PART II

Types and Operations

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
'\naaaaaaaaaaaa\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` `x == y, x != y`	Magnitude comparison, set subset and superset; 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` `x - y`	Addition, concatenation; Subtraction, set difference

`x * y`
`x % y`
`x / y, x // y`
	Multiplication, repetition;
	Remainder, format;
	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).

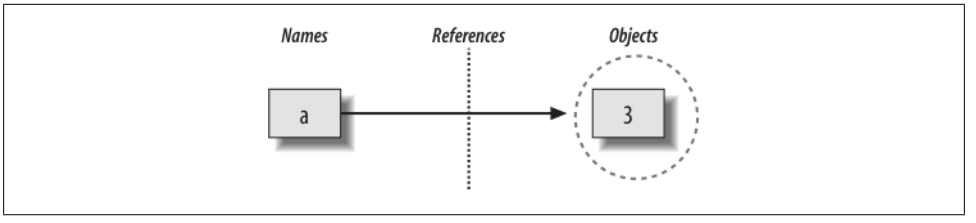

Figure 6-1. Names and objects after running the assignment a = 3. Variable a becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.

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.

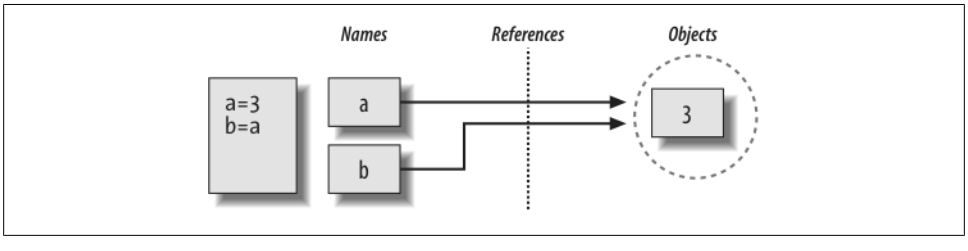

Figure 6-2. Names and objects after next running the assignment b = a. Variable b becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.

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
```

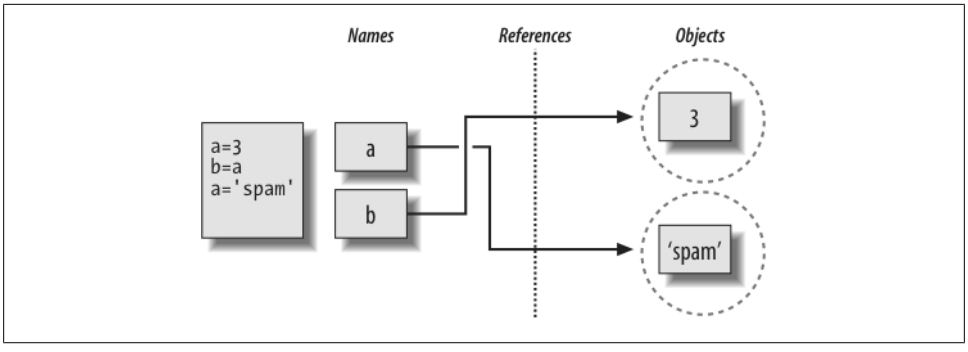

Figure 6-3. Names and objects after finally running the assignment a = 'spam'. Variable a references the new object (i.e., piece of memory) created by running the literal expression 'spam', but variable b still refers to the original object 3. Because this assignment is not an in-place change to the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

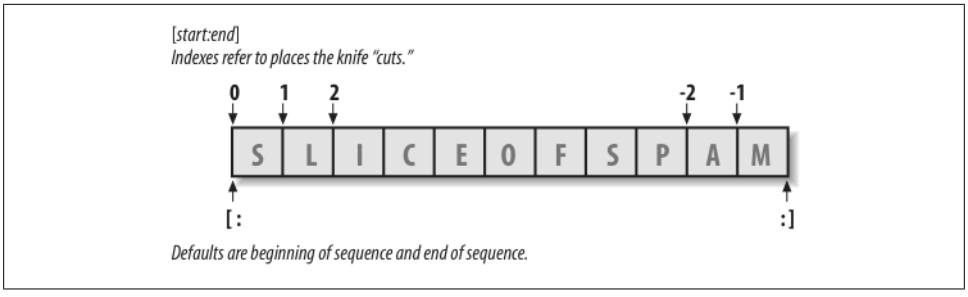

Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset –1 is the last item). Either kind of offset can be used to give positions in indexing and slicing operations.

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

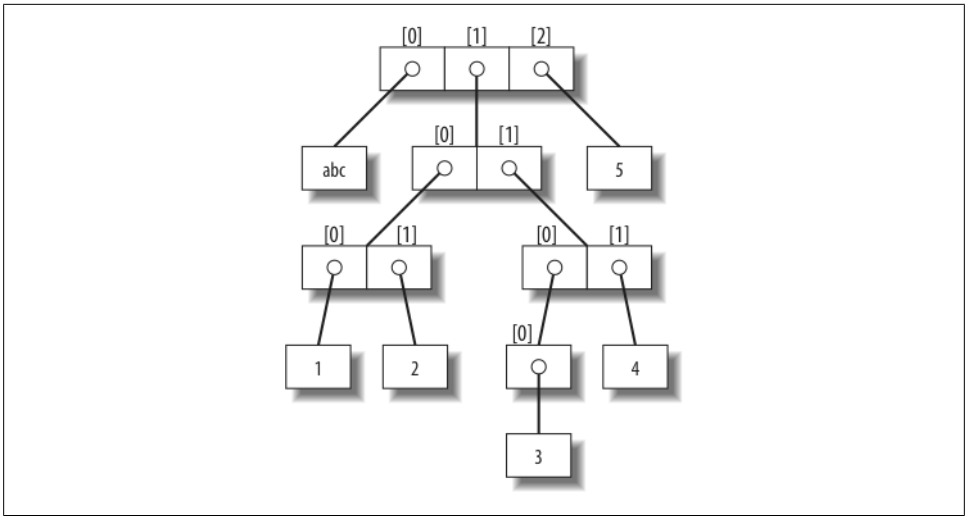

Figure 9-1. A nested object tree with the offsets of its components, created by running the literal expression ['abc', [(1, 2), ([3], 4)], 5]. Syntactically nested objects are internally represented as references (i.e., pointers) to separate pieces of memory.

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

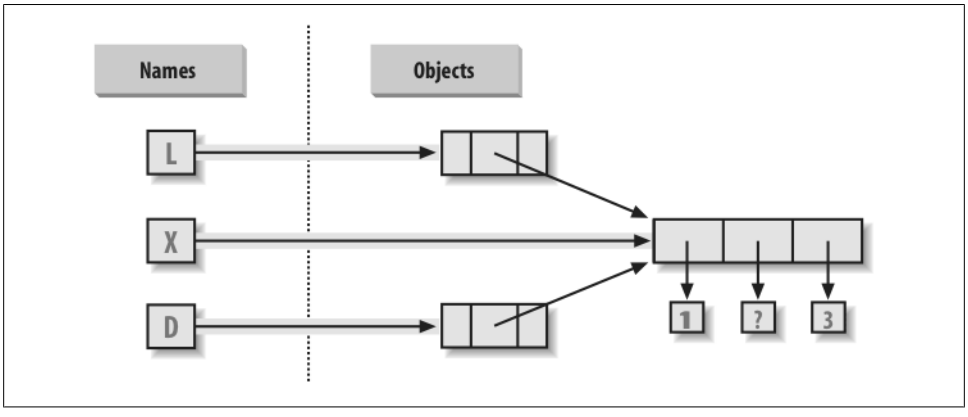

Figure 9-2. Shared object references: because the list referenced by variable X is also referenced from within the objects referenced by L and D, changing the shared list from X makes it look different from L and D, too.

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.

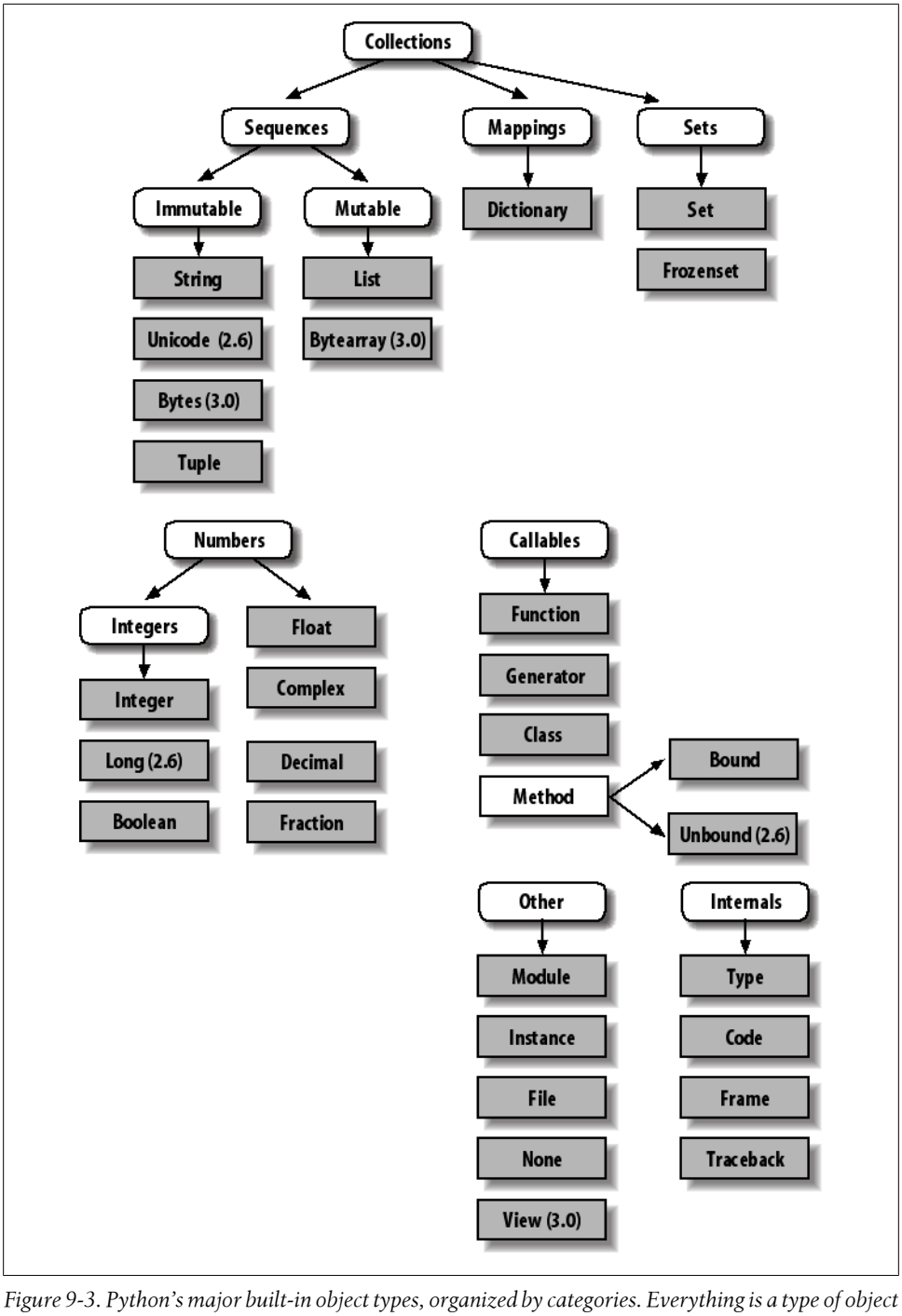

Figure 9-3. Python's major built-in object types, organized by categories. Everything is a type of object in Python, even the type of an object!

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.