated operations.

Because objects are the most fundamental notion in Python programming, we'll start this chapter with a survey of Python's built-in object types.

By way of introduction, however, let's first establish a clear picture of how this chapter fits into the overall Python picture. From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

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

The discussion of modules in Chapter 3 introduced the highest level of this hierarchy. This part's chapters begin at the bottom, exploring both built-in objects and the expressions you can code to use them.

## Why Use Built-in Types?

If you've used lower-level languages such as C or C++, you know that much of your work centers on implementing *objects*—also known as *data structures*—to represent the components in your application's domain. You need to lay out memory structures, manage memory allocation, implement search and access routines, and so on. These chores are about as tedious (and error-prone) as they sound, and they usually distract from your program's real goals.

In typical Python programs, most of this grunt work goes away. Because Python provides powerful object types as an intrinsic part of the language, there's usually no need to code object implementations before you start solving problems. In fact, unless you have a need for special processing that built-in types don't provide, you're almost always better off using a built-in object instead of implementing your own. Here are some reasons why:

- **Built-in objects make programs easy to write**. For simple tasks, built-in types are often all you need to represent the structure of problem domains. Because you get powerful tools such as collections (lists) and search tables (dictionaries) for free, you can use them immediately. You can get a lot of work done with Python's built-in object types alone.
- **Built-in objects are components of extensions**. For more complex tasks, you may need to provide your own objects using Python classes or C language interfaces. But as you'll see in later parts of this book, objects implemented manually are often built on top of built-in types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- **Built-in objects are often more efficient than custom data structures**. Python's built-in types employ already optimized data structure algorithms that are implemented in C for speed. Although you can write similar object types on your own, you'll usually be hard-pressed to get the level of performance built-in object types provide.
- **Built-in objects are a standard part of the language**. In some ways, Python borrows both from languages that rely on built-in tools (e.g., LISP) and languages that rely on the programmer to provide tool implementations or frameworks of their own (e.g., C++). Although you can implement unique object types in Python, you don't need to do so just to get started. Moreover, because Python's built-ins are standard, they're always the same; proprietary frameworks, on the other hand, tend to differ from site to site.

In other words, not only do built-in object types make programming easier, but they're also more powerful and efficient than most of what can be created from scratch. Regardless of whether you implement new object types, built-in objects form the core of every Python program.

## Python's Core Data Types

Table 4-1 previews Python's built-in object types and some of the syntax used to code their *literals*—that is, the expressions that generate these objects.* Some of these types will probably seem familiar if you've used other languages; for instance, numbers and strings represent numeric and textual values, respectively, and files provide an interface for processing files stored on your computer.

*Table 4-1. Built-in objects preview*

| Object type | Example literals/creation |
|---|---|
| Numbers | `1234`, `3.1415`, `3+4j`, `Decimal`, `Fraction` |
| Strings | `'spam'`, `"guido's"`, `b'a\x01c'` |
| Lists | `[1, [2, 'three'], 4]` |
| Dictionaries | `{'food': 'spam', 'taste': 'yum'}` |
| Tuples | `(1, 'spam', 4, 'U')` |
| Files | `myfile = open('eggs', 'r')` |
| Sets | `set('abc'), {'a', 'b', 'c'}` |
| Other core types | Booleans, types, None |
| Program unit types | Functions, modules, classes (Part IV, Part V, Part VI) |
| Implementation-related types | Compiled code, stack tracebacks (Part IV, Part VII) |

Table 4-1 isn't really complete, because *everything* we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we perform network scripting, we use socket objects. These other kinds of objects are generally created by importing and using modules and have behavior all their own.

As we'll see in later parts of the book, *program units* such as functions, modules, and classes are objects in Python too—they are created with statements and expressions such as `def`, `class`, `import`, and `lambda` and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of *implementation-related* types such as compiled code objects, which are generally of interest to tool builders more than application developers; these are also discussed in later parts of this text.

We usually call the other object types in Table 4-1 *core* data types, though, because they are effectively built into the Python language—that is, there is specific expression syntax for generating most of them. For instance, when you run the following code:

```
>>> 'spam'
```

* In this book, the term *literal* simply means an expression whose syntax generates an object—sometimes also called a *constant*. Note that the term "constant" does not imply objects or variables that can never be changed (i.e., this term is unrelated to C++'s `const` or Python's "immutable"—a topic explored in the section "Immutability" on page 82).

you are, technically speaking, running a literal expression that generates and returns a new string object. There is specific Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a list, one in curly braces makes a dictionary, and so on. Even though, as we'll see, there are no type declarations in Python, the syntax of the expressions you run determines the types of objects you create and use. In fact, object-generation expressions like those in Table 4-1 are generally where types originate in the Python language.

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a list. As you'll learn, Python is *dynamically typed* (it keeps track of types for you automatically instead of requiring declaration code), but it is also *strongly typed* (you can perform on an object only operations that are valid for its type).

Functionally, the object types in Table 4-1 are more general and powerful than what you may be accustomed to. For instance, you'll find that lists and dictionaries alone are powerful data representation tools that obviate most of the work you do to support collections and searching in lower-level languages. In short, lists provide ordered collections of other objects, while dictionaries store objects by key; both lists and dictionaries may be nested, can grow and shrink on demand, and may contain objects of any type.

We'll study each of the object types in Table 4-1 in detail in upcoming chapters. Before digging into the details, though, let's begin by taking a quick look at Python's core objects in action. The rest of this chapter provides a preview of the operations we'll explore in more depth in the chapters that follow. Don't expect to find the full story here—the goal of this chapter is just to whet your appetite and introduce some key ideas. Still, the best way to get started is to get started, so let's jump right into some real code.

## Numbers

If you've done any programming or scripting in the past, some of the object types in Table 4-1 will probably seem familiar. Even if you haven't, numbers are fairly straightforward. Python's core objects set includes the usual suspects: integers (numbers without a fractional part), floating-point numbers (roughly, numbers with a decimal point in them), and more exotic numeric types (complex numbers with imaginary parts, fixed-precision decimals, rational fractions with numerator and denominator, and full-featured sets).

Although it offers some fancier options, Python's basic number types are, well, basic. Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, a star (*) is used for multiplication, and two stars (**) are used for exponentiation:

```
>>> 123 + 222                  # Integer addition
345
>>> 1.5 * 4                    # Floating-point multiplication
6.0
>>> 2 ** 100                   # 2 to the power 100
1267650600228229401496703205376
```

Notice the last result here: Python 3.0's integer type automatically provides extra precision for large numbers like this when needed (in 2.6, a separate long integer type handles numbers too large for the normal integer type in similar ways). You can, for instance, compute 2 to the power 1,000,000 as an integer in Python, but you probably shouldn't try to print the result—with more than 300,000 digits, you may be waiting awhile!

```
>>> len(str(2 ** 1000000))     # How many digits in a really BIG number?
301030
```

Once you start experimenting with floating-point numbers, you're likely to stumble across something that may look a bit odd on first glance:

```
>>> 3.1415 * 2                 # repr: as code
6.2830000000000004
>>> print(3.1415 * 2)          # str: user-friendly
6.283
```

The first result isn't a bug; it's a display issue. It turns out that there are two ways to print every object: with full precision (as in the first result shown here), and in a user-friendly form (as in the second). Formally, the first form is known as an object's as-code `repr`, and the second is its user-friendly `str`. The difference can matter when we step up to using classes; for now, if something looks odd, try showing it with a `print` built-in call statement.

Besides expressions, there are a handful of useful numeric modules that ship with Python—*modules* are just packages of additional tools that we import to use:

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

The `math` module contains more advanced numeric tools as functions, while the `random` module performs random number generation and random selections (here, from a Python list, introduced later in this chapter):

```
>>> import random
>>> random.random()
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1
```

Python also includes more exotic numeric objects—such as complex, fixed-precision, and rational numbers, as well as sets and Booleans—and the third-party open source

extension domain has even more (e.g., matrixes and vectors). We'll defer discussion of these types until later in the book.

So far, we've been using Python much like a simple calculator; to do better justice to its built-in types, let's move on to explore strings.

# Strings

Strings are used to record textual information as well as arbitrary collections of bytes. They are our first example of what we call a *sequence* in Python—that is, a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative position. Strictly speaking, strings are sequences of one-character strings; other types of sequences include lists and tuples, covered later.

## Sequence Operations

As sequences, strings support operations that assume a positional ordering among items. For example, if we have a four-character string, we can verify its length with the built-in `len` function and fetch its components with *indexing* expressions:

```
>>> S = 'Spam'
>>> len(S)                # Length
4
>>> S[0]                  # The first item in S, indexing by zero-based position
'S'
>>> S[1]                  # The second item from the left
'p'
```

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on.

Notice how we assign the string to a *variable* named `S` here. We'll go into detail on how this works later (especially in Chapter 6), but Python variables never need to be declared ahead of time. A variable is created when you assign it a value, may be assigned any type of object, and is replaced with its value when it shows up in an expression. It must also have been previously assigned by the time you use its value. For the purposes of this chapter, it's enough to know that we need to assign an object to a variable in order to save it for later use.

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right:

```
>>> S[-1]                 # The last item from the end in S
'm'
>>> S[-2]                 # The second to last item from the end
'a'
```

Formally, a negative index is simply added to the string's size, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

```
>>> S[-1]               # The last item in S
'm'
>>> S[len(S)-1]         # Negative indexing, the hard way
'm'
```

Notice that we can use an arbitrary expression in the square brackets, not just a hard-coded number literal—anywhere that Python expects a value, we can use a literal, a variable, or any expression. Python's syntax is completely general this way.

In addition to simple positional indexing, sequences also support a more general form of indexing known as *slicing*, which is a way to extract an entire section (slice) in a single step. For example:

```
>>> S                   # A 4-character string
'Spam'
>>> S[1:3]              # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

Probably the easiest way to think of slices is that they are a way to extract an entire *column* from a string in a single step. Their general form, `X[I:J]`, means "give me everything in `X` from offset `I` up to but not including offset `J`." The result is returned in a new object. The second of the preceding operations, for instance, gives us all the characters in string `S` from offsets 1 through 2 (that is, 3 – 1) as a new string. The effect is to slice or "parse out" the two characters in the middle.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced. This leads to some common usage variations:

```
>>> S[1:]               # Everything past the first (1:len(S))
'pam'
>>> S                   # S itself hasn't changed
'Spam'
>>> S[0:3]              # Everything but the last
'Spa'
>>> S[:3]               # Same as S[0:3]
'Spa'
>>> S[:-1]              # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:]                # All of S as a top-level copy (0:len(S))
'Spam'
```

Note how negative offsets can be used to give bounds for slices, too, and how the last operation effectively copies the entire string. As you'll learn later, there is no reason to copy a string, but this form can be useful for sequences like lists.

Finally, as sequences, strings also support *concatenation* with the plus sign (joining two strings into a new string) and *repetition* (making a new string by repeating another):

```
>>> S
Spam'
>>> S + 'xyz'           # Concatenation
```

```
'Spamxyz'
>>> S                          # S is unchanged
'Spam'
>>> S * 8                      # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Notice that the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings. This is a general property of Python that we'll call *polymorphism* later in the book—in sum, the meaning of an operation depends on the objects being operated on. As you'll see when we study dynamic typing, this polymorphism property accounts for much of the conciseness and flexibility of Python code. Because types aren't constrained, a Python-coded operation can normally work on many different types of objects automatically, as long as they support a compatible interface (like the + operation here). This turns out to be a huge idea in Python; you'll learn more about it later on our tour.

## Immutability

Notice that in the prior examples, we were not changing the original string with any of the operations we ran on it. Every string operation is defined to produce a new string as its result, because strings are *immutable* in Python—they cannot be changed in-place after they are created. For example, you can't change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python cleans up old objects as you go (as you'll see later), this isn't as inefficient as it may sound:

```
>>> S
'Spam'
>>> S[0] = 'z'              # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]         # But we can run expressions to make new objects
>>> S
'zpam'
```

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core types, numbers, strings, and tuples are immutable; lists and dictionaries are not (they can be changed in-place freely). Among other things, immutability can be used to guarantee that an object remains constant throughout your program.

## Type-Specific Methods

Every string operation we've studied so far is really a sequence operation—that is, these operations will work on other sequences in Python as well, including lists and tuples. In addition to generic sequence operations, though, strings also have operations all their own, available as *methods*—functions attached to the object, which are triggered with a call expression.

For example, the string `find` method is the basic substring search operation (it returns the offset of the passed-in substring, or `-1` if it is not present), and the string `replace` method performs global searches and replacements:

```
>>> S.find('pa')              # Find the offset of a substring
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ')    # Replace occurrences of a substring with another
'SXYZm'
>>> S
'Spam'
```

Again, despite the names of these string methods, we are not changing the original strings here, but creating new strings as the results—because strings are immutable, we have to do it this way. String methods are the first line of text-processing tools in Python. Other methods split a string into substrings on a delimiter (handy as a simple form of parsing), perform case conversions, test the content of the string (digits, letters, and so on), and strip whitespace characters off the ends of the string:

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')           # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'spam'
>>> S.upper()                 # Upper- and lowercase conversions
'SPAM'

>>> S.isalpha()               # Content tests: isalpha, isdigit, etc.
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line = line.rstrip()      # Remove whitespace characters on the right side
>>> line
'aaa,bbb,ccccc,dd'
```

Strings also support an advanced substitution operation known as *formatting*, available as both an expression (the original) and a string method call (new in 2.6 and 3.0):

```
>>> '%s, eggs, and %s' % ('spam', 'SPAM!')           # Formatting expression (all)
'spam, eggs, and SPAM!'

>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!')     # Formatting method (2.6, 3.0)
'spam, eggs, and SPAM!'
```

One note here: although sequence operations are generic, methods are not—although some types share some method names, string method operations generally work only on strings, and nothing else. As a rule of thumb, Python's toolset is layered: generic operations that span multiple types show up as built-in functions or expressions (e.g., `len(X)`, `X[0]`), but type-specific operations are method calls (e.g., `aString.upper()`). Finding the tools you need among all these categories will become more natural as you use Python more, but the next section gives a few tips you can use right now.

## Getting Help

The methods introduced in the prior section are a representative, but small, sample of what is available for string objects. In general, this book is not exhaustive in its look at object methods. For more details, you can always call the built-in `dir` function, which returns a list of all the attributes available for a given object. Because methods are function attributes, they will show up in this list. Assuming `S` is still the string, here are its attributes on Python 3.0 (Python 2.6 varies slightly):

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'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', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

You probably won't care about the names with underscores in this list until later in the book, when we study operator overloading in classes—they represent the implementation of the string object and are available to support customization. In general, leading and trailing double underscores is the naming pattern Python uses for implementation details. The names without the underscores in this list are the callable methods on string objects.

The `dir` function simply gives the methods' names. To ask what they do, you can pass them to the `help` function:

```
>>> help(S.replace)
Help on built-in function replace:

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

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

`help` is one of a handful of interfaces to a system of code that ships with Python known as *PyDoc*—a tool for extracting documentation from objects. Later in the book, you'll see that PyDoc can also render its reports in HTML format.

You can also ask for help on an entire string (e.g., `help(S)`), but you may get more help than you want to see—i.e., information about every string method. It's generally better to ask about a specific method.

For more details, you can also consult Python's standard library reference manual or commercially published reference books, but `dir` and `help` are the first line of documentation in Python.

## Other Ways to Code Strings

So far, we've looked at the string object's sequence operations and type-specific methods. Python also provides a variety of ways for us to code strings, which we'll explore in greater depth later. For instance, special characters can be represented as backslash escape sequences:

```
>>> S = 'A\nB\tC'           # \n is end-of-line, \t is tab
>>> len(S)                  # Each stands for just one character
5

>>> ord('\n')               # \n is a byte with the binary value 10 in ASCII
10

>>> S = 'A\0B\0C'           # \0, a binary zero byte, does not terminate string
>>> len(S)
5
```

Python allows strings to be enclosed in single or double quote characters (they mean the same thing). It also allows multiline string literals enclosed in triple quotes (single or double)—when this form is used, all the lines are concatenated together, and end-of-line characters are added where line breaks appear. This is a minor syntactic convenience, but it's useful for embedding things like HTML and XML code in a Python script:

```
>>> msg = """ aaaaaaaaaaaa
bbb''bbbbbbbbbb""bbbbbbb'bbbb
cccccccccccccc"""
>>> msg
'\naaaaaaaaaaaaa\nbbb\'\'\'bbbbbbbbbb""bbbbbbb\'bbbb\ncccccccccccccc'
```

Python also supports a *raw* string literal that turns off the backslash escape mechanism (such string literals start with the letter `r`), as well as *Unicode* string support that supports internationalization. In 3.0, the basic `str` string type handles Unicode too (which makes sense, given that ASCII text is a simple kind of Unicode), and a `bytes` type represents raw byte strings; in 2.6, Unicode is a separate type, and `str` handles both 8-bit strings and binary data. Files are also changed in 3.0 to return and accept `str` for text and `bytes` for binary data. We'll meet all these special string forms in later chapters.

## Pattern Matching

One point worth noting before we move on is that none of the string object's methods support pattern-based text processing. Text pattern matching is an advanced tool outside this book's scope, but readers with backgrounds in other scripting languages may be interested to know that to do pattern matching in Python, we import a module called

re. This module has analogous calls for searching, splitting, and replacement, but because we can use patterns to specify substrings, we can be much more general:

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello    Python world')
>>> match.group(1)
'Python '
```

This example searches for a substring that begins with the word "Hello," followed by zero or more tabs or spaces, followed by arbitrary characters to be saved as a matched group, terminated by the word "world." If such a substring is found, portions of the substring matched by parts of the pattern enclosed in parentheses are available as groups. The following pattern, for example, picks out three groups separated by slashes:

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
```

Pattern matching is a fairly advanced text-processing tool by itself, but there is also support in Python for even more advanced language processing, including natural language processing. I've already said enough about strings for this tutorial, though, so let's move on to the next type.

# Lists

The Python list object is the most general sequence provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in-place by assignment to offsets as well as a variety of list method calls.

## Sequence Operations

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that the results are usually lists instead of strings. For instance, given a three-item list:

```
>>> L = [123, 'spam', 1.23]          # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3
```

we can index, slice, and so on, just as for strings:

```
>>> L[0]                             # Indexing by position
123

>>> L[:-1]                           # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6]                    # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]
```

```
>>> L                              # We're not changing the original list
[123, 'spam', 1.23]
```

## Type-Specific Operations

Python's lists are related to arrays in other languages, but they tend to be more powerful. For one thing, they have no fixed type constraint—the list we just looked at, for example, contains three objects of completely different types (an integer, a string, and a floating-point number). Further, lists have no fixed size. That is, they can grow and shrink on demand, in response to list-specific operations:

```
>>> L.append('NI')              # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)                    # Shrinking: delete an item in the middle
1.23

>>> L                           # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

Here, the list `append` method expands the list's size and inserts an item at the end; the `pop` method (or an equivalent `del` statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert an item at an arbitrary position (`insert`), remove a given item by value (`remove`), and so on. Because lists are mutable, most list methods also change the list object in-place, instead of creating a new one:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

The list `sort` method here, for example, orders the list in ascending fashion by default, and `reverse` reverses it—in both cases, the methods modify the list directly.

## Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end:

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range
```

```
>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

This is intentional, as it's usually an error to try to assign off the end of a list (and a particularly nasty one in the C language, which doesn't do as much error checking as Python). Rather than silently growing the list in response, Python reports an error. To grow a list, we call list methods such as `append` instead.

## Nesting

One nice feature of Python's core data types is that they support arbitrary nesting—we can nest them in any combination, and as deeply as we like (for example, we can have a list that contains a dictionary, which contains another list, and so on). One immediate application of this feature is to represent matrixes, or "multidimensional arrays" in Python. A list with nested lists will do the job for basic applications:

```
>>> M = [[1, 2, 3],              # A 3 × 3 matrix, as nested lists
        [4, 5, 6],               # Code can span lines if bracketed
        [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Here, we've coded a list that contains three other lists. The effect is to represent a 3 × 3 matrix of numbers. Such a structure can be accessed in a variety of ways:

```
>>> M[1]                         # Get row 2
[4, 5, 6]

>>> M[1][2]                      # Get row 2, then get item 3 within the row
6
```

The first operation here fetches the entire second row, and the second grabs the third item within that row. Stringing together index operations takes us deeper and deeper into our nested-object structure.†

## Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation known as a *list comprehension expression*, which turns out to be a powerful way to process structures like our matrix. Suppose, for instance, that we need to extract the second column of our sample matrix. It's easy to grab rows by simple indexing

† This matrix structure works for small-scale tasks, but for more serious number crunching you will probably want to use one of the numeric extensions to Python, such as the open source *NumPy* system. Such tools can store and process large matrixes much more efficiently than our nested list structure. NumPy has been said to turn Python into the equivalent of a free and more powerful version of the Matlab system, and organizations such as NASA, Los Alamos, and JPMorgan Chase use this tool for scientific and financial tasks. Search the Web for more details.

because the matrix is stored by rows, but it's almost as easy to get a column with a list comprehension:

```
>>> col2 = [row[1] for row in M]          # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                     # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List comprehensions derive from set notation; they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right. List comprehensions are coded in square brackets (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name (`row`, here). The preceding list comprehension means basically what it says: "Give me `row[1]` for each row in matrix `M`, in a new list." The result is a new list containing column 2 of the matrix.

List comprehensions can be more complex in practice:

```
>>> [row[1] + 1 for row in M]                # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]  # Filter out odd items
[2, 8]
```

The first operation here, for instance, adds 1 to each item as it is collected, and the second uses an `if` clause to filter odd numbers out of the result using the `%` modulus expression (remainder of division). List comprehensions make new lists of results, but they can be used to iterate over any iterable object. Here, for instance, we use list comprehensions to step over a hardcoded list of coordinates and a string:

```
>>> diag = [M[i][i] for i in [0, 1, 2]]       # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']         # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

List comprehensions, and relatives like the `map` and `filter` built-in functions, are a bit too involved for me to say more about them here. The main point of this brief introduction is to illustrate that Python includes both simple and advanced tools in its arsenal. List comprehensions are an optional feature, but they tend to be handy in practice and often provide a substantial processing speed advantage. They also work on any type that is a sequence in Python, as well as some types that are not. You'll hear much more about them later in this book.

As a preview, though, you'll find that in recent Pythons, comprehension syntax in parentheses can also be used to create *generators* that produce results on demand (the `sum` built-in, for instance, sums items in a sequence):

```
>>> G = (sum(row) for row in M)          # Create a generator of row sums
>>> next(G)
6
>>> next(G)                              # Run the iteration protocol
15
```

The `map` built-in can do similar work, by generating the results of running items through a function. Wrapping it in `list` forces it to return all its values in Python 3.0:

```
>>> list(map(sum, M))                    # Map sum over items in M
[6, 15, 24]
```

In Python 3.0, comprehension syntax can also be used to create *sets* and *dictionaries*:

```
>>> {sum(row) for row in M}              # Create a set of row sums
{24, 6, 15}

>>> {i : sum(M[i]) for i in range(3)}    # Creates key/value table of row sums
{0: 6, 1: 15, 2: 24}
```

In fact, lists, sets, and dictionaries can all be built with comprehensions in 3.0:

```
>>> [ord(x) for x in 'spaam']            # List of character ordinals
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaam'}            # Sets remove duplicates
{112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'}         # Dictionary keys are unique
{'a': 97, 'p': 112, 's': 115, 'm': 109}
```

To understand objects like generators, sets, and dictionaries, though, we must move ahead.

# Dictionaries

Python dictionaries are something completely different (Monty Python reference intended)—they are not sequences at all, but are instead known as *mappings*. Mappings are also collections of other objects, but they store objects by key instead of by relative position. In fact, mappings don't maintain any reliable left-to-right order; they simply map keys to associated values. Dictionaries, the only mapping type in Python's core objects set, are also mutable: they may be changed in-place and can grow and shrink on demand, like lists.

## Mapping Operations

When written as literals, dictionaries are coded in curly braces and consist of a series of "key: value" pairs. Dictionaries are useful anytime we need to associate a set of values with keys—to describe the properties of something, for instance. As an example, consider the following three-item dictionary (with keys "food," "quantity," and "color"):

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

We can index this dictionary by key to fetch and change the keys' associated values. The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key, not a relative position:

```
>>> D['food']                 # Fetch value of key 'food'
'Spam'

>>> D['quantity'] += 1        # Add 1 to 'quantity' value
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways. The following code, for example, starts with an empty dictionary and fills it out one key at a time. Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys:

```
>>> D = {}
>>> D['name'] = 'Bob'         # Create keys by assignment
>>> D['job']  = 'dev'
>>> D['age']  = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print(D['name'])
Bob
```

Here, we're effectively using dictionary keys as field names in a record that describes someone. In other applications, dictionaries can also be used to replace searching operations—indexing a dictionary by key is often the fastest way to code a search in Python.

## Nesting Revisited

In the prior example, we used a dictionary to describe a hypothetical person, with three keys. Suppose, though, that the information is more complex. Perhaps we need to record a first name and a last name, along with multiple job titles. This leads to another application of Python's object nesting in action. The following dictionary, coded all at once as a literal, captures more structured information:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
           'job':  ['dev', 'mgr'],
           'age':  40.5}
```

Here, we again have a three-key dictionary at the top (keys "name," "job," and "age"), but the values have become more complex: a nested dictionary for the name to support multiple parts, and a nested list for the job to support multiple roles and future expansion. We can access the components of this structure much as we did for our matrix earlier, but this time some of our indexes are dictionary keys, not list offsets:

```
>>> rec['name']                          # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']                  # Index the nested dictionary
'Smith'

>>> rec['job']                           # 'job' is a nested list
['dev', 'mgr']
>>> rec['job'][-1]                       # Index the nested list
'mgr'

>>> rec['job'].append('janitor')         # Expand Bob's job description in-place
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}
```

Notice how the last operation here expands the nested job list—because the job list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (object memory layout will be discussed further later in this book).

The real reason for showing you this example is to demonstrate the *flexibility* of Python's core data types. As you can see, nesting allows us to build up complex information structures directly and easily. Building a similar structure in a low-level language like C would be tedious and require much more code: we would have to lay out and declare structures and arrays, fill out values, link everything together, and so on. In Python, this is all automatic—running the expression creates the entire nested object structure for us. In fact, this is one of the main benefits of scripting languages like Python.

Just as importantly, in a lower-level language we would have to be careful to clean up all of the object's space when we no longer need it. In Python, when we lose the last reference to the object—by assigning its variable to something else, for example—all of the memory space occupied by that object's structure is automatically cleaned up for us:

```
>>> rec = 0                              # Now the object's space is reclaimed
```

Technically speaking, Python has a feature known as *garbage collection* that cleans up unused memory as your program runs and frees you from having to manage such details in your code. In Python, the space is reclaimed immediately, as soon as the last reference to an object is removed. We'll study how this works later in this book; for now, it's enough to know that you can use objects freely, without worrying about creating their space or cleaning up as you go.‡

‡ Keep in mind that the `rec` record we just created really could be a database record, when we employ Python's *object persistence* system—an easy way to store native Python objects in files or access-by-key databases. We won't go into details here, but watch for discussion of Python's `pickle` and `shelve` modules later in this book.

## Sorting Keys: for Loops

As mappings, as we've already seen, dictionaries only support accessing items by key. However, they also support type-specific operations with method calls that are useful in a variety of common use cases.

As mentioned earlier, because dictionaries are not sequences, they don't maintain any dependable left-to-right order. This means that if we make a dictionary and print it back, its keys may come back in a different order than that in which we typed them:

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

What do we do, though, if we do need to impose an ordering on a dictionary's items? One common solution is to grab a list of keys with the dictionary `keys` method, sort that with the list `sort` method, and then step through the result with a Python `for` loop (be sure to press the Enter key twice after coding the `for` loop below—as explained in Chapter 3, an empty line means "go" at the interactive prompt, and the prompt changes to "..." on some interfaces):

```
>>> Ks = list(D.keys())                # Unordered keys list
>>> Ks                                 # A list in 2.6, "view" in 3.0: use list()
['a', 'c', 'b']

>>> Ks.sort()                          # Sorted keys list
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:                     # Iterate though sorted keys
        print(key, '=>', D[key])       # <== press Enter twice here

a => 1
b => 2
c => 3
```

This is a three-step process, although, as we'll see in later chapters, in recent versions of Python it can be done in one step with the newer `sorted` built-in function. The `sorted` call returns the result and sorts a variety of object types, in this case sorting dictionary keys automatically:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
        print(key, '=>', D[key])

a => 1
b => 2
c => 3
```

Besides showcasing dictionaries, this use case serves to introduce the Python `for` loop. The `for` loop is a simple and efficient way to step through all the items in a sequence

and run a block of code for each item in turn. A user-defined loop variable (`key`, here) is used to reference the current item each time through. The net effect in our example is to print the unordered dictionary's keys and values, in sorted-key order.

The `for` loop, and its more general cousin the `while` loop, are the main ways we code repetitive tasks as statements in our scripts. Really, though, the `for` loop (like its relative the list comprehension, which we met earlier) is a sequence operation. It works on any object that is a sequence and, like the list comprehension, even on some things that are not. Here, for example, it is stepping across the characters in a string, printing the uppercase version of each as it goes:

```
>>> for c in 'spam':
        print(c.upper())

S
P
A
M
```

Python's `while` loop is a more general sort of looping tool, not limited to stepping across sequences:

```
>>> x = 4
>>> while x > 0:
        print('spam!' * x)
        x -= 1

spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

We'll discuss looping statements, syntax, and tools in depth later in the book.

## Iteration and Optimization

If the last section's `for` loop looks like the list comprehension expression introduced earlier, it should: both are really general iteration tools. In fact, both will work on any object that follows the *iteration protocol*—a pervasive idea in Python that essentially means a physically stored sequence in memory, or an object that generates one item at a time in the context of an iteration operation. An object falls into the latter category if it responds to the `iter` built-in with an object that advances in response to `next`. The *generator* comprehension expression we saw earlier is such an object.

I'll have more to say about the iteration protocol later in this book. For now, keep in mind that every Python tool that scans an object from left to right uses the iteration protocol. This is why the `sorted` call used in the prior section works on the dictionary directly—we don't have to call the `keys` method to get a sequence because dictionaries are iterable objects, with a `next` that returns successive keys.

This also means that any list comprehension expression, such as this one, which computes the squares of a list of numbers:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

can always be coded as an equivalent `for` loop that builds the result list manually by appending as it goes:

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:          # This is what a list comprehension does
        squares.append(x ** 2)          # Both run the iteration protocol internally

>>> squares
[1, 4, 9, 16, 25]
```

The list comprehension, though, and related functional programming tools like `map` and `filter`, will generally run faster than a `for` loop today (perhaps even twice as fast)—a property that could matter in your programs for large data sets. Having said that, though, I should point out that performance measures are tricky business in Python because it optimizes so much, and performance can vary from release to release.

A major rule of thumb in Python is to code for simplicity and readability first and worry about performance later, after your program is working, and after you've proved that there is a genuine performance concern. More often than not, your code will be quick enough as it is. If you do need to tweak code for performance, though, Python includes tools to help you out, including the `time` and `timeit` modules and the `profile` module. You'll find more on these later in this book, and in the Python manuals.

## Missing Keys: if Tests

One other note about dictionaries before we move on. Although we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                        # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                             # Referencing a nonexistent key is an error
...error text omitted...
KeyError: 'f'
```

This is what we want—it's usually a programming error to fetch something that isn't really there. But in some generic programs, we can't always know what keys will be present when we write our code. How do we handle such cases and avoid errors? One trick is to test ahead of time. The dictionary `in` membership expression allows us to

query the existence of a key and branch on the result with a Python `if` statement (as with the `for`, be sure to press Enter twice to run the `if` interactively here):

```
>>> 'f' in D
False

>>> if not 'f' in D:
       print('missing')

missing
```

I'll have much more to say about the `if` statement and statement syntax in general later in this book, but the form we're using here is straightforward: it consists of the word `if`, followed by an expression that is interpreted as a true or false result, followed by a block of code to run if the test is true. In its full form, the `if` statement can also have an `else` clause for a default case, and one or more `elif` (else if) clauses for other tests. It's the main selection tool in Python, and it's the way we code logic in our scripts.

Still, there are other ways to create dictionaries and avoid accessing nonexistent keys: the `get` method (a conditional index with a default); the Python 2.X `has_key` method (which is no longer available in 3.0); the `try` statement (a tool we'll first meet in Chapter 10 that catches and recovers from exceptions altogether); and the `if`/`else` expression (essentially, an `if` statement squeezed onto a single line). Here are a few examples:

```
>>> value = D.get('x', 0)                  # Index but with a default
>>> value
0
>>> value = D['x'] if 'x' in D else 0       # if/else expression form
>>> value
0
```

We'll save the details on such alternatives until a later chapter. For now, let's move on to tuples.

# Tuples

The tuple object (pronounced "toople" or "tuhple," depending on who you ask) is roughly like a list that cannot be changed—tuples are sequences, like lists, but they are immutable, like strings. Syntactically, they are coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations:

```
>>> T = (1, 2, 3, 4)          # A 4-item tuple
>>> len(T)                    # Length
4

>> T + (5, 6)                 # Concatenation
(1, 2, 3, 4, 5, 6)

>>> T[0]                      # Indexing, slicing, and more
1
```

Tuples also have two type-specific callable methods in Python 3.0, but not nearly as many as lists:

```
>>> T.index(4)                  # Tuple methods: 4 appears at offset 3
3
>>> T.count(4)                  # 4 appears once
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences:

```
>>> T[0] = 2                    # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
```

Like lists and dictionaries, tuples support mixed types and nesting, but they don't grow and shrink because they are immutable:

```
>>> T = ('spam', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

## Why Tuples?

So, why have a type that is like a list, but supports fewer operations? Frankly, tuples are not generally used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in programs larger than those we'll write here. We'll talk more about tuples later in the book. For now, though, let's jump ahead to our last major core type: the file.

# Files

File objects are Python code's main interface to external files on your computer. Files are a core type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, to create a file object, you call the built-in `open` function, passing in an external filename and a processing mode as strings. For example, to create a text output file, you would pass in its name and the `'w'` processing mode string to write data:

```
>>> f = open('data.txt', 'w')       # Make a new file in output mode
>>> f.write('Hello\n')              # Write strings of bytes to it
6
>>> f.write('world\n')              # Returns number of bytes written in Python 3.0
6
>>> f.close()                       # Close to flush output buffers to disk
```

This creates a file in the current directory and writes text to it (the filename can be a full directory path if you need to access a file elsewhere on your computer). To read back what you just wrote, reopen the file in `'r'` processing mode, for reading text input—this is the default if you omit the mode in the call. Then read the file's content into a string, and display it. A file's contents are always a string in your script, regardless of the type of data the file contains:

```
>>> f = open('data.txt')          # 'r' is the default processing mode
>>> text = f.read()               # Read entire file into a string
>>> text
'Hello\nworld\n'

>>> print(text)                   # print interprets control characters
Hello
world

>>> text.split()                  # File content is always a string
['Hello', 'world']
```

Other file object methods support additional features we don't have time to cover here. For instance, file objects provide more ways of reading and writing (`read` accepts an optional byte size, `readline` reads one line at a time, and so on), as well as other tools (`seek` moves to a new file position). As we'll see later, though, the best way to read a file today is to *not read it at all*—files provide an *iterator* that automatically reads line by line in `for` loops and other contexts.

We'll meet the full set of file methods later in this book, but if you want a quick preview now, run a `dir` call on any open file and a `help` on any of the method names that come back:

```
>>> dir(f)
[ ...many names omitted...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
'writelines']

>>>help(f.seek)
...try it and see...
```

Later in the book, we'll also see that files in Python 3.0 draw a sharp distinction between text and binary data. *Text files* represent content as strings and perform Unicode encoding and decoding automatically, while *binary files* represent content as a special `bytes` string type and allow you to access file content unaltered:

```
>>> data = open('data.bin', 'rb').read()      # Open binary file
>>> data                                      # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]
b'spam'
```

Although you won't generally need to care about this distinction if you deal only with ASCII text, Python 3.0's strings and files are an asset if you deal with internationalized applications or byte-oriented data.

### Other File-Like Tools

The `open` function is the workhorse for most file processing you will do in Python. For more advanced tasks, though, Python comes with additional file-like tools: pipes, FIFOs, sockets, keyed-access files, persistent object shelves, descriptor-based files, relational and object-oriented database interfaces, and more. Descriptor files, for instance, support file locking and other low-level tools, and sockets provide an interface for networking and interprocess communication. We won't cover many of these topics in this book, but you'll find them useful once you start programming Python in earnest.

## Other Core Types

Beyond the core types we've seen so far, there are others that may or may not qualify for membership in the set, depending on how broadly it is defined. *Sets*, for example, are a recent addition to the language that are neither mappings nor sequences; rather, they are unordered collections of unique and immutable objects. Sets are created by calling the built-in `set` function or using new set literals and expressions in 3.0, and they support the usual mathematical set operations (the choice of new `{...}` syntax for set literals in 3.0 makes sense, since sets are much like the keys of a valueless dictionary):

```
>>> X = set('spam')                # Make a set out of a sequence in 2.6 and 3.0
>>> Y = {'h', 'a', 'm'}            # Make a set with new 3.0 set literals
>>> X, Y
({'a', 'p', 's', 'm'}, {'a', 'h', 'm'})

>>> X & Y                          # Intersection
{'a', 'm'}

>>> X | Y                          # Union
{'a', 'p', 's', 'h', 'm'}

>>> X - Y                          # Difference
{'p', 's'}

>>> {x ** 2 for x in [1, 2, 3, 4]} # Set comprehensions in 3.0
{16, 1, 4, 9}
```

In addition, Python recently grew a few new numeric types: *decimal* numbers (fixed-precision floating-point numbers) and *fraction* numbers (rational numbers with both a numerator and a denominator). Both can be used to work around the limitations and inherent inaccuracies of floating-point math:

```
>>> 1 / 3                          # Floating-point (use .0 in Python 2.6)
0.33333333333333331
>>> (2/3) + (1/2)
```

```
1.1666666666666665

>>> import decimal                  # Decimals: fixed precision
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')

>>> decimal.getcontext().prec = 2
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')

>>> from fractions import Fraction  # Fractions: numerator+denominator
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)
```

Python also comes with *Booleans* (with predefined `True` and `False` objects that are essentially just the integers 1 and 0 with custom display logic), and it has long supported a special placeholder object called `None` commonly used to initialize names and objects:

```
>>> 1 > 2, 1 < 2                    # Booleans
(False, True)
>>> bool('spam')
True

>>> X = None                        # None placeholder
>>> print(X)
None
>>> L = [None] * 100                # Initialize a list of 100 Nones
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, ...a list of 100 Nones...]
```

## How to Break Your Code's Flexibility

I'll have more to say about all of Python's object types later, but one merits special treatment here. The *type* object, returned by the `type` built-in function, is an object that gives the type of another object; its result differs slightly in 3.0, because types have merged with classes completely (something we'll explore in the context of "new-style" classes in Part VI). Assuming `L` is still the list of the prior section:

```
# In Python 2.6:

>>> type(L)                         # Types: type of L is list type object
<type 'list'>
>>> type(type(L))                   # Even types are objects
<type 'type'>

# In Python 3.0:

>>> type(L)                         # 3.0: types are classes, and vice versa
<class 'list'>
```

```
>>> type(type(L))                # See Chapter 31 for more on class types
<class 'type'>
```

Besides allowing you to explore your objects interactively, the practical application of this is that it allows code to check the types of the objects it processes. In fact, there are at least three ways to do so in a Python script:

```
>>> if type(L) == type([]):      # Type testing, if you must...
        print('yes')

yes
>>> if type(L) == list:          # Using the type name
        print('yes')

yes
>>> if isinstance(L, list):      # Object-oriented tests
        print('yes')

yes
```

Now that I've shown you all these ways to do type testing, however, I am required by law to tell you that doing so is almost always the wrong thing to do in a Python program (and often a sign of an ex-C programmer first starting to use Python!). The reason why won't become completely clear until later in the book, when we start writing larger code units such as functions, but it's a (perhaps *the*) core Python concept. By checking for specific types in your code, you effectively break its flexibility—you limit it to working on just one type. Without such tests, your code may be able to work on a whole range of types.

This is related to the idea of polymorphism mentioned earlier, and it stems from Python's lack of type declarations. As you'll learn, in Python, we code to object *interfaces* (operations supported), not to types. Not caring about specific types means that code is automatically applicable to many of them—any object with a compatible interface will work, regardless of its specific type. Although type checking is supported—and even required, in some rare cases—you'll see that it's not usually the "Pythonic" way of thinking. In fact, you'll find that polymorphism is probably the key idea behind using Python well.

## User-Defined Classes

We'll study *object-oriented programming* in Python—an optional but powerful feature of the language that cuts development time by supporting programming by customization—in depth later in this book. In abstract terms, though, classes define new types of objects that extend the core set, so they merit a passing glance here. Say, for example, that you wish to have a type of object that models employees. Although there is no such specific core type in Python, the following user-defined class might fit the bill:

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # self is the new object
```

```
        self.pay  = pay
    def lastName(self):
        return self.name.split()[-1]          # Split string on blanks
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)           # Update pay in-place
```

This class defines a new kind of object that will have `name` and `pay` attributes (sometimes called *state information*), as well as two bits of behavior coded as functions (normally called *methods*). Calling the class like a function generates instances of our new type, and the class's methods automatically receive the instance being processed by a given method call (in the `self` argument):

```
>>> bob = Worker('Bob Smith', 50000)          # Make two instances
>>> sue = Worker('Sue Jones', 60000)          # Each has name and pay attrs
>>> bob.lastName()                            # Call method: bob is self
'Smith'
>>> sue.lastName()                            # sue is the self subject
'Jones'
>>> sue.giveRaise(.10)                        # Updates sue's pay
>>> sue.pay
66000.0
```

The implied "self" object is why we call this an object-oriented model: there is always an implied subject in functions within a class. In a sense, though, the class-based type simply builds on and uses core types—a user-defined `Worker` object here, for example, is just a collection of a string and a number (`name` and `pay`, respectively), plus functions for processing those two built-in objects.

The larger story of classes is that their inheritance mechanism supports software hierarchies that lend themselves to customization by extension. We extend software by writing new classes, not by changing what already works. You should also know that classes are an optional feature of Python, and simpler built-in types such as lists and dictionaries are often better tools than user-coded classes. This is all well beyond the bounds of our introductory object-type tutorial, though, so consider this just a preview; for full disclosure on user-defined types coded with classes, you'll have to read on to Part VI.

## And Everything Else

As mentioned earlier, everything you can process in a Python script is a type of object, so our object type tour is necessarily incomplete. However, even though everything in Python is an "object," only those types of objects we've met so far are considered part of Python's core type set. Other types in Python either are objects related to program execution (like functions, modules, classes, and compiled code), which we will study later, or are implemented by imported module functions, not language syntax. The latter of these also tend to have application-specific roles—text patterns, database interfaces, network connections, and so on.

Moreover, keep in mind that the objects we've met here are objects, but not necessarily *object-oriented*—a concept that usually requires inheritance and the Python `class` statement, which we'll meet again later in this book. Still, Python's core objects are the workhorses of almost every Python script you're likely to meet, and they usually are the basis of larger noncore types.

## Chapter Summary

And that's a wrap for our concise data type tour. This chapter has offered a brief introduction to Python's core object types and the sorts of operations we can apply to them. We've studied generic operations that work on many object types (sequence operations such as indexing and slicing, for example), as well as type-specific operations available as method calls (for instance, string splits and list appends). We've also defined some key terms, such as immutability, sequences, and polymorphism.

Along the way, we've seen that Python's core object types are more flexible and powerful than what is available in lower-level languages such as C. For instance, Python's lists and dictionaries obviate most of the work you do to support collections and searching in lower-level languages. Lists are ordered collections of other objects, and dictionaries are collections of other objects that are indexed by key instead of by position. Both dictionaries and lists may be nested, can grow and shrink on demand, and may contain objects of any type. Moreover, their space is automatically cleaned up as you go.

I've skipped most of the details here in order to provide a quick tour, so you shouldn't expect all of this chapter to have made sense yet. In the next few chapters, we'll start to dig deeper, filling in details of Python's core object types that were omitted here so you can gain a more complete understanding. We'll start off in the next chapter with an in-depth look at Python numbers. First, though, another quiz to review.

---

## Test Your Knowledge: Quiz

We'll explore the concepts introduced in this chapter in more detail in upcoming chapters, so we'll just cover the big ideas here:

1. Name four of Python's core data types.
2. Why are they called "core" data types?
3. What does "immutable" mean, and which three of Python's core types are considered immutable?
4. What does "sequence" mean, and which three types fall into that category?

5. What does "mapping" mean, and which core type is a mapping?
6. What is "polymorphism," and why should you care?

## Test Your Knowledge: Answers

1. Numbers, strings, lists, dictionaries, tuples, files, and sets are generally considered to be the core object (data) types. Types, `None`, and Booleans are sometimes classified this way as well. There are multiple number types (integer, floating point, complex, fraction, and decimal) and multiple string types (simple strings and Unicode strings in Python 2.X, and text strings and byte strings in Python 3.X).
2. They are known as "core" types because they are part of the Python language itself and are always available; to create other objects, you generally must call functions in imported modules. Most of the core types have specific syntax for generating the objects: `'spam'`, for example, is an expression that makes a string and determines the set of operations that can be applied to it. Because of this, core types are hardwired into Python's syntax. In contrast, you must call the built-in `open` function to create a file object.
3. An "immutable" object is an object that cannot be changed after it is created. Numbers, strings, and tuples in Python fall into this category. While you cannot change an immutable object in-place, you can always make a new one by running an expression.
4. A "sequence" is a positionally ordered collection of objects. Strings, lists, and tuples are all sequences in Python. They share common sequence operations, such as indexing, concatenation, and slicing, but also have type-specific method calls.
5. The term "mapping" denotes an object that maps keys to associated values. Python's dictionary is the only mapping type in the core type set. Mappings do not maintain any left-to-right positional ordering; they support access to data stored by key, plus type-specific method calls.
6. "Polymorphism" means that the meaning of an operation (like a `+`) depends on the objects being operated on. This turns out to be a key idea (perhaps *the* key idea) behind using Python well—not constraining code to specific types makes that code automatically applicable to many types.

CHAPTER 5

# Numeric Types

This chapter begins our in-depth tour of the Python language. In Python, data takes the form of *objects*—either built-in objects that Python provides, or objects we create using Python tools and other languages such as C. In fact, objects are the basis of every Python program you will ever write. Because they are the most fundamental notion in Python programming, objects are also our first focus in this book.

In the preceding chapter, we took a quick pass over Python's core object types. Although essential terms were introduced in that chapter, we avoided covering too many specifics in the interest of space. Here, we'll begin a more careful second look at data type concepts, to fill in details we glossed over earlier. Let's get started by exploring our first data type category: Python's numeric types.

## Numeric Type Basics

Most of Python's number types are fairly typical and will probably seem familiar if you've used almost any other programming language in the past. They can be used to keep track of your bank balance, the distance to Mars, the number of visitors to your website, and just about any other numeric quantity.

In Python, numbers are not really a single object type, but a category of similar types. Python supports the usual numeric types (integers and floating points), as well as literals for creating numbers and expressions for processing them. In addition, Python provides more advanced numeric programming support and objects for more advanced work. A complete inventory of Python's numeric toolbox includes:

- Integers and floating-point numbers
- Complex numbers
- Fixed-precision decimal numbers

- Rational fraction numbers
- Sets
- Booleans
- Unlimited integer precision
- A variety of numeric built-ins and modules

This chapter starts with basic numbers and fundamentals, then moves on to explore the other tools in this list. Before we jump into code, though, the next few sections get us started with a brief overview of how we write and process numbers in our scripts.

## Numeric Literals

Among its basic types, Python provides *integers* (positive and negative whole numbers) and *floating-point* numbers (numbers with a fractional part, sometimes called "floats" for economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited precision (they can grow to have as many digits as your memory space allows). Table 5-1 shows what Python's numeric types look like when written out in a program, as literals.

*Table 5-1. Basic numeric literals*

| Literal | Interpretation |
|---|---|
| `1234, -24, 0, 99999999999999` | Integers (unlimited size) |
| `1.23, 1., 3.14e-10, 4E210, 4.0e+210` | Floating-point numbers |
| `0177, 0x9ff, 0b101010` | Octal, hex, and binary literals in 2.6 |
| `0o177, 0x9ff, 0b101010` | Octal, hex, and binary literals in 3.0 |
| `3+4j, 3.0+4.0j, 3J` | Complex number literals |

In general, Python's numeric type literals are straightforward to write, but a few coding concepts are worth highlighting here:

*Integer and floating-point literals*
: Integers are written as strings of decimal digits. Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an `e` or `E` and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression. Floating-point numbers are implemented as C "doubles," and therefore get as much precision as the C compiler used to build the Python interpreter gives to doubles.

*Integers in Python 2.6: normal and long*

In Python 2.6 there are two integer types, normal (32 bits) and long (unlimited precision), and an integer may end in an `l` or `L` to force it to become a long integer. Because integers are automatically converted to long integers when their values overflow 32 bits, you never need to type the letter `L` yourself—Python automatically converts up to long integer when extra precision is needed.

*Integers in Python 3.0: a single type*

In Python 3.0, the normal and long integer types have been merged—there is only integer, which automatically supports the unlimited precision of Python 2.6's separate long integer type. Because of this, integers can no longer be coded with a trailing `l` or `L`, and integers never print with this character either. Apart from this, most programs are unaffected by this change, unless they do type testing that checks for 2.6 long integers.

*Hexadecimal, octal, and binary literals*

Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2). Hexadecimals start with a leading `0x` or `0X`, followed by a string of hexadecimal digits (`0`–`9` and `A`–`F`). Hex digits may be coded in lower- or uppercase. Octal literals start with a leading `0o` or `0O` (zero and lower- or uppercase letter "o"), followed by a string of digits (`0`–`7`). In 2.6 and earlier, octal literals can also be coded with just a leading `0`, but not in 3.0 (this original octal form is too easily confused with decimal, and is replaced by the new `0o` format). Binary literals, new in 2.6 and 3.0, begin with a leading `0b` or `0B`, followed by binary digits (`0`–`1`).

Note that all of these literals produce integer objects in program code; they are just alternative syntaxes for specifying values. The built-in calls `hex(`*`I`*`)`, `oct(`*`I`*`)`, and `bin(`*`I`*`)` convert an integer to its representation string in these three bases, and `int(`*`str`*`,` *`base`*`)` converts a runtime string to an integer per a given base.

*Complex numbers*

Python complex literals are written as *`realpart`*`+`*`imaginarypart`*, where the *`imaginarypart`* is terminated with a `j` or `J`. The *`realpart`* is technically optional, so the *`imaginarypart`* may appear on its own. Internally, complex numbers are implemented as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers may also be created with the `complex(`*`real`*`,` *`imag`*`)` built-in call.

*Coding other numeric types*

As we'll see later in this chapter, there are additional, more advanced number types not included in Table 5-1. Some of these are created by calling functions in imported modules (e.g., decimals and fractions), and others have literal syntax all their own (e.g., sets).

## Built-in Numeric Tools

Besides the built-in number literals shown in Table 5-1, Python provides a set of tools for processing number objects:

*Expression operators*
: `+`, `-`, `*`, `/`, `>>`, `**`, `&`, etc.

*Built-in mathematical functions*
: `pow`, `abs`, `round`, `int`, `hex`, `bin`, etc.

*Utility modules*
: `random`, `math`, etc.

We'll meet all of these as we go along.

Although numbers are primarily processed with expressions, built-ins, and modules, they also have a handful of type-specific *methods* today, which we'll meet in this chapter as well. Floating-point numbers, for example, have an `as_integer_ratio` method that is useful for the fraction number type, and an `is_integer` method to test if the number is an integer. Integers have various attributes, including a new `bit_length` method in the upcoming Python 3.1 release that gives the number of bits necessary to represent the object's value. Moreover, as part collection and part number, *sets* also support both methods and expressions.

Since expressions are the most essential tool for most number types, though, let's turn to them next.

## Python Expression Operators

Perhaps the most fundamental tool that processes numbers is the *expression*: a combination of numbers (or other objects) and operators that computes a value when executed by Python. In Python, expressions are written using the usual mathematical notation and operator symbols. For instance, to add two numbers `X` and `Y` you would say `X + Y`, which tells Python to apply the `+` operator to the values named by `X` and `Y`. The result of the expression is the sum of `X` and `Y`, another number object.

Table 5-2 lists all the operator expressions available in Python. Many are self-explanatory; for instance, the usual mathematical operators (`+`, `-`, `*`, `/`, and so on) are supported. A few will be familiar if you've used other languages in the past: `%` computes a division remainder, `<<` performs a bitwise left-shift, `&` computes a bitwise AND result, and so on. Others are more Python-specific, and not all are numeric in nature: for example, the `is` operator tests object identity (i.e., address in memory, a strict form of equality), and `lambda` creates unnamed functions.

*Table 5-2. Python expression operators and precedence*

| Operators | Description |
|---|---|
| `yield x` | Generator function `send` protocol |
| `lambda args: expression` | Anonymous function generation |
| `x if y else z` | Ternary selection (x is evaluated only if y is true) |
| `x or y` | Logical OR (y is evaluated only if x is false) |
| `x and y` | Logical AND (y is evaluated only if x is true) |
| `not x` | Logical negation |
| `x in y, x not in y` | Membership (iterables, sets) |
| `x is y, x is not y` | Object identity tests |
| `x < y, x <= y, x > y, x >= y`<br>`x == y, x != y` | Magnitude comparison, set subset and superset;<br>Value equality operators |
| `x \| y` | Bitwise OR, set union |
| `x ^ y` | Bitwise XOR, set symmetric difference |
| `x & y` | Bitwise AND, set intersection |
| `x << y, x >> y` | Shift x left or right by y bits |
| `x + y`<br>`x - y` | Addition, concatenation;<br>Subtraction, set difference |
| `x * y`<br>`x % y`<br>`x / y, x // y` | Multiplication, repetition;<br>Remainder, format;<br>Division: true and floor |
| `-x, +x` | Negation, identity |
| `~x` | Bitwise NOT (inversion) |
| `x ** y` | Power (exponentiation) |
| `x[i]` | Indexing (sequence, mapping, others) |
| `x[i:j:k]` | Slicing |
| `x(...)` | Call (function, method, class, other callable) |
| `x.attr` | Attribute reference |
| `(...)` | Tuple, expression, generator expression |
| `[...]` | List, list comprehension |
| `{...}` | Dictionary, set, set and dictionary comprehensions |

Since this book addresses both Python 2.6 and 3.0, here are some notes about version differences and recent additions related to the operators in Table 5-2:

- In Python 2.6, value inequality can be written as either `X != Y` or `X <> Y`. In Python 3.0, the latter of these options is removed because it is redundant. In either version, best practice is to use `X != Y` for all value inequality tests.
- In Python 2.6, a backquotes expression `` `X` `` works the same as `repr(X)` and converts objects to display strings. Due to its obscurity, this expression is removed in Python 3.0; use the more readable `str` and `repr` built-in functions, described in "Numeric Display Formats" on page 115.
- The `X // Y` floor division expression always truncates fractional remainders in both Python 2.6 and 3.0. The `X / Y` expression performs true division in 3.0 (retaining remainders) and classic division in 2.6 (truncating for integers). See "Division: Classic, Floor, and True" on page 117.
- The syntax `[...]` is used for both list literals and list comprehension expressions. The latter of these performs an implied loop and collects expression results in a new list. See Chapters 4, 14, and 20 for examples.
- The syntax `(...)` is used for tuples and expressions, as well as generator expressions—a form of list comprehension that produces results on demand, instead of building a result list. See Chapters 4 and 20 for examples. The parentheses may sometimes be omitted in all three constructs.
- The syntax `{...}` is used for dictionary literals, and in Python 3.0 for set literals and both dictionary and set comprehensions. See the set coverage in this chapter and Chapters 4, 8, 14, and 20 for examples.
- The `yield` and ternary `if`/`else` selection expressions are available in Python 2.5 and later. The former returns `send(...)` arguments in generators; the latter is shorthand for a multiline `if` statement. `yield` requires parentheses if not alone on the right side of an assignment statement.
- Comparison operators may be chained: `X < Y < Z` produces the same result as `X < Y and Y < X`. See "Comparisons: Normal and Chained" on page 116 for details.
- In recent Pythons, the slice expression `X[I:J:K]` is equivalent to indexing with a slice object: `X[slice(I, J, K)]`.
- In Python 2.X, magnitude comparisons of mixed types—converting numbers to a common type, and ordering other mixed types according to the type name—are allowed. In Python 3.0, nonnumeric mixed-type magnitude comparisons are not allowed and raise exceptions; this includes sorts by proxy.
- Magnitude comparisons for dictionaries are also no longer supported in Python 3.0 (though equality tests are); comparing `sorted(dict.items())` is one possible replacement.

We'll see most of the operators in Table 5-2 in action later; first, though, we need to take a quick look at the ways these operators may be combined in expressions.

### Mixed operators follow operator precedence

As in most languages, in Python, more complex expressions are coded by stringing together the operator expressions in Table 5-2. For instance, the sum of two multiplications might be written as a mix of variables and operators:

```
A * B + C * D
```

So, how does Python know which operation to perform first? The answer to this question lies in *operator precedence*. When you write an expression with more than one operator, Python groups its parts according to what are called *precedence rules*, and this grouping determines the order in which the expression's parts are computed. Table 5-2 is ordered by operator precedence:

- Operators lower in the table have higher precedence, and so bind more tightly in mixed expressions.
- Operators in the same row in Table 5-2 generally group from left to right when combined (except for exponentiation, which groups right to left, and comparisons, which chain left to right).

For example, if you write `X + Y * Z`, Python evaluates the multiplication first (`Y * Z`), then adds that result to `X` because `*` has higher precedence (is lower in the table) than `+`. Similarly, in this section's original example, both multiplications (`A * B` and `C * D`) will happen before their results are added.

### Parentheses group subexpressions

You can forget about precedence completely if you're careful to group parts of expressions with parentheses. When you enclose subexpressions in parentheses, you override Python's precedence rules; Python always evaluates expressions in parentheses first before using their results in the enclosing expressions.

For instance, instead of coding `X + Y * Z`, you could write one of the following to force Python to evaluate the expression in the desired order:

```
(X + Y) * Z
X + (Y * Z)
```

In the first case, `+` is applied to `X` and `Y` first, because this subexpression is wrapped in parentheses. In the second case, the `*` is performed first (just as if there were no parentheses at all). Generally speaking, adding parentheses in large expressions is a good idea—it not only forces the evaluation order you want, but also aids readability.

### Mixed types are converted up

Besides mixing operators in expressions, you can also mix numeric types. For instance, you can add an integer to a floating-point number:

```
40 + 3.14
```

But this leads to another question: what type is the result—integer or floating-point? The answer is simple, especially if you've used almost any other language before: in mixed-type numeric expressions, Python first converts operands *up* to the type of the most complicated operand, and then performs the math on same-type operands. This behavior is similar to type conversions in the C language.

Python ranks the complexity of numeric types like so: integers are simpler than floating-point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first, and floating-point math yields the floating-point result. Similarly, any mixed-type expression where one operand is a complex number results in the other operand being converted up to a complex number, and the expression yields a complex result. (In Python 2.6, normal integers are also converted to long integers whenever their values are too large to fit in a normal integer; in 3.0, integers subsume longs entirely.)

You can force the issue by calling built-in functions to convert types manually:

```
>>> int(3.1415)       # Truncates float to integer
3
>>> float(3)          # Converts integer to float
3.0
```

However, you won't usually need to do this: because Python automatically converts up to the more complex type within an expression, the results are normally what you want.

Also, keep in mind that all these mixed-type conversions apply only when mixing *numeric* types (e.g., an integer and a floating-point) in an expression, including those using numeric and comparison operators. In general, Python does not convert across any other type boundaries automatically. Adding a string to an integer, for example, results in an error, unless you manually convert one or the other; watch for an example when we meet strings in Chapter 7.

In Python 2.6, nonnumeric mixed types can be compared, but no conversions are performed (mixed types compare according to a fixed but arbitrary rule). In 3.0, nonnumeric mixed-type comparisons are not allowed and raise exceptions.

### Preview: Operator overloading and polymorphism

Although we're focusing on built-in numbers right now, all Python operators may be overloaded (i.e., implemented) by Python classes and C extension types to work on objects you create. For instance, you'll see later that objects coded with classes may be added or concatenated with `+` expressions, indexed with `[i]` expressions, and so on.

Furthermore, Python itself automatically overloads some operators, such that they perform different actions depending on the type of built-in objects being processed.

For example, the + operator performs addition when applied to numbers but performs concatenation when applied to sequence objects such as strings and lists. In fact, + can mean anything at all when applied to objects you define with classes.

As we saw in the prior chapter, this property is usually called *polymorphism*—a term indicating that the meaning of an operation depends on the type of the objects being operated on. We'll revisit this concept when we explore functions in Chapter 16, because it becomes a much more obvious feature in that context.

# Numbers in Action

On to the code! Probably the best way to understand numeric objects and expressions is to see them in action, so let's start up the interactive command line and try some basic but illustrative operations (see Chapter 3 for pointers if you need help starting an interactive session).

## Variables and Basic Expressions

First of all, let's exercise some basic math. In the following interaction, we first assign two *variables* (a and b) to integers so we can use them later in a larger expression. Variables are simply names—created by you or Python—that are used to keep track of information in your program. We'll say more about this in the next chapter, but in Python:

- Variables are created when they are first assigned values.
- Variables are replaced with their values when used in expressions.
- Variables must be assigned before they can be used in expressions.
- Variables refer to objects and are never declared ahead of time.

In other words, these assignments cause the variables a and b to spring into existence automatically:

```
% python
>>> a = 3                  # Name created
>>> b = 4
```

I've also used a *comment* here. Recall that in Python code, text after a # mark and continuing to the end of the line is considered to be a comment and is ignored. Comments are a way to write human-readable documentation for your code. Because code you type interactively is temporary, you won't normally write comments in this context, but I've added them to some of this book's examples to help explain the code.* In the next part of the book, we'll meet a related feature—documentation strings—that attaches the text of your comments to objects.

* If you're working along, you don't need to type any of the comment text from the # through to the end of the line; comments are simply ignored by Python and not required parts of the statements we're running.

Now, let's use our new integer objects in some expressions. At this point, the values of `a` and `b` are still `3` and `4`, respectively. Variables like these are replaced with their values whenever they're used inside an expression, and the expression results are echoed back immediately when working interactively:

```
>>> a + 1, a - 1          # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2          # Multiplication (4 * 3), division (4 / 2)
(12, 2.0)
>>> a % 2, b ** 2         # Modulus (remainder), power (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b     # Mixed-type conversions
(6.0, 16.0)
```

Technically, the results being echoed back here are *tuples* of two values because the lines typed at the prompt contain two expressions separated by commas; that's why the results are displayed in parentheses (more on tuples later). Note that the expressions work because the variables `a` and `b` within them have been assigned values. If you use a different variable that has never been assigned, Python reports an error rather than filling in some default value:

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'c' is not defined
```

You don't need to predeclare variables in Python, but they must have been assigned at least once before you can use them. In practice, this means you have to initialize counters to zero before you can add to them, initialize lists to an empty list before you can append to them, and so on.

Here are two slightly larger expressions to illustrate operator grouping and more about conversions:

```
>>> b / 2 + a             # Same as ((4 / 2) + 3)
5.0
>>> print(b / (2.0 + a))  # Same as (4 / (2.0 + 3))
0.8
```

In the first expression, there are no parentheses, so Python automatically groups the components according to its precedence rules—because `/` is lower in Table 5-2 than `+`, it binds more tightly and so is evaluated first. The result is as if the expression had been organized with parentheses as shown in the comment to the right of the code.

Also, notice that all the numbers are integers in the first expression. Because of that, Python 2.6 performs integer division and addition and will give a result of 5, whereas Python 3.0 performs true division with remainders and gives the result shown. If you want integer division in 3.0, code this as `b // 2 + a` (more on division in a moment).

In the second expression, parentheses are added around the `+` part to force Python to evaluate it first (i.e., before the `/`). We also made one of the operands floating-point by adding a decimal point: `2.0`. Because of the mixed types, Python converts the integer

referenced by a to a floating-point value (3.0) before performing the +. If all the numbers in this expression were integers, integer division (4 / 5) would yield the truncated integer 0 in Python 2.6 but the floating-point 0.8 in Python 3.0 (again, stay tuned for division details).

## Numeric Display Formats

Notice that we used a print operation in the last of the preceding examples. Without the print, you'll see something that may look a bit odd at first glance:

```
>>> b / (2.0 + a)          # Auto echo output: more digits
0.80000000000000004

>>> print(b / (2.0 + a))   # print rounds off digits
0.8
```

The full story behind this odd result has to do with the limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits. Because computer architecture is well beyond this book's scope, though, we'll finesse this by saying that all of the digits in the first output are really there in your computer's floating-point hardware—it's just that you're not accustomed to seeing them. In fact, this is really just a display issue—the interactive prompt's automatic result echo shows more digits than the print statement. If you don't want to see all the digits, use print; as the sidebar "str and repr Display Formats" on page 116 will explain, you'll get a user-friendly display.

Note, however, that not all values have so many digits to display:

```
>>> 1 / 2.0
0.5
```

and that there are more ways to display the bits of a number inside your computer than using print and automatic echoes:

```
>>> num = 1 / 3.0
>>> num                    # Echoes
0.33333333333333331
>>> print(num)             # print rounds
0.333333333333

>>> '%e' % num             # String formatting expression
'3.333333e-001'
>>> '%4.2f' % num          # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num) # String formatting method (Python 2.6 and 3.0)
'0.33'
```

The last three of these expressions employ *string formatting*, a tool that allows for format flexibility, which we will explore in the upcoming chapter on strings (Chapter 7). Its results are strings that are typically printed to displays or reports.

### str and repr Display Formats

Technically, the difference between default interactive echoes and `print` corresponds to the difference between the built-in `repr` and `str` functions:

```
>>> num = 1 / 3
>>> repr(num)               # Used by echoes: as-code form
'0.3333333333333331'
>>> str(num)                # Used by print: user-friendly form
'0.333333333333'
```

Both of these convert arbitrary objects to their string representations: `repr` (and the default interactive echo) produces results that look as though they were code; `str` (and the `print` operation) converts to a typically more user-friendly format if available. Some objects have both—a `str` for general use, and a `repr` with extra details. This notion will resurface when we study both strings and operator overloading in classes, and you'll find more on these built-ins in general later in the book.

Besides providing print strings for arbitrary objects, the `str` built-in is also the name of the string data type and may be called with an encoding name to decode a Unicode string from a byte string. We'll study the latter advanced role in Chapter 36 of this book.

## Comparisons: Normal and Chained

So far, we've been dealing with standard numeric operations (addition and multiplication), but numbers can also be compared. Normal comparisons work for numbers exactly as you'd expect—they compare the relative magnitudes of their operands and return a Boolean result (which we would normally test in a larger statement):

```
>>> 1 < 2                   # Less than
True
>>> 2.0 >= 1                # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0              # Equal value
True
>>> 2.0 != 2.0              # Not equal value
False
```

Notice again how mixed types are allowed in numeric expressions (only); in the second test here, Python compares values in terms of the more complex type, float.

Interestingly, Python also allows us to *chain* multiple comparisons together to perform range tests. Chained comparisons are a sort of shorthand for larger Boolean expressions. In short, Python lets us string together magnitude comparison tests to code chained comparisons such as range tests. The expression `(A < B < C)`, for instance, tests whether `B` is between `A` and `C`; it is equivalent to the Boolean test `(A < B and B < C)` but is easier on the eyes (and the keyboard). For example, assume the following assignments:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

The following two expressions have identical effects, but the first is shorter to type, and it may run slightly faster since Python needs to evaluate `Y` only once:

```
>>> X < Y < Z                 # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

The same equivalence holds for false results, and arbitrary chain lengths are allowed:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False

>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

You can use other comparisons in chained tests, but the resulting expressions can become nonintuitive unless you evaluate them the way Python does. The following, for instance, is false just because 1 is not equal to 2:

```
>>> 1 == 2 < 3          # Same as: 1 == 2 and 2 < 3
False                   # Not same as: False < 3 (which means 0 < 3, which is true)
```

Python does not compare the `1 == 2` `False` result to 3—this would technically mean the same as `0 < 3`, which would be `True` (as we'll see later in this chapter, `True` and `False` are just customized 1 and 0).

## Division: Classic, Floor, and True

You've seen how division works in the previous sections, so you should know that it behaves slightly differently in Python 3.0 and 2.6. In fact, there are actually three flavors of division, and two different division operators, one of which changes in 3.0:

`X / Y`
: *Classic* and *true* division. In Python 2.6 and earlier, this operator performs *classic* division, truncating results for integers and keeping remainders for floating-point numbers. In Python 3.0, it performs *true* division, always keeping remainders regardless of types.

`X // Y`
: *Floor* division. Added in Python 2.2 and available in both Python 2.6 and 3.0, this operator always truncates fractional remainders down to their floor, regardless of types.

True division was added to address the fact that the results of the original classic division model are dependent on operand types, and so can be difficult to anticipate in a dynamically typed language like Python. Classic division was removed in 3.0 because of this constraint—the / and // operators implement true and floor division in 3.0.

In sum:

- *In 3.0*, the / now always performs *true* division, returning a float result that includes any remainder, regardless of operand types. The // performs *floor* division, which truncates the remainder and returns an integer for integer operands or a float if any operand is a float.
- *In 2.6*, the / does *classic* division, performing truncating integer division if both operands are integers and float division (keeping remainders) otherwise. The // does *floor* division and works as it does in 3.0, performing truncating division for integers and floor division for floats.

Here are the two operators at work in 3.0 and 2.6:

```
C:\misc> C:\Python30\python
>>>
>>> 10 / 4              # Differs in 3.0: keeps remainder
2.5
>>> 10 // 4             # Same in 3.0: truncates remainder
2
>>> 10 / 4.0            # Same in 3.0: keeps remainder
2.5
>>> 10 // 4.0           # Same in 3.0: truncates to floor
2.0

C:\misc> C:\Python26\python
>>>
>>> 10 / 4
2
>>> 10 // 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4.0
2.0
```

Notice that the data type of the result for // is still dependent on the operand types in 3.0: if either is a float, the result is a float; otherwise, it is an integer. Although this may seem similar to the type-dependent behavior of / in 2.X that motivated its change in 3.0, the type of the return value is much less critical than differences in the return value itself. Moreover, because // was provided in part as a backward-compatibility tool for programs that rely on truncating integer division (and this is more common than you might expect), it must return integers for integers.

### Supporting either Python

Although / behavior differs in 2.6 and 3.0, you can still support both versions in your code. If your programs depend on truncating integer division, use // in both 2.6 and 3.0. If your programs require floating-point results with remainders for integers, use `float` to guarantee that one operand is a float around a / when run in 2.6:

```
X = Y // Z          # Always truncates, always an int result for ints in 2.6 and 3.0

X = Y / float(Z)    # Guarantees float division with remainder in either 2.6 or 3.0
```

Alternatively, you can enable 3.0 / division in 2.6 with a `__future__` import, rather than forcing it with `float` conversions:

```
C:\misc> C:\Python26\python
>>> from __future__ import division          # Enable 3.0 "/" behavior
>>> 10 / 4
2.5
>>> 10 // 4
2
```

### Floor versus truncation

One subtlety: the // operator is generally referred to as *truncating* division, but it's more accurate to refer to it as *floor* division—it truncates the result down to its floor, which means the closest whole number below the true result. The net effect is to round down, not strictly truncate, and this matters for negatives. You can see the difference for yourself with the Python `math` module (modules must be imported before you can use their contents; more on this later):

```
>>> import math
>>> math.floor(2.5)
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)
2
>>> math.trunc(-2.5)
-2
```

When running division operators, you only really truncate for positive results, since truncation is the same as floor; for negatives, it's a floor result (really, they are both floor, but floor is the same as truncation for positives). Here's the case for 3.0:

```
C:\misc> c:\python30\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)

>>> 5 // 2, 5 // -2          # Truncates to floor: rounds to first lower integer
(2, -3)                      # 2.5 becomes 2, -2.5 becomes -3

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
```

```
>>> 5 // 2.0, 5 // -2.0          # Ditto for floats, though result is float too
(2.0, -3.0)
```

The 2.6 case is similar, but / results differ again:

```
C:\misc> c:\python26\python
>>> 5 / 2, 5 / -2               # Differs in 3.0
(2, -3)

>>> 5 // 2, 5 // -2             # This and the rest are the same in 2.6 and 3.0
(2, -3)

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)
```

If you really want truncation regardless of sign, you can always run a float division result through `math.trunc`, regardless of Python version (also see the `round` built-in for related functionality):

```
C:\misc> c:\python30\python
>>> import math
>>> 5 / -2                      # Keep remainder
-2.5
>>> 5 // -2                     # Floor below result
-3
>>> math.trunc(5 / -2)          # Truncate instead of floor
-2

C:\misc> c:\python26\python
>>> import math
>>> 5 / float(-2)               # Remainder in 2.6
-2.5
>>> 5 / -2, 5 // -2             # Floor in 2.6
(-3, -3)
>>> math.trunc(5 / float(-2))   # Truncate in 2.6
-2
```

#### Why does truncation matter?

If you are using 3.0, here is the short story on division operators for reference:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)      # 3.0 true division
(2.5, 2.5, -2.5, -2.5)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)  # 3.0 floor division
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)      # Both
(3.0, 3.0, 3, 3.0)
```

For 2.6 readers, division works as follows:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)      # 2.6 classic division
(2, 2.5, -2.5, -3)
```

```
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)      # 2.6 floor division (same)
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)          # Both
(3, 3.0, 3, 3.0)
```

Although results have yet to come in, it's possible that the nontruncating behavior of / in 3.0 may break a significant number of programs. Perhaps because of a C language legacy, many programmers rely on division truncation for integers and will have to learn to use // in such contexts instead. Watch for a simple prime number `while` loop example in Chapter 13, and a corresponding exercise at the end of Part IV that illustrates the sort of code that may be impacted by this / change. Also stay tuned for more on the special `from` command used in this section; it's discussed further in Chapter 24.

## Integer Precision

Division may differ slightly across Python releases, but it's still fairly standard. Here's something a bit more exotic. As mentioned earlier, Python 3.0 integers support unlimited size:

```
>>> 99999999999999999999999999999 + 1
100000000000000000000000000000
```

Python 2.6 has a separate type for long integers, but it automatically converts any number too large to store in a normal integer to this type. Hence, you don't need to code any special syntax to use longs, and the only way you can tell that you're using 2.6 longs is that they print with a trailing "L":

```
>>> 99999999999999999999999999999 + 1
100000000000000000000000000000L
```

Unlimited-precision integers are a convenient built-in tool. For instance, you can use them to count the U.S. national debt in pennies in Python directly (if you are so inclined, and have enough memory on your computer for this year's budget!). They are also why we were able to raise 2 to such large powers in the examples in Chapter 3. Here are the 3.0 and 2.6 cases:

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376

>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Because Python must do extra work to support their extended precision, integer math is usually substantially slower than normal when numbers grow large. However, if you need the precision, the fact that it's built in for you to use will likely outweigh its performance penalty.

## Complex Numbers

Although less widely used than the types we've been exploring thus far, complex numbers are a distinct core object type in Python. If you know what they are, you know why they are useful; if not, consider this section optional reading.

Complex numbers are represented as two floating-point numbers—the real and imaginary parts—and are coded by adding a `j` or `J` suffix to the imaginary part. We can also write complex numbers with a nonzero real part by adding the two parts with a `+`. For example, the complex number with a real part of `2` and an imaginary part of `-3` is written `2 + -3j`. Here are some examples of complex math at work:

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

Complex numbers also allow us to extract their parts as attributes, support all the usual mathematical expressions, and may be processed with tools in the standard `cmath` module (the complex version of the standard `math` module). Complex numbers typically find roles in engineering-oriented programs. Because they are advanced tools, check Python's language reference manual for additional details.

## Hexadecimal, Octal, and Binary Notation

As described earlier in this chapter, Python integers can be coded in hexadecimal, octal, and binary notation, in addition to the normal base 10 decimal coding. The coding rules were laid out at the start of this chapter; let's look at some live examples here.

Keep in mind that these literals are simply an alternative syntax for specifying the value of an integer object. For example, the following literals coded in Python 3.0 or 2.6 produce normal integers with the specified values in all three bases:

```
>>> 0o1, 0o20, 0o377          # Octal literals
(1, 16, 255)
>>> 0x01, 0x10, 0xFF          # Hex literals
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111  # Binary literals
(1, 16, 255)
```

Here, the octal value `0o377`, the hex value `0xFF`, and the binary value `0b11111111` are all decimal `255`. Python prints in decimal (base 10) by default but provides built-in functions that allow you to convert integers to other bases' digit strings:

```
>>> oct(64), hex(64), bin(64)
('0100', '0x40', '0b1000000')
```

The `oct` function converts decimal to octal, `hex` to hexadecimal, and `bin` to binary. To go the other way, the built-in `int` function converts a string of digits to an integer, and an optional second argument lets you specify the numeric base:

```
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)

>>> int('0x40', 16), int('0b1000000', 2)      # Literals okay too
(64, 64)
```

The `eval` function, which you'll meet later in this book, treats strings as though they were Python code. Therefore, it has a similar effect (but usually runs more slowly—it actually compiles and runs the string as a piece of a program, and it assumes you can trust the source of the string being run; a clever user might be able to submit a string that deletes files on your machine!):

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Finally, you can also convert integers to octal and hexadecimal strings with *string formatting* method calls and expressions:

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)
'100, 40, 1000000'

>>> '%o, %x, %X' % (64, 255, 255)
'100, ff, FF'
```

String formatting is covered in more detail in Chapter 7.

Two notes before moving on. First, Python 2.6 users should remember that you can code octals with simply a leading zero, the original octal format in Python:

```
>>> 0o1, 0o20, 0o377       # New octal format in 2.6 (same as 3.0)
(1, 16, 255)
>>> 01, 020, 0377          # Old octal literals in 2.6 (and earlier)
(1, 16, 255)
```

In 3.0, the syntax in the second of these examples generates an error. Even though it's not an error in 2.6, be careful not to begin a string of digits with a leading zero unless you really mean to code an octal value. Python 2.6 will treat it as base 8, which may not work as you'd expect—`010` is always decimal 8 in 2.6, not decimal 10 (despite what you may or may not think!). This, along with symmetry with the hex and binary forms, is why the octal format was changed in 3.0—you must use `0o010` in 3.0, and probably should in 2.6.

Secondly, note that these literals can produce arbitrarily long integers. The following, for instance, creates an integer with hex notation and then displays it first in decimal and then in octal and binary with converters:

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
>>> X
519229685853482762853049632922009 5L
>>> oct(X)
```

```
'017777777777777777777777777777777777L'
>>> bin(X)
'0b11111111111111111111111111111111111111111111111111111111 ...and so on...
```

Speaking of binary digits, the next section shows tools for processing individual bits.

## Bitwise Operations

Besides the normal numeric operations (addition, subtraction, and so on), Python supports most of the numeric expressions available in the C language. This includes operators that treat integers as strings of binary bits. For instance, here it is at work performing bitwise shift and Boolean operations:

```
>>> x = 1                 # 0001
>>> x << 2                # Shift left 2 bits: 0100
4
>>> x | 2                 # Bitwise OR: 0011
3
>>> x & 1                 # Bitwise AND: 0001
1
```

In the first expression, a binary `1` (in base 2, `0001`) is shifted left two slots to create a binary `4` (`0100`). The last two operations perform a binary OR (`0001|0010` = `0011`) and a binary AND (`0001&0001` = `0001`). Such bit-masking operations allow us to encode multiple flags and other values within a single integer.

This is one area where the binary and hexadecimal number support in Python 2.6 and 3.0 become especially useful—they allow us to code and inspect numbers by bit-strings:

```
>>> X = 0b0001            # Binary literals
>>> X << 2                # Shift left
4
>>> bin(X << 2)           # Binary digits string
'0b100'

>>> bin(X | 0b010)        # Bitwise OR
'0b11'
>>> bin(X & 0b1)          # Bitwise AND
'0b1'

>>> X = 0xFF              # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010        # Bitwise XOR
85
>>> bin(X ^ 0b10101010)
'0b1010101'

>>> int('1010101', 2)     # String to int per base
85
>>> hex(85)               # Hex digit string
'0x55'
```

We won't go into much more detail on "bit-twiddling" here. It's supported if you need it, and it comes in handy if your Python code must deal with things like network packets or packed binary data produced by a C program. Be aware, though, that bitwise operations are often not as important in a high-level language such as Python as they are in a low-level language such as C. As a rule of thumb, if you find yourself wanting to flip bits in Python, you should think about which language you're really coding. In general, there are often better ways to encode information in Python than bit strings.

In the upcoming Python 3.1 release, the integer `bit_length` method also allows you to query the number of bits required to represent a number's value in binary. The same effect can often be achieved by subtracting 2 from the length of the `bin` string using the `len` built-in function we met in Chapter 4, though it may be less efficient:

```
>>> X = 99
>>> bin(X), X.bit_length()
('0b1100011', 7)
>>> bin(256), (256).bit_length()
('0b100000000', 9)
>>> len(bin(256)) - 2
9
```

## Other Built-in Numeric Tools

In addition to its core object types, Python also provides both built-in functions and standard library modules for numeric processing. The `pow` and `abs` built-in functions, for instance, compute powers and absolute values, respectively. Here are some examples of the built-in `math` module (which contains most of the tools in the C language's math library) and a few built-in functions at work:

```
>>> import math
>>> math.pi, math.e                          # Common constants
(3.1415926535897931, 2.7182818284590451)

>>> math.sin(2 * math.pi / 180)              # Sine, tangent, cosine
0.034899496702500969

>>> math.sqrt(144), math.sqrt(2)             # Square root
(12.0, 1.4142135623730951)

>>> pow(2, 4), 2 ** 4                        # Exponentiation (power)
(16, 16)

>>> abs(-42.0), sum((1, 2, 3, 4))            # Absolute value, summation
(42.0, 10)

>>> min(3, 1, 2, 4), max(3, 1, 2, 4)         # Minimum, maximum
(1, 4)
```

The `sum` function shown here works on a sequence of numbers, and `min` and `max` accept either a sequence or individual arguments. There are a variety of ways to drop the

decimal digits of floating-point numbers. We met truncation and floor earlier; we can also round, both numerically and for display purposes:

```
>>> math.floor(2.567), math.floor(-2.567)          # Floor (next-lower integer)
(2, -3)

>>> math.trunc(2.567), math.trunc(-2.567)          # Truncate (drop decimal digits)
(2, -2)

>>> int(2.567), int(-2.567)                        # Truncate (integer conversion)
(2, -2)

>>> round(2.567), round(2.467), round(2.567, 2)    # Round (Python 3.0 version)
(3, 2, 2.5699999999999998)

>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)        # Round for display (Chapter 7)
('2.6', '2.57')
```

As we saw earlier, the last of these produces strings that we would usually print and supports a variety of formatting options. As also described earlier, the second to last test here will output `(3, 2, 2.57)` if we wrap it in a `print` call to request a more user-friendly display. The last two lines still differ, though—`round` rounds a floating-point number but still yields a floating-point number in memory, whereas string formatting produces a string and doesn't yield a modified number:

```
>>> (1 / 3), round(1 / 3, 2), ('%.2f' % (1 / 3))
(0.33333333333333331, 0.33000000000000002, '0.33')
```

Interestingly, there are three ways to compute *square roots* in Python: using a module function, an expression, or a built-in function (if you're interested in performance, we will revisit these in an exercise and its solution at the end of Part IV, to see which runs quicker):

```
>>> import math
>>> math.sqrt(144)              # Module
12.0
>>> 144 ** .5                   # Expression
12.0
>>> pow(144, .5)                # Built-in
12.0

>>> math.sqrt(1234567890)       # Larger numbers
35136.418286444619
>>> 1234567890 ** .5
35136.418286444619
>>> pow(1234567890, .5)
35136.418286444619
```

Notice that standard library modules such as `math` must be imported, but built-in functions such as `abs` and `round` are always available without imports. In other words, modules are external components, but built-in functions live in an implied namespace that Python automatically searches to find names used in your program. This namespace corresponds to the module called `builtins` in Python 3.0 (`__builtin__` in 2.6). There

is much more about name resolution in the function and module parts of this book; for now, when you hear "module," think "import."

The standard library `random` module must be imported as well. This module provides tools for picking a random floating-point number between 0 and 1, selecting a random integer between two numbers, choosing an item at random from a sequence, and more:

```
>>> import random
>>> random.random()
0.44694718823781876
>>> random.random()
0.28970426439292829

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
4

>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
```

The `random` module can be useful for shuffling cards in games, picking images at random in a slideshow GUI, performing statistical simulations, and much more. For more details, see Python's library manual.

# Other Numeric Types

So far in this chapter, we've been using Python's core numeric types—integer, floating point, and complex. These will suffice for most of the number crunching that most programmers will ever need to do. Python comes with a handful of more exotic numeric types, though, that merit a quick look here.

## Decimal Type

Python 2.4 introduced a new core numeric type: the decimal object, formally known as `Decimal`. Syntactically, decimals are created by calling a function within an imported module, rather than running a literal expression. Functionally, decimals are like floating-point numbers, but they have a fixed number of decimal points. Hence, decimals are *fixed-precision* floating-point values.

For example, with decimals, we can have a floating-point value that always retains just two decimal digits. Furthermore, we can specify how to round or truncate the extra decimal digits beyond the object's cutoff. Although it generally incurs a small performance penalty compared to the normal floating-point type, the decimal type is well suited to representing fixed-precision quantities like sums of money and can help you achieve better numeric accuracy.

### The basics

The last point merits elaboration. As you may or may not already know, floating-point math is less than exact, because of the limited space used to store values. For example, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

Printing the result to produce the user-friendly display format doesn't completely help either, because the hardware related to floating-point math is inherently limited in terms of accuracy:

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)
5.55111512313e-17
```

However, with decimals, the result can be dead-on:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

As shown here, we can make decimal objects by calling the `Decimal` constructor function in the `decimal` module and passing in strings that have the desired number of decimal digits for the resulting object (we can use the `str` function to convert floating-point values to strings if needed). When decimals of different precision are mixed in expressions, Python converts up to the largest number of decimal digits automatically:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
```

In Python 3.1 (to be released after this book's publication), it's also possible to create a decimal object from a floating-point object, with a call of the form `decimal.Decimal.from_float(1.25)`. The conversion is exact but can sometimes yield a large number of digits.

### Setting precision globally

Other tools in the `decimal` module can be used to set the precision of all decimal numbers, set up error handling, and more. For instance, a context object in this module allows for specifying precision (number of decimal digits) and rounding modes (down, ceiling, etc.). The precision is applied globally for all decimals created in the calling thread:

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.14285714285714285714285714285714')

>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

This is especially useful for monetary applications, where cents are represented as two decimal digits. Decimals are essentially an alternative to manual rounding and string formatting in this context:

```
>>> 1999 + 1.33
2000.3299999999999
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')
```

### Decimal context manager

In Python 2.6 and 3.0 (and later), it's also possible to reset precision temporarily by using the `with` context manager statement. The precision is reset to its original value on statement exit:

```
C:\misc> C:\Python30\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333')
```

Though useful, this statement requires much more background knowledge than you've obtained at this point; watch for coverage of the `with` statement in Chapter 33.

Because use of the decimal type is still relatively rare in practice, I'll defer to Python's standard library manuals and interactive help for more details. And because decimals address some of the same floating-point accuracy issues as the fraction type, let's move on to the next section to see how the two compare.

## Fraction Type

Python 2.6 and 3.0 debut a new numeric type, `Fraction`, which implements a *rational number* object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math.

### The basics

`Fraction` is a sort of cousin to the existing `Decimal` fixed-precision type described in the prior section, as both can be used to control numerical accuracy by fixing decimal digits and specifying rounding or truncation policies. It's also used in similar ways—like

`Decimal`, `Fraction` resides in a module; import its constructor and pass in a numerator and a denominator to make one. The following interaction shows how:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)                  # Numerator, denominator
>>> y = Fraction(4, 6)                  # Simplified to 2, 3 by gcd

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Once created, `Fraction`s can be used in mathematical expressions as usual:

```
>>> x + y
Fraction(1, 1)
>>> x - y                               # Results are exact: numerator, denominator
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

`Fraction` objects can also be created from floating-point number strings, much like decimals:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

### Numeric accuracy

Notice that this is different from floating-point-type math, which is constrained by the underlying limitations of floating-point hardware. To compare, here are the same operations run with floating-point objects, and notes on their limited accuracy:

```
>>> a = 1 / 3.0                         # Only as accurate as floating-point hardware
>>> b = 4 / 6.0                         # Can lose precision over calculations
>>> a
0.33333333333333331
>>> b
0.66666666666666663

>>> a + b
1.0
>>> a - b
-0.33333333333333331
>>> a * b
0.22222222222222221
```

This floating-point limitation is especially apparent for values that cannot be represented accurately given their limited number of bits in memory. Both `Fraction` and

Decimal provide ways to get exact results, albeit at the cost of some speed. For instance, in the following example (repeated from the prior section), floating-point numbers do not accurately give the zero answer expected, but both of the other types do:

```
>>> 0.1 + 0.1 + 0.1 - 0.3           # This should be zero (close, but not exact)
5.5511151231257827e-17

>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Moreover, fractions and decimals both allow more intuitive and accurate results than floating points sometimes can, in different ways (by using rational representation and by limiting precision):

```
>>> 1 / 3                           # Use 3.0 in Python 2.6 for true "/"
0.33333333333333331

>>> Fraction(1, 3)                  # Numeric accuracy
Fraction(1, 3)

>>> import decimal
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal(1) / decimal.Decimal(3)
Decimal('0.33')
```

In fact, fractions both retain accuracy and automatically simplify results. Continuing the preceding interaction:

```
>>> (1 / 3) + (6 / 12)              # Use ".0" in Python 2.6 for true "/"
0.83333333333333326

>>> Fraction(6, 12)                 # Automatically simplified
Fraction(1, 2)

>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)

>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')

>>> 1000.0 / 1234567890
8.1000000737100011e-07
>>> Fraction(1000, 1234567890)
Fraction(100, 123456789)
```

### Conversions and mixed types

To support fraction conversions, floating-point objects now have a method that yields their numerator and denominator ratio, fractions have a from_float method, and

`float` accepts a `Fraction` as an argument. Trace through the following interaction to see how this pans out (the * in the second test is special syntax that expands a tuple into individual arguments; more on this when we study function argument passing in Chapter 18):

```
>>> (2.5).as_integer_ratio()              # float object method
(5, 2)

>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio())   # Convert float -> fraction: two args
>>> z                                     # Same as Fraction(5, 2)
Fraction(5, 2)

>>> x                                     # x from prior interaction
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                           # 5/2 + 1/3 = 15/6 + 2/6

>>> float(x)                              # Convert fraction -> float
0.3333333333333331
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)             # Convert float -> fraction: other way
Fraction(7, 4)
>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)
```

Finally, some type mixing is allowed in expressions, though `Fraction` must sometimes be manually propagated to retain accuracy. Study the following interaction to see how this works:

```
>>> x
Fraction(1, 3)
>>> x + 2                                 # Fraction + int -> Fraction
Fraction(7, 3)
>>> x + 2.0                               # Fraction + float -> float
2.3333333333333335
>>> x + (1./3)                            # Fraction + float -> float
0.6666666666666663

>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)                    # Fraction + Fraction -> Fraction
Fraction(5, 3)
```

Caveat: although you can convert from floating-point to fraction, in some cases there is an unavoidable precision loss when you do so, because the number is inaccurate in its original floating-point form. When needed, you can simplify such results by limiting the maximum denominator value:

```
>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()                # Precision loss from float
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488.      # 5 / 3 (or close to it!)
1.6666666666666667

>>> a.limit_denominator(10)                     # Simplify to closest fraction
Fraction(5, 3)
```

For more details on the `Fraction` type, experiment further on your own and consult the Python 2.6 and 3.0 library manuals and other documentation.

## Sets

Python 2.4 also introduced a new collection type, the *set*—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. By definition, an item appears only once in a set, no matter how many times it is added. As such, sets have a variety of applications, especially in numeric and database-focused work.

Because sets are collections of other objects, they share some behavior with objects such as lists and dictionaries that are outside the scope of this chapter. For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types. As we'll see, a set acts much like the keys of a valueless dictionary, but it supports extra operations.

However, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets are fundamentally mathematical in nature (and for many readers, may seem more academic and be used much less often than more pervasive objects like dictionaries), we'll explore the basic utility of Python's set objects here.

### Set basics in Python 2.6

There are a few ways to make sets today, depending on whether you are using Python 2.6 or 3.0. Since this book covers both, let's begin with the 2.6 case, which also is available (and sometimes still required) in 3.0; we'll refine this for 3.0 extensions in a moment. To make a set object, pass in a sequence or other iterable object to the built-in `set` function:

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

You get back a set object, which contains all the items in the object passed in (notice that sets do not have a positional ordering, and so are not sequences):

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])                # 2.6 display format
```

Sets made this way support the common mathematical set operations with *expression* operators. Note that we can't perform these expressions on plain sequences—we must create sets from them in order to apply these tools:

```
>>> 'e' in x                                  # Membership
True

>>> x - y                                     # Difference
set(['a', 'c', 'e'])

>>> x | y                                     # Union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                                     # Intersection
set(['b', 'd'])

>>> x ^ y                                     # Symmetric difference (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])

>>> x > y, x < y                              # Superset, subset
(False, False)
```

In addition to expressions, the set object provides *methods* that correspond to these operations and more, and that support set changes—the set `add` method inserts one item, `update` is an in-place union, and `remove` deletes an item by value (run a `dir` call on any set instance or the `set` type name to see all the available methods). Assuming `x` and `y` are still as they were in the prior interaction:

```
>>> z = x.intersection(y)                     # Same as x & y
>>> z
set(['b', 'd'])
>>> z.add('SPAM')                             # Insert one item
>>> z
set(['b', 'd', 'SPAM'])
>>> z.update(set(['X', 'Y']))                 # Merge: in-place union
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])
>>> z.remove('b')                             # Delete one item
>>> z
set(['Y', 'X', 'd', 'SPAM'])
```

As *iterable* containers, sets can also be used in operations such as `len`, `for` loops, and list comprehensions. Because they are unordered, though, they don't support sequence operations like indexing and slicing:

```
>>> for item in set('abc'): print(item * 3)
...
aaa
```

```
ccc
bbb
```

Finally, although the set expressions shown earlier generally require two sets, their method-based counterparts can often work with *any iterable type* as well:

```
>>> S = set([1, 2, 3])

>>> S | set([3, 4])          # Expressions require both to be sets
set([1, 2, 3, 4])
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> S.union([3, 4])          # But their methods allow any iterable
set([1, 2, 3, 4])
>>> S.intersection((1, 3, 5))
set([1, 3])
>>> S.issubset(range(-5, 5))
True
```

For more details on set operations, see Python's library reference manual or a reference book. Although set operations can be coded manually in Python with other types, like lists and dictionaries (and often were in the past), Python's built-in sets use efficient algorithms and implementation techniques to provide quick and standard operation.

### Set literals in Python 3.0

If you think sets are "cool," they recently became noticeably cooler. In Python 3.0 we can still use the `set` built-in to make set objects, but 3.0 also adds a new set literal form, using the curly braces formerly reserved for dictionaries. In 3.0, the following are equivalent:

```
set([1, 2, 3, 4])          # Built-in call
{1, 2, 3, 4}               # 3.0 set literals
```

This syntax makes sense, given that sets are essentially like *valueless dictionaries*—because they are unordered, unique, and immutable, a set's items behave much like a dictionary's keys. This operational similarity is even more striking given that dictionary key lists in 3.0 are *view* objects, which support set-like behavior such as intersections and unions (see Chapter 8 for more on dictionary view objects).

In fact, regardless of how a set is made, 3.0 displays it using the new literal format. The `set` built-in is still required in 3.0 to create empty sets and to build sets from existing iterable objects (short of using set comprehensions, discussed later in this chapter), but the new literal is convenient for initializing sets of known structure:

```
C:\Misc> c:\python30\python
>>> set([1, 2, 3, 4])          # Built-in: same as in 2.6
{1, 2, 3, 4}
>>> set('spam')                # Add all items in an iterable
{'a', 'p', 's', 'm'}

>>> {1, 2, 3, 4}               # Set literals: new in 3.0
```

```
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S.add('alot')
>>> S
{'a', 'p', 's', 'm', 'alot'}
```

All the set processing operations discussed in the prior section work the same in 3.0, but the result sets print differently:

```
>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}                  # Intersection
{1, 3}
>>> {1, 5, 3, 6} | S1            # Union
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}               # Difference
{2}
>>> S1 > {1, 3}                  # Superset
True
```

Note that {} is still a dictionary in Python. *Empty* sets must be created with the `set` built-in, and print the same way:

```
>>> S1 - {1, 2, 3, 4}            # Empty sets print differently
set()
>>> type({})                     # Because {} is an empty dictionary
<class 'dict'>

>>> S = set()                    # Initialize an empty set
>>> S.add(1.23)
>>> S
{1.23}
```

As in Python 2.6, sets created with 3.0 literals support the same methods, some of which allow general iterable operands that expressions do not:

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}

>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

#### Immutable constraints and frozen sets

Sets are powerful and flexible objects, but they do have one constraint in both 3.0 and 2.6 that you should keep in mind—largely because of their implementation, sets can

only contain immutable (a.k.a "hashable") object types. Hence, lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values. Tuples compare by their full values when used in set operations:

```
>>> S
{1.23}
>>> S.add([1, 2, 3])                    # Only mutable objects work in a set
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))
>>> S                                   # No list or dict, but tuple okay
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}         # Union: same as S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S                     # Membership: by complete values
True
>>> (1, 4, 3) in S
False
```

Tuples in a set, for instance, might be used to represent dates, records, IP addresses, and so on (more on tuples later in this part of the book). Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the `frozenset` built-in call works just like `set` but creates an immutable set that cannot change and thus can be embedded in other sets.

### Set comprehensions in Python 3.0

In addition to literals, 3.0 introduces a set comprehension construct; it is similar in form to the list comprehension we previewed in Chapter 4, but is coded in curly braces instead of square brackets and run to make a set instead of a list. Set comprehensions run a loop and collect the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression. The result is a new set created by running the code, with all the normal set behavior:

```
>>> {x ** 2 for x in [1, 2, 3, 4]}      # 3.0 set comprehension
{16, 1, 4, 9}
```

In this expression, the loop is coded on the right, and the collection expression is coded on the left (`x ** 2`). As for list comprehensions, we get back pretty much what this expression says: "Give me a new set containing X squared, for every X in a list." Comprehensions can also iterate across other kinds of objects, such as strings (the first of the following examples illustrates the comprehension-based way to make a set from an existing iterable):

```
>>> {x for x in 'spam'}                 # Same as: set('spam')
{'a', 'p', 's', 'm'}

>>> {c * 4 for c in 'spam'}             # Set of collected expression results
{'ssss', 'aaaa', 'pppp', 'mmmm'}
>>> {c * 4 for c in 'spamham'}
```

```
{'ssss', 'aaaa', 'hhhh', 'pppp', 'mmmm'}

>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmmm', 'xxxx'}
{'ssss', 'aaaa', 'pppp', 'mmmm', 'xxxx'}
>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

Because the rest of the comprehensions story relies upon underlying concepts we're not yet prepared to address, we'll postpone further details until later in this book. In Chapter 8, we'll meet a first cousin in 3.0, the dictionary comprehension, and I'll have much more to say about all comprehensions (list, set, dictionary, and generator) later, especially in Chapters14 and 20. As we'll learn later, all comprehensions, including sets, support additional syntax not shown here, including nested loops and `if` tests, which can be difficult to understand until you've had a chance to study larger statements.

### Why sets?

Set operations have a variety of common uses, some more practical than mathematical. For example, because items are stored only once in a set, sets can be used to filter duplicates out of other collections. Simply convert the collection to a set, and then convert it back again (because sets are iterable, they work in the `list` call here):

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                    # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

Sets can also be used to keep track of where you've already been when traversing a graph or other cyclic structure. For example, the transitive module reloader and inheritance tree lister examples we'll study in Chapters 24 and 30, respectively, must keep track of items visited to avoid loops. Although recording states visited as keys in a dictionary is efficient, sets offer an alternative that's essentially equivalent (and may be more or less intuitive, depending on who you ask).

Finally, sets are also convenient when dealing with large data sets (database query results, for example)—the intersection of two sets contains objects in common to both categories, and the union contains all items in either set. To illustrate, here's a somewhat more realistic example of set operations at work, applied to lists of people in a hypothetical company, using 3.0 set literals (use `set` in 2.6):

```
>>> engineers = {'bob', 'sue', 'ann', 'vic'}
>>> managers  = {'tom', 'sue'}

>>> 'bob' in engineers                  # Is bob an engineer?
True

>>> engineers & managers                # Who is both engineer and manager?
```

```
{'sue'}

>>> engineers | managers                  # All people in either category
{'vic', 'sue', 'tom', 'bob', 'ann'}

>>> engineers - managers                  # Engineers who are not managers
{'vic', 'bob', 'ann'}

>>> managers - engineers                  # Managers who are not engineers
{'tom'}

>>> engineers > managers                  # Are all managers engineers? (superset)
False

>>> {'bob', 'sue'} < engineers            # Are both engineers? (subset)
True

>>> (managers | engineers) > managers     # All people is a superset of managers
True

>>> managers ^ engineers                  # Who is in one but not both?
{'vic', 'bob', 'ann', 'tom'}

>>> (managers | engineers) - (managers ^ engineers)      # Intersection!
{'sue'}
```

You can find more details on set operations in the Python library manual and some mathematical and relational database theory texts. Also stay tuned for Chapter 8's revival of some of the set operations we've seen here, in the context of dictionary view objects in Python 3.0.

## Booleans

Some argue that the Python Boolean type, `bool`, is numeric in nature because its two values, `True` and `False`, are just customized versions of the integers 1 and 0 that print themselves differently. Although that's all most programmers need to know, let's explore this type in a bit more detail.

More formally, Python today has an explicit Boolean data type called `bool`, with the values `True` and `False` available as new preassigned built-in names. Internally, the names `True` and `False` are instances of `bool`, which is in turn just a subclass (in the object-oriented sense) of the built-in integer type `int`. `True` and `False` behave exactly like the integers 1 and 0, except that they have customized printing logic—they print themselves as the words `True` and `False`, instead of the digits `1` and `0`. `bool` accomplishes this by redefining `str` and `repr` string formats for its two objects.

Because of this customization, the output of Boolean expressions typed at the interactive prompt prints as the words `True` and `False` instead of the older and less obvious `1` and `0`. In addition, Booleans make truth values more explicit. For instance, an infinite loop can now be coded as `while True:` instead of the less intuitive `while 1:`. Similarly,

flags can be initialized more clearly with `flag = False`. We'll discuss these statements further in Part III.

Again, though, for all other practical purposes, you can treat `True` and `False` as though they are predefined variables set to integer `1` and `0`. Most programmers used to preassign `True` and `False` to `1` and `0` anyway; the `bool` type simply makes this standard. Its implementation can lead to curious results, though. Because `True` is just the integer `1` with a custom display format, `True + 4` yields `5` in Python:

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1               # Same value
True
>>> True is 1               # But different object: see the next chapter
False
>>> True or False           # Same as: 1 or 0
True
>>> True + 4                # (Hmmm)
5
```

Since you probably won't come across an expression like the last of these in real Python code, you can safely ignore its deeper metaphysical implications....

We'll revisit Booleans in Chapter 9 (to define Python's notion of truth) and again in Chapter 12 (to see how Boolean operators like `and` and `or` work).

## Numeric Extensions

Finally, although Python core numeric types offer plenty of power for most applications, there is a large library of third-party open source extensions available to address more focused needs. Because numeric programming is a popular domain for Python, you'll find a wealth of advanced tools.

For example, if you need to do serious number crunching, an optional extension for Python called *NumPy* (Numeric Python) provides advanced numeric programming tools, such as a matrix data type, vector processing, and sophisticated computation libraries. Hardcore scientific programming groups at places like Los Alamos and NASA use Python with NumPy to implement the sorts of tasks they previously coded in C++, FORTRAN, or Matlab. The combination of Python and NumPy is often compared to a free, more flexible version of Matlab—you get NumPy's performance, plus the Python language and its libraries.

Because it's so advanced, we won't talk further about NumPy in this book. You can find additional support for advanced numeric programming in Python, including graphics and plotting tools, statistics libraries, and the popular *SciPy* package at Python's PyPI site, or by searching the Web. Also note that NumPy is currently an optional extension; it doesn't come with Python and must be installed separately.

## Chapter Summary

This chapter has taken a tour of Python's numeric object types and the operations we can apply to them. Along the way, we met the standard integer and floating-point types, as well as some more exotic and less commonly used types such as complex numbers, fractions, and sets. We also explored Python's expression syntax, type conversions, bitwise operations, and various literal forms for coding numbers in scripts.

Later in this part of the book, I'll fill in some details about the next object type, the string. In the next chapter, however, we'll take some time to explore the mechanics of variable assignment in more detail than we have here. This turns out to be perhaps the most fundamental idea in Python, so make sure you check out the next chapter before moving on. First, though, it's time to take the usual chapter quiz.

---

## Test Your Knowledge: Quiz

1. What is the value of the expression `2 * (3 + 4)` in Python?
2. What is the value of the expression `2 * 3 + 4` in Python?
3. What is the value of the expression `2 + 3 * 4` in Python?
4. What tools can you use to find a number's square root, as well as its square?
5. What is the type of the result of the expression `1 + 2.0 + 3`?
6. How can you truncate and round a floating-point number?
7. How can you convert an integer to a floating-point number?
8. How would you display an integer in octal, hexadecimal, or binary notation?
9. How might you convert an octal, hexadecimal, or binary string to a plain integer?

## Test Your Knowledge: Answers

1. The value will be `14`, the result of 2 * 7, because the parentheses force the addition to happen before the multiplication.
2. The value will be `10`, the result of 6 + 4. Python's operator precedence rules are applied in the absence of parentheses, and multiplication has higher precedence than (i.e., happens before) addition, per Table 5-2.
3. This expression yields `14`, the result of 2 + 12, for the same precedence reasons as in the prior question.
4. Functions for obtaining the square root, as well as *pi*, tangents, and more, are available in the imported `math` module. To find a number's square root, import `math` and call `math.sqrt(N)`. To get a number's square, use either the exponent

expression `X ** 2` or the built-in function `pow(X, 2)`. Either of these last two can also compute the square root when given a power of `0.5` (e.g., `X ** .5`).

5. The result will be a floating-point number: the integers are converted up to floating point, the most complex type in the expression, and floating-point math is used to evaluate it.
6. The `int(N)` and `math.trunc(N)` functions truncate, and the `round(N, digits)` function rounds. We can also compute the floor with `math.floor(N)` and round for display with string formatting operations.
7. The `float(I)` function converts an integer to a floating point; mixing an integer with a floating point within an expression will result in a conversion as well. In some sense, Python 3.0 `/` division converts too—it always returns a floating-point result that includes the remainder, even if both operands are integers.
8. The `oct(I)` and `hex(I)` built-in functions return the octal and hexadecimal string forms for an integer. The `bin(I)` call also returns a number's binary digits string in Python 2.6 and 3.0. The `%` string formatting expression and `format` string method also provide targets for some such conversions.
9. The `int(S, base)` function can be used to convert from octal and hexadecimal strings to normal integers (pass in `8`, `16`, or `2` for the base). The `eval(S)` function can be used for this purpose too, but it's more expensive to run and can have security issues. Note that integers are always stored in binary in computer memory; these are just display string format conversions.

CHAPTER 6

# The Dynamic Typing Interlude

In the prior chapter, we began exploring Python's core object types in depth with a look at Python numbers. We'll resume our object type tour in the next chapter, but before we move on, it's important that you get a handle on what may be the most fundamental idea in Python programming and is certainly the basis of much of both the conciseness and flexibility of the Python language—dynamic typing, and the polymorphism it yields.

As you'll see here and later in this book, in Python, we do not declare the specific types of the objects our scripts use. In fact, programs should not even care about specific types; in exchange, they are naturally applicable in more contexts than we can sometimes even plan ahead for. Because dynamic typing is the root of this flexibility, let's take a brief look at the model here.

## The Case of the Missing Declaration Statements

If you have a background in compiled or statically typed languages like C, C++, or Java, you might find yourself a bit perplexed at this point in the book. So far, we've been using variables without declaring their existence or their types, and it somehow works. When we type `a = 3` in an interactive session or program file, for instance, how does Python know that `a` should stand for an integer? For that matter, how does Python know what `a` is at all?

Once you start asking such questions, you've crossed over into the domain of Python's *dynamic typing* model. In Python, types are determined automatically at runtime, not in response to declarations in your code. This means that you never declare variables ahead of time (a concept that is perhaps simpler to grasp if you keep in mind that it all boils down to variables, objects, and the links between them).

## Variables, Objects, and References

As you've seen in many of the examples used so far in this book, when you run an assignment statement such as `a = 3` in Python, it works even if you've never told Python to use the name `a` as a variable, or that `a` should stand for an integer-type object. In the Python language, this all pans out in a very natural way, as follows:

*Variable creation*
: A variable (i.e., name), like `a`, is created when your code first assigns it a value. Future assignments change the value of the already created name. Technically, Python detects some names before your code runs, but you can think of it as though initial assignments make variables.

*Variable types*
: A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.

*Variable use*
: When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be. Further, all variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

In sum, variables are created when assigned, can reference any type of object, and must be assigned before they are referenced. This means that you never need to declare names used by your script, but you must initialize names before you can update them; counters, for example, must be initialized to zero before you can add to them.

This dynamic typing model is strikingly different from the typing model of traditional languages. When you are first starting out, the model is usually easier to understand if you keep clear the distinction between names and objects. For example, when we say this:

```
>>> a = 3
```

at least conceptually, Python will perform three distinct steps to carry out the request. These steps reflect the operation of all assignments in the Python language:

1. Create an object to represent the value `3`.
2. Create the variable `a`, if it does not yet exist.
3. Link the variable `a` to the new object `3`.

The net result will be a structure inside Python that resembles Figure 6-1. As sketched, variables and objects are stored in different parts of memory and are associated by links (the link is shown as a pointer in the figure). Variables always link to objects and never to other variables, but larger objects may link to other objects (for instance, a list object has links to the objects it contains).

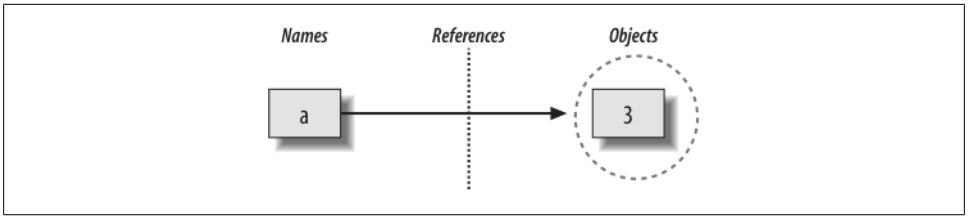


*Figure 6-1. Names and objects after running the assignment a = 3. Variable a becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.*

These links from variables to objects are called *references* in Python—that is, a reference is a kind of association, implemented as a pointer in memory.* Whenever the variables are later used (i.e., referenced), Python automatically follows the variable-to-object links. This is all simpler than the terminology may imply. In concrete terms:

- *Variables* are entries in a system table, with spaces for links to objects.
- *Objects* are pieces of allocated memory, with enough space to represent the values for which they stand.
- *References* are automatically followed pointers from variables to objects.

At least conceptually, each time you generate a new value in your script by running an expression, Python creates a new object (i.e., a chunk of memory) to represent that value. Internally, as an optimization, Python caches and reuses certain kinds of unchangeable objects, such as small integers and strings (each `0` is not really a new piece of memory—more on this caching behavior later). But, from a logical perspective, it works as though each expression's result value is a distinct object and each object is a distinct piece of memory.

Technically speaking, objects have more structure than just enough space to represent their values. Each object also has two standard header fields: a *type designator* used to mark the type of the object, and a *reference counter* used to determine when it's OK to reclaim the object. To understand how these two header fields factor into the model, we need to move on.

## Types Live with Objects, Not Variables

To see how object types come into play, watch what happens if we assign a variable multiple times:

* Readers with a background in C may find Python references similar to C pointers (memory addresses). In fact, references are implemented as pointers, and they often serve the same roles, especially with objects that can be changed in-place (more on this later). However, because references are always automatically dereferenced when used, you can never actually do anything useful with a reference itself; this is a feature that eliminates a vast category of C bugs. You can think of Python references as C "void*" pointers, which are automatically followed whenever used.

```
>>> a = 3                # It's an integer
>>> a = 'spam'           # Now it's a string
>>> a = 1.23             # Now it's a floating point
```

This isn't typical Python code, but it does work—a starts out as an integer, then becomes a string, and finally becomes a floating-point number. This example tends to look especially odd to ex-C programmers, as it appears as though the *type* of a changes from integer to string when we say `a = 'spam'`.

However, that's not really what's happening. In Python, things work more simply. Names have no types; as stated earlier, types live with objects, not names. In the preceding listing, we've simply changed a to reference different objects. Because variables have no type, we haven't actually changed the type of the variable a; we've simply made the variable reference a different type of object. In fact, again, all we can ever say about a variable in Python is that it references a particular object at a particular point in time.

Objects, on the other hand, know what type they are—each object contains a header field that tags the object with its type. The integer object 3, for example, will contain the value 3, plus a designator that tells Python that the object is an integer (strictly speaking, a pointer to an object called int, the name of the integer type). The type designator of the 'spam' string object points to the string type (called str) instead. Because objects know their types, variables don't have to.

To recap, types are associated with objects in Python, not with variables. In typical code, a given variable usually will reference just one kind of object. Because this isn't a requirement, though, you'll find that Python code tends to be much more flexible than you may be accustomed to—if you use Python well, your code might work on many types automatically.

I mentioned that objects have two header fields, a type designator and a reference counter. To understand the latter of these, we need to move on and take a brief look at what happens at the end of an object's life.

## Objects Are Garbage-Collected

In the prior section's listings, we assigned the variable a to different types of objects in each assignment. But when we reassign a variable, what happens to the value it was previously referencing? For example, after the following statements, what happens to the object 3?

```
>>> a = 3
>>> a = 'spam'
```

The answer is that in Python, whenever a name is assigned to a new object, the space held by the prior object is reclaimed (if it is not referenced by any other name or object). This automatic reclamation of objects' space is known as *garbage collection*.

To illustrate, consider the following example, which sets the name x to a different object on each assignment:

```
>>> x = 42
>>> x = 'shrubbery'          # Reclaim 42 now (unless referenced elsewhere)
>>> x = 3.1415               # Reclaim 'shrubbery' now
>>> x = [1, 2, 3]            # Reclaim 3.1415 now
```

First, notice that `x` is set to a different type of object each time. Again, though this is not really the case, the effect is as though the type of `x` is changing over time. Remember, in Python types live with objects, not names. Because names are just generic references to objects, this sort of code works naturally.

Second, notice that references to objects are discarded along the way. Each time `x` is assigned to a new object, Python reclaims the prior object's space. For instance, when it is assigned the string `'shrubbery'`, the object `42` is immediately reclaimed (assuming it is not referenced anywhere else)—that is, the object's space is automatically thrown back into the free space pool, to be reused for a future object.

Internally, Python accomplishes this feat by keeping a counter in every object that keeps track of the number of references currently pointing to that object. As soon as (and exactly when) this counter drops to zero, the object's memory space is automatically reclaimed. In the preceding listing, we're assuming that each time `x` is assigned to a new object, the prior object's reference counter drops to zero, causing it to be reclaimed.

The most immediately tangible benefit of garbage collection is that it means you can use objects liberally without ever needing to free up space in your script. Python will clean up unused space for you as your program runs. In practice, this eliminates a substantial amount of bookkeeping code required in lower-level languages such as C and C++.

Technically speaking, Python's garbage collection is based mainly upon *reference counters*, as described here; however, it also has a component that detects and reclaims objects with *cyclic references* in time. This component can be disabled if you're sure that your code doesn't create cycles, but it is enabled by default.

Because references are implemented as pointers, it's possible for an object to reference itself, or reference another object that does. For example, exercise 3 at the end of Part I and its solution in Appendix B show how to create a cycle by embedding a reference to a list within itself. The same phenomenon can occur for assignments to attributes of objects created from user-defined classes. Though relatively rare, because the reference counts for such objects never drop to zero, they must be treated specially.

For more details on Python's cycle detector, see the documentation for the `gc` module in Python's library manual. Also note that this description of Python's garbage collector applies to the standard CPython only; Jython and IronPython may use different schemes, though the net effect in all is similar—unused space is reclaimed for you automatically.

## Shared References

So far, we've seen what happens as a single variable is assigned references to objects. Now let's introduce another variable into our interaction and watch what happens to its names and objects:

```
>>> a = 3
>>> b = a
```

Typing these two statements generates the scene captured in Figure 6-2. The second line causes Python to create the variable `b`; the variable `a` is being used and not assigned here, so it is replaced with the object it references (`3`), and `b` is made to reference that object. The net effect is that the variables `a` and `b` wind up referencing the same object (that is, pointing to the same chunk of memory). This scenario, with multiple names referencing the same object, is called a *shared reference* in Python.

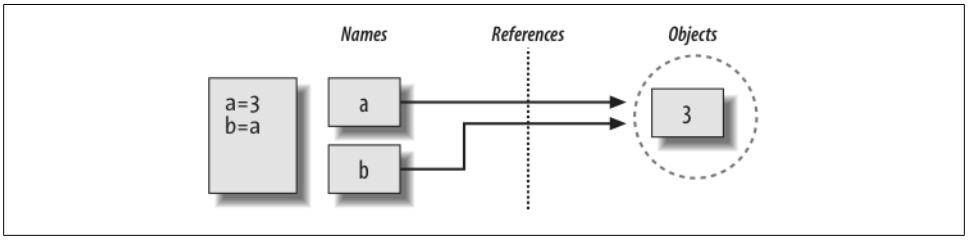


*Figure 6-2. Names and objects after next running the assignment b = a. Variable b becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.*

Next, suppose we extend the session with one more statement:

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

As with all Python assignments, this statement simply makes a new object to represent the string value `'spam'` and sets `a` to reference this new object. It does not, however, change the value of `b`; `b` still references the original object, the integer `3`. The resulting reference structure is shown in Figure 6-3.

The same sort of thing would happen if we changed `b` to `'spam'` instead—the assignment would change only `b`, not `a`. This behavior also occurs if there are no type differences at all. For example, consider these three statements:

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

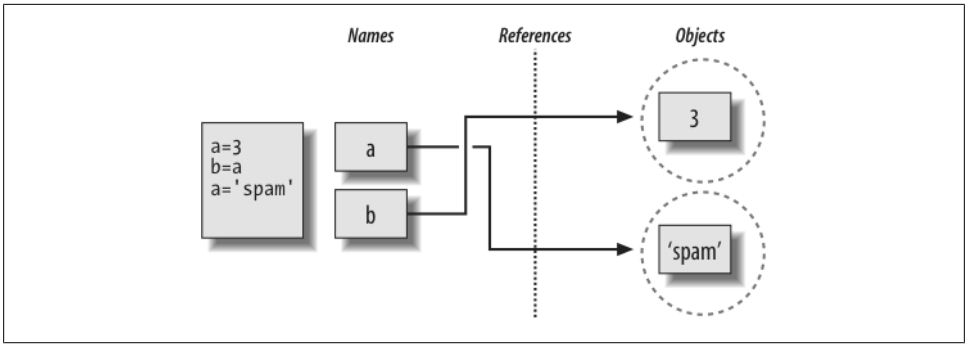


*Figure 6-3. Names and objects after finally running the assignment a = 'spam'. Variable a references the new object (i.e., piece of memory) created by running the literal expression 'spam', but variable b still refers to the original object 3. Because this assignment is not an in-place change to the object 3, it changes only variable a, not b.*

In this sequence, the same events transpire. Python makes the variable `a` reference the object `3` and makes `b` reference the same object as `a`, as in Figure 6-2; as before, the last assignment then sets `a` to a completely different object (in this case, the integer `5`, which is the result of the `+` expression). It does not change `b` as a side effect. In fact, there is no way to ever overwrite the value of the object `3`—as introduced in Chapter 4, integers are immutable and thus can never be changed in-place.

One way to think of this is that, unlike in some languages, in Python variables are always pointers to objects, not labels of changeable memory areas: setting a variable to a new value does not alter the original object, but rather causes the variable to reference an entirely different object. The net effect is that assignment to a variable can impact only the single variable being assigned. When mutable objects and in-place changes enter the equation, though, the picture changes somewhat; to see how, let's move on.

## Shared References and In-Place Changes

As you'll see later in this part's chapters, there are objects and operations that perform in-place object changes. For instance, an assignment to an offset in a list actually changes the list object itself in-place, rather than generating a brand new list object. For objects that support such in-place changes, you need to be more aware of shared references, since a change from one name may impact others.

To further illustrate, let's take another look at the list objects introduced in Chapter 4. Recall that lists, which do support in-place assignments to positions, are simply collections of other objects, coded in square brackets:

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

L1 here is a list containing the objects 2, 3, and 4. Items inside a list are accessed by their positions, so L1[0] refers to object 2, the first item in the list L1. Of course, lists are also objects in their own right, just like integers and strings. After running the two prior assignments, L1 and L2 reference the same object, just like a and b in the prior example (see Figure 6-2). Now say that, as before, we extend this interaction to say the following:

```
>>> L1 = 24
```

This assignment simply sets L1 is to a different object; L2 still references the original list. If we change this statement's syntax slightly, however, it has a radically different effect:

```
>>> L1 = [2, 3, 4]          # A mutable object
>>> L2 = L1                 # Make a reference to the same object
>>> L1[0] = 24              # An in-place change

>>> L1                      # L1 is different
[24, 3, 4]
>>> L2                      # But so is L2!
[24, 3, 4]
```

Really, we haven't changed L1 itself here; we've changed a component of the *object* that L1 references. This sort of change overwrites part of the list object in-place. Because the list object is shared by (referenced from) other variables, though, an in-place change like this doesn't only affect L1—that is, you must be aware that when you make such changes, they can impact other parts of your program. In this example, the effect shows up in L2 as well because it references the same object as L1. Again, we haven't actually changed L2, either, but its value will appear different because it has been overwritten.

This behavior is usually what you want, but you should be aware of how it works, so that it's expected. It's also just the default: if you don't want such behavior, you can request that Python *copy* objects instead of making references. There are a variety of ways to copy a list, including using the built-in `list` function and the standard library `copy` module. Perhaps the most common way is to slice from start to finish (see Chapters 4 and 7 for more on slicing):

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]              # Make a copy of L1
>>> L1[0] = 24

>>> L1
[24, 3, 4]
>>> L2                      # L2 is not changed
[2, 3, 4]
```

Here, the change made through L1 is not reflected in L2 because L2 references a copy of the object L1 references; that is, the two variables point to different pieces of memory.

Note that this slicing technique won't work on the other major mutable core types, dictionaries and sets, because they are not sequences—to copy a dictionary or set, instead use their `X.copy()` method call. Also, note that the standard library `copy` module has a call for copying any object type generically, as well as a call for copying nested object structures (a dictionary with nested lists, for example):

```
import copy
X = copy.copy(Y)          # Make top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)      # Make deep copy of any object Y: copy all nested parts
```

We'll explore lists and dictionaries in more depth, and revisit the concept of shared references and copies, in Chapters 8 and 9. For now, keep in mind that objects that can be changed in-place (that is, mutable objects) are always open to these kinds of effects. In Python, this includes lists, dictionaries, and some objects defined with `class` statements. If this is not the desired behavior, you can simply copy your objects as needed.

## Shared References and Equality

In the interest of full disclosure, I should point out that the garbage-collection behavior described earlier in this chapter may be more conceptual than literal for certain types. Consider these statements:

```
>>> x = 42
>>> x = 'shrubbery'       # Reclaim 42 now?
```

Because Python caches and reuses small integers and small strings, as mentioned earlier, the object `42` here is probably not literally reclaimed; instead, it will likely remain in a system table to be reused the next time you generate a `42` in your code. Most kinds of objects, though, are reclaimed immediately when they are no longer referenced; for those that are not, the caching mechanism is irrelevant to your code.

For instance, because of Python's reference model, there are two different ways to check for equality in a Python program. Let's create a shared reference to demonstrate:

```
>>> L = [1, 2, 3]
>>> M = L                 # M and L reference the same object
>>> L == M                # Same value
True
>>> L is M                # Same object
True
```

The first technique here, the `==` operator, tests whether the two referenced objects have the same values; this is the method almost always used for equality checks in Python. The second method, the `is` operator, instead tests for object identity—it returns `True` only if both names point to the exact same object, so it is a much stronger form of equality testing.

Really, `is` simply compares the pointers that implement references, and it serves as a way to detect shared references in your code if needed. It returns `False` if the names point to equivalent but different objects, as is the case when we run two different literal expressions:

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]          # M and L reference different objects
>>> L == M                 # Same values
True
>>> L is M                 # Different objects
False
```

Now, watch what happens when we perform the same operations on small numbers:

```
>>> X = 42
>>> Y = 42                 # Should be two different objects
>>> X == Y
True
>>> X is Y                 # Same object anyhow: caching at work!
True
```

In this interaction, `X` and `Y` should be `==` (same value), but not `is` (same object) because we ran two different literal expressions. Because small integers and strings are cached and reused, though, `is` tells us they reference the same single object.

In fact, if you really want to look under the hood, you can always ask Python how many references there are to an object: the `getrefcount` function in the standard `sys` module returns the object's reference count. When I ask about the integer object `1` in the IDLE GUI, for instance, it reports 837 reuses of this same object (most of which are in IDLE's system code, not mine):

```
>>> import sys
>>> sys.getrefcount(1)     # 837 pointers to this shared piece of memory
837
```

This object caching and reuse is irrelevant to your code (unless you run the `is` check!). Because you cannot change numbers or strings in-place, it doesn't matter how many references there are to the same object. Still, this behavior reflects one of the many ways Python optimizes its model for execution speed.

## Dynamic Typing Is Everywhere

Of course, you don't really need to draw name/object diagrams with circles and arrows to use Python. When you're starting out, though, it sometimes helps you understand unusual cases if you can trace their reference structures. If a mutable object changes out from under you when passed around your program, for example, chances are you are witnessing some of this chapter's subject matter firsthand.

Moreover, even if dynamic typing seems a little abstract at this point, you probably will care about it eventually. Because *everything* seems to work by assignment and references in Python, a basic understanding of this model is useful in many different

contexts. As you'll see, it works the same in assignment statements, function arguments, `for` loop variables, module imports, class attributes, and more. The good news is that there is just one assignment model in Python; once you get a handle on dynamic typing, you'll find that it works the same everywhere in the language.

At the most practical level, dynamic typing means there is less code for you to write. Just as importantly, though, dynamic typing is also the root of Python's *polymorphism*, a concept we introduced in Chapter 4 and will revisit again later in this book. Because we do not constrain types in Python code, it is highly flexible. As you'll see, when used well, dynamic typing and the polymorphism it provides produce code that automatically adapts to new requirements as your systems evolve.

## Chapter Summary

This chapter took a deeper look at Python's dynamic typing model—that is, the way that Python keeps track of object types for us automatically, rather than requiring us to code declaration statements in our scripts. Along the way, we learned how variables and objects are associated by references in Python; we also explored the idea of garbage collection, learned how shared references to objects can affect multiple variables, and saw how references impact the notion of equality in Python.

Because there is just one assignment model in Python, and because assignment pops up everywhere in the language, it's important that you have a handle on the model before moving on. The following quiz should help you review some of this chapter's ideas. After that, we'll resume our object tour in the next chapter, with strings.

---

## Test Your Knowledge: Quiz

1. Consider the following three statements. Do they change the value printed for `A`?

```
A = "spam"
B = A
B = "shrubbery"
```

2. Consider these three statements. Do they change the printed value of `A`?

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```

3. How about these—is `A` changed now?

```
A = ["spam"]
B = A[:]
B[0] = "shrubbery"
```

## Test Your Knowledge: Answers

1. No: `A` still prints as `"spam"`. When `B` is assigned to the string `"shrubbery"`, all that happens is that the variable `B` is reset to point to the new string object. `A` and `B` initially share (i.e., reference/point to) the same single string object `"spam"`, but two names are never linked together in Python. Thus, setting `B` to a different object has no effect on `A`. The same would be true if the last statement here was `B = B + 'shrubbery'`, by the way—the concatenation would make a new object for its result, which would then be assigned to `B` only. We can never overwrite a string (or number, or tuple) in-place, because strings are immutable.
2. Yes: `A` now prints as `["shrubbery"]`. Technically, we haven't really changed either `A` or `B`; instead, we've changed part of the object they both reference (point to) by overwriting that object in-place through the variable `B`. Because `A` references the same object as `B`, the update is reflected in `A` as well.
3. No: `A` still prints as `["spam"]`. The in-place assignment through `B` has no effect this time because the slice expression made a copy of the list object before it was assigned to `B`. After the second assignment statement, there are two different list objects that have the same value (in Python, we say they are `==`, but not `is`). The third statement changes the value of the list object pointed to by `B`, but not that pointed to by `A`.

# CHAPTER 7
# Strings

The next major type on our built-in object tour is the Python *string*—an ordered collection of characters used to store and represent text-based information. We looked briefly at strings in Chapter 4. Here, we will revisit them in more depth, filling in some of the details we skipped then.

From a functional perspective, strings can be used to represent just about anything that can be encoded as text: symbols and words (e.g., your name), contents of text files loaded into memory, Internet addresses, Python programs, and so on. They can also be used to hold the absolute binary values of bytes, and multibyte Unicode text used in internationalized programs.

You may have used strings in other languages, too. Python's strings serve the same role as character arrays in languages such as C, but they are a somewhat higher-level tool than arrays. Unlike in C, in Python, strings come with a powerful set of processing tools. Also unlike languages such as C, Python has no distinct type for individual characters; instead, you just use one-character strings.

Strictly speaking, Python strings are categorized as immutable sequences, meaning that the characters they contain have a left-to-right positional order and that they cannot be changed in-place. In fact, strings are the first representative of the larger class of objects called *sequences* that we will study here. Pay special attention to the sequence operations introduced in this chapter, because they will work the same on other sequence types we'll explore later, such as lists and tuples.

Table 7-1 previews common string literals and operations we will discuss in this chapter. Empty strings are written as a pair of quotation marks (single or double) with nothing in between, and there are a variety of ways to code strings. For processing, strings support *expression* operations such as concatenation (combining strings), slicing (extracting sections), indexing (fetching by offset), and so on. Besides expressions, Python also provides a set of string *methods* that implement common string-specific tasks, as well as *modules* for more advanced text-processing tasks such as pattern matching. We'll explore all of these later in the chapter.

*Table 7-1. Common string literals and operations*

| Operation | Interpretation |
|---|---|
| `S = ''` | Empty string |
| `S = "spam's"` | Double quotes, same as single |
| `S = 's\np\ta\x00m'` | Escape sequences |
| `S = """..."""` | Triple-quoted block strings |
| `S = r'\temp\spam'` | Raw strings |
| `S = b'spam'` | Byte strings in 3.0 (Chapter 36) |
| `S = u'spam'` | Unicode strings in 2.6 only (Chapter 36) |
| `S1 + S2` | Concatenate, repeat |
| `S * 3` | |
| `S[i]` | Index, slice, length |
| `S[i:j]` | |
| `len(S)` | |
| `"a %s parrot" % kind` | String formatting expression |
| `"a {0} parrot".format(kind)` | String formatting method in 2.6 and 3.0 |
| `S.find('pa')` | String method calls: search, |
| `S.rstrip()` | remove whitespace, |
| `S.replace('pa', 'xx')` | replacement, |
| `S.split(',')` | split on delimiter, |
| `S.isdigit()` | content test, |
| `S.lower()` | case conversion, |
| `S.endswith('spam')` | end test, |
| `'spam'.join(strlist)` | delimiter join, |
| `S.encode('latin-1')` | Unicode encoding, etc. |
| `for x in S: print(x)` | Iteration, membership |
| `'spam' in S` | |
| `[c * 2 for c in S]` | |
| `map(ord, S)` | |

Beyond the core set of string tools in Table 7-1, Python also supports more advanced pattern-based string processing with the standard library's `re` (regular expression) module, introduced in Chapter 4, and even higher-level text processing tools such as XML parsers, discussed briefly in Chapter 36. This book's scope, though, is focused on the fundamentals represented by Table 7-1.

To cover the basics, this chapter begins with an overview of string literal forms and string expressions, then moves on to look at more advanced tools such as string methods and formatting. Python comes with many string tools, and we won't look at them all here; the complete story is chronicled in the Python library manual. Our goal here is to explore enough commonly used tools to give you a representative sample; methods we won't see in action here, for example, are largely analogous to those we will.

*Content note*: Technically speaking, this chapter tells only part of the string story in Python—the part most programmers need to know. It presents the basic `str` string type, which handles ASCII text and works the same regardless of which version of Python you use. That is, this chapter intentionally limits its scope to the string processing essentials that are used in most Python scripts.

From a more formal perspective, ASCII is a simple form of Unicode text. Python addresses the distinction between text and binary data by including distinct object types:

- In Python 3.0 there are three string types: `str` is used for Unicode text (ASCII or otherwise), `bytes` is used for binary data (including encoded text), and `bytearray` is a mutable variant of `bytes`.
- In Python 2.6, `unicode` strings represent wide Unicode text, and `str` strings handle both 8-bit text and binary data.

The `bytearray` type is also available as a back-port in 2.6, but not earlier, and it's not as closely bound to binary data as it is in 3.0. Because most programmers don't need to dig into the details of Unicode encodings or binary data formats, though, I've moved all such details to the Advanced Topics part of this book, in Chapter 36.

If you do need to deal with more advanced string concepts such as alternative character sets or packed binary data and files, see Chapter 36 after reading the material here. For now, we'll focus on the basic string type and its operations. As you'll find, the basics we'll study here also apply directly to the more advanced string types in Python's toolset.

# String Literals

By and large, strings are fairly easy to use in Python. Perhaps the most complicated thing about them is that there are so many ways to write them in your code:

- Single quotes: `'spa"m'`
- Double quotes: `"spa'm"`
- Triple quotes: `'''... spam ...'''`, `"""... spam ..."""`
- Escape sequences: `"s\tp\na\0m"`
- Raw strings: `r"C:\new\test.spm"`

- Byte strings in 3.0 (see Chapter 36): `b'sp\x01am'`
- Unicode strings in 2.6 only (see Chapter 36): `u'eggs\u0020spam'`

The single- and double-quoted forms are by far the most common; the others serve specialized roles, and we're postponing discussion of the last two advanced forms until Chapter 36. Let's take a quick look at all the other options in turn.

## Single- and Double-Quoted Strings Are the Same

Around Python strings, single and double quote characters are interchangeable. That is, string literals can be written enclosed in either two single or two double quotes—the two forms work the same and return the same type of object. For example, the following two strings are identical, once coded:

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
```

The reason for supporting both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash. You may embed a single quote character in a string enclosed in double quote characters, and vice versa:

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

Incidentally, Python automatically concatenates adjacent string literals in any expression, although it is almost as simple to add a `+` operator between them to invoke concatenation explicitly (as we'll see in Chapter 12, wrapping this form in parentheses also allows it to span multiple lines):

```
>>> title = "Meaning " 'of' " Life"          # Implicit concatenation
>>> title
'Meaning of Life'
```

Notice that adding commas between these strings would result in a tuple, not a string. Also notice in all of these outputs that Python prefers to print strings in single quotes, unless they embed one. You can also embed quotes by escaping them with backslashes:

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

To understand why, you need to know how escapes work in general.

## Escape Sequences Represent Special Bytes

The last example embedded a quote inside a string by preceding it with a backslash. This is representative of a general pattern in strings: backslashes are used to introduce special byte codings known as *escape sequences*.

Escape sequences let us embed byte codes in strings that cannot easily be typed on a keyboard. The character \, and one or more characters following it in the string literal, are replaced with a single character in the resulting string object, which has the binary

value specified by the escape sequence. For example, here is a five-character string that embeds a newline and a tab:

```
>>> s = 'a\nb\tc'
```

The two characters `\n` stand for a single character—the byte containing the binary value of the newline character in your character set (usually, ASCII code 10). Similarly, the sequence `\t` is replaced with the tab character. The way this string looks when printed depends on how you print it. The interactive echo shows the special characters as escapes, but `print` interprets them instead:

```
>>> s
'a\nb\tc'
>>> print(s)
a
b       c
```

To be completely sure how many bytes are in this string, use the built-in `len` function—it returns the actual number of bytes in a string, regardless of how it is displayed:

```
>>> len(s)
5
```

This string is five bytes long: it contains an ASCII *a* byte, a newline byte, an ASCII *b* byte, and so on. Note that the original backslash characters are not really stored with the string in memory; they are used to tell Python to store special byte values in the string. For coding such special bytes, Python recognizes a full set of escape code sequences, listed in Table 7-2.

*Table 7-2. String backslash characters*

| Escape | Meaning |
|---|---|
| `\`*`newline`* | Ignored (continuation line) |
| `\\` | Backslash (stores one \) |
| `\'` | Single quote (stores ') |
| `\"` | Double quote (stores ") |
| `\a` | Bell |
| `\b` | Backspace |
| `\f` | Formfeed |
| `\n` | Newline (linefeed) |
| `\r` | Carriage return |
| `\t` | Horizontal tab |
| `\v` | Vertical tab |
| `\x`*hh* | Character with hex value *hh* (at most 2 digits) |
| `\`*ooo* | Character with octal value *ooo* (up to 3 digits) |
| `\0` | Null: binary 0 character (doesn't end string) |

| Escape | Meaning |
|---|---|
| \N{ id } | Unicode database ID |
| \u*hhhh* | Unicode 16-bit hex |
| \U*hhhhhhhh* | Unicode 32-bit hex[a] |
| *other* | Not an escape (keeps both \ and *other*) |

[a] The \U*hhhh*... escape sequence takes exactly eight hexadecimal digits (*h*); both \u and \U can be used only in Unicode string literals.

Some escape sequences allow you to embed absolute binary values into the bytes of a string. For instance, here's a five-character string that embeds two binary zero bytes (coded as octal escapes of one digit):

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

In Python, the zero (null) byte does not terminate a string the way it typically does in C. Instead, Python keeps both the string's length and text in memory. In fact, no character terminates a string in Python. Here's a string that is all absolute binary escape codes—a binary 1 and 2 (coded in octal), followed by a binary 3 (coded in hexadecimal):

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

Notice that Python displays nonprintable characters in hex, regardless of how they were specified. You can freely combine absolute value escapes and the more symbolic escape types in Table 7-2. The following string contains the characters "spam", a tab and newline, and an absolute zero value byte coded in hex:

```
>>> S = "s\tp\na\x00m"
>>> S
's\tp\na\x00m'
>>> len(S)
7
>>> print(S)
s       p
a m
```

This becomes more important to know when you process binary data files in Python. Because their contents are represented as strings in your scripts, it's OK to process binary files that contain any sorts of binary byte values (more on files in Chapter 9).*

* If you need to care about binary data files, the chief distinction is that you open them in binary mode (using open mode flags with a `b`, such as `'rb'`, `'wb'`, and so on). In Python 3.0, binary file content is a `bytes` string, with an interface similar to that of normal strings; in 2.6, such content is a normal `str` string. See also the standard `struct` module introduced in Chapter 9, which can parse binary data loaded from a file, and the extended coverage of binary files and byte strings in Chapter 36.

Finally, as the last entry in Table 7-2 implies, if Python does not recognize the character after a \ as being a valid escape code, it simply keeps the backslash in the resulting string:

```
>>> x = "C:\py\code"          # Keeps \ literally
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Unless you're able to commit all of Table 7-2 to memory, though, you probably shouldn't rely on this behavior.† To code literal backslashes explicitly such that they are retained in your strings, double them up (\\ is an escape for one \) or use raw strings; the next section shows how.

## Raw Strings Suppress Escapes

As we've seen, escape sequences are handy for embedding special byte codes within strings. Sometimes, though, the special treatment of backslashes for introducing escapes can lead to trouble. It's surprisingly common, for instance, to see Python newcomers in classes trying to open a file with a filename argument that looks something like this:

```
myfile = open('C:\new\text.dat', 'w')
```

thinking that they will open a file called *text.dat* in the directory *C:\new*. The problem here is that `\n` is taken to stand for a newline character, and `\t` is replaced with a tab. In effect, the call tries to open a file named *C:(newline)ew(tab)ext.dat*, with usually less than stellar results.

This is just the sort of thing that raw strings are useful for. If the letter `r` (uppercase or lowercase) appears just before the opening quote of a string, it turns off the escape mechanism. The result is that Python retains your backslashes literally, exactly as you type them. Therefore, to fix the filename problem, just remember to add the letter `r` on Windows:

```
myfile = open(r'C:\new\text.dat', 'w')
```

Alternatively, because two backslashes are really an escape sequence for one backslash, you can keep your backslashes by simply doubling them up:

```
myfile = open('C:\\new\\text.dat', 'w')
```

In fact, Python itself sometimes uses this doubling scheme when it prints strings with embedded backslashes:

```
>>> path = r'C:\new\text.dat'
>>> path                      # Show as Python code
'C:\\new\\text.dat'
>>> print(path)               # User-friendly format
```

† In classes, I've met people who have indeed committed most or all of this table to memory; I'd probably think that was really sick, but for the fact that I'm a member of the set, too.

```
C:\new\text.dat
>>> len(path)                    # String length
15
```

As with numeric representation, the default format at the interactive prompt prints results as if they were code, and therefore escapes backslashes in the output. The `print` statement provides a more user-friendly format that shows that there is actually only one backslash in each spot. To verify this is the case, you can check the result of the built-in `len` function, which returns the number of bytes in the string, independent of display formats. If you count the characters in the `print(path)` output, you'll see that there really is just 1 character per backslash, for a total of 15.

Besides directory paths on Windows, raw strings are also commonly used for regular expressions (text pattern matching, supported with the `re` module introduced in Chapter 4). Also note that Python scripts can usually use *forward* slashes in directory paths on Windows and Unix because Python tries to interpret paths portably (i.e., `'C:/new/text.dat'` works when opening files, too). Raw strings are useful if you code paths using native Windows backslashes, though.

Despite its role, even a raw string cannot end in a single backslash, because the backslash escapes the following quote character—you still must escape the surrounding quote character to embed it in the string. That is, `r"...\"` is not a valid string literal—a raw string cannot end in an odd number of backslashes. If you need to end a raw string with a single backslash, you can use two and slice off the second (`r'1\nb\tc\\'[:-1]`), tack one on manually (`r'1\nb\tc' + '\\'`), or skip the raw string syntax and just double up the backslashes in a normal string (`'1\\nb\\tc\\'`). All three of these forms create the same eight-character string containing three backslashes.

## Triple Quotes Code Multiline Block Strings

So far, you've seen single quotes, double quotes, escapes, and raw strings in action. Python also has a triple-quoted string literal format, sometimes called a *block string*, that is a syntactic convenience for coding multiline text data. This form begins with three quotes (of either the single or double variety), is followed by any number of lines of text, and is closed with the same triple-quote sequence that opened it. Single and double quotes embedded in the string's text may be, but do not have to be, escaped—the string does not end until Python sees three unescaped quotes of the same kind used to start the literal. For example:

```
>>> mantra = """Always look
...  on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```

This string spans three lines (in some interfaces, the interactive prompt changes to `...` on continuation lines; IDLE simply drops down one line). Python collects all the triple-quoted text into a single multiline string, with embedded newline characters (`\n`) at the places where your code has line breaks. Notice that, as in the literal, the second line in the result has a leading space, but the third does not—what you type is truly what you get. To see the string with the newlines interpreted, print it instead of echoing:

```
>>> print(mantra)
Always look
 on the bright
side of life.
```

Triple-quoted strings are useful any time you need multiline text in your program; for example, to embed multiline error messages or HTML or XML code in your source code files. You can embed such blocks directly in your scripts without resorting to external text files or explicit concatenation and newline characters.

Triple-quoted strings are also commonly used for documentation strings, which are string literals that are taken as comments when they appear at specific points in your file (more on these later in the book). These don't have to be triple-quoted blocks, but they usually are to allow for multiline comments.

Finally, triple-quoted strings are also sometimes used as a "horribly hackish" way to temporarily disable lines of code during development (OK, it's not really too horrible, and it's actually a fairly common practice). If you wish to turn off a few lines of code and run your script again, simply put three quotes above and below them, like this:

```
X = 1
"""
import os                               # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
```

I said this was hackish because Python really does make a string out of the lines of code disabled this way, but this is probably not significant in terms of performance. For large sections of code, it's also easier than manually adding hash marks before each line and later removing them. This is especially true if you are using a text editor that does not have support for editing Python code specifically. In Python, practicality often beats aesthetics.

# Strings in Action

Once you've created a string with the literal expressions we just met, you will almost certainly want to do things with it. This section and the next two demonstrate string expressions, methods, and formatting—the first line of text-processing tools in the Python language.

## Basic Operations

Let's begin by interacting with the Python interpreter to illustrate the basic string operations listed earlier in Table 7-1. Strings can be concatenated using the + operator and repeated using the * operator:

```
% python
>>> len('abc')              # Length: number of items
3
>>> 'abc' + 'def'           # Concatenation: a new string
'abcdef'
>>> 'Ni!' * 4               # Repetition: like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Formally, adding two string objects creates a new string object, with the contents of its operands joined. Repetition is like adding a string to itself a number of times. In both cases, Python lets you create arbitrarily sized strings; there's no need to predeclare anything in Python, including the sizes of data structures.‡ The `len` built-in function returns the length of a string (or any other object with a length).

Repetition may seem a bit obscure at first, but it comes in handy in a surprising number of contexts. For example, to print a line of 80 dashes, you can count up to 80, or let Python count for you:

```
>>> print('------- ...more... ---')      # 80 dashes, the hard way
>>> print('-' * 80)                      # 80 dashes, the easy way
```

Notice that operator overloading is at work here already: we're using the same + and * operators that perform addition and multiplication when using numbers. Python does the correct operation because it knows the types of the objects being added and multiplied. But be careful: the rules aren't quite as liberal as you might expect. For instance, Python doesn't allow you to mix numbers and strings in + expressions: `'abc'+9` raises an error instead of automatically converting `9` to a string.

As shown in the last row in Table 7-1, you can also iterate over strings in loops using `for` statements and test membership for both characters and substrings with the `in` expression operator, which is essentially a search. For substrings, `in` is much like the `str.find()` method covered later in this chapter, but it returns a Boolean result instead of the substring's position:

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')    # Step through items
...
```

‡ Unlike with C character arrays, you don't need to allocate or manage storage arrays when using Python strings; you can simply create string objects as needed and let Python manage the underlying memory space. As discussed in Chapter 6, Python reclaims unused objects' memory space automatically, using a reference-count garbage-collection strategy. Each object keeps track of the number of names, data structures, etc., that reference it; when the count reaches zero, Python frees the object's space. This scheme means Python doesn't have to stop and scan all the memory to find unused space to free (an additional garbage component also collects cyclic objects).

```
h a c k e r
>>> "k" in myjob                    # Found
True
>>> "z" in myjob                    # Not found
False
>>> 'spam' in 'abcspamdef'          # Substring search, no position returned
True
```

The `for` loop assigns a variable to successive items in a sequence (here, a string) and executes one or more statements for each item. In effect, the variable `c` becomes a cursor stepping across the string here. We will discuss iteration tools like these and others listed in Table 7-1 in more detail later in this book (especially in Chapters 14 and 20).

## Indexing and Slicing

Because strings are defined as ordered collections of characters, we can access their components by position. In Python, characters in a string are fetched by *indexing*—providing the numeric offset of the desired component in square brackets after the string. You get back the one-character string at the specified position.

As in the C language, Python offsets start at 0 and end at one less than the length of the string. Unlike C, however, Python also lets you fetch items from sequences such as strings using *negative* offsets. Technically, a negative offset is added to the length of a string to derive a positive offset. You can also think of negative offsets as counting backward from the end. The following interaction demonstrates:

```
>>> S = 'spam'
>>> S[0], S[-2]                     # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]           # Slicing: extract a section
('pa', 'pam', 'spa')
```

The first line defines a four-character string and assigns it the name `S`. The next line indexes it in two ways: `S[0]` fetches the item at offset 0 from the left (the one-character string `'s'`), and `S[-2]` gets the item at offset 2 back from the end (or equivalently, at offset (4 + (–2)) from the front). Offsets and slices map to cells as shown in Figure 7-1.§

The last line in the preceding example demonstrates *slicing*, a generalized form of indexing that returns an entire *section*, not a single item. Probably the best way to think of slicing is that it is a type of *parsing* (analyzing structure), especially when applied to strings—it allows us to extract an entire section (substring) in a single step. Slices can be used to extract columns of data, chop off leading and trailing text, and more. In fact, we'll explore slicing in the context of text parsing later in this chapter.

The basics of slicing are straightforward. When you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing

§ More mathematically minded readers (and students in my classes) sometimes detect a small asymmetry here: the leftmost item is at offset 0, but the rightmost is at offset –1. Alas, there is no such thing as a distinct –0 value in Python.

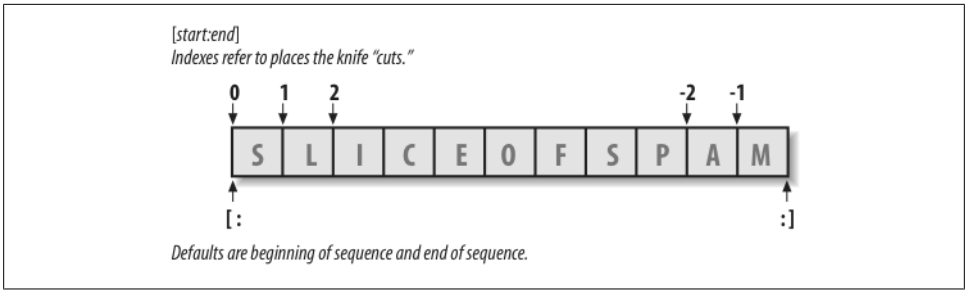


*Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset –1 is the last item). Either kind of offset can be used to give positions in indexing and slicing operations.*

the contiguous section identified by the offset pair. The left offset is taken to be the lower bound (*inclusive*), and the right is the upper bound (*noninclusive*). That is, Python fetches all items from the lower bound up to but not including the upper bound, and returns a new object containing the fetched items. If omitted, the left and right bounds default to 0 and the length of the object you are slicing, respectively.

For instance, in the example we just saw, `S[1:3]` extracts the items at offsets 1 and 2: it grabs the second and third items, and stops before the fourth item at offset 3. Next, `S[1:]` gets *all items beyond the first*—the upper bound, which is not specified, defaults to the length of the string. Finally, `S[:-1]` fetches *all but the last item*—the lower bound defaults to 0, and –1 refers to the last item, noninclusive.

This may seem confusing at first glance, but indexing and slicing are simple and powerful tools to use, once you get the knack. Remember, if you're unsure about the effects of a slice, try it out interactively. In the next chapter, you'll see that it's even possible to change an entire section of another object in one step by assigning to a slice (though not for immutables like strings). Here's a summary of the details for reference:

- *Indexing* (`S[i]`) fetches components at offsets:
  - —The first item is at offset 0.
  - —Negative indexes mean to count backward from the end or right.
  - —`S[0]` fetches the first item.
  - —`S[-2]` fetches the second item from the end (like `S[len(S)-2]`).
- *Slicing* (`S[i:j]`) extracts contiguous sections of sequences:
  - —The upper bound is noninclusive.
  - —Slice boundaries default to 0 and the sequence length, if omitted.
  - —`S[1:3]` fetches items at offsets 1 up to but not including 3.
  - —`S[1:]` fetches items at offset 1 through the end (the sequence length).

— `S[:3]` fetches items at offset 0 up to but not including 3.

— `S[:-1]` fetches items at offset 0 up to but not including the last item.

— `S[:]` fetches items at offsets 0 through the end—this effectively performs a top-level copy of `S`.

The last item listed here turns out to be a very common trick: it makes a full top-level *copy* of a sequence object—an object with the same value, but a distinct piece of memory (you'll find more on copies in Chapter 9). This isn't very useful for immutable objects like strings, but it comes in handy for objects that may be changed in-place, such as lists.

In the next chapter, you'll see that the syntax used to index by offset (square brackets) is used to index dictionaries by key as well; the operations look the same but have different interpretations.

#### Extended slicing: the third limit and slice objects

In Python 2.3 and later, slice expressions have support for an optional third index, used as a *step* (sometimes called a *stride*). The step is added to the index of each item extracted. The full-blown form of a slice is now `X[I:J:K]`, which means "extract all the items in `X`, from offset `I` through `J`–1, by `K`." The third limit, *K*, defaults to 1, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

For instance, `X[1:10:2]` will fetch *every other item* in `X` from offsets 1–9; that is, it will collect the items at offsets 1, 3, 5, 7, and 9. As usual, the first and second limits default to 0 and the length of the sequence, respectively, so `X[::2]` gets every other item from the beginning to the end of the sequence:

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'
```

You can also use a negative stride. For example, the slicing expression `"hello"[::-1]` returns the new string `"olleh"`—the first two bounds default to 0 and the length of the sequence, as before, and a stride of –1 indicates that the slice should go from right to left instead of the usual left to right. The effect, therefore, is to *reverse* the sequence:

```
>>> S = 'hello'
>>> S[::-1]
'olleh'
```

With a negative stride, the meanings of the first two bounds are essentially reversed. That is, the slice `S[5:1:-1]` fetches the items from 2 to 5, in reverse order (the result contains items from offsets 5, 4, 3, and 2):

```
>>> S = 'abcedfg'
>>> S[5:1:-1]
'fdec'
```

Skipping and reversing like this are the most common use cases for three-limit slices, but see Python's standard library manual for more details (or run a few experiments interactively). We'll revisit three-limit slices again later in this book, in conjunction with the `for` loop statement.

Later in the book, we'll also learn that slicing is equivalent to indexing with a *slice object*, a finding of importance to class writers seeking to support both operations:

```
>>> 'spam'[1:3]                    # Slicing syntax
'pa'
>>> 'spam'[slice(1, 3)]            # Slice objects
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, -1)]
'maps'
```

## Why You Will Care: Slices

Throughout this book, I will include common use case sidebars (such as this one) to give you a peek at how some of the language features being introduced are typically used in real programs. Because you won't be able to make much sense of real use cases until you've seen more of the Python picture, these sidebars necessarily contain many references to topics not introduced yet; at most, you should consider them previews of ways that you may find these abstract language concepts useful for common programming tasks.

For instance, you'll see later that the argument words listed on a system command line used to launch a Python program are made available in the `argv` attribute of the built-in `sys` module:

```
# File echo.py
import sys
print(sys.argv)

% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

Usually, you're only interested in inspecting the arguments that follow the program name. This leads to a very typical application of slices: a single slice expression can be used to return all but the first item of a list. Here, `sys.argv[1:]` returns the desired list, `['-a', '-b', '-c']`. You can then process this list without having to accommodate the program name at the front.

Slices are also often used to clean up lines read from input files. If you know that a line will have an end-of-line character at the end (a `\n` newline marker), you can get rid of it with a single expression such as `line[:-1]`, which extracts all but the last character in the line (the lower limit defaults to 0). In both cases, slices do the job of logic that must be explicit in a lower-level language.

Note that calling the `line.rstrip` method is often preferred for stripping newline characters because this call leaves the line intact if it has no newline character at the end—a common case for files created with some text-editing tools. Slicing works if you're sure the line is properly terminated.

## String Conversion Tools

One of Python's design mottos is that it refuses the temptation to guess. As a prime example, you cannot add a number and a string together in Python, even if the string looks like a number (i.e., is all digits):

```
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

This is by design: because `+` can mean both addition and concatenation, the choice of conversion would be ambiguous. So, Python treats this as an error. In Python, magic is generally omitted if it will make your life more complex.

What to do, then, if your script obtains a number as a text string from a file or user interface? The trick is that you need to employ conversion tools before you can treat a string like a number, or vice versa. For instance:

```
>>> int("42"), str(42)          # Convert from/to string
(42, '42')
>>> repr(42)                    # Convert to as-code string
'42'
```

The `int` function converts a string to a number, and the `str` function converts a number to its string representation (essentially, what it looks like when printed). The `repr` function (and the older backquotes expression, removed in Python 3.0) also converts an object to its string representation, but returns the object as a string of code that can be rerun to recreate the object. For strings, the result has quotes around it if displayed with a `print` statement:

```
>>> print(str('spam'), repr('spam'))
('spam', "'spam'")
```

See the sidebar "str and repr Display Formats" on page 116 for more on this topic. Of these, `int` and `str` are the generally prescribed conversion techniques.

Now, although you can't mix strings and number types around operators such as `+`, you can manually convert operands before that operation if needed:

```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: cannot concatenate 'str' and 'int' objects

>>> int(S) + I              # Force addition
43
```

```
>>> S + str(I)              # Force concatenation
'421'
```

Similar built-in functions handle floating-point number conversions to and from strings:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```

Later, we'll further study the built-in `eval` function; it runs a string containing Python expression code and so can convert a string to any kind of object. The functions `int` and `float` convert only to numbers, but this restriction means they are usually faster (and more secure, because they do not accept arbitrary expression code). As we saw briefly in Chapter 5, the string formatting expression also provides a way to convert numbers to strings. We'll discuss formatting further later in this chapter.

#### Character code conversions

On the subject of conversions, it is also possible to convert a single character to its underlying ASCII integer code by passing it to the built-in `ord` function—this returns the actual binary value of the corresponding byte in memory. The `chr` function performs the inverse operation, taking an ASCII integer code and converting it to the corresponding character:

```
>>> ord('s')
115
>>> chr(115)
's'
```

You can use a loop to apply these functions to all characters in a string. These tools can also be used to perform a sort of string-based math. To advance to the next character, for example, convert and do the math in integer:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

At least for single-character strings, this provides an alternative to using the built-in `int` function to convert from string to integer:

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

Such conversions can be used in conjunction with looping statements, introduced in Chapter 4 and covered in depth in the next part of this book, to convert a string of binary digits to their corresponding integer values. Each time through the loop, multiply the current value by 2 and add the next digit's integer value:

```
>>> B = '1101'                 # Convert binary digits to integer with ord
>>> I = 0
>>> while B != '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

A left-shift operation (`I << 1`) would have the same effect as multiplying by 2 here. We'll leave this change as a suggested exercise, though, both because we haven't studied loops in detail yet and because the `int` and `bin` built-ins we met in Chapter 5 handle binary conversion tasks for us in Python 2.6 and 3.0:

```
>>> int('1101', 2)             # Convert binary to integer: built-in
13
>>> bin(13)                    # Convert integer to binary
'0b1101'
```

Given enough time, Python tends to automate most common tasks!

## Changing Strings

Remember the term "immutable sequence"? The *immutable* part means that you can't change a string in-place (e.g., by assigning to an index):

```
>>> S = 'spam'
>>> S[0] = "x"
Raises an error!
```

So, how do you modify text information in Python? To change a string, you need to build and assign a new string using tools such as concatenation and slicing, and then, if desired, assign the result back to the string's original name:

```
>>> S = S + 'SPAM!'            # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

The first example adds a substring at the end of `S`, by concatenation (really, it makes a new string and assigns it back to `S`, but you can think of this as "changing" the original string). The second example replaces four characters with six by slicing, indexing, and concatenating. As you'll see in the next section, you can achieve similar effects with string method calls like `replace`:

```
>>> S = 'splot'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

Like every operation that yields a new string value, string methods generate new string objects. If you want to retain those objects, you can assign them to variable names. Generating a new string object for each string change is not as inefficient as it may sound—remember, as discussed in the preceding chapter, Python automatically garbage collects (reclaims the space of) old unused string objects as you go, so newer objects reuse the space held by prior values. Python is usually more efficient than you might expect.

Finally, it's also possible to build up new text values with string formatting expressions. Both of the following substitute objects into a string, in a sense converting the objects to strings and changing the original string according to a format specification:

```
>>> 'That is %d %s bird!' % (1, 'dead')          # Format expression
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead')    # Format method in 2.6 and 3.0
'That is 1 dead bird!'
```

Despite the substitution metaphor, though, the result of formatting is a new string object, not a modified one. We'll study formatting later in this chapter; as we'll find, formatting turns out to be more general and useful than this example implies. Because the second of the preceding calls is provided as a method, though, let's get a handle on string method calls before we explore formatting further.

As we'll see in Chapter 36, Python 3.0 and 2.6 introduce a new string type known as `bytearray`, which *is* mutable and so may be changed in place. `bytearray` objects aren't really strings; they're sequences of small, 8-bit integers. However, they support most of the same operations as normal strings and print as ASCII characters when displayed. As such, they provide another option for large amounts of text that must be changed frequently. In Chapter 36 we'll also see that `ord` and `chr` handle Unicode characters, too, which might not be stored in single bytes.

# String Methods

In addition to expression operators, strings provide a set of *methods* that implement more sophisticated text-processing tasks. Methods are simply functions that are associated with particular objects. Technically, they are attributes attached to objects that happen to reference callable functions. In Python, expressions and built-in functions may work across a range of types, but methods are generally *specific to object types*—string methods, for example, work only on string objects. The method sets of some types intersect in Python 3.0 (e.g., many types have a `count` method), but they are still more type-specific than other tools.

In finer-grained detail, functions are packages of code, and method calls combine two operations at once (an attribute fetch and a call):

*Attribute fetches*
: An expression of the form *`object.attribute`* means "fetch the value of *`attribute`* in *`object`*."

*Call expressions*
: An expression of the form *`function(arguments)`* means "invoke the code of *`function`*, passing zero or more comma-separated *`argument`* objects to it, and return *`function`*'s result value."

Putting these two together allows us to call a method of an object. The method call expression *`object.method(arguments)`* is evaluated from left to right—Python will first fetch the *`method`* of the *`object`* and then call it, passing in the *`arguments`*. If the method computes a result, it will come back as the result of the entire method-call expression.

As you'll see throughout this part of the book, most objects have callable methods, and all are accessed using this same method-call syntax. To call an object method, as you'll see in the following sections, you have to go through an existing object.

Table 7-3 summarizes the methods and call patterns for built-in string objects in Python 3.0; these change frequently, so be sure to check Python's standard library manual for the most up-to-date list, or run a `help` call on any string interactively. Python 2.6's string methods vary slightly; it includes a `decode`, for example, because of its different handling of Unicode data (something we'll discuss in Chapter 36). In this table, `S` is a string object, and optional arguments are enclosed in square brackets. String methods in this table implement higher-level operations such as splitting and joining, case conversions, content tests, and substring searches and replacements.

*Table 7-3. String method calls in Python 3.0*

| | |
|---|---|
| `S.capitalize()` | `S.ljust(width [, fill])` |
| `S.center(width [, fill])` | `S.lower()` |
| `S.count(sub [, start [, end]])` | `S.lstrip([chars])` |
| `S.encode([encoding [,errors]])` | `S.maketrans(x[, y[, z]])` |
| `S.endswith(suffix [, start [, end]])` | `S.partition(sep)` |
| `S.expandtabs([tabsize])` | `S.replace(old, new [, count])` |
| `S.find(sub [, start [, end]])` | `S.rfind(sub [,start [,end]])` |
| `S.format(fmtstr, *args, **kwargs)` | `S.rindex(sub [, start [, end]])` |
| `S.index(sub [, start [, end]])` | `S.rjust(width [, fill])` |
| `S.isalnum()` | `S.rpartition(sep)` |
| `S.isalpha()` | `S.rsplit([sep[, maxsplit]])` |
| `S.isdecimal()` | `S.rstrip([chars])` |
| `S.isdigit()` | `S.split([sep [,maxsplit]])` |

```
S.isidentifier()                    S.splitlines([keepends])
S.islower()                         S.startswith(prefix [, start [, end]])
S.isnumeric()                       S.strip([chars])
S.isprintable()                     S.swapcase()
S.isspace()                         S.title()
S.istitle()                         S.translate(map)
S.isupper()                         S.upper()
S.join(iterable)                    S.zfill(width)
```

As you can see, there are quite a few string methods, and we don't have space to cover them all; see Python's library manual or reference texts for all the fine points. To help you get started, though, let's work through some code that demonstrates some of the most commonly used methods in action, and illustrates Python text-processing basics along the way.

## String Method Examples: Changing Strings

As we've seen, because strings are immutable, they cannot be changed in-place directly. To make a new text value from an existing string, you construct a new string with operations such as slicing and concatenation. For example, to replace two characters in the middle of a string, you can use code like this:

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

But, if you're really just out to replace a substring, you can use the string `replace` method instead:

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

The `replace` method is more general than this code implies. It takes as arguments the original substring (of any length) and the string (of any length) to replace it with, and performs a global search and replace:

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

In such a role, `replace` can be used as a tool to implement template replacements (e.g., in form letters). Notice that this time we simply printed the result, instead of assigning it to a name—you need to assign results to names only if you want to retain them for later use.

If you need to replace one fixed-size string that can occur at any offset, you can do a replacement again, or search for the substring with the string `find` method and then slice:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')              # Search for position
>>> where                               # Occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

The `find` method returns the offset where the substring appears (by default, searching from the front), or `-1` if it is not found. As we saw earlier, it's a substring search operation just like the `in` expression, but `find` returns the position of a located substring.

Another option is to use `replace` with a third argument to limit it to a single substitution:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')           # Replace all
'xxxxEGGSxxxxEGGSxxxx'

>>> S.replace('SPAM', 'EGGS', 1)        # Replace one
'xxxxEGGSxxxxSPAMxxxx'
```

Notice that `replace` returns a new string object each time. Because strings are immutable, methods never really change the subject strings in-place, even if they are called "replace"!

The fact that concatenation operations and the `replace` method generate new string objects each time they are run is actually a potential downside of using them to change strings. If you have to apply many changes to a very large string, you might be able to improve your script's performance by converting the string to an object that does support in-place changes:

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
```

The built-in `list` function (or an object construction call) builds a new list out of the items in any sequence—in this case, "exploding" the characters of a string into a list. Once the string is in this form, you can make multiple changes to it without generating a new copy for each change:

```
>>> L[3] = 'x'                          # Works for lists, not strings
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

If, after your changes, you need to convert back to a string (e.g., to write to a file), use the string `join` method to "implode" the list back into a string:

```
>>> S = ''.join(L)
>>> S
'spaxxy'
```

The `join` method may look a bit backward at first sight. Because it is a method of strings (not of lists), it is called through the desired delimiter. `join` puts the strings in a list (or other iterable) together, with the delimiter between list items; in this case, it uses an empty string delimiter to convert from a list back to a string. More generally, any string delimiter and iterable of strings will do:

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

In fact, joining substrings all at once this way often runs much faster than concatenating them individually. Be sure to also see the earlier note about the mutable `bytearray` string in Python 3.0 and 2.6, described fully in Chapter 36; because it may be changed in place, it offers an alternative to this `list`/`join` combination for some kinds of text that must be changed often.

## String Method Examples: Parsing Text

Another common role for string methods is as a simple form of text *parsing*—that is, analyzing structure and extracting substrings. To extract substrings at fixed offsets, we can employ slicing techniques:

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

Here, the columns of data appear at fixed offsets and so may be sliced out of the original string. This technique passes for parsing, as long as the components of your data have fixed positions. If instead some sort of delimiter separates the data, you can pull out its components by splitting. This will work even if the data may show up at arbitrary positions within the string:

```
>>> line = 'aaa bbb  ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

The string `split` method chops up a string into a list of substrings, around a delimiter string. We didn't pass a delimiter in the prior example, so it defaults to whitespace—the string is split at groups of one or more spaces, tabs, and newlines, and we get back a list of the resulting substrings. In other applications, more tangible delimiters may separate the data. This example splits (and hence parses) the string at commas, a separator common in data returned by some database tools:

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

Delimiters can be longer than a single character, too:

```
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a', 'lumberjack']
```

Although there are limits to the parsing potential of slicing and splitting, both run very fast and can handle basic text-extraction chores.

## Other Common String Methods in Action

Other string methods have more focused roles—for example, to strip off whitespace at the end of a line of text, perform case conversions, test content, and test for a substring at the end or front:

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
True
```

Alternative techniques can also sometimes be used to achieve the same results as string methods—the `in` membership operator can be used to test for the presence of a substring, for instance, and length and slicing operations can be used to mimic `endswith`:

```
>>> line
'The knights who say Ni!\n'

>>> line.find('Ni') != -1        # Search via method call or expression
True
>>> 'Ni' in line
True

>>> sub = 'Ni!\n'
>>> line.endswith(sub)           # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

See also the `format` string formatting method described later in this chapter; it provides more advanced substitution tools that combine many operations in a single step.

Again, because there are so many methods available for strings, we won't look at every one here. You'll see some additional string examples later in this book, but for more

details you can also turn to the Python library manual and other documentation sources, or simply experiment interactively on your own. You can also check the `help(S.method)` results for a *`method`* of any string object `S` for more hints.

Note that none of the string methods accepts *patterns*—for pattern-based text processing, you must use the Python `re` standard library module, an advanced tool that was introduced in Chapter 4 but is mostly outside the scope of this text (one further example appears at the end of Chapter 36). Because of this limitation, though, string methods may sometimes run more quickly than the `re` module's tools.

## The Original string Module (Gone in 3.0)

The history of Python's string methods is somewhat convoluted. For roughly the first decade of its existence, Python provided a standard library module called `string` that contained functions that largely mirrored the current set of string object methods. In response to user requests, in Python 2.0 these functions were made available as methods of string objects. Because so many people had written so much code that relied on the original `string` module, however, it was retained for backward compatibility.

Today, you should use *only string methods*, not the original `string` module. In fact, the original module-call forms of today's string methods have been removed completely from Python in Release 3.0. However, because you may still see the module in use in older Python code, a brief look is in order here.

The upshot of this legacy is that in Python 2.6, there technically are still two ways to invoke advanced string operations: by calling object methods, or by calling `string` module functions and passing in the objects as arguments. For instance, given a variable `X` assigned to a string object, calling an object method:

```
X.method(arguments)
```

is usually equivalent to calling the same operation through the `string` module (provided that you have already imported the module):

```
string.method(X, arguments)
```

Here's an example of the method scheme in action:

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')
>>> x
'aspambspamcspam'
```

To access the same operation through the `string` module in Python 2.6, you need to import the module (at least once in your process) and pass in the object:

```
>>> import string
>>> y = string.replace(S, '+', 'spam')
>>> y
'aspambspamcspam'
```

Because the module approach was the standard for so long, and because strings are such a central component of most programs, you might see both call patterns in Python 2.X code you come across.

Again, though, today you should always use method calls instead of the older module calls. There are good reasons for this, besides the fact that the module calls have gone away in Release 3.0. For one thing, the module call scheme requires you to import the `string` module (methods do not require imports). For another, the module makes calls a few characters longer to type (when you load the module with `import`, that is, not using `from`). And, finally, the module runs more slowly than methods (the module maps most calls back to the methods and so incurs an extra call along the way).

The original `string` module itself, without its string method equivalents, is retained in Python 3.0 because it contains additional tools, including predefined string constants and a template object system (a relatively obscure tool omitted here—see the Python library manual for details on template objects). Unless you really want to have to change your 2.6 code to use 3.0, though, you should consider the basic string operation calls in it to be just ghosts from the past.

## String Formatting Expressions

Although you can get a lot done with the string methods and sequence operations we've already met, Python also provides a more advanced way to combine string processing tasks—*string formatting* allows us to perform multiple type-specific substitutions on a string in a single step. It's never strictly required, but it can be convenient, especially when formatting text to be displayed to a program's users. Due to the wealth of new ideas in the Python world, string formatting is available in two flavors in Python today:

*String formatting expressions*
: The original technique, available since Python's inception; this is based upon the C language's "printf" model and is used in much existing code.

*String formatting method calls*
: A newer technique added in Python 2.6 and 3.0; this is more unique to Python and largely overlaps with string formatting expression functionality.

Since the method call flavor is new, there is some chance that one or the other of these may become deprecated over time. The expressions are more likely to be deprecated in later Python releases, though this should depend on the future practice of real Python programmers. As they are largely just variations on a theme, though, either technique is valid to use today. Since string formatting expressions are the original in this department, let's start with them.

Python defines the `%` binary operator to work on strings (you may recall that this is also the remainder of division, or modulus, operator for numbers). When applied to strings, the `%` operator provides a simple way to format values as strings according to a format

definition. In short, the % operator provides a compact way to code multiple string substitutions all at once, instead of building and concatenating parts individually.

To format strings:

1. On the left of the % operator, provide a format string containing one or more embedded conversion targets, each of which starts with a % (e.g., %d).
2. On the right of the % operator, provide the object (or objects, embedded in a tuple) that you want Python to insert into the format string on the left in place of the conversion target (or targets).

For instance, in the formatting example we saw earlier in this chapter, the integer 1 replaces the %d in the format string on the left, and the string 'dead' replaces the %s. The result is a new string that reflects these two substitutions:

```
>>> 'That is %d %s bird!' % (1, 'dead')          # Format expression
That is 1 dead bird!
```

Technically speaking, string formatting expressions are usually optional—you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation. It's powerful enough to warrant a few more examples:

```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'

>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

The first example here plugs the string "Ni" into the target on the left, replacing the %s marker. In the second example, three values are inserted into the target string. Note that when you're inserting more than one value, you need to group the values on the right in parentheses (i.e., put them in a tuple). The % formatting expression operator expects either a single item or a tuple of one or more items on its right side.

The third example again inserts three values—an integer, a floating-point object, and a list object—but notice that all of the targets on the left are %s, which stands for conversion to string. As every type of object can be converted to a string (the one used when printing), every object type works with the %s conversion code. Because of this, unless you will be doing some special formatting, %s is often the only code you need to remember for the formatting expression.

Again, keep in mind that formatting always makes a new string, rather than changing the string on the left; because strings are immutable, it must work this way. As before, assign the result to a variable name if you need to retain it.

## Advanced String Formatting Expressions

For more advanced type-specific formatting, you can use any of the conversion type codes listed in Table 7-4 in formatting expressions; they appear after the `%` character in substitution targets. C programmers will recognize most of these because Python string formatting supports all the usual C `printf` format codes (but returns the result, instead of displaying it, like `printf`). Some of the format codes in the table provide alternative ways to format the same type; for instance, `%e`, `%f`, and `%g` provide alternative ways to format floating-point numbers.

*Table 7-4. String formatting type codes*

| Code | Meaning |
|---|---|
| `s` | String (or any object's `str(X)` string) |
| `r` | `s`, but uses `repr`, not `str` |
| `c` | Character |
| `d` | Decimal (integer) |
| `i` | Integer |
| `u` | Same as `d` (obsolete: no longer unsigned) |
| `o` | Octal integer |
| `x` | Hex integer |
| `X` | `x`, but prints uppercase |
| `e` | Floating-point exponent, lowercase |
| `E` | Same as `e`, but prints uppercase |
| `f` | Floating-point decimal |
| `F` | Floating-point decimal |
| `g` | Floating-point `e` or `f` |
| `G` | Floating-point `E` or `F` |
| `%` | Literal `%` |

In fact, conversion targets in the format string on the expression's left side support a variety of conversion operations with a fairly sophisticated syntax all their own. The general structure of conversion targets looks like this:

```
%[(name)][flags][width][.precision]typecode
```

The character type codes in Table 7-4 show up at the end of the target string. Between the `%` and the character code, you can do any of the following: provide a dictionary key; list flags that specify things like left justification (`-`), numeric sign (`+`), and zero fills (`0`); give a total minimum field width and the number of digits after a decimal point; and more. Both ***`width`*** and ***`precision`*** can also be coded as a `*` to specify that they should take their values from the next item in the input values.

Formatting target syntax is documented in full in the Python standard manuals, but to demonstrate common usage, let's look at a few examples. This one formats integers by default, and then in a six-character field with left justification and zero padding:

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234  ...001234'
```

The `%e`, `%f`, and `%g` formats display floating-point numbers in different ways, as the following interaction demonstrates (`%E` is the same as `%e` but the exponent is uppercase):

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'

>>> '%E' % x
'1.234568E+00'
```

For floating-point numbers, you can achieve a variety of additional formatting effects by specifying left justification, zero padding, numeric signs, field width, and digits after the decimal point. For simpler tasks, you might get by with simply converting to strings with a format expression or the `str` built-in function shown earlier:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

When sizes are not known until runtime, you can have the width and precision computed by specifying them with a * in the format string to force their values to be taken from the next item in the inputs to the right of the `%` operator—the 4 in the tuple here gives precision:

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

If you're interested in this feature, experiment with some of these examples and operations on your own for more information.

## Dictionary-Based String Formatting Expressions

String formatting also allows conversion targets on the left to refer to the keys in a dictionary on the right and fetch the corresponding values. I haven't told you much about dictionaries yet, so here's an example that demonstrates the basics:

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}
'1 spam'
```

Here, the (n) and (x) in the format string refer to keys in the dictionary literal on the right and fetch their associated values. Programs that generate text such as HTML or XML often use this technique—you can build up a dictionary of values and substitute them all at once with a single formatting expression that uses key-based references:

```
>>> reply = """                          # Template with substitution targets
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40}  # Build up values to substitute
>>> print(reply % values)                # Perform substitutions

Greetings...
Hello Bob!
Your age squared is 40
```

This trick is also used in conjunction with the vars built-in function, which returns a dictionary containing all the variables that exist in the place it is called:

```
>>> food = 'spam'
>>> age = 40
>>> vars()
{'food': 'spam', 'age': 40, ...many more... }
```

When used on the right of a format operation, this allows the format string to refer to variables by name (i.e., by dictionary key):

```
>>> "%(age)d %(food)s" % vars()
'40 spam'
```

We'll study dictionaries in more depth in Chapter 8. See also Chapter 5 for examples that convert to hexadecimal and octal number strings with the %x and %o formatting target codes.

## String Formatting Method Calls

As mentioned earlier, Python 2.6 and 3.0 introduced a new way to format strings that is seen by some as a bit more Python-specific. Unlike formatting expressions, formatting method calls are not closely based upon the C language's "printf" model, and they are more verbose and explicit in intent. On the other hand, the new technique still relies on some "printf" concepts, such as type codes and formatting specifications. Moreover, it largely overlaps with (and sometimes requires a bit more code than) formatting expressions, and it can be just as complex in advanced roles. Because of this, there is no best-use recommendation between expressions and method calls today, so most programmers would be well served by a cursory understanding of both schemes.

## The Basics

In short, the new string object's `format` method in 2.6 and 3.0 (and later) uses the subject string as a template and takes any number of arguments that represent values to be substituted according to the template. Within the subject string, curly braces designate substitution targets and arguments to be inserted either by position (e.g., `{1}`) or keyword (e.g., `{food}`). As we'll learn when we study argument passing in depth in Chapter 18, arguments to functions and methods may be passed by position or keyword name, and Python's ability to collect arbitrarily many positional and keyword arguments allows for such general method call patterns. In Python 2.6 and 3.0, for example:

```
>>> template = '{0}, {1} and {2}'                    # By position
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}'          # By keyword
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}'             # By both
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'
```

Naturally, the string can also be a literal that creates a temporary string, and arbitrary object types can be substituted:

```
>>> '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
'3.14, 42 and [1, 2]'
```

Just as with the `%` expression and other string methods, `format` creates and returns a new string object, which can be printed immediately or saved for further work (recall that strings are immutable, so `format` really *must* make a new object). String formatting is not just for display:

```
>>> X = '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
>>> X
'3.14, 42 and [1, 2]'

>>> X.split(' and ')
['3.14, 42', '[1, 2]']

>>> Y = X.replace('and', 'but under no circumstances')
>>> Y
'3.14, 42 but under no circumstances [1, 2]'
```

## Adding Keys, Attributes, and Offsets

Like `%` formatting expressions, format calls can become more complex to support more advanced usage. For instance, format strings can name object attributes and dictionary keys—as in normal Python syntax, square brackets name dictionary keys and dots denote object attributes of an item referenced by position or keyword. The first of the

following examples indexes a dictionary on the key "spam" and then fetches the attribute "platform" from the already imported `sys` module object. The second does the same, but names the objects by keyword instead of position:

```
>>> import sys

>>> 'My {1[spam]} runs {0.platform}'.format(sys, {'spam': 'laptop'})
'My laptop runs win32'

>>> 'My {config[spam]} runs {sys.platform}'.format(sys=sys,
                                                   config={'spam': 'laptop'})
'My laptop runs win32'
```

Square brackets in format strings can name list (and other sequence) offsets to perform indexing, too, but only single positive offsets work syntactically within format strings, so this feature is not as general as you might think. As with `%` expressions, to name negative offsets or slices, or to use arbitrary expression results in general, you must run expressions outside the format string itself:

```
>>> somelist = list('SPAM')
>>> somelist
['S', 'P', 'A', 'M']

>>> 'first={0[0]}, third={0[2]}'.format(somelist)
'first=S, third=A'

>>> 'first={0}, last={1}'.format(somelist[0], somelist[-1])    # [-1] fails in fmt
'first=S, last=M'

>>> parts = somelist[0], somelist[-1], somelist[1:3]          # [1:3] fails in fmt
>>> 'first={0}, last={1}, middle={2}'.format(*parts)
"first=S, last=M, middle=['P', 'A']"
```

## Adding Specific Formatting

Another similarity with `%` expressions is that more specific layouts can be achieved by adding extra syntax in the format string. For the formatting method, we use a colon after the substitution target's identification, followed by a format specifier that can name the field size, justification, and a specific type code. Here's the formal structure of what can appear as a substitution target in a format string:

{*fieldname*!*conversionflag*:*formatspec*}

In this substitution target syntax:

- *`fieldname`* is a number or keyword naming an argument, followed by optional ".name" attribute or "[index]" component references.
- *`conversionflag`* can be `r`, `s`, or `a` to call `repr`, `str`, or `ascii` built-in functions on the value, respectively.

- *formatspec* specifies how the value should be presented, including details such as field width, alignment, padding, decimal precision, and so on, and ends with an optional data type code.

The *formatspec* component after the colon character is formally described as follows (brackets denote optional components and are not coded literally):

```
[[fill]align][sign][#][0][width][.precision][typecode]
```

*align* may be `<`, `>`, `=`, or `^`, for left alignment, right alignment, padding after a sign character, or centered alignment, respectively. The *formatspec* also contains nested `{}` format strings with field names only, to take values from the arguments list dynamically (much like the * in formatting expressions).

See Python's library manual for more on substitution syntax and a list of the available type codes—they almost completely overlap with those used in `%` expressions and listed previously in Table 7-4, but the format method also allows a "b" type code used to display integers in binary format (it's equivalent to using the `bin` built-in call), allows a "%" type code to display percentages, and uses only "d" for base-10 integers (not "i" or "u").

As an example, in the following `{0:10}` means the first positional argument in a field 10 characters wide, `{1:<10}` means the second positional argument left-justified in a 10-character-wide field, and `{0.platform:>10}` means the `platform` attribute of the first argument right-justified in a 10-character-wide field:

```
>>> '{0:10} = {1:10}'.format('spam', 123.4567)
'spam       =    123.457'

>>> '{0:>10} = {1:<10}'.format('spam', 123.4567)
'      spam = 123.457   '

>>> '{0.platform:>10} = {1[item]:<10}'.format(sys, dict(item='laptop'))
'     win32 = laptop    '
```

Floating-point numbers support the same type codes and formatting specificity in formatting method calls as in `%` expressions. For instance, in the following `{2:g}` means the third argument formatted by default according to the "g" floating-point representation, `{1:.2f}` designates the "f" floating-point format with just 2 decimal digits, and `{2:06.2f}` adds a field with a width of 6 characters and zero padding on the left:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Hex, octal, and binary formats are supported by the `format` method as well. In fact, string formatting is an alternative to some of the built-in functions that format integers to a given base:

```
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)      # Hex, octal, binary
'FF, 377, 11111111'

>>> bin(255), int('11111111', 2), 0b11111111        # Other to/from binary
('0b11111111', 255, 255)

>>> hex(255), int('FF', 16), 0xFF                   # Other to/from hex
('0xff', 255, 255)

>>> oct(255), int('377', 8), 0o377, 0377            # Other to/from octal
('0377', 255, 255, 255)                             # 0377 works in 2.6, not 3.0!
```

Formatting parameters can either be hardcoded in format strings or taken from the arguments list dynamically by nested format syntax, much like the star syntax in formatting expressions:

```
>>> '{0:.2f}'.format(1 / 3.0)                       # Parameters hardcoded
'0.33'
>>> '%.2f' % (1 / 3.0)
'0.33'

>>> '{0:.{1}f}'.format(1 / 3.0, 4)                  # Take value from arguments
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)                           # Ditto for expression
'0.3333'
```

Finally, Python 2.6 and 3.0 also provide a new built-in `format` function, which can be used to format a single item. It's a more concise alternative to the string `format` method, and is roughly similar to formatting a single item with the `%` formatting expression:

```
>>> '{0:.2f}'.format(1.2345)                        # String method
'1.23'
>>> format(1.2345, '.2f')                           # Built-in function
'1.23'
>>> '%.2f' % 1.2345                                 # Expression
'1.23'
```

Technically, the `format` built-in runs the subject object's `__format__` method, which the `str.format` method does internally for each formatted item. It's still more verbose than the original `%` expression's equivalent, though—which leads us to the next section.

## Comparison to the % Formatting Expression

If you study the prior sections closely, you'll probably notice that at least for positional references and dictionary keys, the string `format` method looks very much like the `%` formatting expression, especially in advanced use with type codes and extra formatting syntax. In fact, in common use cases formatting expressions may be easier to code than formatting method calls, especially when using the generic `%s` print-string substitution target:

```
print('%s=%s' % ('spam', 42))            # 2.X+ format expression

print('{0}={1}'.format('spam', 42))      # 3.0 (and 2.6) format method
```

As we'll see in a moment, though, more complex formatting tends to be a draw in terms of complexity (difficult tasks are generally difficult, regardless of approach), and some see the formatting method as largely redundant.

On the other hand, the formatting method also offers a few potential advantages. For example, the original `%` expression can't handle keywords, attribute references, and binary type codes, although dictionary key references in `%` format strings can often achieve similar goals. To see how the two techniques overlap, compare the following `%` expressions to the equivalent `format` method calls shown earlier:

```
# The basics: with % instead of format()

>>> template = '%s, %s, %s'
>>> template % ('spam', 'ham', 'eggs')                    # By position
    'spam, ham, eggs'

>>> template = '%(motto)s, %(pork)s and %(food)s'
>>> template % dict(motto='spam', pork='ham', food='eggs')    # By key
'spam, ham and eggs'

>>> '%s, %s and %s' % (3.14, 42, [1, 2])                  # Arbitrary types
'3.14, 42 and [1, 2]'


# Adding keys, attributes, and offsets

>>> 'My %(spam)s runs %(platform)s' % {'spam': 'laptop', 'platform': sys.platform}
'My laptop runs win32'

>>> 'My %(spam)s runs %(platform)s' % dict(spam='laptop', platform=sys.platform)
'My laptop runs win32'

>>> somelist = list('SPAM')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
"first=S, last=M, middle=['P', 'A']"
```

When more complex formatting is applied the two techniques approach parity in terms of complexity, although if you compare the following with the `format` method call equivalents listed earlier you'll again find that the `%` expressions tend to be a bit simpler and more concise:

```
# Adding specific formatting

>>> '%-10s = %10s' % ('spam', 123.4567)
'spam       =   123.4567'

>>> '%10s = %-10s' % ('spam', 123.4567)
'      spam = 123.4567  '

>>> '%(plat)10s = %(item)-10s' % dict(plat=sys.platform, item='laptop')
'     win32 = laptop    '
```

```
# Floating-point numbers

>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'


# Hex and octal, but not binary

>>> '%x, %o' % (255, 255)
'ff, 377'
```

The `format` method has a handful of advanced features that the `%` expression does not, but even more involved formatting still seems to be essentially a draw in terms of complexity. For instance, the following shows the same result generated with both techniques, with field sizes and justifications and various argument reference methods:

```
# Hardcoded references in both

>>> import sys

>>> 'My {1[spam]:<8} runs {0.platform:>8}'.format(sys, {'spam': 'laptop'})
'My laptop   runs    win32'

>>> 'My %(spam)-8s runs %(plat)8s' % dict(spam='laptop', plat=sys.platform)
'My laptop   runs    win32'
```

In practice, programs are less likely to hardcode references like this than to execute code that builds up a set of substitution data ahead of time (to collect data to substitute into an HTML template all at once, for instance). When we account for common practice in examples like this, the comparison between the `format` method and the `%` expression is even more direct (as we'll see in Chapter 18, the `**data` in the method call here is special syntax that unpacks a dictionary of keys and values into individual "name=value" keyword arguments so they can be referenced by name in the format string):

```
# Building data ahead of time in both

>>> data = dict(platform=sys.platform, spam='laptop')

>>> 'My {spam:<8} runs {platform:>8}'.format(**data)
'My laptop   runs    win32'

>>> 'My %(spam)-8s runs %(platform)8s' % data
'My laptop   runs    win32'
```

As usual, the Python community will have to decide whether `%` expressions, `format` method calls, or a toolset with both techniques proves better over time. Experiment with these techniques on your own to get a feel for what they offer, and be sure to see the Python 2.6 and 3.0 library manuals for more details.

*String format method enhancements in Python 3.1*: The upcoming 3.1 release (in alpha form as this chapter was being written) will add a thousand-separator syntax for numbers, which inserts commas between three-digit groups. Add a comma before the type code to make this work, as follows:

```
>>> '{0:d}'.format(999999999999)
'999999999999'

>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

Python 3.1 also assigns relative numbers to substitution targets automatically if they are not included explicitly, though using this extension may negate one of the main benefits of the formatting method, as the next section describes:

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'

>>> '{:,d} {:,d}'.format(9999999, 8888888)
'9,999,999 8,888,888'

>>> '{:,.2f}'.format(296999.2567)
'296,999.26'
```

This book doesn't cover 3.1 officially, so you should take this as a preview. Python 3.1 will also address a major performance issue in 3.0 related to the speed of file input/output operations, which made 3.0 impractical for many types of programs. See the 3.1 release notes for more details. See also the *formats.py* comma-insertion and money-formatting function examples in Chapter 24 for a manual solution that can be imported and used prior to Python 3.1.

## Why the New Format Method?

Now that I've gone to such lengths to compare and contrast the two formatting techniques, I need to explain why you might want to consider using the `format` method variant at times. In short, although the formatting method can sometimes require more code, it also:

- Has a few extra features not found in the `%` expression
- Can make substitution value references more explicit
- Trades an operator for an arguably more mnemonic method name
- Does not support different syntax for single and multiple substitution value cases

Although both techniques are available today and the formatting expression is still widely used, the `format` method might eventually subsume it. But because the choice is currently still yours to make, let's briefly expand on some of the differences before moving on.

### Extra features

The method call supports a few extras that the expression does not, such as binary type codes and (coming in Python 3.1) thousands groupings. In addition, the method call supports key and attribute references directly. As we've seen, though, the formatting expression can usually achieve the same effects in other ways:

```
>>> '{0:b}'.format((2 ** 16) -1)
'1111111111111111'

>>> '%b' % ((2 ** 16) -1)
ValueError: unsupported format character 'b' (0x62) at index 1

>>> bin((2 ** 16) -1)
'0b1111111111111111'

>>> '%s' % bin((2 ** 16) -1)[2:]
'1111111111111111'
```

See also the prior examples that compare dictionary-based formatting in the `%` expression to key and attribute references in the `format` method; especially in common practice, the two seem largely variations on a theme.

### Explicit value references

One use case where the `format` method is at least debatably clearer is when there are many values to be substituted into the format string. The *lister.py* classes example we'll meet in Chapter 30, for example, substitutes six items into a single string, and in this case the method's `{i}` position labels seem easier to read than the expression's `%s`:

```
'\n%s<Class %s, address %s:\n%s%s%s>\n' % (...)              # Expression

'\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(...)    # Method
```

On the other hand, using dictionary keys in `%` expressions can mitigate much of this difference. This is also something of a worst-case scenario for formatting complexity, and not very common in practice; more typical use cases seem largely a tossup. Moreover, in Python 3.1 (still in alpha release form as I write these words), numbering substitution values will become optional, thereby subverting this purported benefit altogether:

```
C:\misc> C:\Python31\python
>>> 'The {0} side {1} {2}'.format('bright', 'of', 'life')
'The bright side of life'
>>>
>>> 'The {} side {} {}'.format('bright', 'of', 'life')        # Python 3.1+
'The bright side of life'
>>>
>>> 'The %s side %s %s' % ('bright', 'of', 'life')
'The bright side of life'
```

Using 3.1's automatic relative numbering like this seems to negate a large part of the method's advantage. Compare the effect on floating-point formatting, for example—the formatting expression is still more concise, and still seems less cluttered:

```
C:\misc> C:\Python31\python
>>> '{0:f}, {1:.2f}, {2:05.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 03.14'
>>>
>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
>>>
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

#### Method names and general arguments

Given this 3.1 auto-numbering change, the only clearly remaining potential advantages of the formatting method are that it replaces the `%` operator with a more mnemonic `format` method name and does not distinguish between single and multiple substitution values. The former may make the method appear simpler to beginners at first glance ("format" may be easier to parse than multiple "%" characters), though this is too subjective to call.

The latter difference might be more significant—with the format expression, a single value can be given by itself, but multiple values must be enclosed in a tuple:

```
>>> '%.2f' % 1.2345
'1.23'
>>> '%.2f %s' % (1.2345, 99)
'1.23 99'
```

Technically, the formatting expression accepts either a single substitution value, or a tuple of one or more items. In fact, because a single item can be given *either* by itself or within a tuple, a tuple to be formatted must be provided as nested tuples:

```
>>> '%s' % 1.23
'1.23'
>>> '%s' % (1.23,)
'1.23'
>>> '%s' % ((1.23,),)
'(1.23,)'
```

The formatting method, on the other hand, tightens this up by accepting general function arguments in both cases:

```
>>> '{0:.2f}'.format(1.2345)
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99)
'1.23 99'

>>> '{0}'.format(1.23)
'1.23'
>>> '{0}'.format((1.23,))
'(1.23,)'
```

Consequently, it might be less confusing to beginners and cause fewer programming mistakes. This is still a fairly minor issue, though—if you always enclose values in a tuple and ignore the nontupled option, the expression is essentially the same as the method call here. Moreover, the method incurs an extra price in inflated code size to achieve its limited flexibility. Given that the expression has been used extensively throughout Python's history, it's not clear that this point justifies breaking existing code for a new tool that is so similar, as the next section argues.

#### Possible future deprecation?

As mentioned earlier, there is some risk that Python developers may deprecate the `%` expression in favor of the `format` method in the future. In fact, there is a note to this effect in Python 3.0's manuals.

This has not yet occurred, of course, and both formatting techniques are fully available and reasonable to use in Python 2.6 and 3.0 (the versions of Python this book covers). Both techniques are supported in the upcoming Python 3.1 release as well, so deprecation of either seems unlikely for the foreseeable future. Moreover, because formatting expressions are used extensively in almost all existing Python code written to date, most programmers will benefit from being familiar with both techniques for many years to come.

If this deprecation ever does occur, though, you may need to recode all your `%` expressions as `format` methods, and translate those that appear in this book, in order to use a newer Python release. At the risk of editorializing here, I hope that such a change will be based upon the future common practice of actual Python programmers, not the whims of a handful of core developers—particularly given that the window for Python 3.0's many incompatible changes is now closed. Frankly, this deprecation would seem like trading one complicated thing for another complicated thing—one that is largely equivalent to the tool it would replace! If you care about migrating to future Python releases, though, be sure to watch for developments on this front over time.

## General Type Categories

Now that we've explored the first of Python's collection objects, the string, let's pause to define a few general type concepts that will apply to most of the types we look at from here on. With regard to built-in types, it turns out that operations work the same for all the types in the same category, so we'll only need to define most of these ideas once. We've only examined numbers and strings so far, but because they are representative of two of the three major type categories in Python, you already know more about several other types than you might think.

## Types Share Operation Sets by Categories

As you've learned, strings are immutable sequences: they cannot be changed in-place (the *immutable* part), and they are positionally ordered collections that are accessed by offset (the *sequence* part). Now, it so happens that all the sequences we'll study in this part of the book respond to the same sequence operations shown in this chapter at work on strings—concatenation, indexing, iteration, and so on. More formally, there are three major type (and operation) categories in Python:

*Numbers (integer, floating-point, decimal, fraction, others)*
: Support addition, multiplication, etc.

*Sequences (strings, lists, tuples)*
: Support indexing, slicing, concatenation, etc.

*Mappings (dictionaries)*
: Support indexing by key, etc.

Sets are something of a category unto themselves (they don't map keys to values and are not positionally ordered sequences), and we haven't yet explored mappings on our in-depth tour (dictionaries are discussed in the next chapter). However, many of the other types we will encounter will be similar to numbers and strings. For example, for any sequence objects `X` and `Y`:

- `X + Y` makes a new sequence object with the contents of both operands.
- `X * N` makes a new sequence object with `N` copies of the sequence operand `X`.

In other words, these operations work the same way on any kind of sequence, including strings, lists, tuples, and some user-defined object types. The only difference is that the new result object you get back is of the same type as the operands `X` and `Y`—if you concatenate lists, you get back a new list, not a string. Indexing, slicing, and other sequence operations work the same on all sequences, too; the type of the objects being processed tells Python which flavor of the task to perform.

## Mutable Types Can Be Changed In-Place

The immutable classification is an important constraint to be aware of, yet it tends to trip up new users. If an object type is immutable, you cannot change its value in-place; Python raises an error if you try. Instead, you must run code to make a new object containing the new value. The major core types in Python break down as follows:

*Immutables (numbers, strings, tuples, frozensets)*
: None of the object types in the immutable category support in-place changes, though we can always run expressions to make new objects and assign their results to variables as needed.

*Mutables (lists, dictionaries, sets)*
: Conversely, the mutable types can always be changed in-place with operations that do not create new objects. Although such objects can be copied, in-place changes support direct modification.

Generally, immutable types give some degree of integrity by guaranteeing that an object won't be changed by another part of a program. For a refresher on why this matters, see the discussion of shared object references in Chapter 6. To see how lists, dictionaries, and tuples participate in type categories, we need to move ahead to the next chapter.

## Chapter Summary

In this chapter, we took an in-depth tour of the string object type. We learned about coding string literals, and we explored string operations, including sequence expressions, string method calls, and string formatting with both expressions and method calls. Along the way, we studied a variety of concepts in depth, such as slicing, method call syntax, and triple-quoted block strings. We also defined some core ideas common to a variety of types: sequences, for example, share an entire set of operations.

In the next chapter, we'll continue our types tour with a look at the most general object collections in Python—lists and dictionaries. As you'll find, much of what you've learned here will apply to those types as well. And as mentioned earlier, in the final part of this book we'll return to Python's string model to flesh out the details of Unicode text and binary data, which are of interest to some, but not all, Python programmers. Before moving on, though, here's another chapter quiz to review the material covered here.

---

## Test Your Knowledge: Quiz

1. Can the string `find` method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer to a character?
4. How might you go about changing a string in Python?
5. Given a string `S` with the value `"s,pa,m"`, name two ways to extract the two characters in the middle.
6. How many characters are there in the string `"a\nb\x1f\000d"`?
7. Why might you use the `string` module instead of string method calls?

## Test Your Knowledge: Answers

1. No, because methods are always type-specific; that is, they only work on a single data type. Expressions like `X+Y` and built-in functions like `len(X)` are generic, though, and may work on a variety of types. In this case, for instance, the `in` membership expression has a similar effect as the string `find`, but it can be used to search both strings and lists. In Python 3.0, there is some attempt to group methods by categories (for example, the mutable sequence types `list` and `bytearray` have similar method sets), but methods are still more type-specific than other operation sets.
2. Yes. Unlike methods, expressions are generic and apply to many types. In this case, the slice expression is really a sequence operation—it works on any type of sequence object, including strings, lists, and tuples. The only difference is that when you slice a list, you get back a new list.
3. The built-in `ord(S)` function converts from a one-character string to an integer character code; `chr(I)` converts from the integer code back to a string.
4. Strings cannot be changed; they are immutable. However, you can achieve a similar effect by creating a new string—by concatenating, slicing, running formatting expressions, or using a method call like `replace`—and then assigning the result back to the original variable name.
5. You can slice the string using `S[2:4]`, or split on the comma and index the string using `S.split(',')[1]`. Try these interactively to see for yourself.
6. Six. The string `"a\nb\x1f\000d"` contains the bytes `a`, newline (`\n`), `b`, binary `31` (a hex escape `\x1f`), binary `0` (an octal escape `\000`), and `d`. Pass the string to the built-in `len` function to verify this, and print each of its character's `ord` results to see the actual byte values. See Table 7-2 for more details.
7. You should never use the `string` module instead of string object method calls today—it's deprecated, and its calls are removed completely in Python 3.0. The only reason for using the `string` module at all is for its other tools, such as predefined constants. You might also see it appear in what is now very old and dusty Python code.

# CHAPTER 8
# Lists and Dictionaries

This chapter presents the list and dictionary object types, both of which are collections of other objects. These two types are the main workhorses in almost all Python scripts. As you'll see, both types are remarkably flexible: they can be changed in-place, can grow and shrink on demand, and may contain and be nested in any other kind of object. By leveraging these types, you can build up and process arbitrarily rich information structures in your scripts.

## Lists

The next stop on our built-in object tour is the Python *list*. Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in-place by assignment to offsets and slices, list method calls, deletion statements, and more—they are *mutable* objects.

Python lists do the work of most of the collection data structures you might have to implement manually in lower-level languages such as C. Here is a quick look at their main properties. Python lists are:

*Ordered collections of arbitrary objects*
: From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain (i.e., they are sequences).

*Accessed by offset*
: Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

*Variable-length, heterogeneous, and arbitrarily nestable*
: Unlike strings, lists can grow and shrink in-place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

*Of the category "mutable sequence"*
: In terms of our type category qualifiers, lists are mutable (i.e., can be changed in-place) and can respond to all the sequence operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return new lists instead of new strings when applied to lists. Because lists are mutable, however, they also support other operations that strings don't (such as deletion and index assignment operations, which change the lists in-place).

*Arrays of object references*
: Technically, Python lists contain zero or more references to other objects. Lists might remind you of arrays of pointers (addresses) if you have a background in some other languages. Fetching an item from a Python list is about as fast as indexing a C array; in fact, lists really are arrays inside the standard Python interpreter, not linked structures. As we learned in Chapter 6, though, Python always follows a reference to an object whenever the reference is used, so your program deals only with objects. Whenever you assign an object to a data structure component or variable name, Python always stores a reference to that same object, not a copy of it (unless you request a copy explicitly).

Table 8-1 summarizes common and representative list object operations. As usual, for the full story see the Python standard library manual, or run a `help(list)` or `dir(list)` call interactively for a complete list of list methods—you can pass in a real list, or the word `list`, which is the name of the list data type.

*Table 8-1. Common list literals and operations*

| Operation | Interpretation |
|---|---|
| `L = []` | An empty list |
| `L = [0, 1, 2, 3]` | Four items: indexes 0..3 |
| `L = ['abc', ['def', 'ghi']]` | Nested sublists |
| `L = list('spam')` | Lists of an iterable's items, list of successive integers |
| `L = list(range(-4, 4))` | |
| `L[i]` | Index, index of index, slice, length |
| `L[i][j]` | |
| `L[i:j]` | |
| `len(L)` | |

| Operation | Interpretation |
|---|---|
| `L1 + L2` | Concatenate, repeat |
| `L * 3` | |
| `for x in L: print(x)` | Iteration, membership |
| `3 in L` | |
| `L.append(4)` | Methods: growing |
| `L.extend([5,6,7])` | |
| `L.insert(I, X)` | |
| `L.index(1)` | Methods: searching |
| `L.count(X)` | |
| `L.sort()` | Methods: sorting, reversing, etc. |
| `L.reverse()` | |
| `del L[k]` | Methods, statement: shrinking |
| `del L[i:j]` | |
| `L.pop()` | |
| `L.remove(2)` | |
| `L[i:j] = []` | |
| `L[i] = 1` | Index assignment, slice assignment |
| `L[i:j] = [4,5,6]` | |
| `L = [x**2 for x in range(5)]` | List comprehensions and maps (Chapters 14, 20) |
| `list(map(ord, 'spam'))` | |

When written down as a literal expression, a list is coded as a series of objects (really, expressions that return objects) in square brackets, separated by commas. For instance, the second row in Table 8-1 assigns the variable `L` to a four-item list. A nested list is coded as a nested square-bracketed series (row 3), and the empty list is just a square-bracket pair with nothing inside (row 1).*

Many of the operations in Table 8-1 should look familiar, as they are the same sequence operations we put to work on strings—indexing, concatenation, iteration, and so on. Lists also respond to list-specific method calls (which provide utilities such as sorting, reversing, adding items to the end, etc.), as well as in-place change operations (deleting items, assignment to indexes and slices, and so forth). Lists have these tools for change operations because they are a mutable object type.

* In practice, you won't see many lists written out like this in list-processing programs. It's more common to see code that processes lists constructed dynamically (at runtime). In fact, although it's important to master literal syntax, most data structures in Python are built by running program code at runtime.

# Lists in Action

Perhaps the best way to understand lists is to see them at work. Let's once again turn to some simple interpreter interactions to illustrate the operations in Table 8-1.

## Basic List Operations

Because they are sequences, lists support many of the same operations as strings. For example, lists respond to the + and * operators much like strings—they mean concatenation and repetition here too, except that the result is a new list, not a string:

```
% python
>>> len([1, 2, 3])                    # Length
3
>>> [1, 2, 3] + [4, 5, 6]             # Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                       # Repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Although the + operator works the same for lists and strings, it's important to know that it expects the same sort of sequence on both sides—otherwise, you get a type error when the code runs. For instance, you cannot concatenate a list and a string unless you first convert the list to a string (using tools such as `str` or `%` formatting) or convert the string to a list (the `list` built-in function does the trick):

```
>>> str([1, 2]) + "34"                # Same as "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")               # Same as [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

## List Iteration and Comprehensions

More generally, lists respond to all the sequence operations we used on strings in the prior chapter, including iteration tools:

```
>>> 3 in [1, 2, 3]                    # Membership
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')              # Iteration
...
1 2 3
```

We will talk more formally about `for` iteration and the `range` built-ins in Chapter 13, because they are related to statement syntax. In short, `for` loops step through items in any sequence from left to right, executing one or more statements for each item.

The last items in Table 8-1, list comprehensions and `map` calls, are covered in more detail in Chapter 14 and expanded on in Chapter 20. Their basic operation is straightforward, though—as introduced in Chapter 4, list comprehensions are a way to build a new list

by applying an expression to each item in a sequence, and are close relatives to `for` loops:

```
>>> res = [c * 4 for c in 'SPAM']          # List comprehensions
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

This expression is functionally equivalent to a `for` loop that builds up a list of results manually, but as we'll learn in later chapters, list comprehensions are simpler to code and faster to run today:

```
>>> res = []
>>> for c in 'SPAM':                       # List comprehension equivalent
...     res.append(c * 4)
...
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

As also introduced in Chapter 4, the `map` built-in function does similar work, but applies a function to items in a sequence and collects all the results in a new list:

```
>>> list(map(abs, [-1, -2, 0, 1, 2]))      # map function across sequence
[1, 2, 0, 1, 2]
```

Because we're not quite ready for the full iteration story, we'll postpone further details for now, but watch for a similar comprehension expression for dictionaries later in this chapter.

## Indexing, Slicing, and Matrixes

Becauselists are sequences, indexing and slicing work the same way for lists as they do for strings. However, the result of indexing a list is whatever type of object lives at the offset you specify, while slicing a list always returns a new list:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                                   # Offsets start at zero
'SPAM!'
>>> L[-2]                                  # Negative: count from the right
'Spam'
>>> L[1:]                                  # Slicing fetches sections
['Spam', 'SPAM!']
```

One note here: because you can nest lists and other object types within lists, you will sometimes need to string together index operations to go deeper into a data structure. For example, one of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sublists. Here's a basic 3 × 3 two-dimensional list-based array:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> matrix[1][1]
5
```

Notice in the preceding interaction that lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets (more on syntax in the next part of the book). Later in this chapter, you'll also see a dictionary-based matrix representation. For high-powered numeric work, the NumPy extension mentioned in Chapter 5 provides other ways to handle matrixes.

## Changing Lists In-Place

Because lists are mutable, they support operations that change a list object *in-place*. That is, the operations in this section all modify the list object directly, without requiring that you make a new copy, as you had to for strings. Because Python deals only in object references, this distinction between changing an object in-place and creating a new object matters—as discussed in Chapter 6, if you change an object in-place, you might impact more than one reference to it at the same time.

### Index and slice assignments

When using a list, you can change its contents by assigning to either a particular item (offset) or an entire section (slice):

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                 # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']      # Slice assignment: delete+insert
>>> L                             # Replaces items 0,1
['eat', 'more', 'SPAM!']
```

Both index and slice assignments are in-place changes—they modify the subject list directly, rather than generating a new list object for the result. Index assignment in Python works much as it does in C and most other languages: Python replaces the object reference at the designated offset with a new one.

*Slice assignment*, the last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:

1. *Deletion*. The slice you specify to the left of the `=` is deleted.
2. *Insertion*. The new items contained in the object to the right of the `=` are inserted into the list on the left, at the place where the old slice was deleted.†

This isn't what really happens, but it tends to help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list `L` that has the value `[1,2,3]`, the assignment `L[1:2]=[4,5]` sets `L` to the list `[1,4,5,3]`. Python first deletes the `2` (a one-item slice), then inserts the `4` and `5` where the deleted `2` used to be. This also explains why `L[1:2]=[]` is really a deletion operation—Python deletes the slice (the item at offset 1), and then inserts nothing.

In effect, slice assignment replaces an entire section, or "column," all at once. Because the length of the sequence being assigned does not have to match the length of the slice being assigned to, slice assignment can be used to replace (by overwriting), expand (by inserting), or shrink (by deleting) the subject list. It's a powerful operation, but frankly, one that you may not see very often in practice. There are usually more straightforward ways to replace, insert, and delete (concatenation and the `insert`, `pop`, and `remove` list methods, for example), which Python programmers tend to prefer in practice.

### List method calls

Like strings, Python list objects also support type-specific method calls, many of which change the subject list in-place:

```
>>> L.append('please')              # Append method call: add item at end
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()                        # Sort list items ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

Methods were introduced in Chapter 7. In brief, they are functions (really, attributes that reference functions) that are associated with particular objects. Methods provide type-specific tools; the list methods presented here, for instance, are generally available only for lists.

Perhaps the most commonly used list method is `append`, which simply tacks a single item (object reference) onto the end of the list. Unlike concatenation, `append` expects you to pass in a single object, not a list. The effect of `L.append(X)` is similar to `L+[X]`, but while the former changes `L` in-place, the latter makes a new list.‡

Another commonly seen method, `sort`, orders a list in-place; it uses Python standard comparison tests (here, string comparisons), and by default sorts in ascending order.

† This description needs elaboration when the value and the slice being assigned overlap: `L[2:5]=L[3:6]`, for instance, works fine because the value to be inserted is fetched before the deletion happens on the left.

‡ Unlike `+` concatenation, `append` doesn't have to generate new objects, so it's usually faster. You can also mimic `append` with clever slice assignments: `L[len(L):]=[X]` is like `L.append(X)`, and `L[:0]=[X]` is like appending at the front of a list. Both delete an empty slice and insert `X`, changing `L` in-place quickly, like `append`.

You can modify sort behavior by passing in *keyword arguments*—a special "name=value" syntax in function calls that specifies passing by name and is often used for giving configuration options. In sorts, the `key` argument gives a one-argument function that returns the value to be used in sorting, and the `reverse` argument allows sorts to be made in descending instead of ascending order:

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                              # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                 # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)   # Change sort order
>>> L
['aBe', 'ABD', 'abc']
```

The sort `key` argument might also be useful when sorting lists of dictionaries, to pick out a sort key by indexing each dictionary. We'll study dictionaries later in this chapter, and you'll learn more about keyword function arguments in Part IV.

*Comparison and sorts in 3.0*: In Python 2.6 and earlier, comparisons of differently typed objects (e.g., a string and a list) work—the language defines a fixed ordering among different types, which is deterministic, if not aesthetically pleasing. That is, the ordering is based on the names of the types involved: all integers are less than all strings, for example, because `"int"` is less than `"str"`. Comparisons never automatically convert types, except when comparing numeric type objects.

In Python 3.0, this has changed: comparison of mixed types raises an exception instead of falling back on the fixed cross-type ordering. Because sorting uses comparisons internally, this means that `[1, 2, 'spam'].sort()` succeeds in Python 2.X but will raise an exception in Python 3.0 and later.

Python 3.0 also no longer supports passing in an arbitrary *comparison function* to sorts, to implement different orderings. The suggested workaround is to use the `key=`*`func`* keyword argument to code value transformations during the sort, and use the `reverse=True` keyword argument to change the sort order to descending. These were the typical uses of comparison functions in the past.

One warning here: beware that `append` and `sort` change the associated list object in-place, but don't return the list as a result (technically, they both return a value called `None`). If you say something like `L=L.append(X)`, you won't get the modified value of `L` (in fact, you'll lose the reference to the list altogether!). When you use attributes such as `append` and `sort`, objects are changed as a side effect, so there's no reason to reassign.

Partly because of such constraints, sorting is also available in recent Pythons as a built-in function, which sorts any collection (not just lists) and returns a new list for the result (instead of in-place changes):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)          # Sorting built-in
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)    # Pretransform items: differs!
['abe', 'abd', 'abc']
```

Notice the last example here—we can convert to lowercase prior to the sort with a list comprehension, but the result does not contain the original list's values as it does with the `key` argument. The latter is applied temporarily during the sort, instead of changing the values to be sorted. As we move along, we'll see contexts in which the `sorted` built-in can sometimes be more useful than the `sort` method.

Like strings, lists have other methods that perform other specialized operations. For instance, `reverse` reverses the list in-place, and the `extend` and `pop` methods insert multiple items at the end of and delete an item from the end of the list, respectively. There is also a `reversed` built-in function that works much like `sorted`, but it must be wrapped in a `list` call because it's an iterator (more on iterators later):

```
>>> L = [1, 2]
>>> L.extend([3,4,5])               # Add many items at end
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                         # Delete and return last item
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                     # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))               # Reversal built-in with a result
[1, 2, 3, 4]
```

In some types of programs, the list `pop` method used here is often used in conjunction with `append` to implement a quick last-in-first-out (LIFO) *stack* structure. The end of the list serves as the top of the stack:

```
>>> L = []
>>> L.append(1)                     # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                         # Pop off stack
2
>>> L
[1]
```

The pop method also accepts an optional offset of the item to be deleted and returned (the default is the last item). Other list methods remove an item by value (remove), insert an item at an offset (insert), search for an item's offset (index), and more:

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')                 # Index of an object
1
>>> L.insert(1, 'toast')            # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')                # Delete by value
>>> L
['spam', 'toast', 'ham']
>>> L.pop(1)                        # Delete by position
'toast'
>>> L
['spam', 'ham']
```

See other documentation sources or experiment with these calls interactively on your own to learn more about list methods.

#### Other common list operations

Because lists are mutable, you can use the del statement to delete an item or section in-place:

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]                        # Delete one item
>>> L
['eat', 'more', 'please']
>>> del L[1:]                       # Delete an entire section
>>> L                               # Same as L[1:] = []
['eat']
```

Because slice assignment is a deletion plus an insertion, you can also delete a section of a list by assigning an empty list to a slice (L[i:j]=[]); Python deletes the slice named on the left, and then inserts nothing. Assigning an empty list to an index, on the other hand, just stores a reference to the empty list in the specified slot, rather than deleting it:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there are additional list methods and operations not illustrated here (including methods for inserting and searching). For a comprehensive and up-to-date list of type tools, you should always consult

Python's manuals, Python's `dir` and `help` functions (which we first met in Chapter 4), or one of the reference texts mentioned in the Preface.

I'd also like to remind you one more time that all the in-place change operations discussed here work only for mutable objects: they won't work on strings (or tuples, discussed in Chapter 9), no matter how hard you try. Mutability is an inherent property of each object type.

## Dictionaries

Apart from lists, *dictionaries* are perhaps the most flexible built-in data type in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by positional offset.

Being a built-in type, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—indexing a dictionary is a very fast search operation. Dictionaries also sometimes do the work of records and symbol tables used in other languages, can represent sparse (mostly empty) data structures, and much more. Here's a rundown of their main properties. Python dictionaries are:

*Accessed by key, not offset*
: Dictionaries are sometimes called *associative arrays* or *hashes*. They associate a set of values with keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

*Unordered collections of arbitrary objects*
: Unlike in a list, items stored in a dictionary aren't kept in any particular order; in fact, Python randomizes their left-to-right order to provide quick lookup. Keys provide the symbolic (not physical) locations of items in a dictionary.

*Variable-length, heterogeneous, and arbitrarily nestable*
: Like lists, dictionaries can grow and shrink in-place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on).

*Of the category "mutable mapping"*
: Dictionaries can be changed in-place by assigning to indexes (they are mutable), but they don't support the sequence operations that work on strings and lists. Because dictionaries are unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in representatives of the mapping type category (objects that map keys to values).

*Tables of object references (hash tables)*

If lists are arrays of object references that support access by position, dictionaries are unordered tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies).

Table 8-2 summarizes some of the most common and representative dictionary operations (again, see the library manual or run a `dir(dict)` or `help(dict)` call for a complete list—`dict` is the name of the type). When coded as a literal expression, a dictionary is written as a series of *`key:value`* pairs, separated by commas, enclosed in curly braces.§ An empty dictionary is an empty set of braces, and dictionaries can be nested by writing one as a value inside another dictionary, or within a list or tuple.

*Table 8-2. Common dictionary literals and operations*

| Operation | Interpretation |
|---|---|
| `D = {}` | Empty dictionary |
| `D = {'spam': 2, 'eggs': 3}` | Two-item dictionary |
| `D = {'food': {'ham': 1, 'egg': 2}}` | Nesting |
| `D = dict(name='Bob', age=40)` | Alternative construction techniques: |
| `D = dict(zip(keyslist, valslist))` | keywords, zipped pairs, key lists |
| `D = dict.fromkeys(['a', 'b'])` | |
| `D['eggs']` | Indexing by key |
| `D['food']['ham']` | |
| `'eggs' in D` | Membership: key present test |
| `D.keys()` | Methods: keys, |
| `D.values()` | values, |
| `D.items()` | keys+values, |
| `D.copy()` | copies, |
| `D.get(key, default)` | defaults, |
| `D.update(D2)` | merge, |
| `D.pop(key)` | delete, etc. |
| `len(D)` | Length: number of stored entries |
| `D[key] = 42` | Adding/changing keys |

§ As with lists, you won't often see dictionaries constructed using literals. Lists and dictionaries are grown in different ways, though. As you'll see in the next section, dictionaries are typically built up by assigning to new keys at runtime; this approach fails for lists (lists are commonly grown with `append` instead).

| Operation | Interpretation |
| --- | --- |
| `del D[key]` | Deleting entries by key |
| `list(D.keys())` | Dictionary views (Python 3.0) |
| `D1.keys() & D2.keys()` | |
| `D = {x: x*2 for x in range(10)}` | Dictionary comprehensions (Python 3.0) |

# Dictionaries in Action

As Table 8-2 suggests, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates a dictionary, it stores its items in any left-to-right order it chooses; to fetch a value back, you supply the key with which it is associated, not its relative position. Let's go back to the interpreter to get a feel for some of the dictionary operations in Table 8-2.

## Basic Dictionary Operations

In normal operation, you create dictionaries with literals and store and access items by key with indexing:

```
% python
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}      # Make a dictionary
>>> D['spam']                                 # Fetch a value by key
2
>>> D                                         # Order is scrambled
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Here, the dictionary is assigned to the variable `D`; the value of the key `'spam'` is the integer `2`, and so on. We use the same square bracket syntax to index dictionaries by key as we did to index lists by offset, but here it means access by key, not by position.

Notice the end of this example: the left-to-right order of keys in a dictionary will almost always be different from what you originally typed. This is on purpose: to implement fast key lookup (a.k.a. hashing), keys need to be reordered in memory. That's why operations that assume a fixed left-to-right order (e.g., slicing, concatenation) do not apply to dictionaries; you can fetch values only by key, not by position.

The built-in `len` function works on dictionaries, too; it returns the number of items stored in the dictionary or, equivalently, the length of its keys list. The dictionary `in` membership operator allows you to test for key existence, and the `keys` method returns all the keys in the dictionary. The latter of these can be useful for processing dictionaries sequentially, but you shouldn't depend on the order of the keys list. Because the `keys` result can be used as a normal list, however, it can always be sorted if order matters (more on sorting and dictionaries later):

```
>>> len(D)                                    # Number of entries in dictionary
3
>>> 'ham' in D                                # Key membership test alternative
```

```
True
>>> list(D.keys())                    # Create a new list of my keys
['eggs', 'ham', 'spam']
```

Notice the second expression in this listing. As mentioned earlier, the `in` membership test used for strings and lists also works on dictionaries—it checks whether a key is stored in the dictionary. Technically, this works because dictionaries define *iterators* that step through their keys lists. Other types provide iterators that reflect their common uses; files, for example, have iterators that read line by line. We'll discuss iterators in Chapters 14 and 20.

Also note the syntax of the last example in this listing. We have to enclose it in a `list` call in Python 3.0 for similar reasons—`keys` in 3.0 returns an iterator, instead of a physical list. The `list` call forces it to produce all its values at once so we can print them. In 2.6, `keys` builds and returns an actual list, so the `list` call isn't needed to display results. More on this later in this chapter.

The order of keys in a dictionary is arbitrary and can change from release to release, so don't be alarmed if your dictionaries print in a different order than shown here. In fact, the order has changed for me too—I'm running all these examples with Python 3.0, but their keys had a different order in an earlier edition when displayed. You shouldn't depend on dictionary key ordering, in either programs or books!

## Changing Dictionaries In-Place

Let's continue with our interactive session. Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in-place without making new dictionaries: simply assign a value to a key to change or create an entry. The `del` statement works here, too; it deletes the entry associated with the key specified as an index. Notice also the nesting of a list inside a dictionary in this example (the value of the key `'ham'`). All collection data types in Python can nest inside each other arbitrarily:

```
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}

>>> D['ham'] = ['grill', 'bake', 'fry']          # Change entry
>>> D
{'eggs': 3, 'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> del D['eggs']                                # Delete entry
>>> D
{'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> D['brunch'] = 'Bacon'                        # Add new entry
>>> D
{'brunch': 'Bacon', 'ham': ['grill', 'bake', 'fry'], 'spam': 2}
```

As with lists, assigning to an existing index in a dictionary changes its associated value. Unlike with lists, however, whenever you assign a *new* dictionary key (one that hasn't been assigned before) you create a new entry in the dictionary, as was done in the previous example for the key `'brunch'`. This doesn't work for lists because Python considers an offset beyond the end of a list out of bounds and throws an error. To expand a list, you need to use tools such as the `append` method or slice assignment instead.

## More Dictionary Methods

Dictionary methods provide a variety of tools. For instance, the dictionary `values` and `items` methods return the dictionary's values and (*`key`*`,`*`value`*) pair tuples, respectively (as with `keys`, wrap them in a `list` call in Python 3.0 to collect their values for display):

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> list(D.values())
[3, 1, 2]
>>> list(D.items())
 [('eggs', 3), ('ham', 1), ('spam', 2)]
```

Such lists are useful in loops that need to step through dictionary entries one by one. Fetching a nonexistent key is normally an error, but the `get` method returns a default value (`None`, or a passed-in default) if the key doesn't exist. It's an easy way to fill in a default for a key that isn't present and avoid a missing-key error:

```
>>> D.get('spam')                    # A key that is there
2
>>> print(D.get('toast'))            # A key that is missing
None
>>> D.get('toast', 88)
88
```

The `update` method provides something similar to concatenation for dictionaries, though it has nothing to do with left-to-right ordering (again, there is no such thing in dictionaries). It merges the keys and values of one dictionary into another, blindly overwriting values of the same key:

```
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> D2 = {'toast':4, 'muffin':5}
>>> D.update(D2)
>>> D
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

Finally, the dictionary `pop` method deletes a key from a dictionary and returns the value it had. It's similar to the list `pop` method, but it takes a key instead of an optional position:

```
# pop a dictionary by key
>>> D
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
>>> D.pop('muffin')
```

```
5
>>> D.pop('toast')                      # Delete and return from a key
4
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}

# pop a list by position
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                             # Delete and return from the end
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                            # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

Dictionaries also provide a `copy` method; we'll discuss this in Chapter 9, as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with many more methods than those listed in Table 8-2; see the Python library manual or other documentation sources for a comprehensive list.

## A Languages Table

Let's look at a more realistic dictionary example. The following example creates a table that maps programming language names (the keys) to their creators (the values). You fetch creator names by indexing on language names:

```
>>> table = {'Python':  'Guido van Rossum',
...          'Perl':    'Larry Wall',
...          'Tcl':     'John Ousterhout' }
>>>
>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'

>>> for lang in table:                  # Same as: for lang in table.keys()
...     print(lang, '\t', table[lang])
...
Tcl     John Ousterhout
Python  Guido van Rossum
Perl    Larry Wall
```

The last command uses a `for` loop, which we haven't covered in detail yet. If you aren't familiar with `for` loops, this command simply iterates through each key in the table and prints a tab-separated list of keys and their values. We'll learn more about `for` loops in Chapter 13.

Dictionaries aren't sequences like lists and strings, but if you need to step through the items in a dictionary, it's easy—calling the dictionary `keys` method returns all stored

keys, which you can iterate through with a `for`. If needed, you can index from key to value inside the `for` loop, as was done in this code.

In fact, Python also lets you step through a dictionary's keys list without actually calling the `keys` method in most `for` loops. For any dictionary `D`, saying `for key in D:` works the same as saying the complete `for key in D.keys():`. This is really just another instance of the iterators mentioned earlier, which allow the `in` membership operator to work on dictionaries as well (more on iterators later in this book).

## Dictionary Usage Notes

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

- **Sequence operations don't work**. Dictionaries are mappings, not sequences; because there's no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting a contiguous section) simply don't apply. In fact, Python raises an error when your code runs if you try to do such things.
- **Assigning to new indexes adds entries**. Keys can be created when you write a dictionary literal (in which case they are embedded in the literal itself), or when you assign values to new keys of an existing dictionary object. The end result is the same.
- **Keys need not always be strings**. Our examples so far have used strings as keys, but any other *immutable* objects (i.e., not lists) work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples are sometimes used as dictionary keys too, allowing for compound key values. Class instance objects (discussed in Part VI) can also be used as keys, as long as they have the proper protocol methods; roughly, they need to tell Python that their values are hashable and won't change, as otherwise they would be useless as fixed keys.

### Using dictionaries to simulate flexible lists

The last point in the prior list is important enough to demonstrate with a few examples. When you use lists, it is illegal to assign to an offset that is off the end of the list:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Although you can use repetition to preallocate as big a list as you'll need (e.g., `[0]*100`), you can also do something that looks similar with dictionaries that does not require such space allocations. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

Here, it looks as if `D` is a 100-item list, but it's really a dictionary with a single entry; the value of the key `99` is the string `'spam'`. You can access this structure with offsets much like a list, but you don't have to allocate space for all the positions you might ever need to assign values to in the future. When used like this, dictionaries are like more flexible equivalents of lists.

### Using dictionaries for sparse data structures

In a similar way, dictionary keys are also commonly leveraged to implement *sparse* data structures—for example, multidimensional arrays where only a few positions have values stored in them:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4          # ; separates statements
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Here, we've used a dictionary to represent a three-dimensional array that is empty except for the two positions `(2,3,4)` and `(7,8,9)`. The keys are *tuples* that record the coordinates of nonempty slots. Rather than allocating a large and mostly empty three-dimensional matrix to hold these values, we can use a simple two-item dictionary. In this scheme, accessing an empty slot triggers a nonexistent key exception, as these slots are not physically stored:

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

### Avoiding missing-key errors

Errors for nonexistent key fetches are common in sparse matrixes, but you probably won't want them to shut down your program. There are at least three ways to fill in a default value instead of getting such an error message—you can test for keys ahead of time in `if` statements, use a `try` statement to catch and recover from the exception explicitly, or simply use the dictionary `get` method shown earlier to provide a default for keys that do not exist:

```
>>> if (2,3,6) in Matrix:              # Check for key before fetch
...     print(Matrix[(2,3,6)])         # See Chapter 12 for if/else
```

```
... else:
...     print(0)
...
0
>>> try:
...     print(Matrix[(2,3,6)])          # Try to index
... except KeyError:                    # Catch and recover
...     print(0)                        # See Chapter 33 for try/except
...
0
>>> Matrix.get((2,3,4), 0)             # Exists; fetch and return
88
>>> Matrix.get((2,3,6), 0)             # Doesn't exist; use default arg
0
```

Of these, the `get` method is the most concise in terms of coding requirements; we'll study the `if` and `try` statements in more detail later in this book.

### Using dictionaries as "records"

As you can see, dictionaries can play many roles in Python. In general, they can replace search data structures (because indexing by key is a search operation) and can represent many types of structured information. For example, dictionaries are one of many ways to describe the properties of an item in your program's domain; that is, they can serve the same role as "records" or "structs" in other languages.

The following, for example, fills out a dictionary by assigning to new keys over time:

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age']  = 45
>>> rec['job']  = 'trainer/writer'
>>>
>>> print(rec['name'])
mel
```

Especially when nested, Python's built-in data types allow us to easily represent structured information. This example again uses a dictionary to capture object properties, but it codes it all at once (rather than assigning to each key separately) and nests a list and a dictionary to represent structured property values:

```
>>> mel = {'name': 'Mark',
...        'jobs': ['trainer', 'writer'],
...        'web':  'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip':80513}}
```

To fetch components of nested objects, simply string together indexing operations:

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
```

```
>>> mel['home']['zip']
80513
```

Although we'll learn in Part VI that classes (which group both data and logic) can be better in this record role, dictionaries are an easy-to-use tool for simpler requirements.

### Why You Will Care: Dictionary Interfaces

Dictionaries aren't just a convenient way to store information by key in your programs—some Python extensions also present interfaces that look like and work the same as dictionaries. For instance, Python's interface to DBM access-by-key files looks much like a dictionary that must be opened. Strings are stored and fetched using key indexes:

```
import anydbm
file = anydbm.open("filename") # Link to file
file['key'] = 'data'           # Store data by key
data = file['key']             # Fetch data by key
```

In Chapter 27, you'll see that you can store entire Python objects this way, too, if you replace `anydbm` in the preceding code with `shelve` (shelves are access-by-key databases of persistent Python objects). For Internet work, Python's CGI script support also presents a dictionary-like interface. A call to `cgi.FieldStorage` yields a dictionary-like object with one entry per input field on the client's web page:

```
import cgi
form = cgi.FieldStorage()      # Parse form data
if 'name' in form:
    showReply('Hello, ' + form['name'].value)
```

All of these, like dictionaries, are instances of mappings. Once you learn dictionary interfaces, you'll find that they apply to a variety of built-in tools in Python.

## Other Ways to Make Dictionaries

Finally, note that because dictionaries are so useful, more ways to build them have emerged over time. In Python 2.3 and later, for example, the last two calls to the `dict` constructor (really, type name) shown here have the same effect as the literal and key-assignment forms above them:

```
{'name': 'mel', 'age': 45}              # Traditional literal expression

D = {}                                  # Assign by keys dynamically
D['name'] = 'mel'
D['age']  = 45

dict(name='mel', age=45)                # dict keyword argument form

dict([('name', 'mel'), ('age', 45)])    # dict key/value tuples form
```

All four of these forms create the same two-key dictionary, but they are useful in differing circumstances:

- The first is handy if you can spell out the entire dictionary ahead of time.
- The second is of use if you need to create the dictionary one field at a time on the fly.
- The third involves less typing than the first, but it requires all keys to be strings.
- The last is useful if you need to build up keys and values as sequences at runtime.

We met keyword arguments earlier when sorting; the third form illustrated in this code listing has become especially popular in Python code today, since it has less syntax (and hence there is less opportunity for mistakes). As suggested previously in Table 8-2, the last form in the listing is also commonly used in conjunction with the `zip` function, to combine separate lists of keys and values obtained dynamically at runtime (parsed out of a data file's columns, for instance). More on this option in the next section.

Provided all the key's values are the same initially, you can also create a dictionary with this special form—simply pass in a list of keys and an initial value for all of the values (the default is `None`):

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

Although you could get by with just literals and key assignments at this point in your Python career, you'll probably find uses for all of these dictionary-creation forms as you start applying them in realistic, flexible, and dynamic Python programs.

The listings in this section document the various ways to create dictionaries in both Python 2.6 and 3.0. However, there is yet another way to create dictionaries, available only in Python 3.0 (and later): the *dictionary comprehension* expression. To see how this last form looks, we need to move on to the next section.

## Dictionary Changes in Python 3.0

This chapter has so far focused on dictionary basics that span releases, but the dictionary's functionality has mutated in Python 3.0. If you are using Python 2.X code, you may come across some dictionary tools that either behave differently or are missing altogether in 3.0. Moreover, 3.0 coders have access to additional dictionary tools not available in 2.X. Specifically, dictionaries in 3.0:

- Support a new dictionary comprehension expression, a close cousin to list and set comprehensions
- Return iterable views instead of lists for the methods `D.keys`, `D.values`, and `D.items`
- Require new coding styles for scanning by sorted keys, because of the prior point
- No longer support relative magnitude comparisons directly—compare manually instead
- No longer have the `D.has_key` method—the `in` membership test is used instead

Let's take a look at what's new in 3.0 dictionaries.

### Dictionary comprehensions

As mentioned at the end of the prior section, dictionaries in 3.0 can also be created with dictionary comprehensions. Like the set comprehensions we met in Chapter 5, dictionary comprehensions are available only in 3.0 (not in 2.6). Like the longstanding list comprehensions we met briefly in Chapter 4 and earlier in this chapter, they run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way.

For example, a standard way to initialize a dictionary dynamically in both 2.6 and 3.0 is to zip together its keys and values and pass the result to the `dict` call. As we'll learn in more detail in Chapter 13, the `zip` function is a way to construct a dictionary from key and value lists in a single call. If you cannot predict the set of keys and values in your code, you can always build them up as lists and zip them together:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))          # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]

>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))      # Make a dict from zip result
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

In Python 3.0, you can achieve the same effect with a dictionary comprehension expression. The following builds a new dictionary with a key/value pair for every such pair in the `zip` result (it reads almost the same in Python, but with a bit more formality):

```
C:\misc> c:\python30\python                       # Use a dict comprehension

>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

Comprehensions actually require more code in this case, but they are also more general than this example implies—we can use them to map a single stream of values to dictionaries as well, and keys can be computed with expressions just like values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}          # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'}                 # Loop over any iterable
>>> D
{'A': 'AAAA', 'P': 'PPPP', 'S': 'SSSS', 'M': 'MMMM'}

>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'ham': 'HAM!', 'spam': 'SPAM!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the `fromkeys` method we met at the end of the preceding section:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)          # Initialize dict from keys
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = {k:0 for k in ['a', 'b', 'c']}             # Same, but with a comprehension
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = dict.fromkeys('spam')                      # Other iterators, default value
>>> D
{'a': None, 'p': None, 's': None, 'm': None}

>>> D = {k: None for k in 'spam'}
>>> D
{'a': None, 'p': None, 's': None, 'm': None}
```

Like related tools, dictionary comprehensions support additional syntax not shown here, including nested loops and `if` clauses. Unfortunately, to truly understand dictionary comprehensions, we need to also know more about iteration statements and concepts in Python, and we don't yet have enough information to address that story well. We'll learn much more about all flavors of comprehensions (list, set, and dictionary) in Chapters 14 and 20, so we'll defer further details until later. We'll also study the `zip` built-in we used in this section in more detail in Chapter 13, when we explore `for` loops.

### Dictionary views

In 3.0 the dictionary `keys`, `values`, and `items` methods all return *view objects*, whereas in 2.6 they return actual result lists. View objects are *iterables*, which simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views also retain the original order of dictionary components, reflect future changes to the dictionary, and may support set operations. On the other hand, they are not lists, and they do not support operations like indexing or the list `sort` method; nor do they display their items when printed.

We'll discuss the notion of iterables more formally in Chapter 14, but for our purposes here it's enough to know that we have to run the results of these three methods through the `list` built-in if we want to apply list operations or display their values:

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

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

>>> K[0]                          # List operations fail unless converted
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'a'
```

Apart from when displaying results at the interactive prompt, you will probably rarely even notice this change, because looping constructs in Python automatically force iterable objects to produce one result on each iteration:

```
>>> for k in D.keys(): print(k)     # Iterators used automatically in loops
...
a
c
b
```

In addition, 3.0 dictionaries still have iterators themselves, which return successive keys—as in 2.6, it's still often not necessary to call keys directly:

```
>>> for key in D: print(key)        # Still no need to call keys() to iterate
...
a
c
b
```

Unlike 2.X's list results, though, dictionary views in 3.0 are not carved in stone when created—they *dynamically reflect future changes* made to the dictionary after the view object has been created:

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

>>> K = D.keys()
>>> V = D.values()
>>> list(K)                         # Views maintain same order as dictionary
['a', 'c', 'b']
>>> list(V)
[1, 3, 2]

>>> del D['b']                      # Change the dictionary in-place
>>> D
{'a': 1, 'c': 3}

>>> list(K)                         # Reflected in any current view objects
['a', 'c']
>>> list(V)                         # Not true in 2.X!
[1, 3]
```

### Dictionary views and sets

Also unlike 2.X's list results, 3.0's view objects returned by the `keys` method are *set-like* and support common set operations such as intersection and union; `values` views are not, since they aren't unique, but `items` results are if their (`key, value`) pairs are unique and hashable. Given that sets behave much like valueless dictionaries (and are even coded in curly braces like dictionaries in 3.0), this is a logical symmetry. Like dictionary keys, set items are unordered, unique, and immutable.

Here is what keys lists look like when used in set operations. In set operations, views may be mixed with other views, sets, and dictionaries (dictionaries are treated the same as their `keys` views in this context):

```
>>> K | {'x': 4}                # Keys (and some items) views are set-like
{'a', 'x', 'c'}

>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'

>>> D = {'a':1, 'b':2, 'c':3}
>>> D.keys() & D.keys()          # Intersect keys views
{'a', 'c', 'b'}
>>> D.keys() & {'b'}             # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1}          # Intersect keys and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'}   # Union keys and set
{'a', 'c', 'b', 'd'}
```

Dictionary items views are set-like too if they are hashable—that is, if they contain only immutable objects:

```
>>> D = {'a': 1}
>>> list(D.items())              # Items set-like if hashable
[('a', 1)]
>>> D.items() | D.keys()         # Union view and view
{('a', 1), 'a'}
>>> D.items() | D                # dict treated same as its keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}          # Set of key/value pairs
{('a', 1), ('d', 4), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)})    # dict accepts iterable sets too
{'a': 1, 'c': 3, 'd': 4}
```

For more details on set operations in general, see Chapter 5. Now, let's look at three other quick coding notes for 3.0 dictionaries.

### Sorting dictionary keys

First of all, because `keys` does not return a list, the traditional coding pattern for scanning a dictionary by sorted keys in 2.X won't work in 3.0. You must either convert to a list manually or use the `sorted` call introduced in Chapter 4 and earlier in this chapter on either a `keys` view or the dictionary itself:

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

>>> Ks = D.keys()                          # Sorting a view object doesn't work!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'

>>> Ks = list(Ks)                          # Force it to be a list and then sort
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> Ks = D.keys()                          # Or you can use sorted() on the keys
>>> for k in sorted(Ks): print(k, D[k])    # sorted() accepts any iterable
...                                        # sorted() returns its result
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}                   # Better yet, sort the dict directly
>>> for k in sorted(D): print(k, D[k])     # dict iterators return keys
...
a 1
b 2
c 3
```

### Dictionary magnitude comparisons no longer work

Secondly, while in Python 2.6 dictionaries may be compared for relative magnitude directly with `<`, `>`, and so on, in Python 3.0 this no longer works. However, it can be simulated by comparing sorted keys lists manually:

```
sorted(D1.items()) < sorted(D2.items())       # Like 2.6 D1 < D2
```

Dictionary equality tests still work in 3.0, though. Since we'll revisit this in the next chapter in the context of comparisons at large, we'll defer further details here.

#### The has_key method is dead: long live in!

Finally, the widely used dictionary `has_key` key presence test method is gone in 3.0. Instead, use the `in` membership expression, or a `get` with a default test (of these, `in` is generally preferred):

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D.has_key('c')                                  # 2.X only: True/False
AttributeError: 'dict' object has no attribute 'has_key'

>>> 'c' in D
True
>>> 'x' in D
False
>>> if 'c' in D: print('present', D['c'])           # Preferred in 3.0
...
present 3

>>> print(D.get('c'))
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c'])  # Another option
...
present 3
```

If you work in 2.6 and care about 3.0 compatibility, note that the first two changes (comprehensions and views) can only be coded in 3.0, but the last three (`sorted`, manual comparisons, and `in`) can be coded in 2.6 today to ease 3.0 migration in the future.

## Chapter Summary

In this chapter, we explored the list and dictionary types—probably the two most common, flexible, and powerful collection types you will see and use in Python code. We learned that the list type supports positionally ordered collections of arbitrary objects, and that it may be freely nested and grown and shrunk on demand. The dictionary type is similar, but it stores items by key instead of by position and does not maintain any reliable left-to-right order among its items. Both lists and dictionaries are mutable, and so support a variety of in-place change operations not available for strings: for example, lists can be grown by `append` calls, and dictionaries by assignment to new keys.

In the next chapter, we will wrap up our in-depth core object type tour by looking at tuples and files. After that, we'll move on to statements that code the logic that processes our objects, taking us another step toward writing complete programs. Before we tackle those topics, though, here are some chapter quiz questions to review.

## Test Your Knowledge: Quiz

1. Name two ways to build a list containing five integer zeros.
2. Name two ways to build a dictionary with two keys, `'a'` and `'b'`, each having an associated value of `0`.
3. Name four operations that change a list object in-place.
4. Name four operations that change a dictionary object in-place.

## Test Your Knowledge: Answers

1. A literal expression like `[0, 0, 0, 0, 0]` and a repetition expression like `[0] * 5` will each create a list of five zeros. In practice, you might also build one up with a loop that starts with an empty list and appends `0` to it in each iteration:

   `L.append(0)`. A list comprehension (`[0 for i in range(5)]`) could work here, too, but this is more work than you need to do.
2. A literal expression such as `{'a': 0, 'b': 0}` or a series of assignments like `D = {}`, `D['a'] = 0`, and `D['b'] = 0` would create the desired dictionary. You can also use the newer and simpler-to-code `dict(a=0, b=0)` keyword form, or the more flexible `dict([('a', 0), ('b', 0)])` key/value sequences form. Or, because all the values are the same, you can use the special form `dict.fromkeys('ab', 0)`. In 3.0, you can also use a dictionary comprehension: `{k:0 for k in 'ab'}`.
3. The `append` and `extend` methods grow a list in-place, the `sort` and `reverse` methods order and reverse lists, the `insert` method inserts an item at an offset, the `remove` and `pop` methods delete from a list by value and by position, the `del` statement deletes an item or slice, and index and slice assignment statements replace an item or entire section. Pick any four of these for the quiz.
4. Dictionaries are primarily changed by assignment to a new or existing key, which creates or changes the key's entry in the table. Also, the `del` statement deletes a key's entry, the dictionary `update` method merges one dictionary into another in-place, and `D.pop(key)` removes a key and returns the value it had. Dictionaries also have other, more exotic in-place change methods not listed in this chapter, such as `setdefault`; see reference sources for more details.

CHAPTER 9

# Tuples, Files, and Everything Else

This chapter rounds out our in-depth look at the core object types in Python by exploring the *tuple*, a collection of other objects that cannot be changed, and the *file*, an interface to external files on your computer. As you'll see, the tuple is a relatively simple object that largely performs operations you've already learned about for strings and lists. The file object is a commonly used and full-featured tool for processing files; the basic overview of files here is supplemented by larger examples in later chapters.

This chapter also concludes this part of the book by looking at properties common to all the core object types we've met—the notions of equality, comparisons, object copies, and so on. We'll also briefly explore other object types in the Python toolbox; as you'll see, although we've covered all the primary built-in types, the object story in Python is broader than I've implied thus far. Finally, we'll close this part of the book by taking a look at a set of common object type pitfalls and exploring some exercises that will allow you to experiment with the ideas you've learned.

## Tuples

The last collection type in our survey is the Python tuple. Tuples construct simple groups of objects. They work exactly like lists, except that tuples can't be changed in-place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Although they don't support as many methods, tuples share most of their properties with lists. Here's a quick look at the basics. Tuples are:

*Ordered collections of arbitrary objects*
: Like strings and lists, tuples are positionally ordered collections of objects (i.e., they maintain a left-to-right order among their contents); like lists, they can embed any kind of object.

*Accessed by offset*
: Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

*Of the category "immutable sequence"*
: Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don't support any of the in-place change operations applied to lists.

*Fixed-length, heterogeneous, and arbitrarily nestable*
: Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so support arbitrary nesting.

*Arrays of object references*
: Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick.

Table 9-1 highlights common tuple operations. A tuple is written as a series of objects (technically, expressions that generate objects), separated by commas and normally enclosed in parentheses. An empty tuple is just a parentheses pair with nothing inside.

*Table 9-1. Common tuple literals and operations*

| Operation | Interpretation |
|---|---|
| `()` | An empty tuple |
| `T = (0,)` | A one-item tuple (not an expression) |
| `T = (0, 'Ni', 1.2, 3)` | A four-item tuple |
| `T = 0, 'Ni', 1.2, 3` | Another four-item tuple (same as prior line) |
| `T = ('abc', ('def', 'ghi'))` | Nested tuples |
| `T = tuple('spam')` | Tuple of items in an iterable |
| `T[i]` | Index, index of index, slice, length |
| `T[i][j]` | |
| `T[i:j]` | |
| `len(T)` | |
| `T1 + T2` | Concatenate, repeat |
| `T * 3` | |
| `for x in T: print(x)` | Iteration, membership |
| `'spam' in T` | |
| `[x ** 2 for x in T]` | |
| `T.index('Ni')` | Methods in 2.6 and 3.0: search, count |
| `T.count('Ni')` | |

## Tuples in Action

As usual, let's start an interactive session to explore tuples at work. Notice in Table 9-1 that tuples do not have all the methods that lists have (e.g., an `append` call won't work here). They do, however, support the usual sequence operations that we saw for both strings and lists:

```
>>> (1, 2) + (3, 4)             # Concatenation
(1, 2, 3, 4)

>>> (1, 2) * 4                  # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)            # Indexing, slicing
>>> T[0], T[1:3]
(1, (2, 3))
```

### Tuple syntax peculiarities: Commas and parentheses

The second and fourth entries in Table 9-1 merit a bit more explanation. Because parentheses can also enclose expressions (see Chapter 5), you need to do something special to tell Python when a single object in parentheses is a tuple object and not a simple expression. If you really want a single-item tuple, simply add a trailing comma after the single item, before the closing parenthesis:

```
>>> x = (40)                    # An integer!
>>> x
40
>>> y = (40,)                   # A tuple containing an integer
>>> y
(40,)
```

As a special case, Python also allows you to omit the opening and closing parentheses for a tuple in contexts where it isn't syntactically ambiguous to do so. For instance, the fourth line of Table 9-1 simply lists four items separated by commas. In the context of an assignment statement, Python recognizes this as a tuple, even though it doesn't have parentheses.

Now, some people will tell you to always use parentheses in your tuples, and some will tell you to never use parentheses in tuples (and still others have lives, and won't tell you what to do with your tuples!). The only significant places where the parentheses are *required* are when a tuple is passed as a literal in a function call (where parentheses matter), and when one is listed in a Python 2.X `print` statement (where commas are significant).

For beginners, the best advice is that it's probably easier to use the parentheses than it is to figure out when they are optional. Many programmers (myself included) also find that parentheses tend to aid script readability by making the tuples more explicit, but your mileage may vary.

### Conversions, methods, and immutability

Apart from literal syntax differences, tuple operations (the middle rows in Table 9-1) are identical to string and list operations. The only differences worth noting are that the +, *, and slicing operations return new *tuples* when applied to tuples, and that tuples don't provide the same methods you saw for strings, lists, and dictionaries. If you want to sort a tuple, for example, you'll usually have to either first convert it to a list to gain access to a sorting method call and make it a mutable object, or use the newer `sorted` built-in that accepts any sequence object (and more):

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)                  # Make a list from a tuple's items
>>> tmp.sort()                     # Sort the list
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)                 # Make a tuple from the list's items
>>> T
('aa', 'bb', 'cc', 'dd')

>>> sorted(T)                      # Or use the sorted built-in
['aa', 'bb', 'cc', 'dd']
```

Here, the `list` and `tuple` built-in functions are used to convert the object to a list and then back to a tuple; really, both calls make new objects, but the net effect is like a conversion.

List comprehensions can also be used to convert tuples. The following, for example, makes a list from a tuple, adding 20 to each item along the way:

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

List comprehensions are really sequence operations—they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, and other lists. As we'll see later in the book, they even work on some things that are not physically stored sequences—any iterable objects will do, including files, which are automatically read line by line.

Although tuples don't have the same methods as lists and strings, they do have two of their own as of Python 2.6 and 3.0—`index` and `count` works as they do for lists, but they are defined for tuple objects:

```
>>> T = (1, 2, 3, 2, 4, 2)         # Tuple methods in 2.6 and 3.0
>>> T.index(2)                     # Offset of first appearance of 2
1
>>> T.index(2, 2)                  # Offset of appearance after offset 2
3
>>> T.count(2)                     # How many 2s are there?
3
```

Prior to 2.6 and 3.0, tuples have no methods at all—this was an old Python convention for immutable types, which was violated years ago on grounds of practicality with strings, and more recently with both numbers and tuples.

Also, note that the rule about tuple *immutability* applies only to the top level of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual:

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'                  # This fails: can't change tuple itself
TypeError: object doesn't support item assignment

>>> T[1][0] = 'spam'               # This works: can change mutables inside
>>> T
(1, ['spam', 3], 4)
```

For most programs, this one-level-deep immutability is sufficient for common tuple roles. Which, coincidentally, brings us to the next section.

## Why Lists and Tuples?

This seems to be the first question that always comes up when teaching beginners about tuples: why do we need tuples if we have lists? Some of the reasoning may be historic; Python's creator is a mathematician by training, and he has been quoted as seeing a tuple as a simple association of objects and a list as a data structure that changes over time. In fact, this use of the word "tuple" derives from mathematics, as does its frequent use for a row in a relational database table.

The best answer, however, seems to be that the immutability of tuples provides some *integrity*—you can be sure a tuple won't be changed through another reference elsewhere in a program, but there's no such guarantee for lists. Tuples, therefore, serve a similar role to "constant" declarations in other languages, though the notion of constantness is associated with objects in Python, not variables.

Tuples can also be used in places that lists cannot—for example, as dictionary keys (see the sparse matrix example in Chapter 8). Some built-in operations may also require or imply tuples, not lists, though such operations have often been generalized in recent years. As a rule of thumb, lists are the tool of choice for ordered collections that might need to change; tuples can handle the other cases of fixed associations.

# Files

You may already be familiar with the notion of files, which are named storage compartments on your computer that are managed by your operating system. The last major built-in object type that we'll examine on our object types tour provides a way to access those files inside Python programs.

In short, the built-in `open` function creates a Python file object, which serves as a link to a file residing on your machine. After calling `open`, you can transfer strings of data to and from the associated external file by calling the returned file object's methods.

Compared to the types you've seen so far, file objects are somewhat unusual. They're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks. Most file methods are concerned with performing input from and output to the external file associated with a file object, but other file methods allow us to seek to a new position in the file, flush output buffers, and so on. Table 9-2 summarizes common file operations.

*Table 9-2. Common file operations*

| Operation | Interpretation |
|---|---|
| `output = open(r'C:\spam', 'w')` | Create output file (`'w'` means write) |
| `input = open('data', 'r')` | Create input file (`'r'` means read) |
| `input = open('data')` | Same as prior line (`'r'` is the default) |
| `aString = input.read()` | Read entire file into a single string |
| `aString = input.read(N)` | Read up to next N characters (or bytes) into a string |
| `aString = input.readline()` | Read next line (including `\n` newline) into a string |
| `aList = input.readlines()` | Read entire file into list of line strings (with `\n`) |
| `output.write(aString)` | Write a string of characters (or bytes) into file |
| `output.writelines(aList)` | Write all line strings in a list into file |
| `output.close()` | Manual close (done for you when file is collected) |
| `output.flush()` | Flush output buffer to disk without closing |
| `anyFile.seek(N)` | Change file position to offset N for next operation |
| `for line in open('data'):` *`use line`* | File iterators read line by line |
| `open('f.txt', encoding='latin-1')` | Python 3.0 Unicode text files (`str` strings) |
| `open('f.bin', 'rb')` | Python 3.0 binary bytes files (`bytes` strings) |

## Opening Files

To open a file, a program calls the built-in `open` function, with the external filename first, followed by a processing *mode*. The mode is typically the string `'r'` to open for text input (the default), `'w'` to create and open for text output, or `'a'` to open for appending text to the end. The processing mode argument can specify additional options:

- Adding a `b` to the mode string allows for *binary* data (end-of-line translations and 3.0 Unicode encodings are turned off).

- Adding a `+` opens the file for *both* input and output (i.e., you can both read and write to the same file object, often in conjunction with seek operations to reposition in the file).

Both arguments to `open` must be Python strings, and an optional third argument can be used to control output buffering—passing a zero means that output is unbuffered (it is transferred to the external file immediately on a write method call). The external filename argument may include a platform-specific and absolute or relative directory path prefix; without a directory path, the file is assumed to exist in the current working directory (i.e., where the script runs). We'll cover file fundamentals and explore some basic examples here, but we won't go into all file-processing mode options; as usual, consult the Python library manual for additional details.

## Using Files

Once you make a file object with `open`, you can call its methods to read from or write to the associated external file. In all cases, file text takes the form of strings in Python programs; reading a file returns its text in strings, and text is passed to the write methods as strings. Reading and writing methods come in multiple flavors; Table 9-2 lists the most common. Here are a few fundamental usage notes:

*File iterators are best for reading lines*
: Though the reading and writing methods in the table are common, keep in mind that probably the best way to read lines from a text file today is to not read the file at all—as we'll see in Chapter 14, files also have an *iterator* that automatically reads one line at a time in a `for` loop, list comprehension, or other iteration context.

*Content is strings, not objects*
: Notice in Table 9-2 that data read from a file always comes back to your script as a string, so you'll have to convert it to a different type of Python object if a string is not what you need. Similarly, unlike with the `print` operation, Python does not add any formatting and does not convert objects to strings automatically when you write data to a file—you must send an already formatted string. Because of this, the tools we have already met to convert objects to and from strings (e.g., `int`, `float`, `str`, and the string formatting expression and method) come in handy when dealing with files. Python also includes advanced standard library tools for handling generic object storage (such as the `pickle` module) and for dealing with packed binary data in files (such as the `struct` module). We'll see both of these at work later in this chapter.

`close` *is usually optional*
: Calling the file `close` method terminates your connection to the external file. As discussed in Chapter 6, in Python an object's memory space is automatically reclaimed as soon as the object is no longer referenced anywhere in the program. When file objects are reclaimed, Python also automatically closes the files if they are still open (this also happens when a program shuts down). This means you

don't always need to manually close your files, especially in simple scripts that don't run for long. On the other hand, including manual `close` calls can't hurt and is usually a good idea in larger systems. Also, strictly speaking, this auto-close-on-collection feature of files is not part of the language definition, and it may change over time. Consequently, manually issuing file `close` method calls is a good habit to form. (For an alternative way to guarantee automatic file closes, also see this section's later discussion of the file object's *context manager*, used with the new `with`/`as` statement in Python 2.6 and 3.0.)

*Files are buffered and seekable.*

The prior paragraph's notes about closing files are important, because closing both frees up operating system resources and flushes output buffers. By default, output files are always buffered, which means that text you write may not be transferred from memory to disk immediately—closing a file, or running its `flush` method, forces the buffered data to disk. You can avoid buffering with extra `open` arguments, but it may impede performance. Python files are also random-access on a byte offset basis—their `seek` method allows your scripts to jump around to read and write at specific locations.

## Files in Action

Let's work through a simple example that demonstrates file-processing basics. The following code begins by opening a new text file for output, writing two lines (strings terminated with a newline marker, `\n`), and closing the file. Later, the example opens the same file again in input mode and reads the lines back one at a time with `readline`. Notice that the third `readline` call returns an empty string; this is how Python file methods tell you that you've reached the end of the file (empty lines in the file come back as strings containing just a newline character, not as empty strings). Here's the complete interaction:

```
>>> myfile = open('myfile.txt', 'w')         # Open for text output: create/empty
>>> myfile.write('hello text file\n')         # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                           # Flush output buffers to disk

>>> myfile = open('myfile.txt')              # Open for text input: 'r' is default
>>> myfile.readline()                        # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                        # Empty string: end of file
''
```

Notice that file `write` calls return the number of characters written in Python 3.0; in 2.6 they don't, so you won't see these numbers echoed interactively. This example writes each line of text, including its end-of-line terminator, `\n`, as a string; write

methods don't add the end-of-line character for us, so we must include it to properly terminate our lines (otherwise the next write will simply extend the current line in the file).

If you want to display the file's content with end-of-line characters interpreted, read the entire file into a string *all at once* with the file object's `read` method and print it:

```
>>> open('myfile.txt').read()               # Read all at once into string
'hello text file\ngoodbye text file\n'

>>> print(open('myfile.txt').read())        # User-friendly display
hello text file
goodbye text file
```

And if you want to scan a text file line by line, *file iterators* are often your best option:

```
>>> for line in open('myfile'):             # Use file iterators, not reads
...     print(line, end='')
...
hello text file
goodbye text file
```

When coded this way, the temporary file object created by `open` will automatically read and return one line on each loop iteration. This form is usually easiest to code, good on memory use, and may be faster than some other options (depending on many variables, of course). Since we haven't reached statements or iterators yet, though, you'll have to wait until Chapter 14 for a more complete explanation of this code.

### Text and binary files in Python 3.0

Strictly speaking, the example in the prior section uses text files. In both Python 3.0 and 2.6, file type is determined by the second argument to `open`, the mode string—an included "b" means binary. Python has always supported both text and binary files, but in Python 3.0 there is a sharper distinction between the two:

- *Text files* represent content as normal `str` strings, perform Unicode encoding and decoding automatically, and perform end-of-line translation by default.
- *Binary files* represent content as a special `bytes` string type and allow programs to access file content unaltered.

In contrast, Python 2.6 text files handle both 8-bit text and binary data, and a special string type and file interface (`unicode` strings and `codecs.open`) handles Unicode text. The differences in Python 3.0 stem from the fact that simple and Unicode text have been merged in the normal string type—which makes sense, given that all text is Unicode, including ASCII and other 8-bit encodings.

Because most programmers deal only with ASCII text, they can get by with the basic text file interface used in the prior example, and normal strings. All strings are technically Unicode in 3.0, but ASCII users will not generally notice. In fact, files and strings work the same in 3.0 and 2.6 if your script's scope is limited to such simple forms of text.

If you need to handle internationalized applications or byte-oriented data, though, the distinction in 3.0 impacts your code (usually for the better). In general, you must use `bytes` strings for binary files, and normal `str` strings for text files. Moreover, because text files implement Unicode encodings, you cannot open a binary data file in text mode—decoding its content to Unicode text will likely fail.

Let's look at an example. When you read a binary data file you get back a `bytes` object—a sequence of small integers that represent absolute byte values (which may or may not correspond to characters), which looks and feels almost exactly like a normal string:

```
>>> data = open('data.bin', 'rb').read()     # Open binary file: rb=read binary
>>> data                                     # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]                                # Act like strings
b'spam'
>>> data[0]                                  # But really are small 8-bit integers
115
>>> bin(data[0])                             # Python 3.0 bin() function
'0b1110011'
```

In addition, binary files do not perform any end-of-line translation on data; text files by default map all forms to and from `\n` when written and read and implement Unicode encodings on transfers. Since Unicode and binary data is of marginal interest to many Python programmers, we'll postpone the full story until Chapter 36. For now, let's move on to some more substantial file examples.

### Storing and parsing Python objects in files

Our next example writes a variety of Python objects into a text file on multiple lines. Notice that it must convert objects to strings using conversion tools. Again, file data is always strings in our scripts, and write methods do not do any automatic to-string formatting for us (for space, I'm omitting byte-count return values from `write` methods from here on):

```
>>> X, Y, Z = 43, 44, 45                     # Native Python objects
>>> S = 'Spam'                               # Must be strings to store in file
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w')            # Create output file
>>> F.write(S + '\n')                        # Terminate lines with \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z))        # Convert numbers to strings
>>> F.write(str(L) + '$' + str(D) + '\n')    # Convert and separate with $
>>> F.close()
```

Once we have created our file, we can inspect its contents by opening it and reading it into a string (a single operation). Notice that the interactive echo gives the exact byte contents, while the `print` operation interprets embedded end-of-line characters to render a more user-friendly display:

```
>>> chars = open('datafile.txt').read()      # Raw string display
>>> chars
```

```
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)                          # User-friendly display
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

We now have to use other conversion tools to translate from the strings in the text file to real Python objects. As Python never converts strings to numbers (or other types of objects) automatically, this is required if we need to gain access to normal object tools like indexing, addition, and so on:

```
>>> F = open('datafile.txt')             # Open again
>>> line = F.readline()                  # Read one line
>>> line
'Spam\n'
>>> line.rstrip()                        # Remove end-of-line
'Spam'
```

For this first line, we used the string `rstrip` method to get rid of the trailing end-of-line character; a `line[:-1]` slice would work, too, but only if we can be sure all lines end in the `\n` character (the last line in a file sometimes does not).

So far, we've read the line containing the string. Now let's grab the next line, which contains numbers, and parse out (that is, extract) the objects on that line:

```
>>> line = F.readline()                  # Next line from file
>>> line                                 # It's a string here
'43,44,45\n'
>>> parts = line.split(',')              # Split (parse) on commas
>>> parts
['43', '44', '45\n']
```

We used the string `split` method here to chop up the line on its comma delimiters; the result is a list of substrings containing the individual numbers. We still must convert from strings to integers, though, if we wish to perform math on these:

```
>>> int(parts[1])                        # Convert from string to int
44
>>> numbers = [int(P) for P in parts]    # Convert all in list at once
>>> numbers
[43, 44, 45]
```

As we have learned, `int` translates a string of digits into an integer object, and the list comprehension expression introduced in Chapter 4 can apply the call to each item in our list all at once (you'll find more on list comprehensions later in this book). Notice that we didn't have to run `rstrip` to delete the `\n` at the end of the last part; `int` and some other converters quietly ignore whitespace around digits.

Finally, to convert the stored list and dictionary in the third line of the file, we can run them through `eval`, a built-in function that treats a string as a piece of executable program code (technically, a string containing a Python expression):

```
>>> line = F.readline()
>>> line
```

```
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$')                    # Split (parse) on $
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> eval(parts[0])                             # Convert to any object type
[1, 2, 3]
>>> objects = [eval(P) for P in parts]         # Do same for all in list
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Because the end result of all this parsing and converting is a list of normal Python objects instead of strings, we can now apply list and dictionary operations to them in our script.

### Storing native Python objects with pickle

Using `eval` to convert from strings to objects, as demonstrated in the preceding code, is a powerful tool. In fact, sometimes it's *too* powerful. `eval` will happily run any Python expression—even one that might delete all the files on your computer, given the necessary permissions! If you really want to store native Python objects, but you can't trust the source of the data in the file, Python's standard library `pickle` module is ideal.

The `pickle` module is an advanced tool that allows us to store almost any Python object in a file directly, with no to- or from-string conversion requirement on our part. It's like a super-general data formatting and parsing utility. To store a dictionary in a file, for instance, we pickle it directly:

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                          # Pickle any object to file
>>> F.close()
```

Then, to get the dictionary back later, we simply use `pickle` again to re-create it:

```
>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)                         # Load any object from file
>>> E
{'a': 1, 'b': 2}
```

We get back an equivalent dictionary object, with no manual splitting or converting required. The `pickle` module performs what is known as *object serialization*—converting objects to and from strings of bytes—but requires very little work on our part. In fact, `pickle` internally translates our dictionary to a string form, though it's not much to look at (and may vary if we pickle in other data protocol modes):

```
>>> open('datafile.pkl', 'rb').read()          # Format is prone to change!
b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'
```

Because `pickle` can reconstruct the object from this format, we don't have to deal with that ourselves. For more on the `pickle` module, see the Python standard library manual, or import `pickle` and pass it to `help` interactively. While you're exploring, also take a look at the `shelve` module. `shelve` is a tool that uses `pickle` to store Python objects in an access-by-key filesystem, which is beyond our scope here (though you will get to see

an example of `shelve` in action in Chapter 27, and other `pickle` examples in Chapters 30 and 36).

Note that I opened the file used to store the pickled object in *binary mode*; binary mode is always required in Python 3.0, because the pickler creates and uses a `bytes` string object, and these objects imply binary-mode files (text-mode files imply `str` strings in 3.0). In earlier Pythons it's OK to use text-mode files for protocol 0 (the default, which creates ASCII text), as long as text mode is used consistently; higher protocols require binary-mode files. Python 3.0's default protocol is 3 (binary), but it creates `bytes` even for protocol 0. See Chapter 36, Python's library manual, or reference books for more details on this.

Python 2.6 also has a `cPickle` module, which is an optimized version of `pickle` that can be imported directly for speed. Python 3.0 renames this module `_pickle` and uses it automatically in `pickle`—scripts simply import `pickle` and let Python optimize itself.

### Storing and parsing packed binary data in files

One other file-related note before we move on: some advanced applications also need to deal with packed binary data, created perhaps by a C language program. Python's standard library includes a tool to help in this domain—the `struct` module knows how to both compose and parse packed binary data. In a sense, this is another data-conversion tool that interprets strings in files as binary data.

To create a packed binary data file, for example, open it in `'wb'` (write binary) mode, and pass `struct` a format string and some Python objects. The format string used here means pack as a 4-byte integer, a 4-character string, and a 2-byte integer, all in big-endian form (other format codes handle padding bytes, floating-point numbers, and more):

```
>>> F = open('data.bin', 'wb')                    # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'spam', 8)     # Make packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                                 # Write byte string
>>> F.close()
```

Python creates a binary `bytes` data string, which we write out to the file normally—this one consists mostly of nonprintable characters printed in hexadecimal escapes, and is the same binary file we met earlier. To parse the values out to normal Python objects, we simply read the string back and unpack it using the same format string. Python extracts the values into normal Python objects—integers and a string:

```
>>> F = open('data.bin', 'rb')
>>> data = F.read()                               # Get packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
```

```
>>> values = struct.unpack('>i4sh', data)          # Convert to Python objects
>>> values
(7, 'spam', 8)
```

Binary data files are advanced and somewhat low-level tools that we won't cover in more detail here; for more help, see Chapter 36, consult the Python library manual, or import `struct` and pass it to the `help` function interactively. Also note that the binary file-processing modes `'wb'` and `'rb'` can be used to process a simpler binary file such as an image or audio file as a whole without having to unpack its contents.

#### File context managers

You'll also want to watch for Chapter 33's discussion of the file's *context manager* support, new in Python 3.0 and 2.6. Though more a feature of exception processing than files themselves, it allows us to wrap file-processing code in a logic layer that ensures that the file will be closed automatically on exit, instead of relying on the auto-close on garbage collection:

```
with open(r'C:\misc\data.txt') as myfile:          # See Chapter 33 for details
    for line in myfile:
        ...use line here...
```

The `try/finally` statement we'll look at in Chapter 33 can provide similar functionality, but at some cost in extra code—three extra lines, to be precise (though we can often avoid both options and let Python close files for us automatically):

```
myfile = open(r'C:\misc\data.txt')
try:
    for line in myfile:
        ...use line here...
finally:
    myfile.close()
```

Since both these options require more information than we have yet obtained, we'll postpone details until later in this book.

## Other File Tools

There are additional, more advanced file methods shown in Table 9-2, and even more that are not in the table. For instance, as mentioned earlier, `seek` resets your current position in a file (the next read or write happens at that position), `flush` forces buffered output to be written out to disk (by default, files are always buffered), and so on.

The Python standard library manual and the reference books described in the Preface provide complete lists of file methods; for a quick look, run a `dir` or `help` call interactively, passing in an open file object (in Python 2.6 but not 3.0, you can pass in the name `file` instead). For more file-processing examples, watch for the sidebar "Why You Will Care: File Scanners" on page 340. It sketches common file-scanning loop code patterns with statements we have not covered enough yet to use here.

Also, note that although the `open` function and the file objects it returns are your main interface to external files in a Python script, there are additional file-like tools in the Python toolset. Also available, to name a few, are:

*Standard streams*
: Preopened file objects in the `sys` module, such as `sys.stdout` (see "Print Operations" on page 297)

*Descriptor files in the `os` module*
: Integer file handles that support lower-level tools such as file locking

*Sockets, pipes, and FIFOs*
: File-like objects used to synchronize processes or communicate over networks

*Access-by-key files known as "shelves"*
: Used to store unaltered Python objects directly, by key (used in Chapter 27)

*Shell command streams*
: Tools such as `os.popen` and `subprocess.Popen` that support spawning shell commands and reading and writing to their standard streams

The third-party open source domain offers even more file-like tools, including support for communicating with serial ports in the *PySerial* extension and interactive programs in the *pexpect* system. See more advanced Python texts and the Web at large for additional information on file-like tools.

*Version skew note*: In Python 2.5 and earlier, the built-in name `open` is essentially a synonym for the name `file`, and files may technically be opened by calling either `open` or `file` (though `open` is generally preferred for opening). In Python 3.0, the name `file` is no longer available, because of its redundancy with `open`.

Python 2.6 users may also use the name `file` as the file object type, in order to customize files with object-oriented programming (described later in this book). In Python 3.0, files have changed radically. The classes used to implement file objects live in the standard library module `io`. See this module's documentation or code for the classes it makes available for customization, and run a `type(F)` call on open files `F` for hints.

# Type Categories Revisited

Now that we've seen all of Python's core built-in types in action, let's wrap up our object types tour by reviewing some of the properties they share. Table 9-3 classifies all the major types we've seen so far according to the type categories introduced earlier. Here are some points to remember:

- Objects share operations according to their category; for instance, strings, lists, and tuples all share sequence operations such as concatenation, length, and indexing.
- Only mutable objects (lists, dictionaries, and sets) may be changed in-place; you cannot change numbers, strings, or tuples in-place.
- Files export only methods, so mutability doesn't really apply to them—their state may be changed when they are processed, but this isn't quite the same as Python core type mutability constraints.
- "Numbers" in Table 9-3 includes all number types: integer (and the distinct long integer in 2.6), floating-point, complex, decimal, and fraction.
- "Strings" in Table 9-3 includes `str`, as well as `bytes` in 3.0 and `unicode` in 2.6; the `bytearray` string type in 3.0 is mutable.
- Sets are something like the keys of a valueless dictionary, but they don't map to values and are not ordered, so sets are neither a mapping nor a sequence type; `frozenset` is an immutable variant of `set`.
- In addition to type category operations, as of Python 2.6 and 3.0 all the types in Table 9-3 have callable methods, which are generally specific to their type.

*Table 9-3. Object classifications*

| Object type | Category | Mutable? |
|---|---|---|
| Numbers (all) | Numeric | No |
| Strings | Sequence | No |
| Lists | Sequence | Yes |
| Dictionaries | Mapping | Yes |
| Tuples | Sequence | No |
| Files | Extension | N/A |
| Sets | Set | Yes |
| `frozenset` | Set | No |
| `bytearray` (3.0) | Sequence | Yes |

## Why You Will Care: Operator Overloading

In Part VI of this book, we'll see that objects we implement with classes can pick and choose from these categories arbitrarily. For instance, if we want to provide a new kind of specialized sequence object that is consistent with built-in sequences, we can code a class that overloads things like indexing and concatenation:

```
class MySequence:
     def __getitem__(self, index):
         # Called on self[index], others
     def __add__(self, other):
         # Called on self + other
```

and so on. We can also make the new object mutable or not by selectively implementing methods called for in-place change operations (e.g., `__setitem__` is called on `self[index]=value` assignments). Although it's beyond this book's scope, it's also possible to implement new objects in an external language like C as C extension types. For these, we fill in C function pointer slots to choose between number, sequence, and mapping operation sets.

## Object Flexibility

This part of the book introduced a number of compound object types (collections with components). In general:

- Lists, dictionaries, and tuples can hold any kind of object.
- Lists, dictionaries, and tuples can be arbitrarily nested.
- Lists and dictionaries can dynamically grow and shrink.

Because they support arbitrary structures, Python's compound object types are good at representing complex information in programs. For example, values in dictionaries may be lists, which may contain tuples, which may contain dictionaries, and so on. The nesting can be as deep as needed to model the data to be processed.

Let's look at an example of nesting. The following interaction defines a tree of nested compound sequence objects, shown in Figure 9-1. To access its components, you may include as many index operations as required. Python evaluates the indexes from left to right, and fetches a reference to a more deeply nested object at each step. Figure 9-1 may be a pathologically complicated data structure, but it illustrates the syntax used to access nested objects in general:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

## References Versus Copies

Chapter 6 mentioned that assignments always store references to objects, not copies of those objects. In practice, this is usually what you want. Because assignments can generate multiple references to the same object, though, it's important to be aware that changing a mutable object in-place may affect other references to the same object

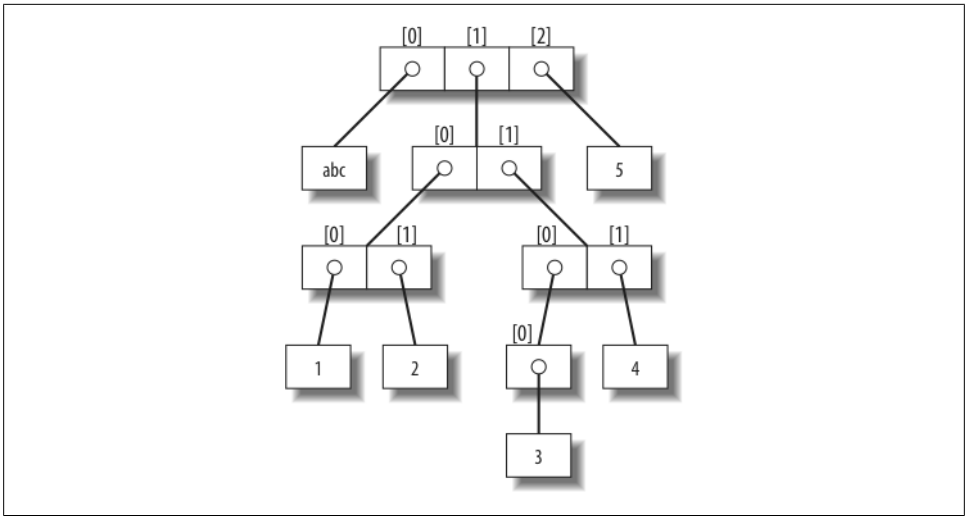


*Figure 9-1. A nested object tree with the offsets of its components, created by running the literal expression ['abc', [(1, 2), ([3], 4)], 5]. Syntactically nested objects are internally represented as references (i.e., pointers) to separate pieces of memory.*

elsewhere in your program. If you don't want such behavior, you'll need to tell Python to copy the object explicitly.

We studied this phenomenon in Chapter 6, but it can become more subtle when larger objects come into play. For instance, the following example creates a list assigned to `X`, and another list assigned to `L` that embeds a reference back to list `X`. It also creates a dictionary `D` that contains another reference back to list `X`:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']              # Embed references to X's object
>>> D = {'x':X, 'y':2}
```

At this point, there are three references to the first list created: from the name `X`, from inside the list assigned to `L`, and from inside the dictionary assigned to `D`. The situation is illustrated in Figure 9-2.

Because lists are mutable, changing the shared list object from any of the three references also changes what the other two reference:

```
>>> X[1] = 'surprise'              # Changes all three references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

References are a higher-level analog of pointers in other languages. Although you can't grab hold of the reference itself, it's possible to store the same reference in more than one place (variables, lists, and so on). This is a feature—you can pass a large object

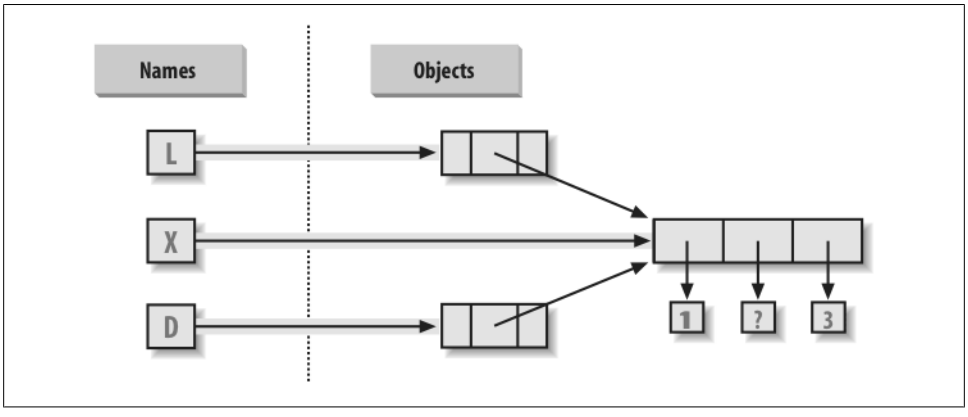


*Figure 9-2. Shared object references: because the list referenced by variable X is also referenced from within the objects referenced by L and D, changing the shared list from X makes it look different from L and D, too.*

around a program without generating expensive copies of it along the way. If you really do want copies, however, you can request them:

- Slice expressions with empty limits (`L[:]`) copy sequences.
- The dictionary and set `copy` method (`X.copy()`) copies a dictionary or set.
- Some built-in functions, such as `list`, make copies (`list(L)`).
- The `copy` standard library module makes full copies.

For example, say you have a list and a dictionary, and you don't want their values to be changed through other variables:

```
>>> L = [1,2,3]
>>> D = {'a':1, 'b':2}
```

To prevent this, simply assign copies to the other variables, not references to the same objects:

```
>>> A = L[:]                    # Instead of A = L (or list(L))
>>> B = D.copy()                # Instead of B = D (ditto for sets)
```

This way, changes made from the other variables will change the copies, not the originals:

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

In terms of our original example, you can avoid the reference side effects by slicing the original list instead of simply naming it:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']          # Embed copies of X's object
>>> D = {'x':X[:], 'y':2}
```

This changes the picture in Figure 9-2—`L` and `D` will now point to different lists than `X`. The net effect is that changes made through `X` will impact only `X`, not `L` and `D`; similarly, changes to `L` or `D` will not impact `X`.

One final note on copies: empty-limit slices and the dictionary `copy` method only make *top-level* copies; that is, they do not copy nested data structures, if any are present. If you need a complete, fully independent copy of a deeply nested data structure, use the standard `copy` module: include an `import copy` statement and say `X = copy.deepcopy(Y)` to fully copy an arbitrarily nested object `Y`. This call recursively traverses objects to copy all their parts. This is a much more rare case, though (which is why you have to say more to make it go). References are usually what you will want; when they are not, slices and copy methods are usually as much copying as you'll need to do.

## Comparisons, Equality, and Truth

All Python objects also respond to comparisons: tests for equality, relative magnitude, and so on. Python comparisons always inspect all parts of compound objects until a result can be determined. In fact, when nested objects are present, Python automatically traverses data structures to apply comparisons *recursively* from left to right, and as deeply as needed. The first difference found along the way determines the comparison result.

For instance, a comparison of list objects compares all their components automatically:

```
>>> L1 = [1, ('a', 3)]           # Same value, unique objects
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2           # Equivalent? Same object?
(True, False)
```

Here, `L1` and `L2` are assigned lists that are equivalent but distinct objects. Because of the nature of Python references (studied in Chapter 6), there are two ways to test for equality:

- **The `==` operator tests value equivalence**. Python performs an equivalence test, comparing all nested objects recursively.
- **The `is` operator tests object identity**. Python tests whether the two are really the same object (i.e., live at the same address in memory).

In the preceding example, `L1` and `L2` pass the `==` test (they have equivalent values because all their components are equivalent) but fail the `is` check (they reference two different objects, and hence two different pieces of memory). Notice what happens for short strings, though:

```
>>> S1 = 'spam'
>>> S2 = 'spam'
```

```
>>> S1 == S2, S1 is S2
(True, True)
```

Here, we should again have two distinct objects that happen to have the same value: `==` should be true, and `is` should be false. But because Python internally caches and reuses some strings as an optimization, there really is just a single string `'spam'` in memory, shared by `S1` and `S2`; hence, the `is` identity test reports a true result. To trigger the normal behavior, we need to use longer strings:

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

Of course, because strings are immutable, the object caching mechanism is irrelevant to your code—strings can't be changed in-place, regardless of how many variables refer to them. If identity tests seem confusing, see Chapter 6 for a refresher on object reference concepts.

As a rule of thumb, the `==` operator is what you will want to use for almost all equality checks; `is` is reserved for highly specialized roles. We'll see cases where these operators are put to use later in the book.

Relative magnitude comparisons are also applied recursively to nested data structures:

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2          # Less, equal, greater: tuple of results
(False, False, True)
```

Here, `L1` is greater than `L2` because the nested `3` is greater than `2`. The result of the last line is really a tuple of three objects—the results of the three expressions typed (an example of a tuple without its enclosing parentheses).

In general, Python compares types as follows:

- Numbers are compared by relative magnitude.
- Strings are compared lexicographically, character by character (`"abc" < "ac"`).
- Lists and tuples are compared by comparing each component from left to right.
- Dictionaries compare as equal if their sorted (*`key, value`*) lists are equal. Relative magnitude comparisons are not supported for dictionaries in Python 3.0, but they work in 2.6 and earlier as though comparing sorted (*`key, value`*) lists.
- Nonnumeric mixed-type comparisons (e.g., `1 < 'spam'`) are errors in Python 3.0. They are allowed in Python 2.6, but use a fixed but arbitrary ordering rule. By proxy, this also applies to sorts, which use comparisons internally: nonnumeric mixed-type collections cannot be sorted in 3.0.

In general, comparisons of structured objects proceed as though you had written the objects as literals and compared all their parts one at a time from left to right. In later chapters, we'll see other object types that can change the way they get compared.

## Python 3.0 Dictionary Comparisons

The second to last point in the preceding section merits illustration. In Python 2.6 and earlier, dictionaries support magnitude comparisons, as though you were comparing sorted key/value lists:

```
C:\misc> c:\python26\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
True
```

In Python 3.0, magnitude comparisons for dictionaries are removed because they incur too much overhead when equality is desired (equality uses an optimized scheme in 3.0 that doesn't literally compare sorted key/value lists). The alternative in 3.0 is to either write loops to compare values by key or compare the sorted key/value lists manually—the `items` dictionary methods and `sorted` built-in suffice:

```
C:\misc> c:\python30\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()

>>> list(D1.items())
[('a', 1), ('b', 2)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]

>>> sorted(D1.items()) < sorted(D2.items())
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

In practice, most programs requiring this behavior will develop more efficient ways to compare data in dictionaries than either this workaround or the original behavior in Python 2.6.

## The Meaning of True and False in Python

Notice that the test results returned in the last two examples represent true and false values. They print as the words `True` and `False`, but now that we're using logical tests like these in earnest, I should be a bit more formal about what these names really mean.

In Python, as in most programming languages, an integer `0` represents false, and an integer `1` represents true. In addition, though, Python recognizes any empty data structure as false and any nonempty data structure as true. More generally, the notions of

true and false are intrinsic properties of every object in Python—each object is either true or false, as follows:

- Numbers are true if nonzero.
- Other objects are true if nonempty.

Table 9-4 gives examples of true and false objects in Python.

*Table 9-4. Example object truth values*

| Object | Value |
|---|---|
| `"spam"` | `True` |
| `""` | `False` |
| `[]` | `False` |
| `{}` | `False` |
| `1` | `True` |
| `0.0` | `False` |
| `None` | `False` |

As one application, because objects are true or false themselves, it's common to see Python programmers code tests like `if X:`, which, assuming `X` is a string, is the same as `if X != '':`. In other words, you can test the object itself, instead of comparing it to an empty object. (More on `if` statements in Part III.)

### The None object

As shown in the last item in Table 9-4, Python also provides a special object called `None`, which is always considered to be false. `None` was introduced in Chapter 4; it is the only value of a special data type in Python and typically serves as an empty placeholder (much like a `NULL` pointer in C).

For example, recall that for lists you cannot assign to an offset unless that offset already exists (the list does not magically grow if you make an out-of-bounds assignment). To preallocate a 100-item list such that you can add to any of the 100 offsets, you can fill it with `None` objects:

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

This doesn't limit the size of the list (it can still grow and shrink later), but simply presets an initial size to allow for future index assignments. You could initialize a list with zeros the same way, of course, but best practice dictates using `None` if the list's contents are not yet known.

Keep in mind that `None` does not mean "undefined." That is, `None` is something, not nothing (despite its name!)—it is a real object and piece of memory, given a built-in name by Python. Watch for other uses of this special object later in the book; it is also the default return value of functions, as we'll see in Part IV.

### The bool type

Also keep in mind that the Python Boolean type `bool`, introduced in Chapter 5, simply augments the notions of true and false in Python. As we learned in Chapter 5, the built-in words `True` and `False` are just customized versions of the integers `1` and `0`—it's as if these two words have been preassigned to `1` and `0` everywhere in Python. Because of the way this new type is implemented, this is really just a minor extension to the notions of true and false already described, designed to make truth values more explicit:

- When used explicitly in truth test code, the words `True` and `False` are equivalent to `1` and `0`, but they make the programmer's intent clearer.
- Results of Boolean tests run interactively print as the words `True` and `False`, instead of as `1` and `0`, to make the type of result clearer.

You are not required to use only Boolean types in logical statements such as `if`; all objects are still inherently true or false, and all the Boolean concepts mentioned in this chapter still work as described if you use other types. Python also provides a `bool` built-in function that can be used to test the Boolean value of an object (i.e., whether it is True—that is, nonzero or nonempty):

```
>>> bool(1)
True
>>> bool('spam')
True
>>> bool({})
False
```

In practice, though, you'll rarely notice the Boolean type produced by logic tests, because Boolean results are used automatically by `if` statements and other selection tools. We'll explore Booleans further when we study logical statements in Chapter 12.

## Python's Type Hierarchies

Figure 9-3 summarizes all the built-in object types available in Python and their relationships. We've looked at the most prominent of these; most of the other kinds of objects in Figure 9-3 correspond to program units (e.g., functions and modules) or exposed interpreter internals (e.g., stack frames and compiled code).

The main point to notice here is that *everything* in a Python system is an object type and may be processed by your Python programs. For instance, you can pass a class to a function, assign it to a variable, stuff it in a list or dictionary, and so on.

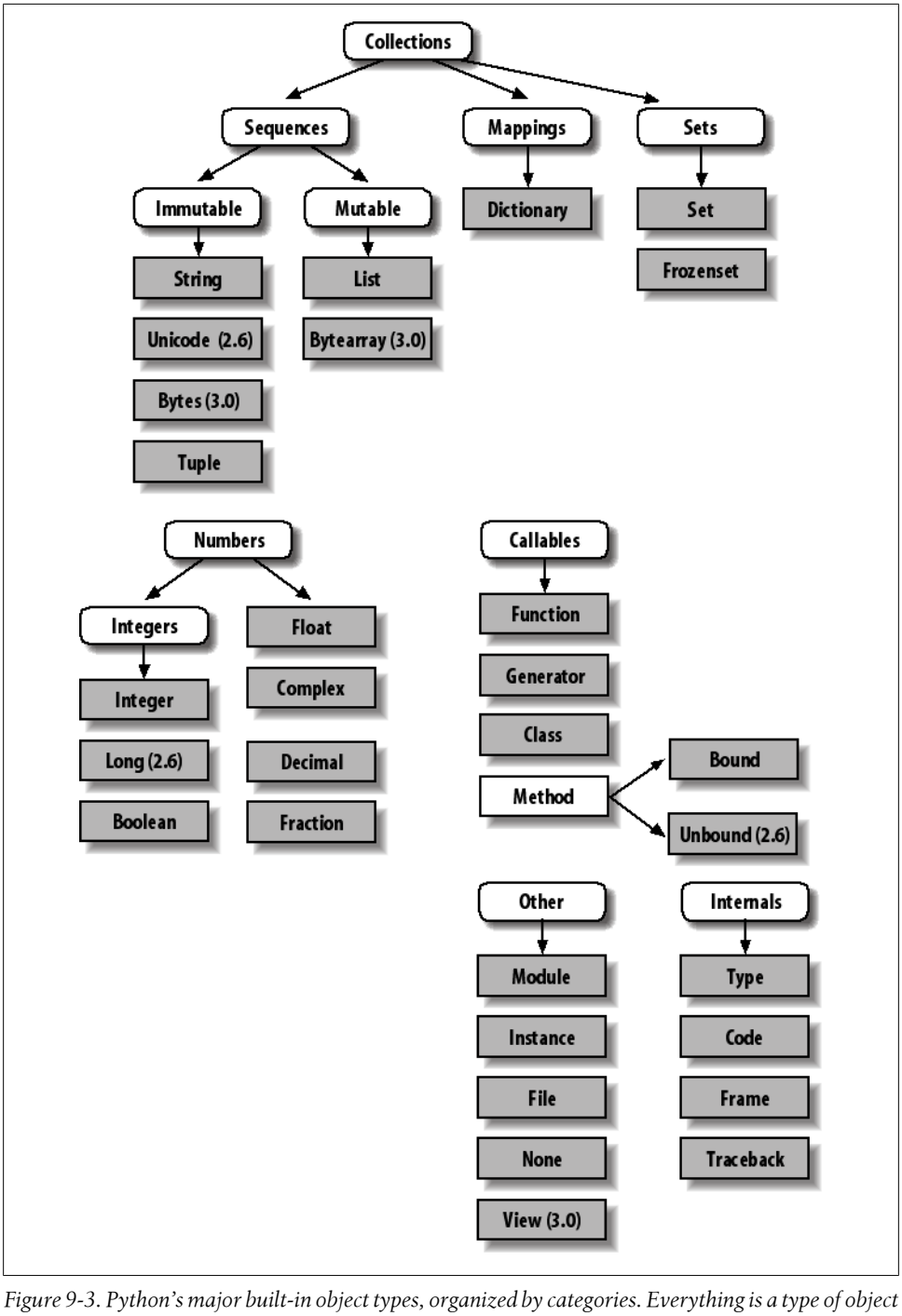


*Figure 9-3. Python's major built-in object types, organized by categories. Everything is a type of object in Python, even the type of an object!*

## Type Objects

In fact, even types themselves are an object type in Python: the type of an object is an object of type `type` (say that three times fast!). Seriously, a call to the built-in function `type(X)` returns the type object of object `X`. The practical application of this is that type objects can be used for manual type comparisons in Python `if` statements. However, for reasons introduced in Chapter 4, manual type testing is usually not the right thing to do in Python, since it limits your code's flexibility.

One note on type names: as of Python 2.2, each core type has a new built-in name added to support type customization through object-oriented subclassing: `dict`, `list`, `str`, `tuple`, `int`, `float`, `complex`, `bytes`, `type`, `set`, and more (in Python 2.6 but not 3.0, `file` is also a type name and a synonym for `open`). Calls to these names are really object constructor calls, not simply conversion functions, though you can treat them as simple functions for basic usage.

In addition, the `types` standard library module in Python 3.0 provides additional type names for types that are not available as built-ins (e.g., the type of a function; in Python 2.6 but not 3.0, this module also includes synonyms for built-in type names), and it is possible to do type tests with the `isinstance` function. For example, all of the following type tests are true:

```
type([1]) == type([])                # Type of another list
type([1]) == list                    # List type name
isinstance([1], list)                # List or customization thereof

import types                         # types has names for other types
def f(): pass
type(f) == types.FunctionType
```

Because types can be subclassed in Python today, the `isinstance` technique is generally recommended. See Chapter 31 for more on subclassing built-in types in Python 2.2 and later.

Also in Chapter 31, we will explore how `type(X)` and type-testing in general apply to instances of user-defined *classes*. In short, in Python 3.0 and for new-style classes in Python 2.6, the type of a class instance is the class from which the instance was made. For classic classes in Python 2.6 and earlier, all class instances are of the type "instance," and we must compare instance `__class__` attributes to compare their types meaningfully. Since we're not ready for classes yet, we'll postpone the rest of this story until Chapter 31.

# Other Types in Python

Besides the core objects studied in this part of the book, and the program-unit objects such as functions, modules, and classes that we'll meet later, a typical Python installation has dozens of additional object types available as linked-in C extensions or

Python classes—regular expression objects, DBM files, GUI widgets, network sockets, and so on.

The main difference between these extra tools and the built-in types we've seen so far is that the built-ins provide special language creation syntax for their objects (e.g., `4` for an integer, `[1,2]` for a list, the `open` function for files, and `def` and `lambda` for functions). Other tools are generally made available in standard library modules that you must first import to use. For instance, to make a regular expression object, you import `re` and call `re.compile()`. See Python's library reference for a comprehensive guide to all the tools available to Python programs.

# Built-in Type Gotchas

That's the end of our look at core data types. We'll wrap up this part of the book with a discussion of common problems that seem to bite new users (and the occasional expert), along with their solutions. Some of this is a review of ideas we've already covered, but these issues are important enough to warn about again here.

## Assignment Creates References, Not Copies

Because this is such a central concept, I'll mention it again: you need to understand what's going on with shared references in your program. For instance, in the following example, the list object assigned to the name `L` is referenced from `L` and from inside the list assigned to the name `M`. Changing `L` in-place changes what `M` references, too:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']          # Embed a reference to L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                   # Changes M too
>>> M
['X', [1, 0, 3], 'Y']
```

This effect usually becomes important only in larger programs, and shared references are often exactly what you want. If they're not, you can avoid sharing objects by copying them explicitly. For lists, you can always make a top-level copy by using an empty-limits slice:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']       # Embed a copy of L
>>> L[1] = 0                   # Changes only L, not M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Remember, slice limits default to 0 and the length of the sequence being sliced; if both are omitted, the slice extracts every item in the sequence and so makes a top-level copy (a new, unshared object).

## Repetition Adds One Level Deep

Repeating a sequence is like adding it to itself a number of times. However, when mutable sequences are nested, the effect might not always be what you expect. For instance, in the following example `X` is assigned to `L` repeated four times, whereas `Y` is assigned to a list *containing* `L` repeated four times:

```
>>> L = [4, 5, 6]
>>> X = L * 4                  # Like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4                # [L] + [L] + ... = [L, L,...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Because `L` was nested in the second repetition, `Y` winds up embedding references back to the original list assigned to `L`, and so is open to the same sorts of side effects noted in the last section:

```
>>> L[1] = 0                   # Impacts Y but not X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

The same solutions to this problem apply here as in the previous section, as this is really just another way to create the shared mutable object reference case. If you remember that repetition, concatenation, and slicing copy only the top level of their operand objects, these sorts of cases make much more sense.

## Beware of Cyclic Data Structures

We actually encountered this concept in a prior exercise: if a collection object contains a reference to itself, it's called a *cyclic object*. Python prints a `[...]` whenever it detects a cycle in the object, rather than getting stuck in an infinite loop:

```
>>> L = ['grail']              # Append reference to same object
>>> L.append(L)                # Generates cycle in object: [...]
>>> L
['grail', [...]]
```

Besides understanding that the three dots in square brackets represent a cycle in the object, this case is worth knowing about because it can lead to gotchas—cyclic structures may cause code of your own to fall into unexpected loops if you don't anticipate them. For instance, some programs keep a list or dictionary of already visited items and

check it to determine whether they're in a cycle. See the solutions to the "Test Your Knowledge: Part I Exercises" in Appendix B for more on this problem, and check out the *reloadall.py* program in Chapter 24 for a solution.

Don't use cyclic references unless you really need to. There are good reasons to create cycles, but unless you have code that knows how to handle them, you probably won't want to make your objects reference themselves very often in practice.

### Immutable Types Can't Be Changed In-Place

You can't change an immutable object in-place. Instead, you construct a new object with slicing, concatenation, and so on, and assign it back to the original reference, if needed:

```
T = (1, 2, 3)

T[2] = 4                # Error!

T = T[:2] + (4,)        # OK: (1, 2, 4)
```

That might seem like extra coding work, but the upside is that the previous gotchas can't happen when you're using immutable objects such as tuples and strings; because they can't be changed in-place, they are not open to the sorts of side effects that lists are.

## Chapter Summary

This chapter explored the last two major core object types—the tuple and the file. We learned that tuples support all the usual sequence operations, have just a few methods, and do not allow any in-place changes because they are immutable. We also learned that files are returned by the built-in `open` function and provide methods for reading and writing data. We explored how to translate Python objects to and from strings for storing in files, and we looked at the `pickle` and `struct` modules for advanced roles (object serialization and binary data). Finally, we wrapped up by reviewing some properties common to all object types (e.g., shared references) and went through a list of common mistakes ("gotchas") in the object type domain.

In the next part, we'll shift gears, turning to the topic of statement syntax in Python—we'll explore all of Python's basic procedural statements in the chapters that follow. The next chapter kicks off that part of the book with an introduction to Python's general syntax model, which is applicable to all statement types. Before moving on, though, take the chapter quiz, and then work through the end-of-part lab exercises to review type concepts. Statements largely just create and process objects, so make sure you've mastered this domain by working through all the exercises before reading on.

## Test Your Knowledge: Quiz

1. How can you determine how large a tuple is? Why is this tool located where it is?
2. Write an expression that changes the first item in a tuple. `(4, 5, 6)` should become `(1, 5, 6)` in the process.
3. What is the default for the processing mode argument in a file `open` call?
4. What module might you use to store Python objects in a file without converting them to strings yourself?
5. How might you go about copying all parts of a nested structure at once?
6. When does Python consider an object true?
7. What is your quest?

## Test Your Knowledge: Answers

1. The built-in `len` function returns the length (number of contained items) for any container object in Python, including tuples. It is a built-in function instead of a type method because it applies to many different types of objects. In general, built-in functions and expressions may span many object types; methods are specific to a single object type, though some may be available on more than one type (`index`, for example, works on lists and tuples).
2. Because they are immutable, you can't really change tuples in-place, but you can generate a new tuple with the desired value. Given `T = (4, 5, 6)`, you can change the first item by making a new tuple from its parts by slicing and concatenating: `T = (1,) + T[1:]`. (Recall that single-item tuples require a trailing comma.) You could also convert the tuple to a list, change it in-place, and convert it back to a tuple, but this is more expensive and is rarely required in practice—simply use a list if you know that the object will require in-place changes.
3. The default for the processing mode argument in a file `open` call is `'r'`, for reading text input. For input text files, simply pass in the external file's name.
4. The `pickle` module can be used to store Python objects in a file without explicitly converting them to strings. The `struct` module is related, but it assumes the data is to be in packed binary format in the file.
5. Import the `copy` module, and call `copy.deepcopy(X)` if you need to copy all parts of a nested structure `X`. This is also rarely seen in practice; references are usually the desired behavior, and shallow copies (e.g., `aList[:]`, `aDict.copy()`) usually suffice for most copies.

6. An object is considered true if it is either a nonzero number or a nonempty collection object. The built-in words `True` and `False` are essentially predefined to have the same meanings as integer `1` and `0`, respectively.
7. Acceptable answers include "To learn Python," "To move on to the next part of the book," or "To seek the Holy Grail."

## Test Your Knowledge: Part II Exercises

This session asks you to get your feet wet with built-in object fundamentals. As before, a few new ideas may pop up along the way, so be sure to flip to the answers in Appendix B when you're done (or when you're not, if necessary). If you have limited time, I suggest starting with exercises 10 and 11 (the most practical of the bunch), and then working from first to last as time allows. This is all fundamental material, though, so try to do as many of these as you can.

1. *The basics*. Experiment interactively with the common type operations found in the various operation tables in this part of the book. To get started, bring up the Python interactive interpreter, type each of the following expressions, and try to explain what's happening in each case. Note that the semicolon in some of these is being used as a statement separator, to squeeze multiple statements onto a single line: for example, `X=1;X` assigns and then prints a variable (more on statement syntax in the next part of the book). Also remember that a comma between expressions usually builds a tuple, even if there are no enclosing parentheses: `X,Y,Z` is a three-item tuple, which Python prints back to you in parentheses.

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
'green {0} and {1}'.format('eggs', S)

('x',)[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
```

```
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D

[[]], ["",[],(),{},None]
```

2. *Indexing and slicing*. At the interactive prompt, define a list named `L` that contains four strings or numbers (e.g., `L=[0,1,2,3]`). Then, experiment with some boundary cases; you may not ever see these cases in real programs, but they are intended to make you think about the underlying model, and some may be useful in less artificial forms:
   a. What happens when you try to index out of bounds (e.g., `L[4]`)?
   b. What about slicing out of bounds (e.g., `L[-1000:100]`)?
   c. Finally, how does Python handle it if you try to extract a sequence in reverse, with the lower bound greater than the higher bound (e.g., `L[3:1]`)? Hint: try assigning to this slice (`L[3:1]=['?']`), and see where the value is put. Do you think this may be the same phenomenon you saw when slicing out of bounds?
3. *Indexing, slicing, and `del`*. Define another list `L` with four items, and assign an empty list to one of its offsets (e.g., `L[2]=[]`). What happens? Then, assign an empty list to a slice (`L[2:3]=[]`). What happens now? Recall that slice assignment deletes the slice and inserts the new value where it used to be.

   The `del` statement deletes offsets, keys, attributes, and names. Use it on your list to delete an item (e.g., `del L[0]`). What happens if you delete an entire slice (`del L[1:]`)? What happens when you assign a nonsequence to a slice (`L[1:2]=1`)?
4. *Tuple assignment*. Type the following lines:

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
```

   What do you think is happening to `X` and `Y` when you type this sequence?
5. *Dictionary keys*. Consider the following code fragments:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

   You've learned that dictionaries aren't accessed by offsets, so what's going on here? Does the following shed any light on the subject? (Hint: strings, integers, and tuples share which type category?)

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

6. *Dictionary indexing*. Create a dictionary named `D` with three entries, for keys `'a'`, `'b'`, and `'c'`. What happens if you try to index a nonexistent key (`D['d']`)? What does Python do if you try to assign to a nonexistent key `'d'` (e.g., `D['d']='spam'`)? How does this compare to out-of-bounds assignments and references for lists? Does this sound like the rule for variable names?
7. *Generic operations*. Run interactive tests to answer the following questions:
   a. What happens when you try to use the `+` operator on different/mixed types (e.g., string `+` list, list `+` tuple)?
   b. Does `+` work when one of the operands is a dictionary?
   c. Does the `append` method work for both lists and strings? How about using the `keys` method on lists? (Hint: what does `append` assume about its subject object?)
   d. Finally, what type of object do you get back when you slice or concatenate two lists or two strings?
8. *String indexing*. Define a string `S` of four characters: `S = "spam"`. Then type the following expression: `S[0][0][0][0][0]`. Any clue as to what's happening this time? (Hint: recall that a string is a collection of characters, but Python characters are one-character strings.) Does this indexing expression still work if you apply it to a list such as `['s', 'p', 'a', 'm']`? Why?
9. *Immutable types*. Define a string `S` of four characters again: `S = "spam"`. Write an assignment that changes the string to `"slam"`, using only slicing and concatenation. Could you perform the same operation using just indexing and concatenation? How about index assignment?
10. *Nesting*. Write a data structure that represents your personal information: name (first, middle, last), age, job, address, email address, and phone number. You may build the data structure with any combination of built-in object types you like (lists, tuples, dictionaries, strings, numbers). Then, access the individual components of your data structures by indexing. Do some structures make more sense than others for this object?
11. *Files*. Write a script that creates a new output file called *myfile.txt* and writes the string `"Hello file world!"` into it. Then write another script that opens *myfile.txt* and reads and prints its contents. Run your two scripts from the system command line. Does the new file show up in the directory where you ran your scripts? What if you add a different directory path to the filename passed to `open`? Note: file `write` methods do not add newline characters to your strings; add an explicit `\n` at the end of the string if you want to fully terminate the line in the file.

PART III

# Statements and Syntax

CHAPTER 10

# Introducing Python Statements

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

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

## Python Program Structure Revisited

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

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

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

## Python's Statements

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

*Table 10-1. Python 3.0 statements*

| Statement | Role | Example |
|---|---|---|
| Assignment | Creating references | `a, *b = 'good', 'bad', 'ugly'` |
| Calls and other expressions | Running functions | `log.write("spam, ham")` |
| `print` calls | Printing objects | `print('The Killer', joke)` |
| `if/elif/else` | Selecting actions | `if "python" in text:`<br>`  print(text)` |
| `for/else` | Sequence iteration | `for x in mylist:`<br>`  print(x)` |
| `while/else` | General loops | `while X > Y:`<br>`  print('hello')` |
| `pass` | Empty placeholder | `while True:`<br>`  pass` |
| `break` | Loop exit | `while True:`<br>`  if exittest(): break` |
| `continue` | Loop continue | `while True:`<br>`  if skiptest(): continue` |
| `def` | Functions and methods | `def f(a, b, c=1, *d):`<br>`  print(a+b+c+d[0])` |
| `return` | Functions results | `def f(a, b, c=1, *d):`<br>`  return a+b+c+d[0]` |
| `yield` | Generator functions | `def gen(n):`<br>`  for i in n: yield i*2` |
| `global` | Namespaces | `x = 'old'`<br>`def function():`<br>`  global x, y; x = 'new'` |
| `nonlocal` | Namespaces (3.0+) | `def outer():`<br>`  x = 'old'`<br>`  def function():`<br>`  nonlocal x; x = 'new'` |
| `import` | Module access | `import sys` |
| `from` | Attribute access | `from sys import stdin` |
| `class` | Building objects | `class Subclass(Superclass):`<br>`  staticData = []`<br>`  def method(self): pass` |

| Statement | Role | Example |
|---|---|---|
| try/except/finally | Catching exceptions | try:<br>    action()<br>except:<br>    print('action error') |
| raise | Triggering exceptions | raise EndSearch(location) |
| assert | Debugging checks | assert X > Y, 'X too small' |
| with/as | Context managers (2.6+) | with open('data') as myfile:<br>    process(myfile) |
| del | Deleting references | del data[k]<br>del data[i:j]<br>del obj.attr<br>del variable |

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

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

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

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

# A Tale of Two ifs

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

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

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

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

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

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

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

## What Python Adds

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

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

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

## What Python Removes

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

### Parentheses are optional

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

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

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

```
if x < y
```

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

### End of line is end of statement

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

```
x = 1;
```

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

```
x = 1
```

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

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

#### End of indentation is end of block

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

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

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

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

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

## Why Indentation Syntax?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## A Few Special Cases

As mentioned previously, in Python's syntax model:

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

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

### Statement rule special cases

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

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

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

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

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

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

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

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

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

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

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

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

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

#### Block rule special case

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

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

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

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

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

# A Quick Example: Interactive Loops

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

## A Simple Interactive Loop

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

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

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

This code makes use of a few new ideas:

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

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

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

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

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

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

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

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

## Doing Math on User Inputs

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

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

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

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

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

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

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

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

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

## Handling Errors by Testing Inputs

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

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

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

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

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

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

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

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

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

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

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

## Handling Errors with try Statements

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

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

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

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

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

## Nesting Code Three Levels Deep

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

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

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

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

## Chapter Summary

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

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

---

## Test Your Knowledge: Quiz

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

## Test Your Knowledge: Answers

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

CHAPTER 11

# Assignments, Expressions, and Prints

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

## Assignment Statements

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

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

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

## Assignment Statement Forms

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

*Table 11-1. Assignment statement forms*

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

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

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

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

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

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

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

## Sequence Assignments

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

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

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

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

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

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

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

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

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

### Advanced sequence assignment patterns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Extended Sequence Unpacking in Python 3.0

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

### Extended unpacking in action

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> S = 'spam'

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

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

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

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

### Boundary cases

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

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

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

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

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

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

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

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

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

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

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

### A useful convenience

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

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

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

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

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

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

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

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

### Application to for loops

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

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

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

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

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

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

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

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

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

## Multiple-Target Assignments

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

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

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

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

### Multiple-target assignment and shared references

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

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

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

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

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

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

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

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

## Augmented Assignments

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

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

*Table 11-2. Augmented assignment statements*

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

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

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

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

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

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

Augmented assignments have three advantages:*

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

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

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

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

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

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

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

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

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

### Augmented assignment and shared references

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

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

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

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

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

## Variable Name Rules

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

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

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

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

*Table 11-3. Python 3.0 reserved words*

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

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

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

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

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

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

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

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

### Python's Deprecation Protocol

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

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

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

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

### Naming conventions

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

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

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

### Names have no type, but objects do

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

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

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

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

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

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

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

## Expression Statements

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

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

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

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

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

*Table 11-4. Common Python expression statements*

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

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

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

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

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

## Expression Statements and In-Place Changes

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

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

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

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

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

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

# Print Operations

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

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

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

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

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

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

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

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

## The Python 3.0 print Function

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

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

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

### Call format

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

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

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

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

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

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

### The 3.0 print function in action

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## The Python 2.6 print Statement

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

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

### Statement forms

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

*Table 11-5. Python 2.6 print statement forms*

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

### The 2.6 print statement in action

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

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

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

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

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

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

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

## Print Stream Redirection

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

### The Python "hello world" program

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

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

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

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

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

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

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

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

#### Manual stream redirection

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

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

is equivalent to the longer:

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

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

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

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

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

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

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

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

### Automatic stream redirection

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Version-Neutral Printing

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

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

```
from __future__ import print_function
```

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

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

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

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

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

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

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

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

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

## Why You Will Care: print and stdout

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

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

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

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

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

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

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

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

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

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

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

## Chapter Summary

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

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

---

## Test Your Knowledge: Quiz

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

## Test Your Knowledge: Answers

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

2. If you assign them this way:

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

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

CHAPTER 12

# if Tests and Syntax Rules

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

## if Statements

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

### General Format

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

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

## Basic Examples

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

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

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

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

## Multiway Branching

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Python Syntax Rules

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

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

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

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

## Block Delimiters: Indentation Rules

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

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

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

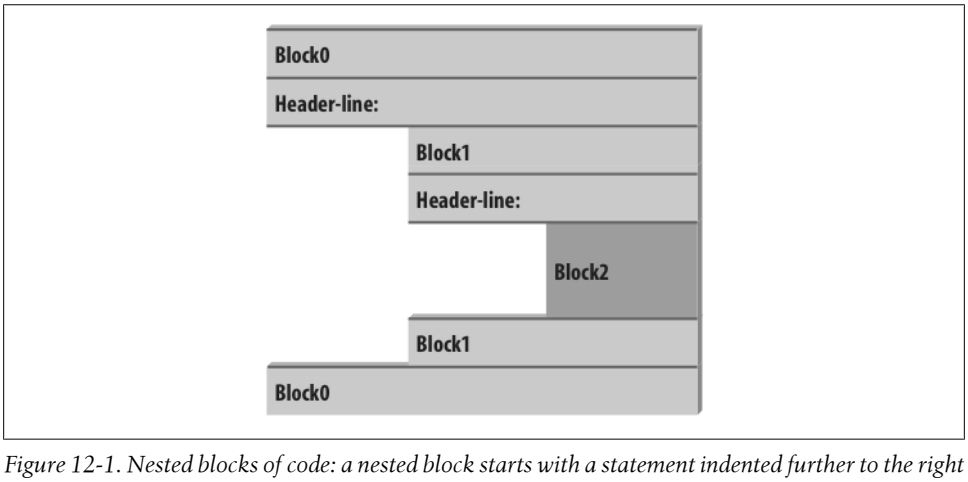


*Figure 12-1. Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.*

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

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

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

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

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

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

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

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

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

#### Avoid mixing tabs and spaces: New error checking in 3.0

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

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

## Statement Delimiters: Lines and Continuations

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

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

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

## A Few Special Cases

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Truth Tests

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## The if/else Ternary Expression

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

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

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

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

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

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

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

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

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

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

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

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

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

For example:

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

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

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

## Why You Will Care: Booleans

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

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

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

```
X = A or default
```

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

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

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

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

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

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

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

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

## Chapter Summary

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

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

---

## Test Your Knowledge: Quiz

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

## Test Your Knowledge: Answers

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

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

# CHAPTER 13
# while and for Loops

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

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

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

## while Loops

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

## General Format

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

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

## Examples

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

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

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

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

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

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

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

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

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

# break, continue, pass, and the Loop else

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

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

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

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

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

## General Loop Format

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

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

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

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

## pass

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

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

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

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

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

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

def func2():
    pass
```

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

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

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

def func2():
    ...

func1()                             # Does nothing if called
```

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

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

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

```
>>> X
Ellipsis
```

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

## continue

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

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

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

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

## break

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

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

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

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

## Loop else

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

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

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

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

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

### More on the loop else

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

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

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

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

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

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

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

### Why You Will Care: Emulating C while Loops

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

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

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

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

or move the assignment into the loop with tests:

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

or move the first assignment outside the loop:

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

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

# for Loops

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

## General Format

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

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

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

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

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

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

## Examples

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

### Basic usage

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

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

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

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

### Other data types

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

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

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

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

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

### Tuple assignment in for loops

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### Python 3.0 extended sequence assignment in for loops

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

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

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

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

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

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

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

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

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

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

See Chapter 11 for more on this assignment form.

### Nested for loops

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

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

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

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

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

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

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

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

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

## Why You Will Care: File Scanners

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

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

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

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

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

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

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

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

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

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

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

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

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

## Loop Coding Techniques

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

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

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

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

## Counter Loops: while and range

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Nonexhaustive Traversals: range and Slices

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

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

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

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

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

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

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

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

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

## Changing Lists: range

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

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

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

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

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

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

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

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

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

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

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

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

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

## Parallel Traversals: zip and map

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

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

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

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

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

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

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

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

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

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

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

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

### map equivalence in Python 2.6

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

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

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

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

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

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

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

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

#### Dictionary construction with zip

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

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

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

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

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

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

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

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

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

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

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

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

## Generating Both Offsets and Items: enumerate

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

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

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

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

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

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

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

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

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

## Chapter Summary

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

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

---

## Test Your Knowledge: Quiz

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

## Test Your Knowledge: Answers

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

CHAPTER 14

# Iterations and Comprehensions, Part 1

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

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

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

## Iterators: A First Look

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

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

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

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

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

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

## The Iteration Protocol: File Iterators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Manual Iteration: iter and next

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Other Built-in Type Iterators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## List Comprehensions: A First Look

Now that we've seen how the iteration protocol works, let's turn to a very common use case. Together with `for` loops, list comprehensions are one of the most prominent contexts in which the iteration protocol is applied.

In the previous chapter, we learned how to use `range` to change a list as we step across it:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

This works, but as I mentioned there, it may not be the optimal "best-practice" approach in Python. Today, the list comprehension expression makes many such prior use cases obsolete. Here, for example, we can replace the loop with a single expression that produces the desired result list:

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

The net result is the same, but it requires less coding on our part and is likely to run substantially faster. The list comprehension isn't exactly the same as the `for` loop statement version because it makes a *new* list object (which might matter if there are multiple references to the original list), but it's close enough for most applications and is a common and convenient enough approach to merit a closer look here.

## List Comprehension Basics

We met the list comprehension briefly in Chapter 4. Syntactically, its syntax is derived from a construct in set theory notation that applies an operation to each item in a set, but you don't have to know set theory to use this tool. In Python, most people find that a list comprehension simply looks like a backward `for` loop.

To get a handle on the syntax, let's dissect the prior section's example in more detail:

```
>>> L = [x + 10 for x in L]
```

List comprehensions are written in square brackets because they are ultimately a way to construct a new list. They begin with an arbitrary expression that we make up, which uses a loop variable that we make up (`x + 10`). That is followed by what you should now recognize as the header of a `for` loop, which names the loop variable, and an iterable object (`for x in L`).

To run the expression, Python executes an iteration across `L` inside the interpreter, assigning `x` to each item in turn, and collects the results of running the items through the expression on the left side. The result list we get back is exactly what the list comprehension says—a new list containing `x + 10`, for every `x` in `L`.

Technically speaking, list comprehensions are never really required because we can always build up a list of expression results manually with `for` loops that append results as we go:

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[21, 22, 23, 24, 25]
```

In fact, this is exactly what the list comprehension does internally.

However, list comprehensions are more concise to write, and because this code pattern of building up result lists is so common in Python work, they turn out to be very handy in many contexts. Moreover, list comprehensions can run much faster than manual `for` loop statements (often roughly twice as fast) because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code; especially for larger data sets, there is a major performance advantage to using them.

## Using List Comprehensions on Files

Let's work through another common use case for list comprehensions to explore them in more detail. Recall that the file object has a `readlines` method that loads the file into a list of line strings all at once:

```
>>> f = open('script1.py')
>>> lines = f.readlines()
```

```
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
```

This works, but the lines in the result all include the newline character (`\n`) at the end. For many programs, the newline character gets in the way—we have to be careful to avoid double-spacing when printing, and so on. It would be nice if we could get rid of these newlines all at once, wouldn't it?

Any time we start thinking about performing an operation on each item in a sequence, we're in the realm of list comprehensions. For example, assuming the variable `lines` is as it was in the prior interaction, the following code does the job by running each line in the list through the string `rstrip` method to remove whitespace on the right side (a `line[:-1]` slice would work, too, but only if we can be sure all lines are properly terminated):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This works as planned. Because list comprehensions are an iteration context just like `for` loop statements, though, we don't even have to open the file ahead of time. If we open it inside the expression, the list comprehension will automatically use the iteration protocol we met earlier in this chapter. That is, it will read one line from the file at a time by calling the file's `next` method, run the line through the `rstrip` expression, and add it to the result list. Again, we get what we ask for—the `rstrip` result of a line, for every line in the file:

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This expression does a lot implicitly, but we're getting a lot of work for free here—Python scans the file and builds a list of operation results automatically. It's also an efficient way to code this operation: because most of this work is done inside the Python interpreter, it is likely much faster than an equivalent `for` statement. Again, especially for large files, the speed advantages of list comprehensions can be significant.

Besides their efficiency, list comprehensions are also remarkably expressive. In our example, we can run any string operation on a file's lines as we iterate. Here's the list comprehension equivalent to the file iterator uppercase example we met earlier, along with a few others (the method chaining in the second of these examples works because string methods return a new string, to which we can apply another string method):

```
>>> [line.upper() for line in open('script1.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> [line.rstrip().upper() for line in open('script1.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(2 ** 33)']

>>> [line.split() for line in open('script1.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(2', '**','33)']]
```

```
>>> [line.replace(' ', '!') for line in open('script1.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(2!**!33)\n']

>>> [('sys' in line, line[0]) for line in open('script1.py')]
[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]
```

## Extended List Comprehension Syntax

In fact, list comprehensions can be even more advanced in practice. As one particularly useful extension, the `for` loop nested in the expression can have an associated `if` clause to filter out of the result items for which the test is not true.

For example, suppose we want to repeat the prior section's file-scanning example, but we need to collect only lines that begin with the letter *p* (perhaps the first character on each line is an action code of some sort). Adding an `if` filter clause to our expression does the trick:

```
>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>> lines
['print(sys.path)', 'print(2 ** 33)']
```

Here, the `if` clause checks each line read from the file to see whether its first character is *p*; if not, the line is omitted from the result list. This is a fairly big expression, but it's easy to understand if we translate it to its simple `for` loop statement equivalent. In general, we can always translate a list comprehension to a `for` statement by appending as we go and further indenting each successive part:

```
>>> res = []
>>> for line in open('script1.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(2 ** 33)']
```

This `for` statement equivalent works, but it takes up four lines instead of one and probably runs substantially slower.

List comprehensions can become even more complex if we need them to—for instance, they may contain nested loops, coded as a series of `for` clauses. In fact, their full syntax allows for any number of `for` clauses, each of which can have an optional associated `if` clause (we'll be more formal about their syntax in Chapter 20).

For example, the following builds a list of the concatenation of `x + y` for every `x` in one string and every `y` in another. It effectively collects the permutation of the characters in two strings:

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Again, one way to understand this expression is to convert it to statement form by indenting its parts. The following is an equivalent, but likely slower, alternative way to achieve the same effect:

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Beyond this complexity level, though, list comprehension expressions can often become too compact for their own good. In general, they are intended for simple types of iterations; for more involved work, a simpler `for` statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it's probably not a good idea.

We'll revisit list comprehensions in Chapter 20, in the context of functional programming tools; as we'll see, they turn out to be just as related to functions as they are to looping statements.

## Other Iteration Contexts

Later in the book, we'll see that user-defined classes can implement the iteration protocol too. Because of this, it's sometimes important to know which built-in tools make use of it—any tool that employs the iteration protocol will automatically work on any built-in type or user-defined class that provides it.

So far, I've been demonstrating iterators in the context of the `for` loop statement, because this part of the book is focused on statements. Keep in mind, though, that every tool that scans from left to right across objects uses the iteration protocol. This includes the `for` loops we've seen:

```
>>> for line in open('script1.py'):          # Use file iterators
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

However, list comprehensions, the `in` membership test, the `map` built-in function, and other built-ins such as the `sorted` and `zip` calls also leverage the iteration protocol. When applied to a file, all of these use the file object's iterator automatically to scan line by line:

```
>>> uppers = [line.upper() for line in open('script1.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']
```

```
>>> map(str.upper, open('script1.py'))        # map is an iterable in 3.0
<map object at 0x02660710>

>>> list( map(str.upper, open('script1.py')) )
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> 'y = 2\n' in open('script1.py')
False
>>> 'x = 2\n' in open('script1.py')
True
```

We introduced the `map` call used here in the preceding chapter; it's a built-in that applies a function call to each item in the passed-in iterable object. `map` is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also *returns* an iterable object itself in Python 3.0, so we must wrap it in a `list` call to force it to give us all its values at once; more on this change later in this chapter. Because `map`, like the list comprehension, is related to both `for` loops and functions, we'll also explore both again in Chapters 19 and 20.

Python includes various additional built-ins that process iterables, too: `sorted` sorts items in an iterable, `zip` combines items from iterables, `enumerate` pairs items in an iterable with relative positions, `filter` selects items for which a function is true, and `reduce` runs pairs of items in an iterable through a function. All of these accept iterables, and `zip`, `enumerate`, and `filter` also return an iterable in Python 3.0, like `map`. Here they are in action running the file's iterator automatically to scan line by line:

```
>>> sorted(open('script1.py'))
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']

>>> list(zip(open('script1.py'), open('script1.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(2 ** 33)\n')]

>>> list(enumerate(open('script1.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
(3, 'print(2 ** 33)\n')]

>>> list(filter(bool, open('script1.py')))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('script1.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(2 ** 33)\n'
```

All of these are iteration tools, but they have unique roles. We met `zip` and `enumerate` in the prior chapter; `filter` and `reduce` are in Chapter 19's functional programming domain, so we'll defer details for now.

We first saw the `sorted` function used here at work in Chapter 4, and we used it for dictionaries in Chapter 8. `sorted` is a built-in that employs the iteration protocol—it's like the original list `sort` method, but it returns the new sorted list as a result and runs

on any iterable object. Notice that, unlike `map` and others, `sorted` returns an actual *list* in Python 3.0 instead of an iterable.

Other built-in functions support the iteration protocol as well (but frankly, are harder to cast in interesting examples related to files). For example, the `sum` call computes the sum of all the numbers in any iterable; the `any` and `all` built-ins return `True` if any or all items in an iterable are `True`, respectively; and `max` and `min` return the largest and smallest item in an iterable, respectively. Like `reduce`, all of the tools in the following examples accept any iterable as an argument and use the iteration protocol to scan it, but return a single result:

```
>>> sum([3, 2, 4, 1, 5, 0])                # sum expects numbers only
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

Strictly speaking, the `max` and `min` functions can be applied to files as well—they automatically use the iteration protocol to scan the file and pick out the lines with the highest and lowest string values, respectively (though I'll leave valid use cases to your imagination):

```
>>> max(open('script1.py'))                 # Line with max/min string value
'x = 2\n'
>>> min(open('script1.py'))
'import sys\n'
```

Interestingly, the iteration protocol is even more pervasive in Python today than the examples so far have demonstrated—*everything* in Python's built-in toolset that scans an object from left to right is defined to use the iteration protocol on the subject object. This even includes more esoteric tools such as the `list` and `tuple` built-in functions (which build new objects from iterables), the string `join` method (which puts a substring between strings contained in an iterable), and even sequence assignments. Consequently, all of these will also work on an open file and automatically read one line at a time:

```
>>> list(open('script1.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> tuple(open('script1.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n')

>>> '&&'.join(open('script1.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(2 ** 33)\n'

>>> a, b, c, d = open('script1.py')
>>> a, d
```

```
('import sys\n', 'print(2 ** 33)\n')

>>> a, *b = open('script1.py')                  # 3.0 extended form
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n'])
```

Earlier, we saw that the built-in `dict` call accepts an iterable `zip` result, too. For that matter, so does the `set` call, as well as the new *set* and *dictionary comprehension expressions* in Python 3.0, which we met in Chapters 4, 5, and 8:

```
>>> set(open('script1.py'))
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {line for line in open('script1.py')}
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {ix: line for ix, line in enumerate(open('script1.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(2 ** 33)\n'}
```

In fact, both set and dictionary comprehensions support the extended syntax of list comprehensions we met earlier in this chapter, including `if` tests:

```
>>> {line for line in open('script1.py') if line[0] == 'p'}
{'print(sys.path)\n', 'print(2 ** 33)\n'}

>>> {ix: line for (ix, line) in enumerate(open('script1.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(2 ** 33)\n'}
```

Like the list comprehension, both of these scan the file line by line and pick out lines that begin with the letter "p." They also happen to build sets and dictionaries in the end, but we get a lot of work "for free" by combining file iteration and comprehension syntax.

There's one last iteration context that's worth mentioning, although it's a bit of a preview: in Chapter 18, we'll learn that a special `*arg` form can be used in function calls to unpack a collection of values into individual arguments. As you can probably predict by now, this accepts any iterable, too, including files (see Chapter 18 for more details on the call syntax):

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4])                    # Unpacks into arguments
1&2&3&4

>>> f(*open('script1.py'))              # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(2 ** 33)
```

In fact, because this argument-unpacking syntax in calls accepts iterables, it's also possible to use the `zip` built-in to *unzip* zipped tuples, by making prior or nested `zip` results

arguments for another `zip` call (warning: you probably shouldn't read the following example if you plan to operate heavy machinery anytime soon!):

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))                  # Zip tuples: returns an iterable
[(1, 3), (2, 4)]
>>>
>>> A, B = zip(*zip(X, Y))           # Unzip a zip!
>>> A
(1, 2)
>>> B
(3, 4)
```

Still other tools in Python, such as the `range` built-in and dictionary view objects, return iterables instead of processing them. To see how these have been absorbed into the iteration protocol in Python 3.0 as well, we need to move on to the next section.

# New Iterables in Python 3.0

One of the fundamental changes in Python 3.0 is that it has a stronger emphasis on iterators than 2.X. In addition to the iterators associated with built-in types such as files and dictionaries, the dictionary methods `keys`, `values`, and `items` return iterable objects in Python 3.0, as do the built-in functions `range`, `map`, `zip`, and `filter`. As shown in the prior section, the last three of these functions both return iterators and process them. All of these tools produce results on demand in Python 3.0, instead of constructing result lists as they do in 2.6.

Although this saves memory space, it can impact your coding styles in some contexts. In various places in this book so far, for example, we've had to wrap up various function and method call results in a `list(...)` call in order to force them to produce all their results at once:

```
>>> zip('abc', 'xyz')                # An iterable in Python 3.0 (a list in 2.6)
<zip object at 0x02E66710>

>>> list(zip('abc', 'xyz'))          # Force list of results in 3.0 to display
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

This isn't required in 2.6, because functions like `zip` return lists of results. In 3.0, though, they return iterable objects, producing results on demand. This means extra typing is required to display the results at the interactive prompt (and possibly in some other contexts), but it's an asset in larger programs—delayed evaluation like this conserves memory and avoids pauses while large result lists are computed. Let's take a quick look at some of the new 3.0 iterables in action.

## The range Iterator

We studied the `range` built-in's basic behavior in the prior chapter. In 3.0, it returns an iterator that generates numbers in the range on demand, instead of building the result list in memory. This subsumes the older 2.X `xrange` (see the upcoming version skew note), and you must use `list(range(...))` to force an actual range list if one is needed (e.g., to display results):

```
C:\\misc> c:\python30\python
>>> R = range(10)                  # range returns an iterator, not a list
>>> R
range(0, 10)

>>> I = iter(R)                    # Make an iterator from the range
>>> next(I)                        # Advance to next result
0                                  # What happens in for loops, comprehensions, etc.
>>> next(I)
1
>>> next(I)
2

>>> list(range(10))                # To force a list if required
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Unlike the list returned by this call in 2.X, `range` objects in 3.0 support only iteration, indexing, and the `len` function. They do not support any other sequence operations (use `list(...)` if you require more list tools):

```
>>> len(R)                         # range also does len and indexing, but no others
10
>>> R[0]
0
>>> R[-1]
9

>>> next(I)                        # Continue taking from iterator, where left off
3
>>> I.__next__()                   # .next() becomes .__next__(), but use new next()
4
```

*Version skew note*: Python 2.X also has a built-in called `xrange`, which is like `range` but produces items on demand instead of building a list of results in memory all at once. Since this is exactly what the new iterator-based `range` does in Python 3.0, `xrange` is no longer available in 3.0—it has been subsumed. You may still see it in 2.X code, though, especially since `range` builds result lists there and so is not as efficient in its memory usage. As noted in a sidebar in the prior chapter, the `file.xreadlines()` method used to minimize memory use in 2.X has been dropped in Python 3.0 for similar reasons, in favor of file iterators.

## The map, zip, and filter Iterators

Like `range`, the `map`, `zip`, and `filter` built-ins also become iterators in 3.0 to conserve space, rather than producing a result list all at once in memory. All three not only process iterables, as in 2.X, but also return iterable results in 3.0. Unlike `range`, though, they are their own iterators—after you step through their results once, they are exhausted. In other words, you can't have multiple iterators on their results that maintain different positions in those results.

Here is the case for the `map` built-in we met in the prior chapter. As with other iterators, you can force a list with `list(...)` if you really need one, but the default behavior can save substantial space in memory for large result sets:

```
>>> M = map(abs, (-1, 0, 1))            # map returns an iterator, not a list
>>> M
<map object at 0x0276B890>
>>> next(M)                             # Use iterator manually: exhausts results
1                                       # These do not support len() or indexing
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration

>>> for x in M: print(x)                # map iterator is now empty: one pass only
...

>>> M = map(abs, (-1, 0, 1))            # Make a new iterator to scan again
>>> for x in M: print(x)                # Iteration contexts auto call next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1)))          # Can force a real list if needed
[1, 0, 1]
```

The `zip` built-in, introduced in the prior chapter, returns iterators that work the same way:

```
>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip is the same: a one-pass iterator
>>> Z
<zip object at 0x02770EE0>

>>> list(Z)
[(1, 10), (2, 20), (3, 30)]

>>> for pair in Z: print(pair)          # Exhausted after one pass
...

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)          # Iterator used automatically or manually
...
(1, 10)
```

```
(2, 20)
(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)
```

The `filter` built-in, which we'll study in the next part of this book, is also analogous. It returns items in an iterable for which a passed-in function returns `True` (as we've learned, in Python `True` includes nonempty objects):

```
>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x0269C6D0>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']
```

Like most of the tools discussed in this section, `filter` both accepts an iterable to process and returns an iterable to generate results in 3.0.

## Multiple Versus Single Iterators

It's interesting to see how the `range` object differs from the built-ins described in this section—it supports `len` and indexing, it is not its own iterator (you make one with `iter` when iterating manually), and it supports multiple iterators over its result that remember their positions independently:

```
>>> R = range(3)                        # range allows multiple iterators
>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)
>>> next(I1)
0
>>> next(I1)
1
>>> I2 = iter(R)                        # Two iterators on one range
>>> next(I2)
0
>>> next(I1)                            # I1 is at a different spot than I2
2
```

By contrast, `zip`, `map`, and `filter` do not support multiple active iterators on the same result:

```
>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                        # Two iterators on one zip
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                            # I2 is at same spot as I1!
```

```
(3, 12)

>>> M = map(abs, (-1, 0, 1))                # Ditto for map (and filter)
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)
StopIteration

>>> R = range(3)                            # But range allows many iterators
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)
0
```

When we code our own iterable objects with classes later in the book (Chapter 29), we'll see that multiple iterators are usually supported by returning new objects for the `iter` call; a single iterator generally means an object returns itself. In Chapter 20, we'll also find that *generator functions and expressions* behave like `map` and `zip` instead of `range` in this regard, supporting a single active iteration. In that chapter, we'll see some subtle implications of one-shot iterators in loops that attempt to scan multiple times.

## Dictionary View Iterators

As we saw briefly in Chapter 8, in Python 3.0 the dictionary `keys`, `values`, and `items` methods return iterable *view* objects that generate result items one at a time, instead of producing result lists all at once in memory. View items maintain the same physical ordering as that of the dictionary and reflect changes made to the underlying dictionary. Now that we know more about iterators, here's the rest of the story:

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()                            # A view object in 3.0, not a list
>>> K
<dict_keys object at 0x026D83C0>

>>> next(K)                                 # Views are not iterators themselves
TypeError: dict_keys object is not an iterator

>>> I = iter(K)                             # Views have an iterator,
>>> next(I)                                 # which can be used manually
'a'                                         # but does not support len(), index
>>> next(I)
'c'

>>> for k in D.keys(): print(k, end=' ')    # All iteration contexts use auto
...
a c b
```

As for all iterators, you can always force a 3.0 dictionary view to build a real list by passing it to the `list` built-in. However, this usually isn't required except to display results interactively or to apply list operations like indexing:

```
>>> K = D.keys()
>>> list(K)                              # Can still force a real list if needed
['a', 'c', 'b']

>>> V = D.values()                       # Ditto for values() and items() views
>>> V
<dict_values object at 0x026D8260>
>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 c 3 b 2
```

In addition, 3.0 dictionaries still have iterators themselves, which return successive keys. Thus, it's not often necessary to call `keys` directly in this context:

```
>>> D                                    # Dictionaries still have own iterator
{'a': 1, 'c': 3, 'b': 2}                 # Returns next key on each iteration
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'

>>> for key in D: print(key, end=' ')    # Still no need to call keys() to iterate
...                                      # But keys is an iterator in 3.0 too!
a c b
```

Finally, remember again that because `keys` no longer returns a list, the traditional coding pattern for scanning a dictionary by sorted keys won't work in 3.0. Instead, convert keys views first with a `list` call, or use the `sorted` call on either a keys view or the dictionary itself, as follows:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D): print(k, D[k], end=' ')     # Best practice key sorting
...
a 1 b 2 c 3
```

## Other Iterator Topics

We'll learn more about both list comprehensions and iterators in Chapter 20, in conjunction with functions, and again in Chapter 29 when we study classes. As you'll see later:

- User-defined functions can be turned into iterable *generator functions*, with `yield` statements.
- List comprehensions morph into iterable *generator expressions* when coded in parentheses.
- User-defined classes are made iterable with `__iter__` or `__getitem__` *operator overloading*.

In particular, user-defined iterators defined with classes allow arbitrary objects and operations to be used in any of the iteration contexts we've met here.

## Chapter Summary

In this chapter, we explored concepts related to looping in Python. We took our first substantial look at the *iteration protocol* in Python—a way for nonsequence objects to take part in iteration loops—and at *list comprehensions*. As we saw, a list comprehension is an expression similar to a `for` loop that applies another expression to all the items in any iterable object. Along the way, we also saw other built-in iteration tools at work and studied recent iteration additions in Python 3.0.

This wraps up our tour of specific procedural statements and related tools. The next chapter closes out this part of the book by discussing documentation options for Python code; documentation is also part of the general syntax model, and it's an important component of well-written programs. In the next chapter, we'll also dig into a set of exercises for this part of the book before we turn our attention to larger structures such as functions. As usual, though, let's first exercise what we've learned here with a quiz.

---

## Test Your Knowledge: Quiz

1. How are `for` loops and iterators related?
2. How are `for` loops and list comprehensions related?
3. Name four iteration contexts in the Python language.
4. What is the best way to read line by line from a text file today?
5. What sort of weapons would you expect to see employed by the Spanish Inquisition?

## Test Your Knowledge: Answers

1. The `for` loop uses the *iteration protocol* to step through items in the object across which it is iterating. It calls the object's `__next__` method (run by the `next` built-in) on each iteration and catches the `StopIteration` exception to determine when to stop looping. Any object that supports this model works in a `for` loop and in other iteration contexts.
2. Both are iteration tools. List comprehensions are a concise and efficient way to perform a common `for` loop task: collecting the results of applying an expression to all items in an iterable object. It's always possible to translate a list comprehension to a `for` loop, and part of the list comprehension expression looks like the header of a `for` loop syntactically.
3. Iteration contexts in Python include the `for` loop; list comprehensions; the `map` built-in function; the `in` membership test expression; and the built-in functions `sorted`, `sum`, `any`, and `all`. This category also includes the `list` and `tuple` built-ins, string `join` methods, and sequence assignments, all of which use the iteration protocol (the `__next__` method) to step across iterable objects one item at a time.
4. The best way to read lines from a text file today is to not read it explicitly at all: instead, open the file within an iteration context such as a `for` loop or list comprehension, and let the iteration tool automatically scan one line at a time by running the file's `next` method on each iteration. This approach is generally best in terms of coding simplicity, execution speed, and memory space requirements.
5. I'll accept any of the following as correct answers: fear, intimidation, nice red uniforms, a comfy chair, and soft pillows.

CHAPTER 15

# The Documentation Interlude

This part of the book concludes with a look at techniques and tools used for documenting Python code. Although Python code is designed to be readable, a few well-placed human-readable comments can do much to help others understand the workings of your programs. Python includes syntax and tools to make documentation easier.

Although this is something of a tools-related concept, the topic is presented here partly because it involves Python's syntax model, and partly as a resource for readers struggling to understand Python's toolset. For the latter purpose, I'll expand here on documentation pointers first given in Chapter 4. As usual, in addition to the chapter quiz this concluding chapter ends with some warnings about common pitfalls and a set of exercises for this part of the text.

## Python Documentation Sources

By this point in the book, you're probably starting to realize that Python comes with an amazing amount of prebuilt functionality—built-in functions and exceptions, predefined object attributes and methods, standard library modules, and more. And we've really only scratched the surface of each of these categories.

One of the first questions that bewildered beginners often ask is: how do I find information on all the built-in tools? This section provides hints on the various documentation sources available in Python. It also presents *documentation strings* (docstrings) and the *PyDoc* system that makes use of them. These topics are somewhat peripheral to the core language itself, but they become essential knowledge as soon as your code reaches the level of the examples and exercises in this part of the book.

As summarized in Table 15-1, there are a variety of places to look for information on Python, with generally increasing verbosity. Because documentation is such a crucial tool in practical programming, we'll explore each of these categories in the sections that follow.

*Table 15-1. Python documentation sources*

| Form | Role |
|---|---|
| # comments | In-file documentation |
| The `dir` function | Lists of attributes available in objects |
| Docstrings: `__doc__` | In-file documentation attached to objects |
| PyDoc: The `help` function | Interactive help for objects |
| PyDoc: HTML reports | Module documentation in a browser |
| The standard manual set | Official language and library descriptions |
| Web resources | Online tutorials, examples, and so on |
| Published books | Commercially available reference texts |

## # Comments

Hash-mark comments are the most basic way to document your code. Python simply ignores all the text following a `#` (as long as it's not inside a string literal), so you can follow this character with words and descriptions meaningful to programmers. Such comments are accessible only in your source files, though; to code comments that are more widely available, you'll need to use docstrings.

In fact, current best practice generally dictates that docstrings are best for larger functional documentation (e.g., "my file does this"), and `#` comments are best limited to smaller code documentation (e.g., "this strange expression does that"). More on docstrings in a moment.

## The dir Function

The built-in `dir` function is an easy way to grab a list of all the attributes available inside an object (i.e., its methods and simpler data items). It can be called on any object that has attributes. For example, to find out what's available in the standard library's `sys` module, import it and pass it to `dir` (these results are from Python 3.0; they might vary slightly on 2.6):

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding',
...more names omitted...]
```

Only some of the many names are displayed here; run these statements on your machine to see the full list.

To find out what attributes are provided in built-in object types, run `dir` on a literal (or existing instance) of the desired type. For example, to see list and string attributes, you can pass empty objects:

```
>>> dir([])
['__add__', '__class__', '__contains__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>> dir('')
['__add__', '__class__', '__contains__', ...more...
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', '
maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
...more names omitted...]
```

`dir` results for any built-in type include a set of attributes that are related to the implementation of that type (technically, operator overloading methods); they all begin and end with double underscores to make them distinct, and you can safely ignore them at this point in the book.

Incidentally, you can achieve the same effect by passing a type name to `dir` instead of a literal:

```
>>> dir(str) == dir('')          # Same result as prior example
True
>>> dir(list) == dir([])
True
```

This works because names like `str` and `list` that were once type converter functions are actually names of types in Python today; calling one of these invokes its constructor to generate an instance of that type. I'll have more to say about constructors and operator overloading methods when we discuss classes in Part VI.

The `dir` function serves as a sort of memory-jogger—it provides a list of attribute names, but it does not tell you anything about what those names mean. For such extra information, we need to move on to the next documentation source.

## Docstrings: __doc__

Besides # comments, Python supports documentation that is automatically attached to objects and retained at runtime for inspection. Syntactically, such comments are coded as strings at the tops of module files and function and class statements, before any other executable code (# comments are OK before them). Python automatically stuffs the strings, known as *docstrings*, into the `__doc__` attributes of the corresponding objects.

### User-defined docstrings

For example, consider the following file, *docstrings.py*. Its docstrings appear at the beginning of the file and at the start of a function and a class within it. Here, I've used triple-quoted block strings for multiline comments in the file and the function, but any sort of string will work. We haven't studied the `def` or `class` statements in detail yet, so ignore everything about them except the strings at their tops:

```
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
    """
    function documentation
    can we have your liver then?
    """
    return x ** 2          # square

class Employee:
    "class documentation"
    pass

print(square(4))
print(square.__doc__)
```

The whole point of this documentation protocol is that your comments are retained for inspection in `__doc__` attributes after the file is imported. Thus, to display the docstrings associated with the module and its objects, we simply import the file and print their `__doc__` attributes, where Python has saved the text:

```
>>> import docstrings
16

    function documentation
    can we have your liver then?

>>> print(docstrings.__doc__)

Module documentation
Words Go Here

>>> print(docstrings.square.__doc__)

    function documentation
    can we have your liver then?

>>> print(docstrings.Employee.__doc__)
    class documentation
```

Note that you will generally want to use `print` to print docstrings; otherwise, you'll get a single string with embedded newline characters.

You can also attach docstrings to *methods* of classes (covered in Part VI), but because these are just `def` statements nested in `class` statements, they're not a special case. To fetch the docstring of a method function inside a class within a module, you would simply extend the path to go through the class: `module.class.method.__doc__` (we'll see an example of method docstrings in Chapter 28).

### Docstring standards

There is no broad standard about what should go into the text of a docstring (although some companies have internal standards). There have been various markup language and template proposals (e.g., HTML or XML), but they don't seem to have caught on in the Python world. And frankly, convincing Python programmers to document their code using handcoded HTML is probably not going to happen in our lifetimes!

Documentation tends to have a low priority amongst programmers in general. Usually, if you get any comments in a file at all, you count yourself lucky. I strongly encourage you to document your code liberally, though—it really is an important part of well-written programs. The point here is that there is presently no standard on the structure of docstrings; if you want to use them, anything goes today.

### Built-in docstrings

As it turns out, built-in modules and objects in Python use similar techniques to attach documentation above and beyond the attribute lists returned by `dir`. For example, to see an actual human-readable description of a built-in module, import it and print its `__doc__` string:

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...more text omitted...
```

Functions, classes, and methods within built-in modules have attached descriptions in their `__doc__` attributes as well:

```
>>> print(sys.getrefcount.__doc__)
getrefcount(object) -> integer

Return the reference count of object.  The count returned is generally
one higher than you might expect, because it includes the (temporary)
...more text omitted...
```

You can also read about built-in functions via their docstrings:

```
>>> print(int.__doc__)
int(x[, base]) -> integer

Convert a string or number to an integer, if possible.  A floating
point argument will be truncated towards zero (this does not include a
...more text omitted...

>>> print(map.__doc__)
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
```

You can get a wealth of information about built-in tools by inspecting their docstrings this way, but you don't have to—the `help` function, the topic of the next section, does this automatically for you.

## PyDoc: The help Function

The docstring technique proved to be so useful that Python now ships with a tool that makes docstrings even easier to display. The standard *PyDoc* tool is Python code that knows how to extract docstrings and associated structural information and format them into nicely arranged reports of various types. Additional tools for extracting and formatting docstrings are available in the open source domain (including tools that may support structured text—search the Web for pointers), but Python ships with PyDoc in its standard library.

There are a variety of ways to launch PyDoc, including command-line script options (see the Python library manual for details). Perhaps the two most prominent PyDoc interfaces are the built-in `help` function and the PyDoc GUI/HTML interface. The `help` function invokes PyDoc to generate a simple textual report (which looks much like a "manpage" on Unix-like systems):

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:

getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object.  The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    ...more omitted...
```

Note that you do not have to import `sys` in order to call `help`, but you do have to import `sys` to get help on `sys`; it expects an object reference to be passed in. For larger objects such as modules and classes, the `help` display is broken down into multiple sections, a few of which are shown here. Run this interactively to see the full report:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...more omitted...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it in builtins.
        ...more omitted...

DATA
    __stderr__ = <io.TextIOWrapper object at 0x0236E950>
    __stdin__ = <io.TextIOWrapper object at 0x02366550>
    __stdout__ = <io.TextIOWrapper object at 0x02366E30>
    ...more omitted...
```

Some of the information in this report is docstrings, and some of it (e.g., function call patterns) is structural information that PyDoc gleans automatically by inspecting objects' internals, when available. You can also use `help` on built-in functions, methods, and types. To get help for a built-in type, use the type name (e.g., `dict` for dictionary, `str` for string, `list` for list). You'll get a large display that describes all the methods available for that type:

```
>>> help(dict)
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary.
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 ...more omitted...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    ...more omitted...

>>> help(ord)
```

```
Help on built-in function ord in module builtins:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Finally, the `help` function works just as well on your modules as it does on built-ins. Here it is reporting on the *docstrings.py* file we coded earlier. Again, some of this is docstrings, and some is information automatically extracted by inspecting objects' structures:

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    function documentation
    can we have your liver then?

>>> help(docstrings.Employee)
Help on class Employee in module docstrings:

class Employee(builtins.object)
 |  class documentation
 |
 |  Data descriptors defined here:
 ...more omitted...

>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\misc\docstrings.py

DESCRIPTION
    Module documentation
    Words Go Here

CLASSES
    builtins.object
        Employee

    class Employee(builtins.object)
     |  class documentation
     |
     |  Data descriptors defined here:
     ...more omitted...

FUNCTIONS
    square(x)
        function documentation
```

```
            can we have your liver then?

    DATA
        spam = 40
```

## PyDoc: HTML Reports

The `help` function is nice for grabbing documentation when working interactively. For a more grandiose display, however, PyDoc also provides a GUI interface (a simple but portable Python/tkinter script) and can render its report in HTML page format, viewable in any web browser. In this mode, PyDoc can run locally or as a remote server in client/server mode; reports contain automatically created hyperlinks that allow you to click your way through the documentation of related components in your application.

To start PyDoc in this mode, you generally first launch the search engine GUI captured in Figure 15-1. You can start this either by selecting the "Module Docs" item in Python's Start button menu on Windows, or by launching the *pydoc.py* script in Python's standard library directory: *Lib* on Windows (run `pydoc.py` with a `-g` command-line argument). Enter the name of a module you're interested in, and press the Enter key; PyDoc will march down your module import search path (`sys.path`) looking for references to the requested module.

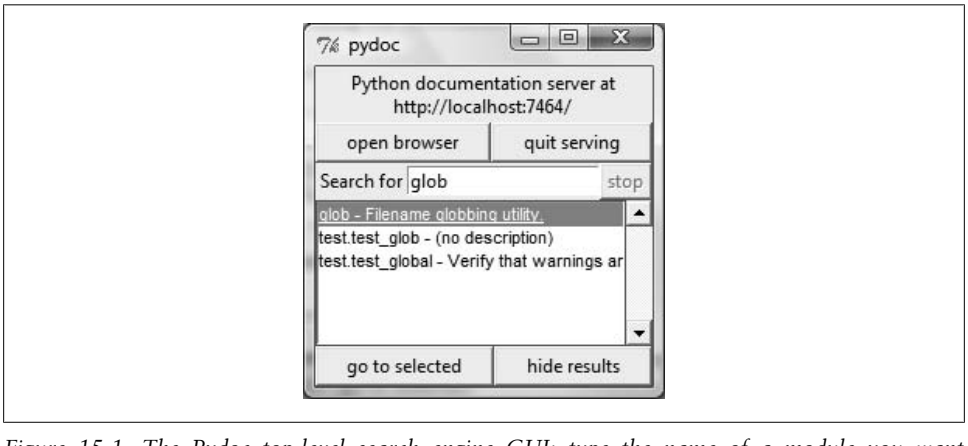


*Figure 15-1. The Pydoc top-level search engine GUI: type the name of a module you want documentation for, press Enter, select the module, and then press "go to selected" (or omit the module name and press "open browser" to see all available modules).*

Once you've found a promising entry, select it and click "go to selected." PyDoc will spawn a web browser on your machine to display the report rendered in HTML format. Figure 15-2 shows the information PyDoc displays for the built-in `glob` module.

Notice the hyperlinks in the Modules section of this page—you can click these to jump to the PyDoc pages for related (imported) modules. For larger pages, PyDoc also generates hyperlinks to sections within the page.

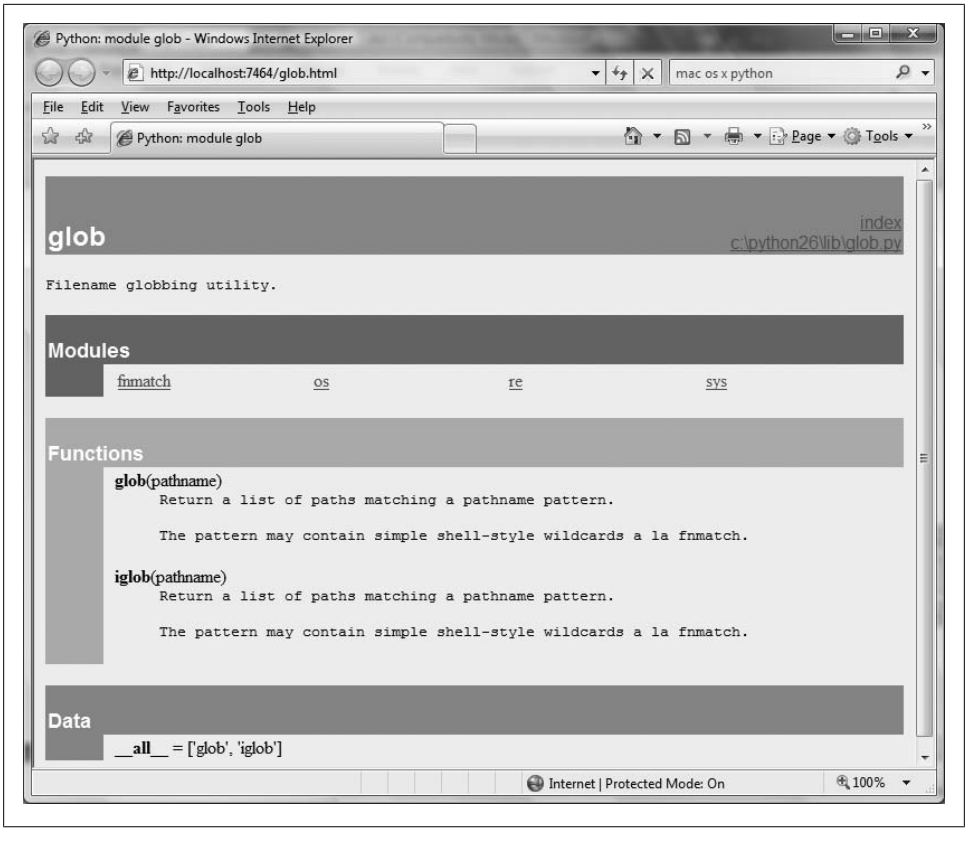


*Figure 15-2. When you find a module in the Figure 15-1 GUI (such as this built-in standard library module) and press "go to selected," the module's documentation is rendered in HTML and displayed in a web browser window like this one.*

Like the `help` function interface, the GUI interface works on user-defined modules as well as built-ins. Figure 15-3 shows the page generated for our *docstrings.py* module file.

PyDoc can be customized and launched in various ways we won't cover here; see its entry in Python's standard library manual for more details. The main thing to take away from this section is that PyDoc essentially gives you implementation reports "for free"—if you are good about using docstrings in your files, PyDoc does all the work of collecting and formatting them for display. PyDoc only helps for objects like functions and modules, but it provides an easy way to access a middle level of documentation for such tools—its reports are more useful than raw attribute lists, and less exhaustive than the standard manuals.

*Cool PyDoc trick of the day*: If you leave the module name empty in the top input field of the window in Figure 15-1 and press the "open browser" button, PyDoc will produce a web page containing a hyperlink to every module you can possibly import on your computer. This includes Python standard library modules, modules of third-party

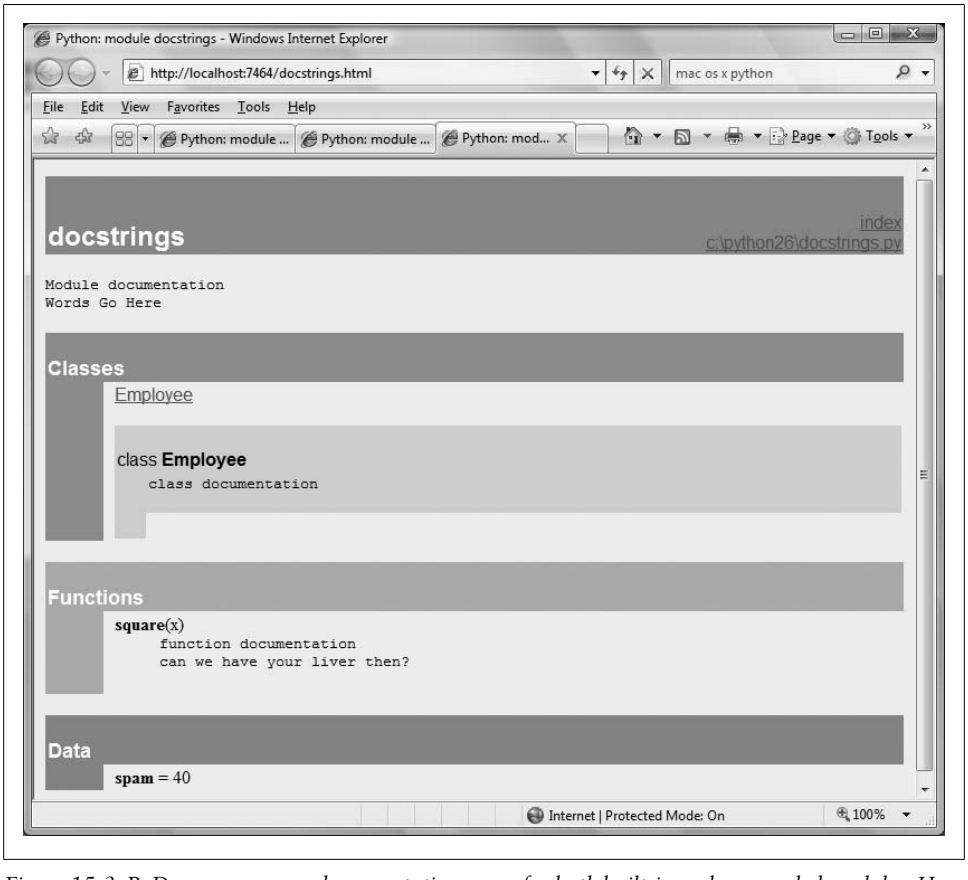


*Figure 15-3. PyDoc can serve up documentation pages for both built-in and user-coded modules. Here is the page for a user-defined module, showing all its documentation strings (docstrings) extracted from the source file.*

extensions you may have installed, user-defined modules on your import search path, and even statically or dynamically linked-in C-coded modules. Such information is hard to come by otherwise without writing code that inspects a set of module sources.

PyDoc can also be run to save the HTML documentation for a module in a file for later viewing or printing; see its documentation for pointers. Also, note that PyDoc might not work well if run on scripts that read from standard input—PyDoc imports the target module to inspect its contents, and there may be no connection for standard input text when it is run in GUI mode. Modules that can be imported without immediate input requirements will always work under PyDoc, though.

## The Standard Manual Set

For the complete and most up-to-date description of the language and its toolset, Python's standard manuals stand ready to serve. Python's manuals ship in HTML and other formats, and they are installed with the Python system on Windows—they are available in your Start button's menu for Python, and they can also be opened from the Help menu within IDLE. You can also fetch the manual set separately from *http://www.python.org* in a variety of formats, or read them online at that site (follow the Documentation link). On Windows, the manuals are a compiled help file to support searches, and the online versions at the Python website include a web-based search page.

When opened, the Windows format of the manuals displays a root page like that in Figure 15-4. The two most important entries here are most likely the Library Reference (which documents built-in types, functions, exceptions, and standard library modules) and the Language Reference (which provides a formal description of language-level details). The tutorial listed on this page also provides a brief introduction for newcomers.


*Figure 15-4. Python's standard manual set, available online at http://www.python.org, from IDLE's Help menu, and in the Windows Start button menu. It's a searchable help file on Windows, and there is a search engine for the online version. Of these, the Library Reference is the one you'll want to use most of the time.*

## Web Resources

At the official Python website (*http://www.python.org*), you'll find links to various Python resources, some of which cover special topics or domains. Click the Documentation link to access an online tutorial and the Beginners Guide to Python. The site also lists non-English Python resources.

You will find numerous Python wikis, blogs, websites, and a host of other resources on the Web today. To sample the online community, try searching for a term like "Python programming" in Google.

## Published Books

As a final resource, you can choose from a large collection of reference books for Python. Bear in mind that books tend to lag behind the cutting edge of Python changes, partly because of the work involved in writing, and partly because of the natural delays built into the publishing cycle. Usually, by the time a book comes out, it's three or more months behind the current Python state. Unlike standard manuals, books are also generally not free.

Still, for many, the convenience and quality of a professionally published text is worth the cost. Moreover, Python changes so slowly that books are usually still relevant years after they are published, especially if their authors post updates on the Web. See the Preface for pointers to other Python books.

# Common Coding Gotchas

Before the programming exercises for this part of the book, let's run through some of the most common mistakes beginners make when coding Python statements and programs. Many of these are warnings I've thrown out earlier in this part of the book, collected here for ease of reference. You'll learn to avoid these pitfalls once you've gained a bit of Python coding experience, but a few words now might help you avoid falling into some of these traps initially:

- **Don't forget the colons**. Always remember to type a `:` at the end of compound statement headers (the first line of an `if`, `while`, `for`, etc.). You'll probably forget at first (I did, and so have most of my 3,000 Python students over the years), but you can take some comfort from the fact that it will soon become an unconscious habit.
- **Start in column 1**. Be sure to start top-level (unnested) code in column 1. That includes unnested code typed into module files, as well as unnested code typed at the interactive prompt.

- **Blank lines matter at the interactive prompt**. Blank lines in compound statements are always ignored in module files, but when you're typing code at the interactive prompt, they end the statement. In other words, blank lines tell the interactive command line that you've finished a compound statement; if you want to continue, don't hit the Enter key at the `...` prompt (or in IDLE) until you're really done.
- **Indent consistently**. Avoid mixing tabs and spaces in the indentation of a block, unless you know what your text editor does with tabs. Otherwise, what you see in your editor may not be what Python sees when it counts tabs as a number of spaces. This is true in any block-structured language, not just Python—if the next programmer has her tabs set differently, she will not understand the structure of your code. It's safer to use all tabs or all spaces for each block.
- **Don't code C in Python**. A reminder for C/C++ programmers: you don't need to type parentheses around tests in `if` and `while` headers (e.g., `if (X==1):`). You can, if you like (any expression can be enclosed in parentheses), but they are fully superfluous in this context. Also, do not terminate all your statements with semicolons; it's technically legal to do this in Python as well, but it's totally useless unless you're placing more than one statement on a single line (the end of a line normally terminates a statement). And remember, don't embed assignment statements in `while` loop tests, and don't use `{}` around blocks (indent your nested code blocks consistently instead).
- **Use simple `for` loops instead of `while` or `range`**. Another reminder: a simple `for` loop (e.g., `for x in seq:`) is almost always simpler to code and quicker to run than a `while`- or `range`-based counter loop. Because Python handles indexing internally for a simple `for`, it can sometimes be twice as fast as the equivalent `while`. Avoid the temptation to count things in Python!
- **Beware of mutables in assignments**. I mentioned this in Chapter 11: you need to be careful about using mutables in a multiple-target assignment (`a = b = []`), as well as in an augmented assignment (`a += [1, 2]`). In both cases, in-place changes may impact other variables. See Chapter 11 for details.
- **Don't expect results from functions that change objects in-place**. We encountered this one earlier, too: in-place change operations like the `list.append` and `list.sort` methods introduced in Chapter 8 do not return values (other than `None`), so you should call them without assigning the result. It's not uncommon for beginners to say something like `mylist = mylist.append(X)` to try to get the result of an `append`, but what this actually does is assign `mylist` to `None`, not to the modified list (in fact, you'll lose your reference to the list altogether).

  A more devious example of this pops up in Python 2.X code when trying to step through dictionary items in a sorted fashion. It's fairly common to see code like `for k in D.keys().sort():`. This almost works—the `keys` method builds a keys list, and the `sort` method orders it—but because the `sort` method returns `None`, the loop fails because it is ultimately a loop over `None` (a nonsequence). This fails even

sooner in Python 3.0, because dictionary keys are views, not lists! To code this correctly, either use the newer `sorted` built-in function, which returns the sorted list, or split the method calls out to statements: `Ks = list(D.keys())`, then `Ks.sort()`, and finally, `for k in Ks:`. This, by the way, is one case where you'll still want to call the `keys` method explicitly for looping, instead of relying on the dictionary iterators—iterators do not sort.

- **Always use parentheses to call a function**. You must add parentheses after a function name to call it, whether it takes arguments or not (e.g., use `function()`, not `function`). In Part IV, we'll see that functions are simply objects that have a special operation—a call that you trigger with the parentheses.

  In classes, this problem seems to occur most often with files; it's common to see beginners type `file.close` to close a file, rather than `file.close()`. Because it's legal to reference a function without calling it, the first version with no parentheses succeeds silently, but it does not close the file!

- **Don't use extensions or paths in imports and reloads**. Omit directory paths and file suffixes in `import` statements (e.g., say `import mod`, not `import mod.py`). (We discussed module basics in Chapter 3 and will continue studying modules in Part V.) Because modules may have other suffixes besides *.py* (*.pyc*, for instance), hardcoding a particular suffix is not only illegal syntax, but doesn't make sense. Any platform-specific directory path syntax comes from module search path settings, not the `import` statement.

## Chapter Summary

This chapter took us on a tour of program documentation—both documentation we write ourselves for our own programs, and documentation available for built-in tools. We met docstrings, explored the online and manual resources for Python reference, and learned how PyDoc's `help` function and web page interface provide extra sources of documentation. Because this is the last chapter in this part of the book, we also reviewed common coding mistakes to help you avoid them.

In the next part of this book, we'll start applying what we already know to larger program constructs: functions. Before moving on, however, be sure to work through the set of lab exercises for this part of the book that appear at the end of this chapter. And even before that, let's run through this chapter's quiz.

---

## Test Your Knowledge: Quiz

1. When should you use documentation strings instead of hash-mark comments?
2. Name three ways you can view documentation strings.

3. How can you obtain a list of the available attributes in an object?
4. How can you get a list of all available modules on your computer?
5. Which Python book should you purchase after this one?

## Test Your Knowledge: Answers

1. Documentation strings (docstrings) are considered best for larger, functional documentation, describing the use of modules, functions, classes, and methods in your code. Hash-mark comments are today best limited to micro-documentation about arcane expressions or statements. This is partly because docstrings are easier to find in a source file, but also because they can be extracted and displayed by the PyDoc system.
2. You can see docstrings by printing an object's `__doc__` attribute, by passing it to PyDoc's `help` function, and by selecting modules in PyDoc's GUI search engine in client/server mode. Additionally, PyDoc can be run to save a module's documentation in an HTML file for later viewing or printing.
3. The built-in `dir(X)` function returns a list of all the attributes attached to any object.
4. Run the PyDoc GUI interface, leave the module name blank, and select "open browser"; this opens a web page containing a link to every module available to your programs.
5. Mine, of course. (Seriously, the Preface lists a few recommended follow-up books, both for reference and for application tutorials.)

## Test Your Knowledge: Part III Exercises

Now that you know how to code basic program logic, the following exercises will ask you to implement some simple tasks with statements. Most of the work is in exercise 4, which lets you explore coding alternatives. There are always many ways to arrange statements, and part of learning Python is learning which arrangements work better than others.

See Part III in Appendix B for the solutions.

1. *Coding basic loops*.
   a. Write a `for` loop that prints the ASCII code of each character in a string named `S`. Use the built-in function `ord(character)` to convert each character to an ASCII integer. (Test it interactively to see how it works.)
   b. Next, change your loop to compute the sum of the ASCII codes of all the characters in a string.

c. Finally, modify your code again to return a new list that contains the ASCII codes of each character in the string. Does the expression `map(ord, S)` have a similar effect? (Hint: see Chapter 14.)

2. *Backslash characters*. What happens on your machine when you type the following code interactively?

```
for i in range(50):
    print('hello %d\n\a' % i)
```

Beware that if it's run outside of the IDLE interface this example may beep at you, so you may not want to run it in a crowded lab. IDLE prints odd characters instead of beeping (see the backslash escape characters in Table 7-2).

3. *Sorting dictionaries*. In Chapter 8, we saw that dictionaries are unordered collections. Write a `for` loop that prints a dictionary's items in sorted (ascending) order. (Hint: use the dictionary `keys` and list `sort` methods, or the newer `sorted` built-in function.)

4. *Program logic alternatives*. Consider the following code, which uses a `while` loop and `found` flag to search a list of powers of 2 for the value of 2 raised to the fifth power (32). It's stored in a module file called *power.py*.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = False
i = 0
while not found and i < len(L):
   if 2 ** X == L[i]:
       found = True
   else:
       i = i+1

if found:
    print('at index', i)
else:
    print(X, 'not found')

C:\book\tests> python power.py
at index 5
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps outlined here to improve it (for all the transformations, you may either type your code interactively or store it in a script file run from the system command line—using a file makes this exercise much easier):

a. First, rewrite this code with a `while` loop `else` clause to eliminate the `found` flag and final `if` statement.

b. Next, rewrite the example to use a `for` loop with an `else` clause, to eliminate the explicit list-indexing logic. (Hint: to get the index of an item, use the list `index` method—`L.index(X)` returns the offset of the first `X` in list `L`.)

c. Next, remove the loop completely by rewriting the example with a simple `in` operator membership expression. (See Chapter 8 for more details, or type this to test: `2 in [1,2,3]`.)

d. Finally, use a `for` loop and the list `append` method to generate the powers-of-2 list (`L`) instead of hardcoding a list literal.

Deeper thoughts:

e. Do you think it would improve performance to move the `2 ** X` expression outside the loops? How would you code that?

f. As we saw in exercise 1, Python includes a `map(function, list)` tool that can generate a powers-of-2 list, too: `map(lambda x: 2 ** x, range(7))`. Try typing this code interactively; we'll meet `lambda` more formally in Chapter 19.

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):`<br>`  return a + b + c[0]` |
| global | `def changer():`<br>`  global x; x = 'new'` |
| nonlocal | `def changer():`<br>`  nonlocal x; x = 'new'` |
| yield | `def squares(x):`<br>`  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!

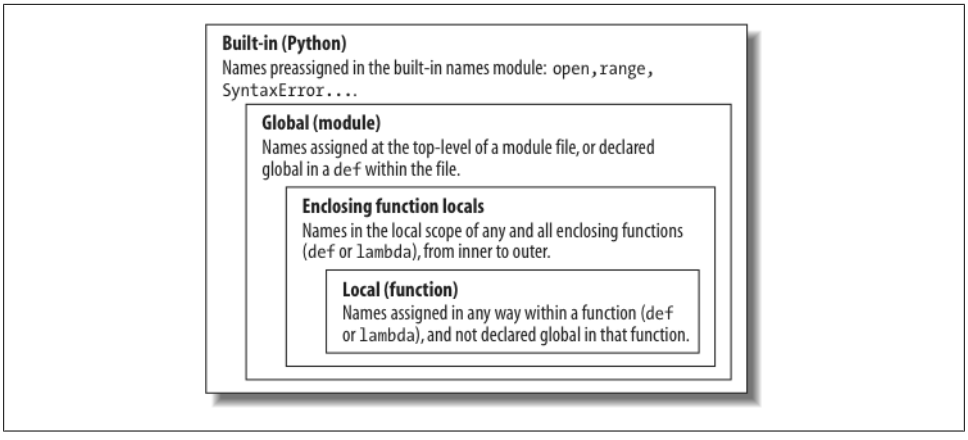


*Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned 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]
```

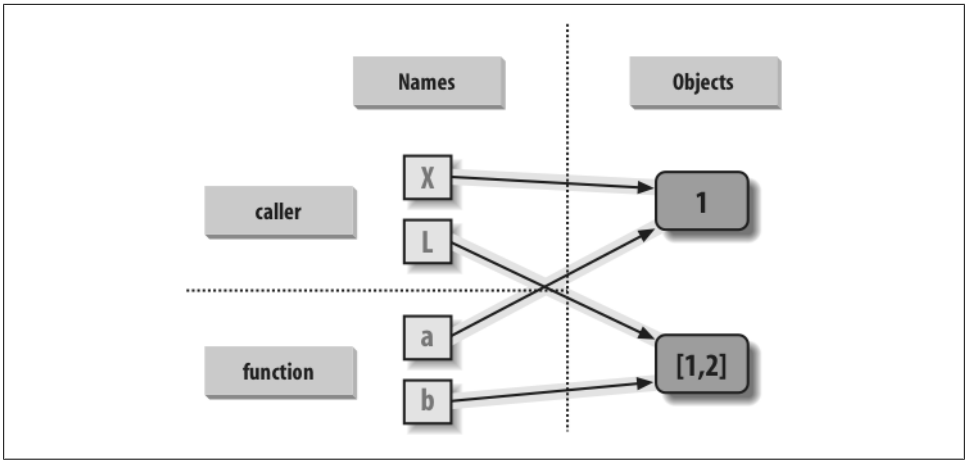


*Figure 18-1. References: arguments. Because arguments are passed by assignment, argument names in the function may share objects with variables in the scope of the call. Hence, in-place changes to mutable arguments in a function can impact the caller. Here, a and b in the function initially reference 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.

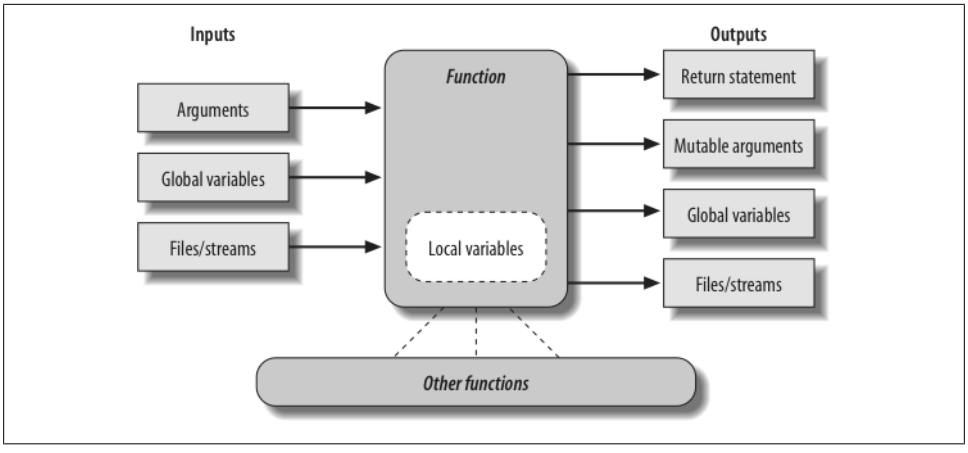


*Figure 19-1. Function execution environment. Functions may obtain input and produce output in a variety of ways, though functions are usually easier to understand and maintain if you use arguments for input and return statements and anticipated mutable argument changes for output. In Python 3, outputs 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?