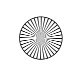
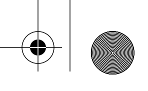
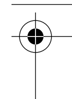
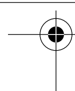
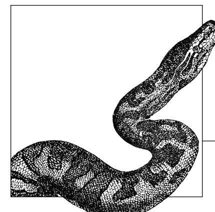







4
The Python Language

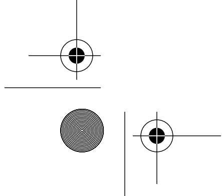
This chapter is a quick guide to the Python language. To learn Python from scratch, I suggest you start with *Learning Python*, by Mark Lutz and David Ascher (O'Reilly). If you already know other programming languages and just want to learn the specific differences of Python, this chapter is for you. However, I'm not trying to teach Python here, so we're going to cover a lot of ground at a pretty fast pace. I focus on teaching the rules, and only secondarily on pointing out best practices and recommended style; for a standard Python style guide, see *http://python.org/doc/peps/pep-0008/*.

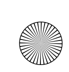
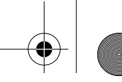
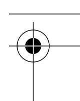
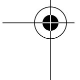
Lexical Structure

The lexical structure of a programming language is the set of basic rules that govern how you write programs in that language. It is the lowest-level syntax of the language and specifies such things as what variable names look like and which characters denote comments. Each Python source file, like any other text file, is a sequence of characters. You can also usefully consider it as a sequence of lines, tokens, or statements. These different lexical views complement and reinforce each other. Python is very particular about program layout, especially with regard to lines and indentation, so you'll want to pay attention to this information if you are coming to Python from another language.

Lines and Indentation

A Python program is composed of a sequence of *logical lines*, each made up of one or more *physical lines*. Each physical line may end with a comment. A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them. A line containing only whitespace, possibly with a comment,



is known as a *blank line*, and Python totally ignores it. In an interactive interpreter session, you must enter an empty physical line (without any whitespace or comment) to terminate a multiline statement.

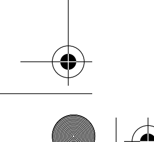
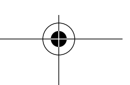
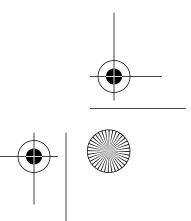
In Python, the end of a physical line marks the end of most statements. Unlike in other languages, you don't normally terminate Python statements with a delimiter, such as a semicolon (;). When a statement is too long to fit on a single physical line, you can join two adjacent physical lines into a logical line by ensuring that the first physical line has no comment and ends with a backslash (\). However, Python automatically joins adjacent physical lines into one logical line if an open parenthesis ((), bracket ([), or brace ({) has not yet been closed, and taking advantage of this mechanism, generally produces more readable code instead of explicitly inserting backslashes at physical line ends. Triple-quoted string literals can also span physical lines. Physical lines after the first one in a logical line are known as *continuation lines*. The indentation issues covered next do not apply to continuation lines but only to the first physical line of each logical line.

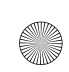
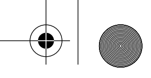
Python uses indentation to express the block structure of a program. Unlike other languages, Python does not use braces, or other begin/end delimiters, around blocks of statements; indentation is the only way to denote such blocks. Each logical line in a Python program is *indented* by the whitespace on its left. A block is a contiguous sequence of logical lines, all indented by the same amount; a logical line with less indentation ends the block. All statements in a block must have the same indentation, as must all clauses in a compound statement. The first statement in a source file must have no indentation (i.e., must not begin with any whitespace). Statements that you type at the interactive interpreter primary prompt >>> (covered in "Interactive Sessions" on page 25) must also have no indentation.

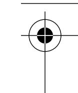
Python logically replaces each tab by up to eight spaces, so that the next character after the tab falls into logical column 9, 17, 25, etc. Standard Python style is to use four spaces (*never* tabs) per indentation level. Don't mix spaces and tabs for indentation, since different tools (e.g., editors, email systems, printers) treat tabs differently. The *-t* and *-tt* options to the Python interpreter (covered in "Command-Line Syntax and Options" on page 23) ensure against inconsistent tab and space usage in Python source code. I recommend you configure your favorite text editor to expand tabs to spaces, so that all Python source code you write always contains just spaces, not tabs. This way, you know that all tools, including Python itself, are going to be perfectly consistent in handling indentation in your Python source files. Optimal Python style is to indent by exactly four spaces.

Character Sets

Normally, a Python source file must be entirely made up of characters from the ASCII set (character codes between 0 and 127). However, you may choose to tell Python that in a certain source file you are using a character set that is a superset of ASCII. In this case, Python allows that specific source file to contain characters



outside the ASCII set, but only in comments and string literals. To accomplish this, start your source file with a comment whose form must be as rigid as the following:

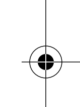
```
# -*- coding: utf-8 -*-
```

Between the `coding:` and the `-*-`, write the name of a codec known to Python, such as `utf-8` or `iso-8859-1`. Note that this *coding directive* comment is taken as such only if it is at the start of a source file (possibly after the "shebang line" covered in "Running Python Programs" on page 28), and that the *only* effect of a coding directive is to let you use non-ASCII characters in string literals and comments.

Tokens

Python breaks each logical line into a sequence of elementary lexical components known as *tokens*. Each token corresponds to a substring of the logical line. The normal token types are *identifiers*, *keywords*, *operators*, *delimiters*, and *literals*, as covered in the following sections. You may freely use whitespace between tokens to separate them. Some whitespace separation is necessary between logically adjacent identifiers or keywords; otherwise, Python would parse them as a single, longer identifier. For example, `printx` is a single identifier; to write the keyword `print` followed by the identifier `x`, you need to insert some whitespace (e.g., `print x`).

Identifiers

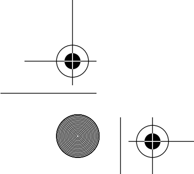
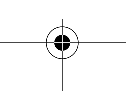
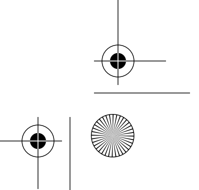


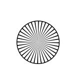
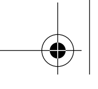
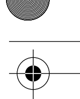
An *identifier* is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter (`A` to `Z` or `a` to `z`) or an underscore (_) followed by zero or more letters, underscores, and digits (`0` to `9`). Case is significant in Python: lowercase and uppercase letters are distinct. Python does not allow punctuation characters such as `@`, `$`, and `%` within identifiers.

Normal Python style is to start class names with an uppercase letter and all other identifiers with a lowercase letter. Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private. Starting an identifier with two leading underscores indicates a strongly private identifier; if the identifier also ends with two trailing underscores, the identifier is a language-defined special name. The identifier `_` (a single underscore) is special in interactive interpreter sessions: the interpreter binds `_` to the result of the last expression statement it has evaluated interactively, if any.

Keywords

Python has 30 keywords, which are identifiers that Python reserves for special syntactic uses. Keywords contain lowercase letters only. You cannot use keywords as regular identifiers. Some keywords begin simple statements or clauses of compound statements, while other keywords are operators. All the keywords are

,ch04.20817 Page 36 Wednesday, July 5, 2006 12:12 PM

covered in detail in this book, either in this chapter, or in Chapters 5, 6, and 7. The keywords in Python are:

```
and        del       for      is      raise
assert     elif      from     lambda  return
break      else      global   not     try
class      except    if       or      while
continue   exec      import   pass    with (2.5)
def        finally   in       print   yield
```

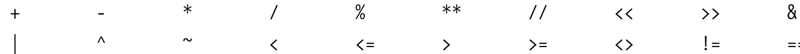
The identifier `with` is a new keyword starting with Python 2.5 (up to Python 2.4, it is a completely normal identifier). The identifier `as`, which is not, strictly speaking, a keyword, is used as a pseudokeyword as part of some statements (specifically, the statements `from`, `import`, and, in Python 2.5, `with`). In Python 2.5, using `with` or `as` as normal identifiers produces warnings. To enable `with` usage as a keyword (and therefore to enable the new `with` statement) in Python 2.5, begin your source file with the statement:

```
from __future__ import with_statement
```

This "import from the future" enables use of the `with` statement in this module.

Operators

Python uses nonalphanumeric characters and character combinations as operators. Python recognizes the following operators, which are covered in detail in "Expressions and Operators" on page 50:



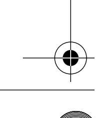
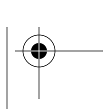
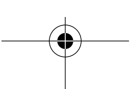
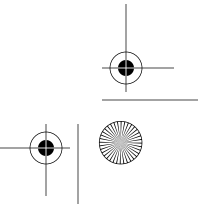
```
+    -    *    /    %    **   //   <<   >>   &
|    ^    ~    <    <=   >    >=   <>   !=   ==
```

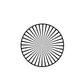
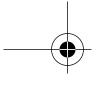
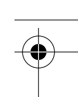
Delimiters

Python uses the following symbols and symbol combinations as delimiters in expressions, lists, dictionaries, various aspects of statements, and strings, among other purposes:

```
(     )     [     ]     {     }
,     :     .     `     =     ;
+=    -=    *=    /=    //=   %=
&=    |=    ^=    >>=   <<=   **=
```

The period (`.`) can also appear in floating-point literals (e.g., `2.3`) and imaginary literals (e.g., `2.3j`). The last two rows list the augmented assignment operators, which lexically are delimiters but also perform an operation. I discuss the syntax for the various delimiters when I introduce the objects or statements with which they are used.



,ch04.20817 Page 37 Wednesday, July 5, 2006 12:12 PM


The following characters have special meanings as part of other tokens:

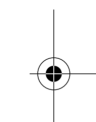
```
'        "        #        \
```

The characters `$` and `?`, all control characters except whitespace, and all characters with ISO codes above 126 (i.e., non-ASCII characters, such as accented letters) can never be part of the text of a Python program, except in comments or string literals (to use non-ASCII characters in comments or string literals, you must start your Python source file with a "coding directive," as covered in "Character Sets" on page 34). This also applies to the character `@` in Python 2.3; however, in Python 2.4, `@` indicates *decorators*, as covered in "Decorators" on page 115.

Literals

A *literal* is a number or string that appears directly in a program. The following are all literals in Python:

```
42                    # Integer literal
3.14                  # Floating-point literal
1.0j                  # Imaginary literal
'hello'               # String literal
"world"               # Another string literal
"""Good
night"""              # Triple-quoted string literal
```



Using literals and delimiters, you can create data values of some other fundamental types:

```
[ 42, 3.14, 'hello' ]    # List
( 100, 200, 300 )        # Tuple
{ 'x':42, 'y':3.14 }     # Dictionary
```

The syntax for literals and other fundamental-type data values is covered in detail in "Data Types" on page 38, when I discuss the various data types Python supports.

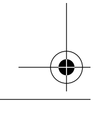
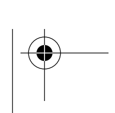
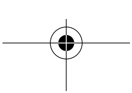
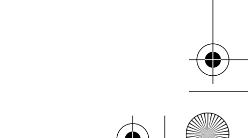
Statements

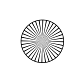
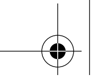
You can consider a Python source file as a sequence of simple and compound statements. Unlike other languages, Python has no declarations or other top-level syntax elements, just statements.

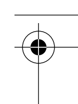
Simple statements

A *simple statement* is one that contains no other statements. A simple statement lies entirely within a logical line. As in other languages, you may place more than one simple statement on a single logical line, with a semicolon (`;`) as the separator. However, one statement per line is the usual Python style, and makes programs more readable.

Any *expression* can stand on its own as a simple statement (I'll discuss expressions in detail in "Expressions and Operators" on page 50). The interactive interpreter shows



the result of an expression statement you enter at the prompt (>>>) and binds the result to a variable named _ (a single underscore). Apart from interactive sessions, expression statements are useful only to call functions (and other *callables*) that have side effects (e.g., ones that perform output, change global variables, or raise exceptions).

An *assignment* is a simple statement that assigns values to variables, as I'll discuss in "Assignment Statements" on page 47. Unlike in some other languages, an assignment in Python is a statement and can never be part of an expression.

Compound statements

A *compound statement* contains one or more other statements and controls their execution. A compound statement has one or more *clauses*, aligned at the same indentation. Each clause has a *header* starting with a keyword and ending with a colon (`:`), followed by a *body*, which is a sequence of one or more statements. When the body contains multiple statements, also known as a *block*, these statements should be placed on separate logical lines after the header line, indented four spaces rightward. The block lexically ends when the indentation returns to that of the clause header (or further left from there, to the indentation of some enclosing compound statement). Alternatively, the body can be a single simple statement, following the `:` on the same logical line as the header. The body may also consist of several simple statements on the same line with semicolons between them, but, as I've already indicated, this is not good style.

Data Types

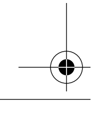
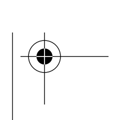
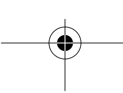
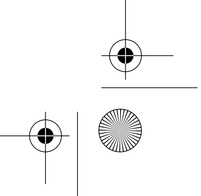
The operation of a Python program hinges on the data it handles. All data values in Python are objects, and each object, or value, has a *type*. An object's type determines which operations the object supports, or, in other words, which operations you can perform on the data value. The type also determines the object's attributes and items (if any) and whether the object can be altered. An object that can be altered is known as a *mutable object*, while one that cannot be altered is an *immutable object*. I cover object attributes and items in detail in "Object attributes and items" on page 46.

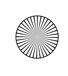
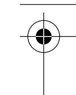
The built-in `type(`*`obj`*`)` accepts any object as its argument and returns the type object that is the type of *`obj`*. Built-in function `isinstance(`*`obj, type`*`)` returns `True` if object *`obj`* has type *`type`* (or any subclass thereof); otherwise, it returns `False`.

Python has built-in types for fundamental data types such as numbers, strings, tuples, lists, and dictionaries, as covered in the following sections. You can also create user-defined types, known as *classes*, as discussed in "Classes and Instances" on page 82.

Numbers

The built-in number objects in Python support integers (plain and long), floating-point numbers, and complex numbers. In Python 2.4, the standard library also offers decimal floating-point numbers, covered in "The decimal Module" on page 372. All numbers in Python are immutable objects, meaning that when you

perform any operation on a number object, you always produce a new number object. Operations on numbers, also known as arithmetic operations, are covered in "Numeric Operations" on page 52.

Note that numeric literals do not include a sign: a leading + or -, if present, is a separate operator, as discussed in "Arithmetic Operations" on page 52.

Integer numbers

Integer literals can be decimal, octal, or hexadecimal. A decimal literal is represented by a sequence of digits in which the first digit is nonzero. To denote an octal literal, use 0 followed by a sequence of octal digits (0 to 7). To indicate a hexadecimal literal, use 0x followed by a sequence of hexadecimal digits (0 to 9 and A to F, in either upper- or lowercase). For example:

```
1, 23, 3493                 # Decimal integers
01, 027, 06645              # Octal integers
0x1, 0x17, 0xDA5            # Hexadecimal integers
```

In practice, you don't need to worry about the distinction between plain and long integers in modern Python, since operating on plain integers produces results that are long integers when needed (i.e., when the result would not fit within the range of plain integers). However, you may choose to terminate any kind of integer literal with a letter L (or l) to explicitly denote a long integer. For instance:

```
1L, 23L, 99999333493L        # Long decimal integers
01L, 027L, 01351033136165L   # Long octal integers
0x1L, 0x17L, 0x17486CBC75L   # Long hexadecimal integers
```

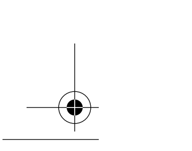
Use uppercase L here, not lowercase l, which might look like the digit 1. The difference between long and plain integers is one of implementation. A long integer has no predefined size limit; it may be as large as memory allows. A plain integer takes up just a few bytes of memory and its minimum and maximum values are dictated by machine architecture. sys.maxint is the largest positive plain integer available, while -sys.maxint-1 is the largest negative one. On 32-bit machines, sys.maxint is 2147483647.

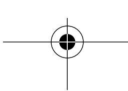
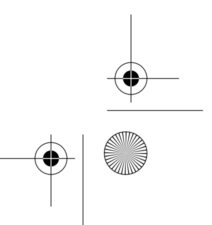
Floating-point numbers

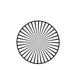
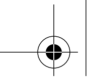
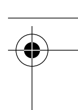
A floating-point literal is represented by a sequence of decimal digits that includes a decimal point (.), an exponent part (an e or E, optionally followed by + or -, followed by one or more digits), or both. The leading character of a floating-point literal cannot be e or E; it may be any digit or a period (.). For example:

```
0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0e0
```

A Python floating-point value corresponds to a C double and shares its limits of range and precision, typically 53 bits of precision on modern platforms. (Python offers no way to find out the exact range and precision of floating-point values on your platform.)



Complex numbers

A complex number is made up of two floating-point values, one each for the real and imaginary parts. You can access the parts of a complex object `z` as read-only attributes `z.real` and `z.imag`. You can specify an imaginary literal as a floating-point or decimal literal followed by a `j` or `J`:

```
0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

The `j` at the end of the literal indicates the square root of `-1`, as commonly used in electrical engineering (some other disciplines use `i` for this purpose, but Python has chosen `j`). There are no other complex literals. To denote any constant complex number, add or subtract a floating-point (or integer) literal and an imaginary one. For example, to denote the complex number that equals one, use expressions like `1+0j` or `1.0+0.0j`.

Sequences

A *sequence* is an ordered container of items, indexed by nonnegative integers. Python provides built-in sequence types known as strings (plain and Unicode), tuples, and lists. Library and extension modules provide other sequence types, and you can write yet others yourself (as discussed in "Sequences" on page 109). You can manipulate sequences in a variety of ways, as discussed in "Sequence Operations" on page 53.

Iterables

A Python concept that generalizes the idea of "sequence" is that of *iterables*, covered in "The for Statement" on page 64 and "Iterators" on page 65. All sequences are iterable: whenever I say that you can use an iterable, you can, in particular, use a sequence (for example, a list).

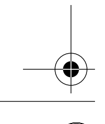
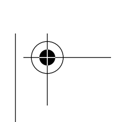
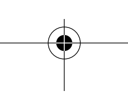
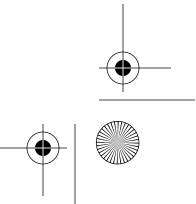
Also, when I say that you can use an iterable, I mean, in general, a *bounded* iterable, which is an iterable that eventually stops yielding items. All sequences are bounded. Iterables, in general, can be unbounded, but if you try to use an unbounded iterable without special precautions, you could easily produce a program that never terminates, or one that exhausts all available memory.

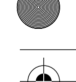
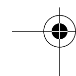
Strings

A built-in string object (plain or Unicode) is a sequence of characters used to store and represent text-based information (plain strings are also sometimes used to store and represent arbitrary sequences of binary bytes). Strings in Python are *immutable*, meaning that when you perform an operation on strings, you always produce a new string object, rather than mutating an existing string. String objects provide many methods, as discussed in detail in "Methods of String Objects" on page 186.

A string literal can be quoted or triple-quoted. A quoted string is a sequence of zero or more characters enclosed in matching quotes, single (`'`) or double (`"`). For example:

```
'This is a literal string'
"This is another string"
```

The two different kinds of quotes function identically; having both allows you to include one kind of quote inside of a string specified with the other kind without needing to escape them with the backslash character (\):

```
'I\'m a Python fanatic'          # a quote can be escaped
"I'm a Python fanatic"           # this way is more readable
```

All other things being equal, using single quotes to denote string literals is a more common Python style. To have a string literal span multiple physical lines, you can use a backslash as the last character of a line to indicate that the next line is a continuation:

```
"A not very long string\
that spans two lines"            # comment not allowed on previous line
```

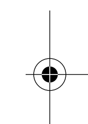
To make the string output on two lines, you can embed a newline in the string:

```
"A not very long string\n\
that prints on two lines"        # comment not allowed on previous line
```

A better approach is to use a triple-quoted string, which is enclosed by matching triplets of quote characters (''' or """):

```
"""An even bigger
string that spans
three lines"""                   # comments not allowed on previous lines
```

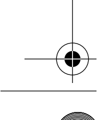
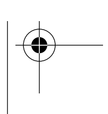
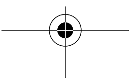
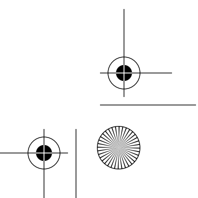
In a triple-quoted string literal, line breaks in the literal are preserved as newline characters in the resulting string object.

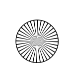
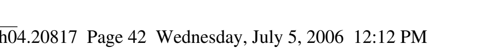


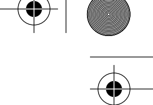
The only character that cannot be part of a triple-quoted string is an unescaped backslash, while a quoted string cannot contain unescaped backslashes, nor line ends, nor the quote character that encloses it. The backslash character starts an escape sequence, which lets you introduce any character in either kind of string. Python's string escape sequences are listed in Table 4-1.

Table 4-1. String escape sequences

Sequence	Meaning	ASCII/ISO code
\<newline>	End of line is ignored	None
\\	Backslash	0x5c
\'	Single quote	0x27
\"	Double quote	0x22
\a	Bell	0x07
\b	Backspace	0x08
\f	Form feed	0x0c
\n	Newline	0x0a
\r	Carriage return	0x0d
\t	Tab	0x09
\v	Vertical tab	0x0b
DDD	Octal value *DDD*	As given
\x*XX*	Hexadecimal value *XX*	As given
other	Any other character	0x5c + as given



A variant of a string literal is a *raw string*. The syntax is the same as for quoted or triple-quoted string literals, except that an `r` or `R` immediately precedes the leading quote. In raw strings, escape sequences are not interpreted as in Table 4-1, but are literally copied into the string, including backslashes and newline characters. Raw string syntax is handy for strings that include many backslashes, as in regular expressions (see "Pattern-String Syntax" on page 201). A raw string cannot end with an odd number of backslashes; the last one would be taken as escaping the terminating quote.

Unicode string literals have the same syntax as other string literals, with a `u` or `U` immediately before the leading quote. Unicode string literals can use `\u` followed by four hex digits to denote Unicode characters and can include the escape sequences listed in Table 4-1. Unicode literals can also include the escape sequence `\N{name}`, where *name* is a standard Unicode name, as listed at *http://www.unicode.org/charts/*. For example, `\N{Copyright Sign}` indicates a Unicode copyright sign character (©). Raw Unicode string literals start with `ur`, not `ru`. Note that raw strings are *not* a different type from ordinary strings: raw strings are just an alternative syntax for literals of the usual two string types, plain (a.k.a. byte strings) and Unicode.

Multiple string literals of any kind (quoted, triple-quoted, raw, Unicode) can be adjacent, with optional whitespace in between. The compiler concatenates such adjacent string literals into a single string object. If any literal in the concatenation is Unicode, the whole result is Unicode. Writing a long string literal in this way lets you present it readably across multiple physical lines and gives you an opportunity to insert comments about parts of the string. For example:

```
marypop = ('supercalifragilistic'  # Open paren -> logical line continues
           'expialidocious')       # Indentation ignored in continuation
```

The string assigned to `marypop` is a single word of 34 characters.

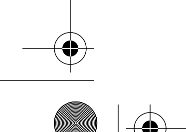
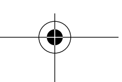
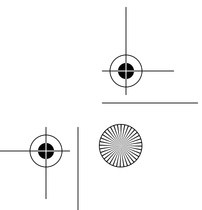
Tuples

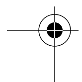
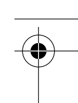
A *tuple* is an immutable ordered sequence of items. The items of a tuple are arbitrary objects and may be of different types. To specify a tuple, use a series of expressions (the *items* of the tuple) separated by commas (`,`). You may optionally place a redundant comma after the last item. You may group tuple items within parentheses, but the parentheses are necessary only where the commas would otherwise have another meaning (e.g., in function calls), or to denote empty or nested tuples. A tuple with exactly two items is often known as a *pair*. To create a tuple of one item (often known as a *singleton*), add a comma to the end of the expression. To denote an empty tuple, use an empty pair of parentheses. Here are some tuples, all enclosed in the optional parentheses:

```
(100, 200, 300)            # Tuple with three items
(3.14,)                    # Tuple with one item
()                         # Empty tuple (parentheses NOT optional!)
```

You can also call the built-in type `tuple` to create a tuple. For example:

```
tuple('wow')
```

This builds a tuple equal to:

```
('w', 'o', 'w')
```

`tuple( )` without arguments creates and returns an empty tuple. When *x* is iterable, `tuple(x)` returns a tuple whose items are the same as the items in *x*.

Lists

A *list* is a mutable ordered sequence of items. The items of a list are arbitrary objects and may be of different types. To specify a list, use a series of expressions (the *items* of the list) separated by commas (`,`) and within brackets (`[]`). You may optionally place a redundant comma after the last item. To denote an empty list, use an empty pair of brackets. Here are some example lists:

```
[42, 3.14, 'hello']       # List with three items
[100]                     # List with one item
[]                        # Empty list
```

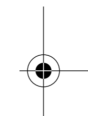
You can also call the built-in type `list` to create a list. For example:

```
list('wow')
```

This builds a list equal to:

```
['w', 'o', 'w']
```

`list( )` without arguments creates and returns an empty list. When *x* is iterable, `list(x)` creates and returns a new list whose items are the same as the items in *x*. You can also build lists with list comprehensions, as discussed in "List comprehensions" on page 67.



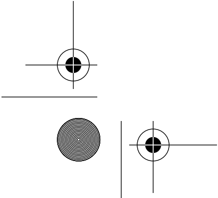
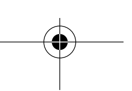
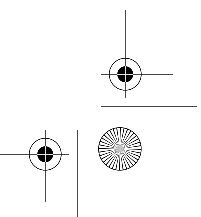
Sets

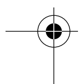
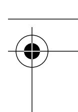
Python 2.4 introduces two built-in set types, `set` and `frozenset`, to represent arbitrarily ordered collections of unique items. These types are equivalent to classes `Set` and `ImmutableSet` found in standard library module `sets`, which also exists in Python 2.3. To ensure that your module uses the best available sets, in any release of Python from 2.3 onwards, place the following code at the start of your module:

```
try:
  set
except NameError:
  from sets import Set as set, ImmutableSet as frozenset
```

Items in a set may be of different types, but they must be *hashable* (see `hash` on page 162). Instances of type `set` are mutable, and therefore not hashable; instances of type `frozenset` are immutable and hashable. So you can't have a set whose items are sets, but you can have a set (or frozenset) whose items are frozensets. Sets and frozensets are *not* ordered.

To create a set, call the built-in type `set` with no argument (this means an empty set) or one argument that is iterable (this means a set whose items are the items of the iterable).

Dictionaries

A *mapping* is an arbitrary collection of objects indexed by nearly arbitrary values called *keys*. Mappings are mutable and, unlike sequences, are *not* ordered.

Python provides a single built-in mapping type, the dictionary type. Library and extension modules provide other mapping types, and you can write others yourself (as discussed in "Mappings" on page 110). Keys in a dictionary may be of different types, but they must be *hashable* (see hash on page 162). Values in a dictionary are arbitrary objects and may be of different types. An *item* in a dictionary is a key/value pair. You can think of a dictionary as an associative array (known in other languages as a "map," "hash table," or "hash").

To specify a dictionary, you can use a series of pairs of expressions (the pairs are the items of the dictionary) separated by commas (,) within braces ({}). You may optionally place a redundant comma after the last item. Each item in a dictionary is written as *key*:*value*, where *key* is an expression giving the item's key and *value* is an expression giving the item's value. If a key appears more than once in a dictionary literal, only one of the items with that key is kept in the resulting dictionary object—dictionaries do not allow duplicate keys. To denote an empty dictionary, use an empty pair of braces. Here are some dictionaries:

```
{'x':42, 'y':3.14, 'z':7 }   # Dictionary with three items and string keys
{1:2, 3:4 }                  # Dictionary with two items and integer keys
{}                           # Empty dictionary
```

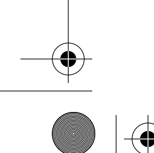
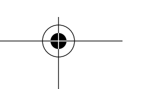
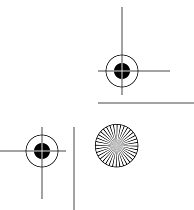
You can also call the built-in type dict to create a dictionary in a way that, while less concise, can sometimes be more readable. For example, the dictionaries in this last snippet can also, equivalently, be written as, respectively:

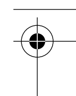
```
dict(x=42, y=3.14, z=7)      # Dictionary with three items and string keys
dict([[1, 2], [3, 4]])       # Dictionary with two items and integer keys
dict( )                      # Empty dictionary
```

dict( ) without arguments creates and returns an empty dictionary. When the argument *x* to dict is a mapping, dict returns a new dictionary object with the same keys and values as *x*. When *x* is iterable, the items in *x* must be pairs, and dict(*x*) returns a dictionary whose items (key/value pairs) are the same as the items in *x*. If a key appears more than once in *x*, only the *last* item with that key is kept in the resulting dictionary.

When you call dict, in addition to or instead of the positional argument *x* you may pass *named arguments*, each with the syntax *name*=*value*, where *name* is an identifier to use as an item's key and *value* is an expression giving the item's value. When you call dict and pass both a positional argument and one or more named arguments, if a key appears both in the positional argument and as a named argument, Python associates to that key the value given with the named argument (i.e., the named argument "wins").

You can also create a dictionary by calling dict.fromkeys. The first argument is an iterable whose items become the keys of the dictionary; the second argument is the value that corresponds to each key (all keys initially have the same corresponding



value). If you omit the second argument, the value corresponding to each key is None. For example:

```
dict.fromkeys('hello', 2)   # same as {'h':2, 'e':2, 'l':2, 'o':2}
dict.fromkeys([1, 2, 3])    # same as {1:None, 2:None, 3:None}
```

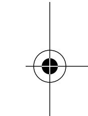
None

The built-in None denotes a null object. None has no methods or other attributes. You can use None as a placeholder when you need a reference but you don't care what object you refer to, or when you need to indicate that no object is there. Functions return None as their result unless they have specific return statements coded to return other values.

Callables

In Python, callable types are those whose instances support the function call operation (see "Calling Functions" on page 73). Functions are callable. Python provides several built-in functions (see "Built-in Functions" on page 158) and supports user-defined functions (see "The def Statement" on page 70). Generators are also callable (see "Generators" on page 78).

Types are also callable, as we already saw for the dict, list, and tuple built-in types. (See "Built-in Types" on page 154 for a complete list of built-in types.) As we'll discuss in "Python Classes" on page 82, class objects (user-defined types) are also callable. Calling a type normally creates and returns a new instance of that type.

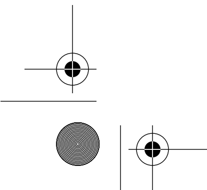


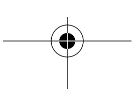
Other callables are *methods*, which are functions bound to class attributes and instances of classes that supply a special method named `__call__`.

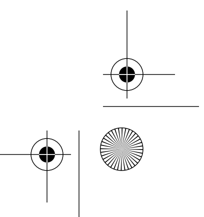
Boolean Values

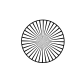
Every data value in Python can be taken as a truth value: true or false. Any nonzero number or nonempty container (e.g., string, tuple, list, set, or dictionary) is true. 0 (of any numeric type), None, and empty containers are false. Be careful about using a floating-point number as a truth value: such use is equivalent to comparing the number for exact equality with zero, and floating-point numbers should almost never be compared for exact equality!

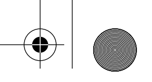
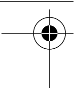
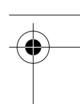
Built-in type bool is a subclass of int. The only two values of type bool are True and False, which have string representations of 'True' and 'False', but also numerical values of 1 and 0, respectively. Several built-in functions return bool results, as do comparison operators. You can call bool(*x*) with any *x* as the argument. The result is True if *x* is true and False if *x* is false. Good Python style is not to use such calls when they are redundant: *always* write `if x:`, *never* `if bool(x):`, `if x==True:`, `if bool(x)==True`, and so on.









Variables and Other References

A Python program accesses data values through references. A *reference* is a name that refers to the location in memory of a value (object). References take the form of variables, attributes, and items. In Python, a variable or other reference has no intrinsic type. The object to which a reference is bound at a given time always has a type, but a given reference may be bound to objects of various types during the program's execution.

Variables

In Python there are no declarations. The existence of a variable begins with a statement that *binds* the variable, or, in other words, sets a name to hold a reference to some object. You can also *unbind* a variable, resetting the name so it no longer holds a reference. Assignment statements are the most common way to bind variables and other references. The `del` statement unbinds references.

Binding a reference that was already bound is also known as *rebinding* it. Whenever I mention binding in this book, I implicitly include rebinding except where I explicitly exclude it. Rebinding or unbinding a reference has no effect on the object to which the reference was bound, except that an object disappears when nothing refers to it. The automatic cleanup of objects bereft of references is known as *garbage collection*.

You can name a variable with any identifier except the 30 that are reserved as Python's keywords (see "Keywords" on page 35). A variable can be global or local. A *global variable* is an attribute of a module object (Chapter 7 covers modules). A *local variable* lives in a function's local namespace (see "Namespaces" on page 76).

Object attributes and items

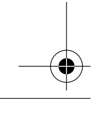
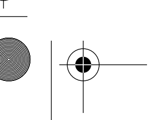
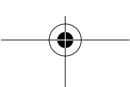
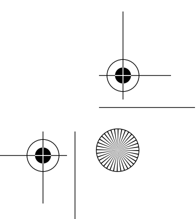
The main distinction between the attributes and items of an object is in the syntax you use to access them. An *attribute* of an object is denoted by a reference to the object, followed by a period (`.`), followed by an identifier known as the *attribute name* (for example, `x.y` refers to one of the attributes of the object bound to name `x`, specifically that attribute which is named `y`).

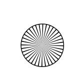
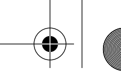
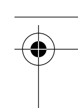
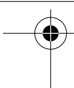
An *item* of an object is denoted by a reference to the object, followed by an expression within brackets (`[]`). The expression in brackets is known as the item's *index* or *key*, and the object is known as the item's *container* (for example, `x[y]` refers to the item at the key or index bound to name `y`, within the container object bound to name `x`).

Attributes that are callable are also known as *methods*. Python draws no strong distinctions between callable and noncallable attributes, as some other languages do. All rules about attributes also apply to callable attributes (methods).

Accessing nonexistent references

A common programming error is trying to access a reference that does not exist. For example, a variable may be unbound, or an attribute name or item index may

not be valid for the object to which you apply it. The Python compiler, when it analyzes and compiles source code, diagnoses only syntax errors. Compilation does not diagnose semantic errors, such as trying to access an unbound attribute, item, or variable. Python diagnoses semantic errors only when the errant code executes, i.e., *at runtime*. When an operation is a Python semantic error, attempting it raises an exception (see Chapter 6). Accessing a nonexistent variable, attribute, or item, just like any other semantic error, raises an exception.

Assignment Statements

Assignment statements can be plain or augmented. Plain assignment to a variable (e.g., *name*=*value*) is how you create a new variable or rebind an existing variable to a new value. Plain assignment to an object attribute (e.g., *x.attr*=*value*) is a request to object *x* to create or rebind attribute *attr*. Plain assignment to an item in a container (e.g., *x*[*k*]=*value*) is a request to container *x* to create or rebind the item with index *k*.

Augmented assignment (e.g., *name*+=*value*) cannot, per se, create new references. Augmented assignment can rebind a variable, ask an object to rebind one of its existing attributes or items, or request the target object to modify itself (an object may, of course, create whatever it wants in response to such requests). When you make a request to an object, it is up to the object to decide whether to honor the request or raise an exception.

Language

Plain assignment

A plain assignment statement in the simplest form has the syntax:

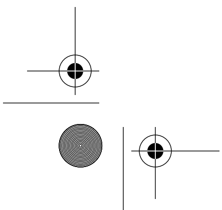
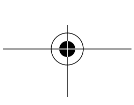
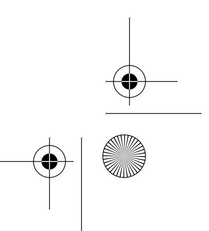
```
target = expression
```

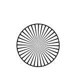
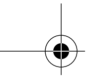
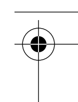
The target is also known as the lefthand side (LHS), and the expression is the righthand side (RHS). When the assignment executes, Python evaluates the RHS expression, then binds the expression's value to the LHS target. The binding does not depend on the type of the value. In particular, Python draws no strong distinction between callable and noncallable objects, as some other languages do, so you can bind functions, methods, types, and other callables to variables, just as you can numbers, strings, lists, and so on.

Details of the binding do depend on the kind of target, however. The target in an assignment may be an identifier, an attribute reference, an indexing, or a slicing:

An identifier
: Is a variable's name. Assignment to an identifier binds the variable with this name.

An attribute reference
: Has the syntax *obj.name*. *obj* is an arbitrary expression, and *name* is an identifier, known as an *attribute name* of the object. Assignment to an attribute reference asks object *obj* to bind its attribute named *name*.

An indexing

Has the syntax *obj*[*expr*]. *obj* and *expr* are arbitrary expressions. Assignment to an indexing asks container *obj* to bind its item indicated by the value of *expr*, also known as the index or key of the item in the container.

A slicing

Has the syntax *obj*[*start*:*stop*] or *obj*[*start*:*stop*:*stride*]. *obj*, *start*, *stop*, and *stride* are arbitrary expressions. *start*, *stop*, and *stride* are all optional (i.e., *obj*[:*stop*:] and *obj*[:*stop*] are also syntactically correct slicings, equivalent to *obj*[None:*stop*:None]). Assignment to a slicing asks container *obj* to bind or unbind some of its items. Assigning to a slicing such as *obj*[*start*:*stop*:*stride*] is equivalent to assigning to the indexing *obj*[slice(*start*, *stop*, *stride*)], where slice is a Python built-in type (see slice on page 156) whose instances represent slices.

I'll come back to indexing and slicing targets when I discuss operations on lists, in "Modifying a list" on page 56, and on dictionaries, in "Indexing a Dictionary" on page 60.

When the target of the assignment is an identifier, the assignment statement specifies the binding of a variable. This is never disallowed: when you request it, it takes place. In all other cases, the assignment statement specifies a request to an object to bind one or more of its attributes or items. An object may refuse to create or rebind some (or all) attributes or items, raising an exception if you attempt a disallowed creation or rebinding (see also __setattr__ on page 108 and __setitem__ on page 112).

You can give multiple targets and equals signs (=) in a plain assignment. For example:

```
a = b = c = 0
```

binds variables a, b, and c to the same value, 0. Each time the statement executes, the RHS expression is evaluated just once, no matter how many targets are part of the statement. Each target then gets bound to the single object returned by the expression, just as if several simple assignments executed one after the other.

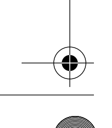
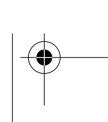
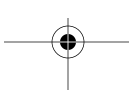
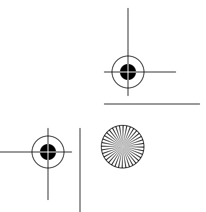
The target in a plain assignment can list two or more references separated by commas, optionally enclosed in parentheses or brackets. For example:

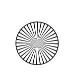
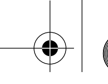
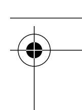
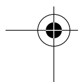
```
a, b, c = x
```

This statement requires x to be an iterable with exactly three items, and binds a to the first item, b to the second, and c to the third. This kind of assignment is known as an *unpacking assignment*. The RHS expression must be an iterable with exactly as many items as there are references in the target; otherwise, Python raises an exception. Each reference in the target gets bound to the corresponding item in the RHS. An unpacking assignment can also be used to swap references:

```
a, b = b, a
```

This assignment statement rebinds name *a* to what name *b* was bound to, and vice versa.

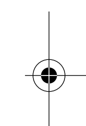





Augmented assignment

An augmented assignment differs from a plain assignment in that, instead of an equals sign (=) between the target and the expression, it uses an *augmented operator*, which is a binary operator followed by =. The augmented operators are +=, -=, *=, /=, //=, %=, **=, |=, >>=, <<=, &=, and ^=. An augmented assignment can have only one target on the LHS; augmented assignment doesn't support multiple targets.

In an augmented assignment, just as in a plain one, Python first evaluates the RHS expression. Then, if the LHS refers to an object that has a special method for the appropriate *in-place* version of the operator, Python calls the method with the RHS value as its argument. It is up to the method to modify the LHS object appropriately and return the modified object ("Special Methods" on page 104 covers special methods). If the LHS object has no appropriate in-place special method, Python applies the corresponding binary operator to the LHS and RHS objects, then rebinds the target reference to the operator's result. For example, `x+=y` is like `x=x.__iadd__(y)` when `x` has special method `__iadd__`. Otherwise, `x+=y` is like `x=x+y`.

Language



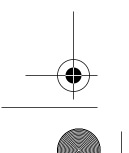
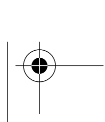
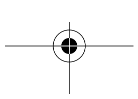
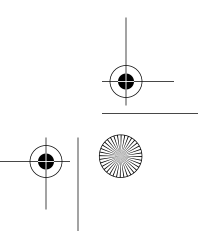
Augmented assignment never creates its target reference; the target must already be bound when augmented assignment executes. Augmented assignment can rebind the target reference to a new object or modify the same object to which the target reference was already bound. Plain assignment, in contrast, can create or rebind the LHS target reference, but it never modifies the object, if any, to which the target reference was previously bound. The distinction between objects and references to objects is crucial here. For example, `x=x+y` does not modify the object to which name `x` was originally bound. Rather, it rebinds the name `x` to refer to a new object. `x+=y`, in contrast, modifies the object to which the name `x` is bound when that object has special method `__iadd__`; otherwise, `x+=y` rebinds the name `x` to a new object, just like `x=x+y`.

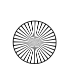
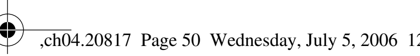
del Statements

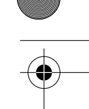
Despite its name, a `del` statement does not delete objects; rather, it unbinds references. Object deletion may automatically follow as a consequence, by garbage collection, when no more references to an object exist.

A `del` statement consists of the keyword `del`, followed by one or more target references separated by commas (,). Each target can be a variable, attribute reference, indexing, or slicing, just like for assignment statements, and must be bound at the time `del` executes. When a `del` target is an identifier, the `del` statement means to unbind the variable. If the identifier was bound, unbinding it is never disallowed; when requested, it takes place.

In all other cases, the `del` statement specifies a request to an object to unbind one or more of its attributes or items. An object may refuse to unbind some (or all) attributes or items, raising an exception if you attempt a disallowed unbinding (see also `__delattr__` on page 106 and `__delitem__` on page 112). Unbinding a slicing normally has the same effect as assigning an empty sequence to that slice, but it is up to the container object to implement this equivalence.



Expressions and Operators

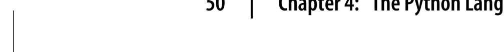
An *expression* is a phrase of code that Python evaluates to produce a value. The simplest expressions are literals and identifiers. You build other expressions by joining subexpressions with the operators and/or delimiters in Table 4-2. This table lists operators in decreasing order of precedence, higher precedence before lower. Operators listed together have the same precedence. The third column lists the associativity of the operator: L (left-to-right), R (right-to-left), or NA (nonassociative).

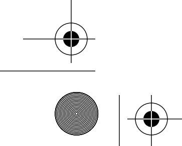
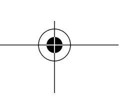
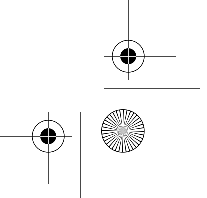
Table 4-2. Operator precedence in expressions

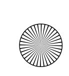
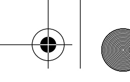
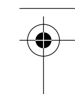
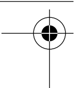
Operator	Description	Associativity
`` `expr,...` ``	String conversion	NA
`{key:expr,...}`	Dictionary creation	NA
`[expr,...]`	List creation	NA
`(expr,...)`	Tuple creation or just parentheses	NA
`f(expr,...)`	Function call	L
`x[index:index]`	Slicing	L
`x[index]`	Indexing	L
`x.attr`	Attribute reference	L
`x**y`	Exponentiation (*x* to *y*th power)	R
`~x`	Bitwise NOT	NA
`+x`, `-x`	Unary plus and minus	NA
`x*y`, `x/y`, `x//y`, `x%y`	Multiplication, division, truncating division, remainder	L
`x+y`, `x-y`	Addition, subtraction	L
`x<<y`, `x>>y`	Left-shift, right-shift	L
`x&y`	Bitwise AND	L
`x^y`	Bitwise XOR	L
`x|y`	Bitwise OR	L
`x<y`, `x<=y`, `x>y`, `x>=y`, `x<>y`, `x!=y`, `x==y`	Comparisons (less than, less than or equal, greater than, greater than or equal, inequality, equality) [a]	NA
`x is y`, `x is not y`	Identity tests	NA
`x in y`, `x not in y`	Membership tests	NA
`not x`	Boolean NOT	NA
`x and y`	Boolean AND	L
`x or y`	Boolean OR	L
`lambda arg,...: expr`	Anonymous simple function	NA

[a] `<>` and `!=` are alternate forms of the same operator. `!=` is the preferred version; `<>` is obsolete.

In Table 4-2, *expr*, *key*, *f*, *index*, *x*, and *y* indicate any expression, while *attr* and *arg* indicate any identifier. The notation `,...` means commas join zero or more repetitions, except for string conversion, where you need one or more repetitions.



A trailing comma is allowed and innocuous in all such cases, except for string conversion, where it's forbidden. The string conversion operator, with its quirky behavior, is not recommended; use built-in function `repr` (covered in `repr` on page 166) instead.

Comparison Chaining

You can *chain* comparisons, implying a logical `and`. For example:

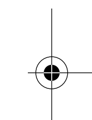
```
a < b <= c < d
```

has the same meaning as:

```
a < b and b <= c and c < d
```

The chained form is more readable and evaluates each subexpression once at the most.

Short-Circuiting Operators



Operators `and` and `or` *short-circuit* their operands' evaluation: the righthand operand evaluates only if its value is needed to get the truth value of the entire `and` or `or` operation.

In other words, *`x`* `and` *`y`* first evaluates *`x`*. If *`x`* is false, the result is *`x`*; otherwise, the result is *`y`*. Similarly, *`x`* `or` *`y`* first evaluates *`x`*. If *`x`* is true, the result is *`x`*; otherwise, the result is *`y`*.

`and` and `or` don't force their results to be `True` or `False`, but rather return one or the other of their operands. This lets you use these operators more generally, not just in Boolean contexts. `and` and `or`, because of their short-circuiting semantics, differ from all other operators, which fully evaluate all operands before performing the operation. `and` and `or` let the left operand act as a *guard* for the right operand.

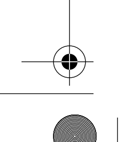
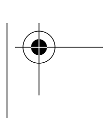
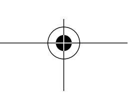
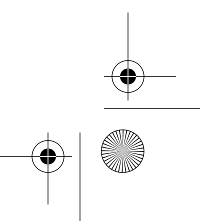
The Python 2.5 ternary operator

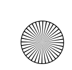
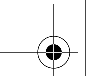
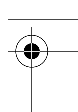
Python 2.5 introduces another short-circuiting operator, the ternary operator `if`/`else`:

```
whentrue if condition else whenfalse
```

Each of *`whentrue`*, *`whenfalse`*, and *`condition`* is an arbitrary expression. *`condition`* evaluates first. If *`condition`* is true, the result is *`whentrue`*; otherwise, the result is *`whenfalse`*. Only one of the two subexpressions *`whentrue`* and *`whenfalse`* evaluates, depending on the truth value of *`condition`*.

The order of the three subexpressions in this new ternary operator may be a bit confusing. Also, a recommended style is to always place parentheses around the whole expression.

Numeric Operations

Python supplies the usual numeric operations, as we've just seen in Table 4-2. Numbers are immutable objects: when you perform numeric operations on number objects, you always produce a new number object and never modify existing ones. You can access the parts of a complex object `z` as read-only attributes `z.real` and `z.imag`. Trying to rebind these attributes on a complex object raises an exception.

A number's optional `+` or `-` sign, and the `+` that joins a floating-point literal to an imaginary one to make a complex number, are not part of the literals' syntax. They are ordinary operators, subject to normal operator precedence rules (see Table 4-2). For example, `-2**2` evaluates to `-4`: exponentiation has higher precedence than unary minus, so the whole expression parses as `-(2**2)`, not as `(-2)**2`.

Numeric Conversions

You can perform arithmetic operations and comparisons between any two numbers of Python built-in types. If the operands' types differ, *coercion* applies: Python converts the operand with the "smaller" type to the "larger" type. The types, in order from smallest to largest, are integers, long integers, floating-point numbers, and complex numbers.

You can request an explicit conversion by passing a noncomplex numeric argument to any of the built-in number types: `int`, `long`, `float`, and `complex`. `int` and `long` drop their argument's fractional part, if any (e.g., `int(9.8)` is `9`). You can also call `complex` with two numeric arguments, giving real and imaginary parts. You cannot convert a `complex` to another numeric type in this way, because there is no single unambiguous way to convert a complex number into, e.g., a `float`.

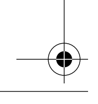
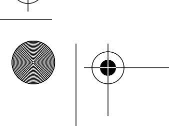
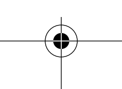
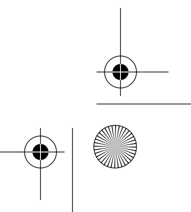
Each built-in numeric type can also take a string argument with the syntax of an appropriate numeric literal, with small extensions: the argument string may have leading and/or trailing whitespace, may start with a sign, and, for complex numbers, may sum or subtract a real part and an imaginary one. `int` and `long` can also be called with two arguments: the first one a string to convert, and the second the *radix*, an integer between 2 and 36 to use as the base for the conversion (e.g., `int('101', 2)` returns `5`, the value of `'101'` in base `2`).

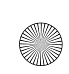
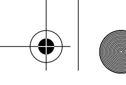
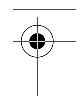
Arithmetic Operations

Python arithmetic operations behave in rather obvious ways, with the possible exception of division and exponentiation.

Division

If the right operand of `/`, `//`, or `%` is `0`, Python raises a runtime exception. The `//` operator performs truncating division, which means it returns an integer result (converted to the same type as the wider operand) and ignores the remainder, if any. When both operands are integers (plain or long), the `/` operator behaves like `//` if the switch *-Qold* was used on the Python command line (*-Qold* is the default in Python 2.3, 2.4, and 2.5). Otherwise, `/` performs true division, returning a

floating-point result (or a complex result if either operand is a complex number). To have / perform true division on integer operands in Python 2.3, 2.4, or 2.5, use the switch -Q*new* on the Python command line, or begin your source file with the statement:

```
from __future__ import division
```

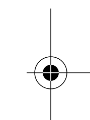
This statement ensures that operator / (within the module that starts with this statement only) works without truncation on operands of any type.

To ensure that the behavior of division does *not* depend on the -Q switch, and on the exact version of Python you're using, always use // when you want truncating division. When you do not want truncation, use /, but also ensure that at least one operand is *not* an integer. For example, instead of using just a/b, code 1.*a/b to avoid making any assumption on the types of a and b. To check whether your program has version dependencies in its use of division, use the switch -Q*warn* on the Python command line to get runtime warnings about all uses of / on integer operands.

The built-in divmod function takes two numeric arguments and returns a pair whose items are the quotient and remainder, so you don't have to use both // for the quotient and % for the remainder.

Exponentiation

The exponentiation ("raise to power") operation, *a****b*, raises an exception if *a* is less than zero and *b* is a floating-point value with a nonzero fractional part. The built-in pow(*a*, *b*) function returns the same result as *a****b*. With three arguments, pow(*a*, *b*, *c*) returns the same result as (*a****b*)%*c* but faster.



Comparisons

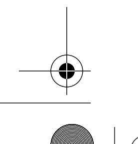
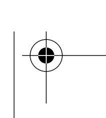
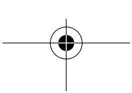
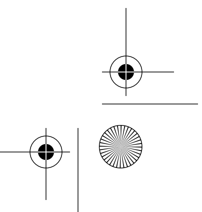
All objects, including numbers, can be compared for equality (==) and inequality (!=). Comparisons requiring order (<, <=, >, >=) may be used between any two numbers, unless either operand is complex, in which case they raise runtime exceptions. All these operators return Boolean values (True or False).

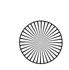
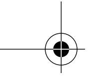
Bitwise Operations on Integers

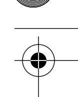
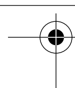
Integers and long integers can be taken as strings of bits and used with the bitwise operations shown in Table 4-2. Bitwise operators have lower priority than arithmetic operators. Positive integers are conceptually extended by an infinite string of 0 bits on the left. Negative integers are represented in two's complement notation, and therefore are conceptually extended by an infinite string of 1 bits on the left.

Sequence Operations

Python supports a variety of operations applicable to all sequences, including strings, lists, and tuples. Some sequence operations apply to all containers (including, for example, sets and dictionaries, which are not sequences), and some

apply to all iterables (meaning "any object on which you can loop," as covered in "Iterables" on page 40; all containers, be they sequences or otherwise, are iterable, and so are many objects that are not containers, such as files, covered in "File Objects" on page 216, and generators, covered in "Generators" on page 78). In the following, I use the terms *sequence*, *container*, and *iterable*, quite precisely and specifically, to indicate exactly which operations apply to each category.

Sequences in General

Sequences are containers with items that are accessible by indexing or slicing. The built-in `len` function takes any container as an argument and returns the number of items in the container. The built-in `min` and `max` functions take one argument, a nonempty iterable whose items are comparable, and return the smallest and largest items, respectively. You can also call `min` and `max` with multiple arguments, in which case they return the smallest and largest arguments, respectively. The built-in `sum` function takes one argument, an iterable whose items are numbers, and returns the sum of the numbers.

Sequence conversions

There is no implicit conversion between different sequence types, except that plain strings are converted to Unicode strings if needed. (String conversion is covered in detail in "Unicode" on page 198.) You can call the built-ins `tuple` and `list` with a single argument (any iterable) to get a new instance of the type you're calling, with the same items (in the same order) as in the argument.

Concatenation and repetition

You can concatenate sequences of the same type with the `+` operator. You can multiply a sequence *S* by an integer *n* with the * operator. *S***n* or *n***S* is the concatenation of *n* copies of *S*. When *n*<=0, *S***n* is an empty sequence of the same type as *S*.

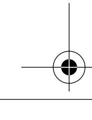
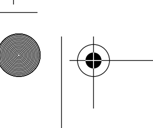
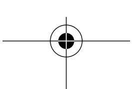
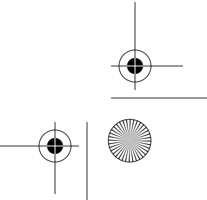
Membership testing

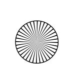
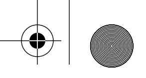
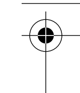
The *x* `in` *S* operator tests to check whether object *x* equals any item in the sequence (or other kind of container or iterable) *S*. It returns `True` if it does and `False` if it doesn't. The *x* `not in` *S* operator is just like `not (`*x* `in` *S*`)`. In the specific case of strings, though, *x* `in` *S* is more widely applicable; in this case, the operator tests whether *x* equals any *substring* of string *S*, not just any single *character*.

Indexing a sequence

The *n*th item of a sequence *S* is denoted by an *indexing*: *S*`[`*n*`]`. Indexing is zero-based (*S*'s first item is *S*`[0]`). If *S* has *L* items, the index *n* may be `0`, `1`…up to and including *L*`-1`, but no larger. *n* may also be `-1`, `-2`…down to and including `-`*L*, but no smaller. A negative *n* indicates the same item in *S* as *L*+*n* does. In other words, *S*`[-1]`, like *S*`[L-1]`, is the last element of *S*, *S*`[-2]` is the next-to-last one, and so on. For example:

```
x = [1, 2, 3, 4]
x[1]                  # 2
x[-1]                 # 4
```

Using an index >=*L* or <-*L* raises an exception. Assigning to an item with an invalid index also raises an exception. You can add one or more elements to a list, but to do so you assign to a slice, not an item, as I'll discuss shortly.

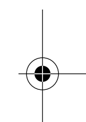
Slicing a sequence

To indicate a subsequence of *S*, you can use a *slicing*, with the syntax *S*[*i*:*j*], where *i* and *j* are integers. *S*[*i*:*j*] is the subsequence of *S* from the *i*th item, included, to the *j*th item, excluded. In Python, ranges always include the lower bound and exclude the upper bound. A slice is an empty subsequence if *j* is less than or equal to *i*, or if *i* is greater than or equal to *L*, the length of *S*. You can omit *i* if it is equal to 0, so that the slice begins from the start of *S*. You can omit *j* if it is greater than or equal to *L*, so that the slice extends all the way to the end of *S*. You can even omit both indices, to mean a copy of the entire sequence: *S*[:]. Either or both indices may be less than 0. A negative index indicates the same spot in *S* as *L*+*n*, just like in indexing. An index greater than or equal to *L* means the end of *S*, while a negative index less than or equal to -*L* means the start of *S*. Here are some examples:

```
x = [1, 2, 3, 4]
x[1:3]                 # [2, 3]
x[1:]                  # [2, 3, 4]
x[:2]                  # [1, 2]
```

Language

Slicing can also use the extended syntax *S*[*i*:*j*:*k*]. *k* is the *stride* of the slice, or the distance between successive indices. *S*[*i*:*j*] is equivalent to *S*[*i*:*j*:1], *S*[::2] is the subsequence of *S* that includes all items that have an even index in *S*, and *S*[::-1] has the same items as *S*, but in reverse order.



Strings

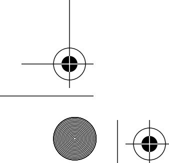
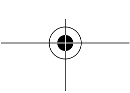
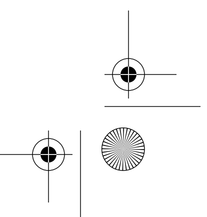
String objects are immutable, so attempting to rebind or delete an item or slice of a string raises an exception. The items of a string object (corresponding to each of the characters in the string) are themselves strings, each of length 1. The slices of a string object are also strings. String objects have several methods, which are covered in "Methods of String Objects" on page 186.

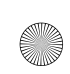
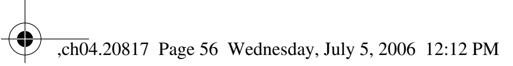
Tuples

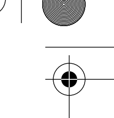
Tuple objects are immutable, so attempting to rebind or delete an item or slice of a tuple raises an exception. The items of a tuple are arbitrary objects and may be of different types. The slices of a tuple are also tuples. Tuples have no normal (nonspecial) methods, only some of the special methods covered in "Special Methods" on page 104.

Lists

List objects are mutable, so you may rebind or delete items and slices of a list. The items of a list are arbitrary objects and may be of different types. The slices of a list are lists.



Modifying a list

You can modify a list by assigning to an indexing. For instance:

```
x = [1, 2, 3, 4]
x[1] = 42               # x is now [1, 42, 3, 4]
```

Another way to modify a list object `L` is to use a slice of `L` as the target (LHS) of an assignment statement. The RHS of the assignment must then be an iterable. If the LHS slice is in extended form, then the RHS must have just as many items as the number of items in the LHS slice. However, if the LHS slice is in normal (nonextended) form, then the LHS slice and the RHS may each be of any length; assigning to a (nonextended) slice of a list can add or remove list items. For example:

```
x = [1, 2, 3, 4]
x[1:3] = [22, 33, 44]    # x is now [1, 22, 33, 44, 4]
x[1:4] = [8, 9]          # x is now [1, 8, 9, 4]
```

Here are some important special cases of assignment to slices:

- Using the empty list `[]` as the RHS expression removes the target slice from `L`. In other words, `L[i:j]=[]` has the same effect as `del L[i:j]`.
- Using an empty slice of `L` as the LHS target inserts the items of the RHS at the appropriate spot in `L`. In other words, `L[i:i]=['a','b']` inserts the items `'a'` and `'b'` before the item that was at index *i* in `L` before the assignment.
- Using a slice that covers the entire list object, `L[:]`, as the LHS target totally replaces the contents of `L`.

You can delete an item or a slice from a list with `del`. For instance:

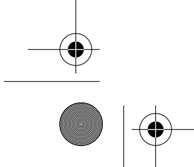
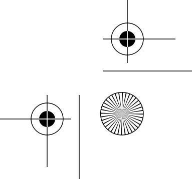
```
x = [1, 2, 3, 4, 5]
del x[1]                # x is now [1, 3, 4, 5]
del x[::2]              # x is now [3, 5]
```

In-place operations on a list

List objects define in-place versions of the `+` and `*` operators, which you can use via augmented assignment statements. The augmented assignment statement `L+=L1` has the effect of adding the items of iterable `L1` to the end of `L`. `L*=n` has the effect of adding `n-1` copies of `L` to the end of `L`; if `n<=0`, `L*=n` empties the contents of `L`, like `L[:]=[]`.

List methods

List objects provide several methods, as shown in Table 4-3. *Nonmutating methods* return a result without altering the object to which they apply, while *mutating methods* may alter the object to which they apply. Many of the mutating methods behave like assignments to appropriate slices of the list. In Table 4-3, `L` indicates any list object, `i` any valid index in `L`, `s` any iterable, and `x` any object.

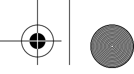
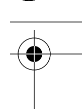
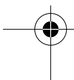




Table 4-3. List object methods

Method	Description
Nonmutating methods	
L.count(*x*)	Returns the number of items of *L* that are equal to *x*.
L.index(*x*)	Returns the index of the first occurrence of an item in *L* that is equal to *x*, or raises an exception if *L* has no such item.
Mutating methods	
L.append(*x*)	Appends item *x* to the end of *L* ; e.g., *L*[len(*L*):]=[*x*].
L.extend(*s*)	Appends all the items of iterable *s* to the end of *L*; e.g., *L*[len(*L*):]=*s*.
L.insert(*i*, *x*)	Inserts item *x* in *L* before the item at index *i*, moving following items of *L* (if any) "rightward" to make space (increases len(*L*) by one, does not replace any item, does not raise exceptions: acts just like *L*[*i*:*i*]=[*x*]).
L.remove(*x*)	Removes from *L* the first occurrence of an item in *L* that is equal to *x*, or raises an exception if *L* has no such item.
L.pop([*i*])	Returns the value of the item at index *i* and removes it from *L*; if *i* is omitted, removes and returns the last item; raises an exception if *L* is empty or *i* is an invalid index in *L*.
L.reverse( )	Reverses, in place, the items of *L*.
L.sort([*f*]) (2.3)	Sorts, in place, the items of *L*, comparing items pairwise via function *f*; if *f* is omitted, comparison is via the built-in function cmp. For more details, see "Sorting a list" on page 57.
L.sort(cmp=cmp, key=None, reverse=False)(2.4)	Sorts, in-place, the items of *L*, comparing items pairwise via the function passed as *cmp* (by default, the built-in function cmp). When argument *key* is not None, what gets compared for each item *x* is *key*(*x*), not *x* itself. For more details, see "Sorting a list" on page 57.

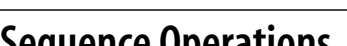
All mutating methods of list objects, except pop, return None.

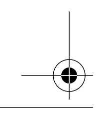
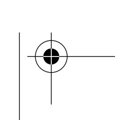
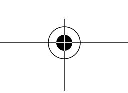
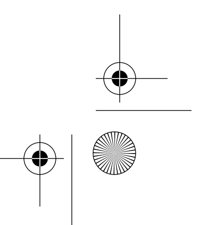
Sorting a list

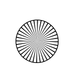
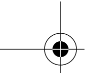
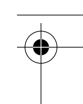
A list's method sort causes the list to be sorted in-place (its items are reordered to place them in increasing order) in a way that is guaranteed to be stable (elements that compare equal are not exchanged). In practice, sort is extremely fast, often *preternaturally* fast, as it can exploit any order or reverse order that may be present in any sublist (the advanced algorithm sort uses, known as *timsort* to honor its developer, Tim Peters, is technically a "non-recursive adaptive stable natural mergesort/binary insertion sort hybrid").

In Python 2.3, a list's sort method takes one optional argument. If present, the argument must be a function that, when called with any two list items as arguments, returns -1, 0, or 1, depending on whether the first item is to be considered less than, equal to, or greater than the second item for sorting purposes. Passing the argument slows down the sort, although it makes it easy to sort small lists in flexible ways. The decorate-sort-undecorate idiom, covered in "Searching and sorting" on page 485 is faster, at least as flexible, and often less error-prone than passing an argument to sort.

In Python 2.4, the sort method takes three optional arguments, which may be passed with either positional or named-argument syntax. Argument *cmp* plays just





,ch04.20817 Page 58 Wednesday, July 5, 2006 12:12 PM


the same role as the only (unnamed) optional argument in Python 2.3. Argument *key*, if not `None`, must be a function that can be called with any list item as its only argument. In this case, to compare any two items *x* and *y*, Python uses `cmp(key(x),key(y))` rather than `cmp(x,y)` (in practice, this is implemented in the same way as the decorate-sort-undecorate idiom presented in "Searching and sorting" on page 485, and can be even faster). Argument *reverse*, if `True`, causes the result of each comparison to be reversed; this is not the same thing as reversing *L* after sorting because the sort is stable (elements that compare equal are never exchanged) whether argument *reverse* is true or false.

Python 2.4 also provides a new built-in function `sorted` (covered in `sorted` on page 167) to produce a sorted list from any input iterable. `sorted` accepts the same input arguments as a list's method `sort`. Moreover, Python 2.4 adds to module `operator` (covered in "The operator Module" on page 368) two new higher-order functions, `attrgetter` and `itemgetter`, which produce functions particularly suitable for the *key*= optional argument of lists' method `sort` and the new built-in function `sorted`. In Python 2.5, an identical *key*= optional argument has also been added to built-in functions `min` and `max`, and to functions `nsmallest` and `nlargest` in standard library module `heapq`, covered in "The heapq Module" on page 177.

Set Operations

Python provides a variety of operations applicable to sets. Since sets are containers, the built-in `len` function can take a set as its single argument and return the number of items in the set object. A set is iterable, so you can pass it to any function or method that takes an iterable argument. In this case, the items of the set are iterated upon, in some arbitrary order. For example, for any set *S*, `min(`*S*`)` returns the smallest item in *S*.

Set Membership

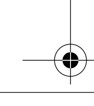
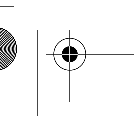
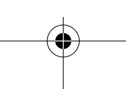
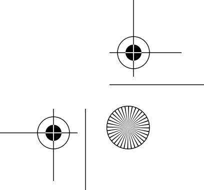
The *k* `in` *S* operator checks whether object *k* is one of the items of set *S*. It returns `True` if it is and `False` if it isn't. Similarly, *k* `not in` *S* is just like `not (`*k* `in` *S*`)`.

Set Methods

Set objects provide several methods, as shown in Table 4-4. Nonmutating methods return a result without altering the object to which they apply and can also be called on instances of type `frozenset`, while mutating methods may alter the object to which they apply and can be called only on instances of type `set`. In Table 4-4, *S* and *S1* indicate any set object, and *x* any hashable object.

Table 4-4. Set object methods

Method	Description
Nonmutating methods	
S.copy( )	Returns a shallow copy of the set (a copy whose items are the same objects as *S*'s, not copies thereof)
S.difference(*S1*)	Returns the set of all items of *S* that aren't in *S1*

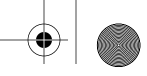
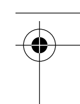
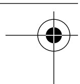




Table 4-4. Set object methods (continued)

Method	Description
`S.intersection(S1)`	Returns the set of all items of *S* that are also in *S1*
`S.issubset(S1)`	Returns `True` if all items of *S* are also in *S1*; otherwise, returns `False`
`S.issuperset(S1)`	Returns `True` if all items of *S1* are also in *S*; otherwise, returns `False` (like `S1.issubset(S)`)
`S.symmetric_difference(S1)`	Returns the set of all items that are in either *S* or *S1*, but not in both sets
`S.union(S1)`	Returns the set of all items that are in *S*, *S1*, or in both sets
Mutating methods	
`S.add(x)`	Adds *x* as an item to *S*; no effect if *x* was already an item in *S*
`S.clear( )`	Removes all items from *S*, leaving *S* empty
`S.discard(x)`	Removes *x* as an item of *S*; no effect if *x* was not an item of *S*
`S.pop( )`	Removes and returns an arbitrary item of *S*
`S.remove(x)`	Removes *x* as an item of *S*; raises a `KeyError` exception if *x* is not an item of *S*

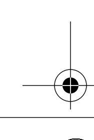
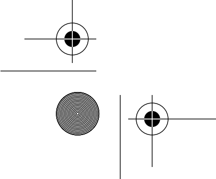
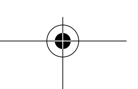
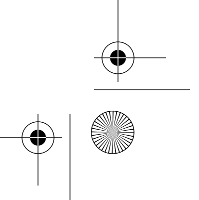
All mutating methods of set objects, except `pop`, return `None`.

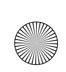
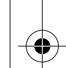
The `pop` method can be used for destructive iteration on a set, consuming little extra memory. The memory savings make `pop` usable for a loop on a huge set, when what you want is to “consume” the set in the course of the loop.

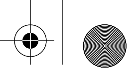
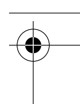
Sets also have mutating methods named `difference_update`, `intersection_update`, `symmetric_difference_update`, and `update` (corresponding to nonmutating method `union`). Each such mutating method performs the same operation as the corresponding nonmutating method, but it performs the operation in place, altering the set on which you call it, and returns `None`. These four nonmutating methods are also accessible with operator syntax: respectively, `S-S1`, `S&S1`, `S^S1`, and `S|S1`; the corresponding mutating methods are also accessible with augmented assignment syntax: respectively, `S-=S1`, `S&=S1`, `S^=S1`, and `S|=S1`. When you use operator or augmented assignment syntax, both operands must be sets or frozensets; however, when you call the named methods, argument *S1* can be any iterable with hashable items, and the semantics are just the same as if the argument you passed was `set(S1)`.

Dictionary Operations

Python provides a variety of operations applicable to dictionaries. Since dictionaries are containers, the built-in `len` function can take a dictionary as its single argument and return the number of items (key/value pairs) in the dictionary object. A dictionary is iterable, so you can pass it to any function or method that takes an iterable argument. In this case, only the keys of the dictionary are iterated upon, in some arbitrary order. For example, for any dictionary *D*, `min(D)` returns the smallest key in *D*.

Dictionary Membership

The *k* in *D* operator checks whether object *k* is one of the keys of the dictionary *D*. It returns True if it is and False if it isn't. *k* not in *D* is just like not (*k* in *D*).

Indexing a Dictionary

The value in a dictionary *D* that is currently associated with key *k* is denoted by an indexing: *D*[*k*]. Indexing with a key that is not present in the dictionary raises an exception. For example:

```
d = { 'x':42, 'y':3.14, 'z':7 }
d['x']                          # 42
d['z']                          # 7
d['a']                          # raises KeyError exception
```

Plain assignment to a dictionary indexed with a key that is not yet in the dictionary (e.g., *D*[*newkey*]=*value*) is a valid operation and adds the key and value as a new item in the dictionary. For instance:

```
d = { 'x':42, 'y':3.14}
d['a'] = 16                     # d is now {'x':42, 'y':3.14,'a':16}
```

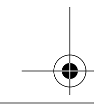
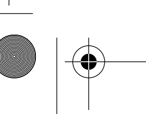
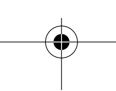
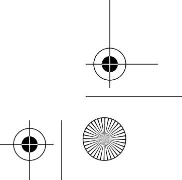
The del statement, in the form del *D*[*k*], removes from the dictionary the item whose key is *k*. If *k* is not a key in dictionary *D*, del *D*[*k*] raises an exception.

Dictionary Methods

Dictionary objects provide several methods, as shown in Table 4-5. Nonmutating methods return a result without altering the object to which they apply, while mutating methods may alter the object to which they apply. In Table 4-5, *D* and *D1* indicate any dictionary object, *k* any hashable object, and *x* any object.

Table 4-5. Dictionary object methods

Method	Description
Nonmutating methods	
D.copy( )	Returns a shallow copy of the dictionary (a copy whose items are the same objects as *D*'s, not copies thereof)
D.has_key(*k*)	Returns True if *k* is a key in *D*; otherwise, returns False, just like *k* in *D*
D.items( )	Returns a new list with all items (key/value pairs) in *D*
D.keys( )	Returns a new list with all keys in *D*
D.values( )	Returns a new list with all values in *D*
D.iteritems( )	Returns an iterator on all items (key/value pairs) in *D*
D.iterkeys( )	Returns an iterator on all keys in *D*
D.itervalues( )	Returns an iterator on all values in *D*
D.get(*k*[, *x*])	Returns *D*[*k*] if *k* is a key in *D*; otherwise, returns *x* (or None, if *x* is not given)
Mutating methods	
D.clear( )	Removes all items from *D*, leaving *D* empty
D.update(*D1*)	For each *k* in *D1*, sets *D*[*k*] equal to *D1*[*k*]
D.setdefault(*k*[, *x*])	Returns *D*[*k*] if *k* is a key in *D*; otherwise, sets *D*[*k*] equal to *x* and returns *x*

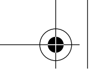
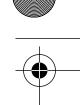
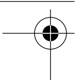
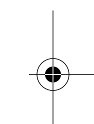




Table 4-5. Dictionary object methods (continued)

Method	Description
D.pop(*k*[, *x*])	Removes and returns *D*[*k*] if *k* is a key in *D*; otherwise, returns *x* (or raises an exception if *x* is not given)
D.popitem( )	Removes and returns an arbitrary item (key/value pair)

The items, keys, and values methods return their resulting lists in arbitrary order. If you call more than one of these methods without any intervening change to the dictionary, however, the order of the results is the same for all. The iteritems, iterkeys, and itervalues methods return iterators equivalent to these lists (iterators are discussed in "Iterators" on page 65). An iterator consumes less memory than a list, but you must never modify the set of keys in a dictionary (i.e., you must never add nor remove keys) while iterating on any of that dictionary's iterators. Iterating on the lists returned by items, keys, or values carries no such constraint. Iterating directly on a dictionary *D* is exactly like iterating on *D*.iterkeys( ).

The popitem method can be used for destructive iteration on a dictionary. Both items and popitem return dictionary items as key/value pairs, but using popitem consumes less memory, as it does not rely on a separate list of items. The memory savings make the idiom usable for a loop on a huge dictionary, when what you want is to "consume" the dictionary in the course of the loop.



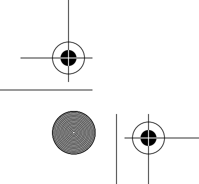
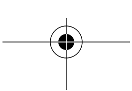
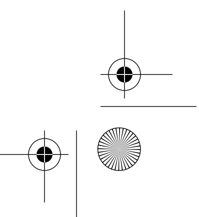
The setdefault method returns the same result as get, but if *k* is not a key in *D*, setdefault also has the side effect of binding *D*[*k*] to the value *x*. Similarly, the pop method returns the same result as get, but if *k* is a key in *D*, pop also has the side effect of removing *D*[*k*] (when *x* is not specified, and *k* is not a key in *D*, get returns None, but pop raises an exception).

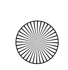
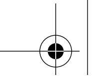
In Python 2.4, the update method can also accept an iterable of key/value pairs as an alternative argument instead of a mapping, and can accept keyword arguments instead of or in addition to its only positional argument; the semantics are the same as for passing such arguments when calling the built-in dict type, as covered in "Dictionaries" on page 44.

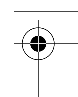
The print Statement

A print statement is denoted by the keyword print followed by zero or more expressions separated by commas. print is a handy, simple way to output values in text form, mostly for debugging purposes. print outputs each expression *x* as a string that's just like the result of calling str(*x*) (covered in str on page 157). print implicitly outputs a space between expressions, and implicitly outputs \n after the last expression, unless the last expression is followed by a trailing comma (,). Here are some examples of print statements:

```
letter = 'c'
print "give me a", letter, "..."          # prints: give me a c ...
answer = 42
print "the answer is:", answer            # prints: the answer is: 42
```



The destination of print's output is the file or file-like object that is the value of the stdout attribute of the sys module (covered in "The sys Module" on page 168). If you want to direct the output from a certain print statement to a specific file object *f* (which must be open for writing), you can use the special syntax:

```
print >>f, rest of print statement
```

(if *f* is None, the destination is sys.stdout, just as it would be without the >>*f*). You can also use the write or writelines methods of file objects, as covered in "Attributes and Methods of File Objects" on page 218. However, print is very simple to use, and simplicity is important in the common case where all you need are the simple output strategies that print supplies—in particular, this is often the case for the kind of simple output statements you may temporarily add to a program for debugging purposes. "The print Statement" on page 256 has more advice and examples concerning the use of print.

Control Flow Statements

A program's *control flow* is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls. (This section covers the if statement and for and while loops; functions are covered in "Functions" on page 70.) Raising and handling exceptions also affects control flow; exceptions are covered in Chapter 6.

The if Statement

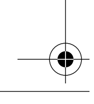
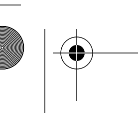
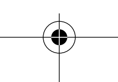
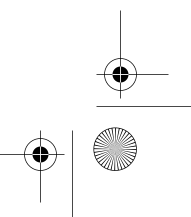
Often, you need to execute some statements only if some condition holds, or choose statements to execute depending on several mutually exclusive conditions. The Python compound statement if, comprising if, elif, and else clauses, lets you conditionally execute blocks of statements. Here's the syntax for the if statement:

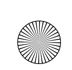
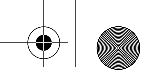
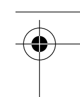
```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

The elif and else clauses are optional. Note that, unlike some languages, Python does not have a switch statement. Use if, elif, and else for all conditional processing.

Here's a typical if statement with all three kinds of clauses:

```
if x < 0: print "x is negative"
elif x % 2: print "x is positive and odd"
else: print "x is even and non-negative"
```

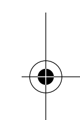
When there are multiple statements in a clause (i.e., the clause controls a block of statements), the statements are placed on separate logical lines after the line containing the clause's keyword (known as the *header line* of the clause), indented rightward from the header line. The block terminates when the indentation returns to that of the clause header (or further left from there). When there is just a single simple statement, as here, it can follow the `:` on the same logical line as the header, but it can also be on a separate logical line, immediately after the header line and indented rightward from it. Most Python programmers prefer the separate-line style, with four-space indents for the guarded statements. Such a style is considered more general and more readable.

```
if x < 0:
    print "x is negative"
elif x % 2:
    print "x is positive and odd"
else:
    print "x is even and non-negative"
```

You can use any Python expression as the condition in an `if` or `elif` clause. Using an expression this way is known as using it "in a Boolean context." In a Boolean context, any value is taken as either true or false. As mentioned earlier, any nonzero number or nonempty container (string, tuple, list, dictionary, set) evaluates as true; zero (of any numeric type), `None`, and empty containers evaluate as false. When you want to test a value *x* in a Boolean context, use the following coding style:

```
if x:
```

This is the clearest and most Pythonic form. Do *not* use any of the following:



```
if x is True:
if x == True:
if bool(x):
```

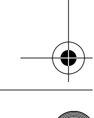
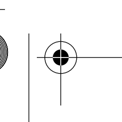
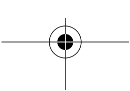
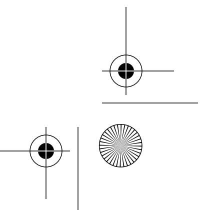
There is a crucial difference between saying that an expression *returns* `True` (meaning the expression returns the value `1` with the `bool` type) and saying that an expression *evaluates as* true (meaning the expression returns any result that is true in a Boolean context). When testing an expression, you care about the latter condition, not the former.

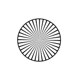
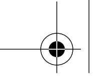
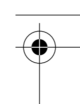
If the expression for the `if` clause evaluates as true, the statements following the `if` clause execute, and the entire `if` statement ends. Otherwise, Python evaluates the expressions for each `elif` clause, in order. The statements following the first `elif` clause whose condition evaluates as true, if any, execute, and the entire `if` statement ends. Otherwise, if an `else` clause exists, the statements following it execute.

The while Statement

The `while` statement in Python supports repeated execution of a statement or block of statements that are controlled by a conditional expression. Here's the syntax for the `while` statement:

```
while expression:
    statement(s)
```

A `while` statement can also include an `else` clause, covered in "The else Clause on Loop Statements" on page 69, and `break` and `continue` statements, covered in "The break Statement" on page 68 and "The continue Statement" on page 68.

Here's a typical `while` statement:

```
count = 0
while x > 0:
    x = x // 2           # truncating division
    count += 1
print "The approximate log2 is", count
```

First, Python evaluates *`expression`*, which is known as the *loop condition*. If the condition is false, the `while` statement ends. If the loop condition is satisfied, the statement or statements that make up the *loop body* execute. When the loop body finishes executing, Python evaluates the loop condition again to check whether another iteration should execute. This process continues until the loop condition is false, at which point the `while` statement ends.

The loop body should contain code that eventually makes the loop condition false; otherwise, the loop will never end (unless an exception is raised or the loop body executes a `break` statement). A loop that is in a function's body also ends if a `return` statement executes in the loop body, since the whole function ends in this case.

The for Statement

The `for` statement in Python supports repeated execution of a statement, or block of statements, controlled by an iterable expression. Here's the syntax for the `for` statement:

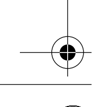
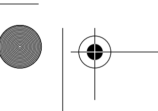
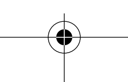
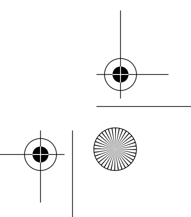
```
for target in iterable:
    statement(s)
```

The `in` keyword is part of the syntax of the `for` statement and is distinct from the `in` operator, which tests membership. A `for` statement can also include an `else` clause, covered in "The else Clause on Loop Statements" on page 69, and `break` and `continue` statements, covered in "The break Statement" on page 68 and "The continue Statement" on page 68.

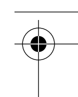
Here's a typical `for` statement:

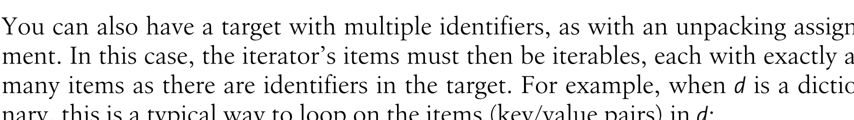
```
for letter in "ciao":
    print "give me a", letter, "..."
```

`iterable` may be any Python expression suitable as an argument to built-in function `iter`, which returns an iterator object (explained in detail in the next section). In particular, any sequence is iterable. *`target`* is normally an identifier that names the *control variable* of the loop; the `for` statement successively rebinds this variable to each item of the iterator, in order. The statement or statements that make up the *loop body* execute once for each item in *`iterable`* (unless the loop ends because an exception is raised or a `break` or `return` statement executes). Note that, since the loop body may contain a `break` statement to terminate the loop, this is one case in which you may want to use an *unbounded* iterable—one that, per se, would never cease yielding items.

,ch04.20817 Page 65 Wednesday, July 5, 2006 12:12 PM





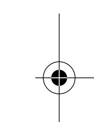
You can also have a target with multiple identifiers, as with an unpacking assignment. In this case, the iterator's items must then be iterables, each with exactly as many items as there are identifiers in the target. For example, when *d* is a dictionary, this is a typical way to loop on the items (key/value pairs) in *d*:

```
for key, value in d.items():
    if not key or not value:      # keep only true keys and values
        del d[key]
```

The `items` method returns a list of key/value pairs, so we can use a `for` loop with two identifiers in the target to unpack each item into `key` and `value`.

When an iterator has a mutable underlying object, you must not alter that object during a `for` loop on it. For example, the previous example cannot use `iteritems` instead of `items`. `iteritems` returns an iterator whose underlying object is `d`, so the loop body cannot mutate `d` (by executing `del d[key]`). `items` returns a list so that `d` is not the underlying object of the iterator; therefore, the loop body can mutate `d`. Specifically:

- When looping on a list, do not insert, append, or delete items (rebinding an item at an existing index is OK).
- When looping on a dictionary, do not add or delete items (rebinding the value for an existing key is OK).
- When looping on a set, do not add or delete items (no alteration is permitted).



The control variable may be rebound in the loop body but is rebound again to the next item in the iterator at the next iteration of the loop. The loop body does not execute at all if the iterator yields no items. In this case, the control variable is not bound or rebound in any way by the `for` statement. If the iterator yields at least one item, however, when the loop statement terminates, the control variable remains bound to the last value to which the loop statement has bound it. The following code is therefore correct, as long as `someseq` is not empty:

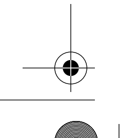
```
for x in someseq:
    process(x)
print "Last item processed was", x
```

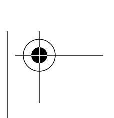
Iterators

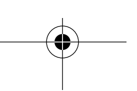
An *iterator* is an object *i* such that you can call *i*`.next()` with no arguments. *i*`.next()` returns the next item of iterator *i* or, when iterator *i* has no more items, raises a `StopIteration` exception. When you write a class (see "Classes and Instances" on page 82), you can allow instances of the class to be iterators by defining such a method `next`. Most iterators are built by implicit or explicit calls to built-in function `iter`, covered in `iter` on page 163. Calling a generator also returns an iterator, as we'll discuss in "Generators" on page 78.

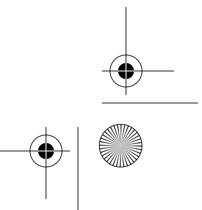
The `for` statement implicitly calls `iter` to get an iterator. The following statement:

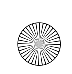
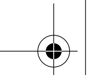
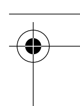
```
for x in c:
    statement(s)
```







is exactly equivalent to:

```
_temporary_iterator = iter(c)
while True:
    try: x = _temporary_iterator.next()
    except StopIteration: break
    statement(s)
```

where *_temporary_iterator* is some arbitrary name that is not used elsewhere in the current scope.

Thus, if `iter(c)` returns an iterator *i* such that `i.next()` never raises `StopIteration` (an *unbounded iterator*), the loop `for x in c` never terminates (unless the statements in the loop body include suitable `break` or `return` statements, or raise or propagate exceptions). `iter(c)`, in turn, calls special method `c.__iter__()` to obtain and return an iterator on *c*. I'll talk more about the special method `__iter__` in `__iter__` on page 112.

Many of the best ways to build and manipulate iterators are found in standard library module `itertools`, covered in "The itertools Module" on page 183.

range and xrange

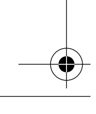
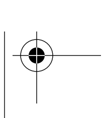
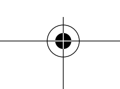
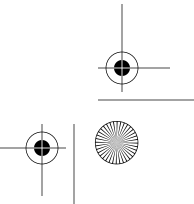
Looping over a sequence of integers is a common task, so Python provides built-in functions `range` and `xrange` to generate and return integer sequences. The simplest way to loop *n* times in Python is:

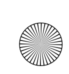
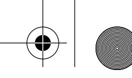
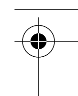
```
for i in xrange(n):
    statement(s)
```

`range(x)` returns a list whose items are consecutive integers from `0` (included) up to *x* (excluded). `range(x, y)` returns a list whose items are consecutive integers from *x* (included) up to *y* (excluded). The result is the empty list if *x* is greater than or equal to *y*. `range(x, y, step)` returns a list of integers from *x* (included) up to *y* (excluded), such that the difference between each two adjacent items in the list is *step*. If *step* is less than `0`, `range` counts down from *x* to *y*. `range` returns the empty list when *x* is greater than or equal to *y* and *step* is greater than `0`, or when *x* is less than or equal to *y* and *step* is less than `0`. When *step* equals `0`, `range` raises an exception.

While `range` returns a normal list object, usable for all purposes, `xrange` returns a special-purpose object, specifically intended for use in iterations like the `for` statement shown previously (unfortunately, to keep backward compatibility with old versions of Python, `xrange` does not return an iterator, as would be natural in today's Python; however, you can easily obtain such an iterator, if you need one, by calling `iter(xrange(...))`). The special-purpose object `xrange` returns consumes less memory (for wide ranges, *much* less memory) than the list object `range` returns, but the overhead of looping on the special-purpose object is slightly higher than that of looping on a list. Apart from performance and memory consumption issues, you can use `range` wherever you could use `xrange`, but not vice versa. For example:

```
>>> print range(1, 5)
[1, 2, 3, 4]
>>> print xrange(1, 5)
xrange(1, 5)
```

Here, `range` returns a perfectly ordinary list, which displays quite normally, but `xrange` returns a special-purpose object, which displays in its own special way.

List comprehensions

A common use of a `for` loop is to inspect each item in an iterable and build a new list by appending the results of an expression computed on some or all of the items. The expression form known as a *list comprehension* lets you code this common idiom concisely and directly. Since a list comprehension is an expression (rather than a block of statements), you can use it wherever you need an expression (e.g., as an argument in a function call, in a `return` statement, or as a subexpression for some other expression).

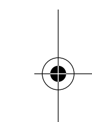
A list comprehension has the following syntax:

```
[ expression for target in iterable lc-clauses ]
```

`target` and *`iterable`* are the same as in a regular `for` statement. You must enclose the *`expression`* in parentheses if it indicates a tuple.

`lc-clauses` is a series of zero or more clauses, each with one of the following forms:

```
for target in iterable
if expression
```



`target` and *`iterable`* in each `for` clause of a list comprehension have the same syntax and meaning as those in a regular `for` statement, and the *`expression`* in each `if` clause of a list comprehension has the same syntax and meaning as the *`expression`* in a regular `if` statement.

A list comprehension is equivalent to a `for` loop that builds the same list by repeated calls to the resulting list's `append` method. For example (assigning the list comprehension result to a variable for clarity):

```
result1 = [x+1 for x in some_sequence]
```

is the same as the `for` loop:

```
result2 = []
for x in some_sequence:
    result2.append(x+1)
```

Here's a list comprehension that uses an `if` clause:

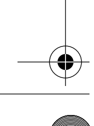
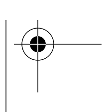
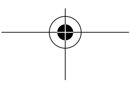
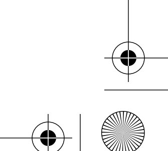
```
result3 = [x+1 for x in some_sequence if x>23]
```

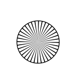
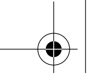
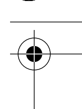
This list comprehension is the same as a `for` loop that contains an `if` statement:

```
result4 = []
for x in some_sequence:
    if x>23:
        result4.append(x+1)
```

And here's a list comprehension that uses a `for` clause:

```
result5 = [x+y for x in alist for y in another]
```



,ch04.20817 Page 68 Wednesday, July 5, 2006 12:12 PM


This is the same as a for loop with another for loop nested inside:

```
result6 = []
for x in alist:
    for y in another:
        result6.append(x+y)
```

As these examples show, the order of for and if in a list comprehension is the same as in the equivalent loop, but in the list comprehension, the nesting remains implicit.

The break Statement

The break statement is allowed only inside a loop body. When break executes, the loop terminates. If a loop is nested inside other loops, a break in it terminates only the innermost nested loop. In practical use, a break statement is usually inside some clause of an if statement in the loop body so that break executes conditionally.

One common use of break is in the implementation of a loop that decides whether it should keep looping only in the middle of each loop iteration:

```
while True:                    # this loop can never terminate naturally
    x = get_next()
    y = preprocess(x)
    if not keep_looping(x, y): break
    process(x, y)
```

The continue Statement

The continue statement is allowed only inside a loop body. When continue executes, the current iteration of the loop body terminates, and execution continues with the next iteration of the loop. In practical use, a continue statement is usually inside some clause of an if statement in the loop body so that continue executes conditionally.

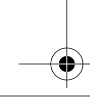
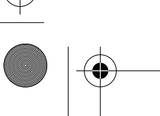
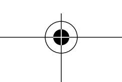
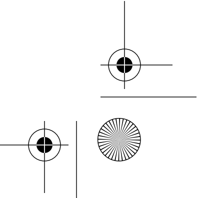
Sometimes, a continue statement can take the place of nested if statements within a loop. For example:

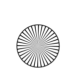
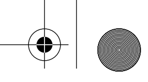
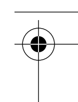
```
for x in some_container:
    if not seems_ok(x): continue
    lowbound, highbound = bounds_to_test()
    if x<lowbound or x>=highbound: continue
    if final_check(x):
        do_processing(x)
```

This equivalent code does conditional processing without continue:

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test()
        if lowbound <= x < highbound:
            if final_check(x):
                do_processing(x)
```

Both versions function identically, so which one you use is a matter of personal preference and style.

The else Clause on Loop Statements

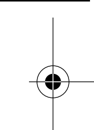
while and for statements may optionally have a trailing else clause. The statement or block under that else executes when the loop terminates *naturally* (at the end of the for iterator, or when the while loop condition becomes false), but not when the loop terminates *prematurely* (via break, return, or an exception). When a loop contains one or more break statements, you often need to check whether the loop terminates naturally or prematurely. You can use an else clause on the loop for this purpose:

```
for x in some_container:
    if is_ok(x): break            # item x is satisfactory, terminate loop
else:
    print "Warning: no satisfactory item was found in container"
    x = None
```

The pass Statement

The body of a Python compound statement cannot be empty; it must always contain at least one statement. You can use a pass statement, which performs no action, as a placeholder when a statement is syntactically required but you have nothing to do. Here's an example of using pass in a conditional statement as a part of somewhat convoluted logic to test mutually exclusive conditions:

```
if condition1(x):
    process1(x)
elif x>23 or condition2(x) and x<5:
    pass                              # nothing to be done in this case
elif condition3(x):
    process3(x)
else:
    process_default(x)
```



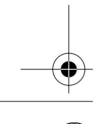
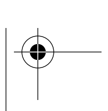
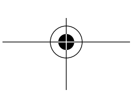
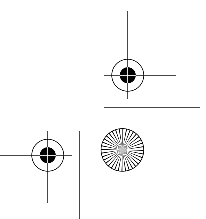
Note that, as the body of an otherwise empty def or class statement, you may use a docstring, covered in "Docstrings" on page 72; if you do write a docstring, then you do not need to also add a pass statement, although you are still allowed to do so if you wish.

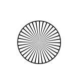
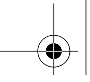
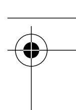
The try and raise Statements

Python supports exception handling with the try statement, which includes try, except, finally, and else clauses. A program can explicitly raise an exception with the raise statement. As I discuss in detail in "Exception Propagation" on page 126, when an exception is raised, normal control flow of the program stops, and Python looks for a suitable exception handler.

The with Statement

In Python 2.5, a with statement has been added as a more readable alternative to the try/finally statement. I discuss it in detail in "The with statement" on page 125.

Functions

Most statements in a typical Python program are grouped and organized into functions (code in a function body may be faster than at a module's top level, as covered in "Avoiding exec and from…import *" on page 486, so there are excellent practical reasons to put most of your code into functions). A *function* is a group of statements that execute upon request. Python provides many built-in functions and allows programmers to define their own functions. A request to execute a function is known as a *function call*. When you call a function, you can pass arguments that specify data upon which the function performs its computation. In Python, a function always returns a result value, either `None` or a value that represents the results of the computation. Functions defined within `class` statements are also known as *methods*. Issues specific to methods are covered in "Bound and Unbound Methods" on page 91; the general coverage of functions in this section, however, also applies to methods.

In Python, functions are objects (values) that are handled like other objects. Thus, you can pass a function as an argument in a call to another function. Similarly, a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. Functions can also be keys into a dictionary. For example, if you need to quickly find a function's inverse given the function, you could define a dictionary whose keys and values are functions and then make the dictionary bidirectional. Here's a small example of this idea, using some functions from module `math`, covered in "The math and cmath Modules" on page 365:

```
inverse = {sin:asin, cos:acos, tan:atan, log:exp}
for f in inverse.keys(): inverse[inverse[f]] = f
```

The fact that functions are ordinary objects in Python is often expressed by saying that functions are *first-class* objects.

The def Statement

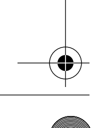
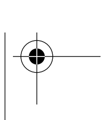
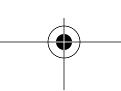
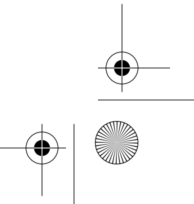
The `def` statement is the most common way to define a function. `def` is a single-clause compound statement with the following syntax:

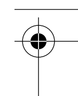
```
def function-name(parameters):
    statement(s)
```

`function-name` is an identifier. It is a variable that gets bound (or rebound) to the function object when `def` executes.

`parameters` is an optional list of identifiers, known as *formal parameters* or just parameters, that get bound to the values supplied as arguments when the function is called. In the simplest case, a function doesn't have any formal parameters, which means the function doesn't take any arguments when it is called. In this case, the function definition has empty parentheses after *`function-name`*.

When a function does take arguments, *`parameters`* contains one or more identifiers, separated by commas (`,`). In this case, each call to the function supplies values, known as *arguments*, corresponding to the parameters listed in the function definition. The parameters are local variables of the function (as we'll discuss



later in this section), and each call to the function binds these local variables to the corresponding values that the caller supplies as arguments.

The nonempty sequence of statements, known as the *function body*, does not execute when the `def` statement executes. Rather, the function body executes later, each time the function is called. The function body can contain zero or more occurrences of the `return` statement, as we'll discuss shortly.

Here's an example of a simple function that returns a value that is twice the value passed to it each time it's called:

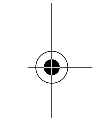
```
def double(x):
    return x*2
```

Parameters

Formal parameters that are just identifiers indicate *mandatory parameters*. Each call to the function must supply a corresponding value (argument) for each mandatory parameter.

In the comma-separated list of parameters, zero or more mandatory parameters may be followed by zero or more *optional parameters*, where each optional parameter has the syntax:

```
identifier=expression
```

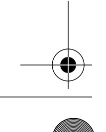


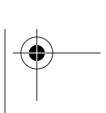
The `def` statement evaluates each such *expression* and saves a reference to the expression's value, known as the *default value* for the parameter, among the attributes of the function object. When a function call does not supply an argument corresponding to an optional parameter, the call binds the parameter's identifier to its default value for that execution of the function. Note that each default value gets computed when the `def` statement evaluates, *not* when the resulting function gets called. In particular, this means that the *same* object, the default value, gets bound to the optional parameter whenever the caller does not supply a corresponding argument. This can be tricky when the default value is a mutable object and the function body alters the parameter. For example:

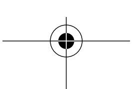
```
def f(x, y=[]):
    y.append(x)
    return y
print f(23)                # prints: [23]
prinf f(42)                # prints: [23, 42]
```

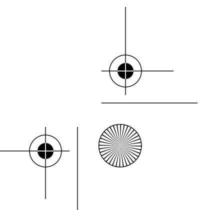
The second `print` statement prints `[23, 42]` because the first call to `f` altered the default value of `y`, originally an empty list `[]`, by appending `23` to it. If you want `y` to be bound to a new empty list object each time `f` is called with a single argument, use the following style instead:

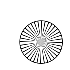
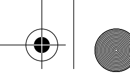
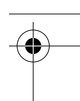
```
def f(x, y=None):
    if y is None: y = []
    y.append(x)
    return y
print f(23)                # prints: [23]
prinf f(42)                # prints: [42]
```







At the end of the parameters, you may optionally use either or both of the special forms *`identifier1` and **`identifier2`. If both forms are present, the form with two asterisks must be last. *`identifier1` specifies that any call to the function may supply any number of extra positional arguments, while **`identifier2` specifies that any call to the function may supply any number of extra named arguments (positional and named arguments are covered in "Calling Functions" on page 73). Every call to the function binds `identifier1` to a tuple whose items are the extra positional arguments (or the empty tuple, if there are none). Similarly, `identifier2` gets bound to a dictionary whose items are the names and values of the extra named arguments (or the empty dictionary, if there are none). Here's a function that accepts any number of positional arguments and returns their sum:

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(23, 42)          # prints: 65
```

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether (at the end of the parameters) either or both of the single- and double-asterisk special forms are present, collectively form a specification known as the function's *signature*. A function's signature defines the ways in which you can call the function.

Attributes of Function Objects

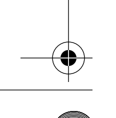
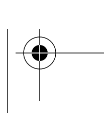
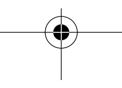
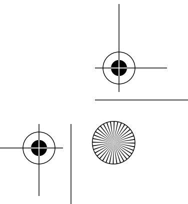
The `def` statement sets some attributes of a function object. The attribute `func_name`, also accessible as `__name__`, refers to the identifier string given as the function name in the `def` statement. In Python 2.3, this is a read-only attribute (trying to rebind or unbind it raises a runtime exception); in Python 2.4, you may rebind the attribute to any string value, but trying to unbind it raises an exception. The attribute `func_defaults`, which you may freely rebind or unbind, refers to the tuple of default values for the optional parameters (or the empty tuple, if the function has no optional parameters).

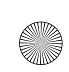
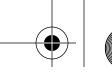
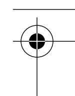
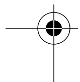
Docstrings

Another function attribute is the *documentation string*, also known as the *docstring*. You may use or rebind a function's docstring attribute as either `func_doc` or `__doc__`. If the first statement in the function body is a string literal, the compiler binds that string as the function's docstring attribute. A similar rule applies to classes (see "Class documentation strings" on page 85) and modules (see "Module documentation strings" on page 142). Docstrings most often span multiple physical lines, so you normally specify them in triple-quoted string literal form. For example:

```
def sum_args(*numbers):
    '''Accept arbitrary numerical arguments and return their sum.
    The arguments are zero or more numbers.  The result is their sum.'''
    return sum(numbers)
```

Documentation strings should be part of any Python code you write. They play a role similar to that of comments in any programming language, but their applicability is wider, since they remain available at runtime. Development environments

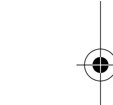





and tools can use docstrings from function, class, and module objects to remind the programmer how to use those objects. The `doctest` module (covered in "The doctest Module" on page 454) makes it easy to check that sample code present in docstrings is accurate and correct.

To make your docstrings as useful as possible, you should respect a few simple conventions. The first line of a docstring should be a concise summary of the function's purpose, starting with an uppercase letter and ending with a period. It should not mention the function's name, unless the name happens to be a natural-language word that comes naturally as part of a good, concise summary of the function's operation. If the docstring is multiline, the second line should be empty, and the following lines should form one or more paragraphs, separated by empty lines, describing the function's parameters, preconditions, return value, and side effects (if any). Further explanations, bibliographical references, and usage examples (which you should check with `doctest`) can optionally follow toward the end of the docstring.

Other attributes of function objects



In addition to its predefined attributes, a function object may have other arbitrary attributes. To create an attribute of a function object, bind a value to the appropriate attribute reference in an assignment statement after the `def` statement executes. For example, a function could count how many times it gets called:

```
def counter( ):
    counter.count += 1
    return counter.count
counter.count = 0
```

Note that this is *not* common usage. More often, when you want to group together some state (data) and some behavior (code), you should use the object-oriented mechanisms covered in Chapter 5. However, the ability to associate arbitrary attributes with a function can sometimes come in handy.

The return Statement

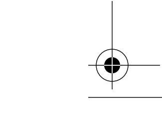
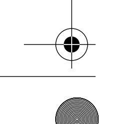
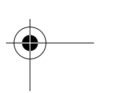
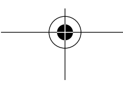
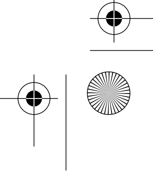
The `return` statement in Python is allowed only inside a function body and can optionally be followed by an expression. When `return` executes, the function terminates, and the value of the expression is the function's result. A function returns `None` if it terminates by reaching the end of its body or by executing a `return` statement that has no expression (or, of course, by executing `return None`).

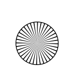
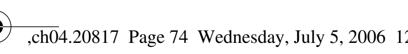
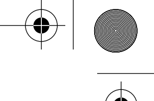
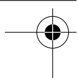
As a matter of style, you should *never* write a `return` statement without an expression at the end of a function body. If some `return` statements in a function have an expression, all `return` statements should have an expression. `return None` should only be written explicitly to meet this style requirement. Python does not enforce these stylistic conventions, but your code will be clearer and more readable if you follow them.

Calling Functions

A function call is an expression with the following syntax:

```
function-object(arguments)
```



,ch04.20817 Page 74 Wednesday, July 5, 2006 12:12 PM



function-object may be any reference to a function (or other callable) object; most often, it's the function's name. The parentheses denote the function-call operation itself. *arguments*, in the simplest case, is a series of zero or more expressions separated by commas (,), giving values for the function's corresponding parameters. When the function call executes, the parameters are bound to the argument values, the function body executes, and the value of the function-call expression is whatever the function returns.

Note that just *mentioning* a function (or other callable object) does *not* call it. To *call* a function (or other object) without arguments, you must use `()` after the function's name.

The semantics of argument passing

In traditional terms, all argument passing in Python is *by value*. For example, if you pass a variable as an argument, Python passes to the function the object (value) to which the variable currently refers, not "the variable itself." Thus, a function cannot rebind the caller's variables. However, if you pass a mutable object as an argument, the function may request changes to that object because Python passes the object itself, not a copy. Rebinding a variable and mutating an object are totally disjoint concepts. For example:

```
def f(x, y):
    x = 23
    y.append(42)
a = 77
b = [99]
f(a, b)
print a, b                  # prints: 77 [99, 42]
```

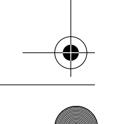
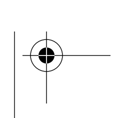
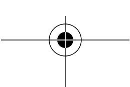
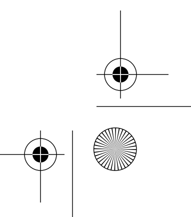
The `print` statement shows that `a` is still bound to `77`. Function `f`'s rebinding of its parameter `x` to `23` has no effect on `f`'s caller, nor, in particular, on the binding of the caller's variable that happened to be used to pass `77` as the parameter's value. However, the `print` statement also shows that `b` is now bound to `[99, 42]`. `b` is still bound to the same list object as before the call, but that object has mutated, as `f` has appended `42` to that list object. In either case, `f` has not altered the caller's bindings, nor can `f` alter the number `77`, since numbers are immutable. However, `f` can alter a list object, since list objects are mutable. In this example, `f` mutates the list object that the caller passes to `f` as the second argument by calling the object's `append` method.

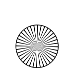
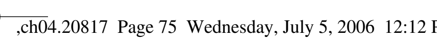
Kinds of arguments

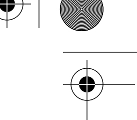
Arguments that are just expressions are known as *positional arguments*. Each positional argument supplies the value for the parameter that corresponds to it by position (order) in the function definition.

In a function call, zero or more positional arguments may be followed by zero or more *named arguments*, each with the following syntax:

```
identifier=expression
```



The *identifier* must be one of the parameter names used in the def statement for the function. The *expression* supplies the value for the parameter of that name. Most built-in functions do not accept named arguments, you must call such functions with positional arguments only. However, all normal functions coded in Python accept named as well as positional arguments, so you may call them in different ways.

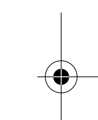
A function call must supply, via a positional or a named argument, exactly one value for each mandatory parameter, and zero or one value for each optional parameter. For example:

```
def divide(divisor, dividend):
    return dividend // divisor
print divide(12, 94)                          # prints: 7
print divide(dividend=94, divisor=12)         # prints: 7
```

As you can see, the two calls to divide are equivalent. You can pass named arguments for readability purposes whenever you think that identifying the role of each argument and controlling the order of arguments enhances your code's clarity.

A common use of named arguments is to bind some optional parameters to specific values, while letting other optional parameters take default values:

```
def f(middle, begin='init', end='finis'):
    return begin+middle+end
print f('tini', end='')                      # prints: inittini
```



Thanks to named argument end='', the caller can specify a value, the empty string '', for f's third parameter, end, and still let f's second parameter, begin, use its default value, the string 'init'.

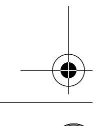
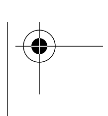
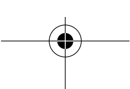
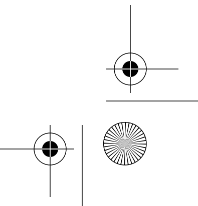
At the end of the arguments in a function call, you may optionally use either or both of the special forms **seq* and ***dct*. If both forms are present, the form with two asterisks must be last. **seq* passes the items of *seq* to the function as positional arguments (after the normal positional arguments, if any, that the call gives with the usual syntax). *seq* may be any iterable. ***dct* passes the items of *dct* to the function as named arguments, where *dct* must be a dictionary whose keys are all strings. Each item's key is a parameter name, and the item's value is the argument's value.

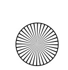
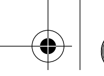
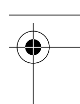
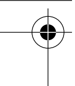
Sometimes you want to pass an argument of the form **seq* or ***dct* when the parameters use similar forms, as described earlier in "Parameters" on page 71. For example, using the function sum_args defined in that section (and shown again here), you may want to print the sum of all the values in dictionary *d*. This is easy with **seq*:

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(*d.values( ))
```

(Of course, in this case, print sum(d.values( )) would be simpler and more direct!)

However, you may also pass arguments of the form **seq* or ***dct* when calling a function that does not use the corresponding forms in its parameters. In that case,



,ch04.20817 Page 76 Wednesday, July 5, 2006 12:12 PM



of course, you must ensure that iterable *seq* has the right number of items, or, respectively, that dictionary *dct* uses the right names as its keys; otherwise, the call operation raises an exception.

Namespaces

A function's parameters, plus any variables that are bound (by assignment or by other binding statements, such as def) in the function body, make up the function's *local namespace*, also known as *local scope*. Each of these variables is known as a *local variable* of the function.

Variables that are not local are known as *global variables* (in the absence of nested function definitions, which we'll discuss shortly). Global variables are attributes of the module object, as covered in "Attributes of module objects" on page 140. Whenever a function's local variable has the same name as a global variable, that name, within the function body, refers to the local variable, not the global one. We express this by saying that the local variable *hides* the global variable of the same name throughout the function body.

The global statement

By default, any variable that is bound within a function body is a local variable of the function. If a function needs to rebind some global variables, the first statement of the function must be:

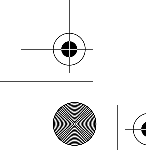
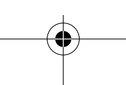
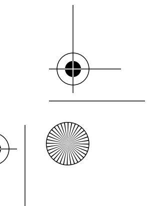
```
global identifiers
```

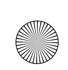
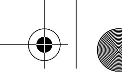
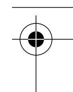
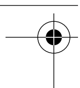
where *identifiers* is one or more identifiers separated by commas (,). The identifiers listed in a global statement refer to the global variables (i.e., attributes of the module object) that the function needs to rebind. For example, the function counter that we saw in "Other attributes of function objects" on page 73 could be implemented using global and a global variable, rather than an attribute of the function object:

```
_count = 0
def counter():
    global _count
    _count += 1
    return _count
```

Without the global statement, the counter function would raise an UnboundLocal-Error exception because _count would then be an uninitialized (unbound) local variable. While the global statement enables this kind of programming, this style is often inelegant and unadvisable. As I mentioned earlier, when you want to group together some state and some behavior, the object-oriented mechanisms covered in Chapter 5 are usually best.

Don't use global if the function body just *uses* a global variable (including mutating the object bound to that variable if the object is mutable). Use a global statement only if the function body *rebinds* a global variable (generally by assigning to the variable's name). As a matter of style, don't use global unless it's strictly necessary, as its presence will cause readers of your program to assume the

statement is there for some useful purpose. In particular, never use `global` except as the first statement in a function body.

Nested functions and nested scopes

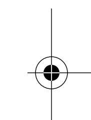
A `def` statement within a function body defines a *nested function*, and the function whose body includes the `def` is known as an *outer function* to the nested one. Code in a nested function's body may access (but not rebind) local variables of an outer function, also known as *free variables* of the nested function.

The simplest way to let a nested function access a value is often not to rely on nested scopes, but rather to explicitly pass that value as one of the function's arguments. If necessary, the argument's value can be bound when the nested function is defined by using the value as the default for an optional argument. For example:

```
def percent1(a, b, c):
    def pc(x, total=a+b+c): return (x*100.0) / total
    print "Percentages are:", pc(a), pc(b), pc(c)
```

Here's the same functionality using nested scopes:

```
def percent2(a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    print "Percentages are:", pc(a), pc(b), pc(c)
```

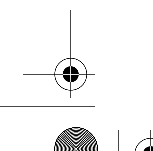
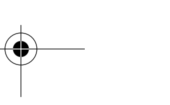
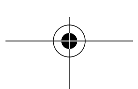
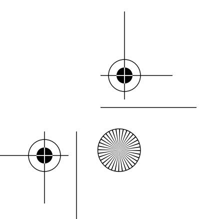


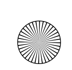
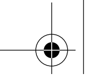
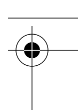
In this specific case, `percent1` has a tiny advantage: the computation of *a+b+c* happens only once, while `percent2`'s inner function `pc` repeats the computation three times. However, if the outer function rebinds its local variables between calls to the nested function, repeating the computation can be necessary. It's therefore advisable to be aware of both approaches, and choose the most appropriate one case by case.

A nested function that accesses values from outer local variables is also known as a *closure*. The following example shows how to build a closure:

```
def make_adder(augend):
    def add(addend):
        return addend+augend
    return add
```

Closures are an exception to the general rule that the object-oriented mechanisms covered in Chapter 5 are the best way to bundle together data and code. When you need specifically to construct callable objects, with some parameters fixed at object construction time, closures can be simpler and more effective than classes. For example, the result of `make_adder(7)` is a function that accepts a single argument and adds `7` to that argument. An outer function that returns a closure is a "factory" for members of a family of functions distinguished by some parameters, such as the value of argument *augend* in the previous example, and may often help you avoid code duplication.

lambda Expressions

If a function body is a single `return` *`expression`* statement, you may choose to replace the function with the special `lambda` expression form:

```
lambda parameters: expression
```

A `lambda` expression is the anonymous equivalent of a normal function whose body is a single `return` statement. Note that the `lambda` syntax does not use the `return` keyword. You can use a `lambda` expression wherever you could use a reference to a function. `lambda` can sometimes be handy when you want to use a simple function as an argument or return value. Here's an example that uses a `lambda` expression as an argument to the built-in `filter` function (covered in `filter` on page 161):

```
aList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
filter(lambda x, l=low, h=high: h>x>l, aList)    # returns: [4, 5, 6]
```

As an alternative, you can always use a local `def` statement that gives the function object a name. You can then use this name as the argument or return value. Here's the same `filter` example using a local `def` statement:

```
aList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
def within_bounds(value, l=low, h=high):
    return h>value>l
filter(within_bounds, aList)                     # returns: [4, 5, 6]
```

While `lambda` can occasionally be useful, many Python users prefer `def`, which is more general, and may make your code more readable if you choose a reasonable name for the function.

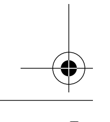
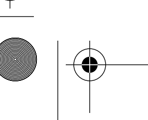
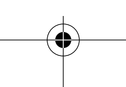
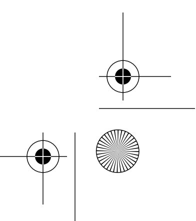
Generators

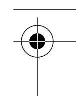
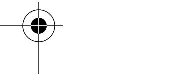
When the body of a function contains one or more occurrences of the keyword `yield`, the function is known as a *generator*. When you call a generator, the function body does not execute. Instead, calling the generator returns a special iterator object that wraps the function body, its local variables (including its parameters), and the current point of execution, which is initially the start of the function.

When the `next` method of this iterator object is called, the function body executes up to the next `yield` statement, which takes the form:

```
yield expression
```

When a `yield` statement executes, the function execution is "frozen," with current point of execution and local variables intact, and the expression following `yield` is returned as the result of the `next` method. When `next` is called again, execution of the function body resumes where it left off, again up to the next `yield` statement. If the function body ends, or executes a `return` statement, the iterator raises a `StopIteration` exception to indicate that the iteration is finished. `return` statements in a generator cannot contain expressions.

A generator is a very handy way to build an iterator. Since the most common way to use an iterator is to loop on it with a `for` statement, you typically call a generator like this:

```
for avariable in somegenerator(arguments):
```

For example, say that you want a sequence of numbers counting up from `1` to *`N`* and then down to `1` again. A generator can help:

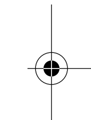
```
def updown(N):
    for x in xrange(1, N): yield x
    for x in xrange(N, 0, -1): yield x
for i in updown(3): print i                    # prints: 1 2 3 2 1
```

Here is a generator that works somewhat like the built-in `xrange` function, but returns a sequence of floating-point values instead of a sequence of integers:

```
def frange(start, stop, step=1.0):
    while start < stop:
        yield start
        start += step
```

This `frange` example is only *somewhat* like `xrange` because, for simplicity, it makes arguments `start` and `stop` mandatory, and silently assumes `step` is positive.

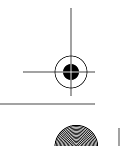
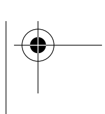
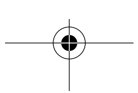
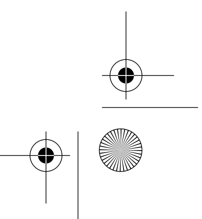
Generators are more flexible than functions that returns lists. A generator may build an *unbounded* iterator, meaning one that returns an infinite stream of results (to use only in loops that terminate by other means, e.g., via a `break` statement). Further, a generator-built iterator performs *lazy evaluation*: the iterator computes each successive item only when and if needed, just in time, while the equivalent function does all computations in advance and may require large amounts of memory to hold the results list. Therefore, if all you need is the ability to iterate on a computed sequence, it is often best to compute the sequence in a generator rather than in a function that returns a list. If the caller needs a list of all the items produced by some bounded generator *`G(arguments)`*, the caller can simply use the following code:

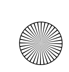
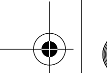
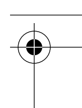
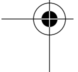


```
resulting_list = list(G(arguments))
```

Generator expressions

Python 2.4 introduces an even simpler way to code particularly simple generators: *generator expressions*, commonly known as *genexps*. The syntax of a genexp is just like that of a list comprehension (as covered in "List comprehensions" on page 67) except that a genexp is enclosed in parentheses (`()`) instead of brackets (`[]`); the semantics of a genexp are the same as those of the corresponding list comprehension, except that a genexp produces an iterator yielding one item at a time, while a list comprehension produces a list of all results in memory (therefore, using a genexp, when appropriate, saves memory). For example, to sum the squares of all single-digit integers, in any modern Python, you can code `sum([x*x for x in xrange(10)])`; in Python 2.4, you can express this functionality even better, coding it as `sum(x*x for x in xrange(10))` (just the same, but omitting the brackets), and obtain exactly the same result while consuming less memory. Note

that the parentheses that indicate the function call also "do double duty" and enclose the genexp (no need for extra parentheses).

Generators in Python 2.5

In Python 2.5, generators are further enhanced, with the possibility of receiving a value (or an exception) back from the caller as each `yield` executes. These advanced features allow generators in 2.5 to implement full-fledged co-routines, as explained at *http://www.python.org/peps/pep-0342.html*. The main change is that, in 2.5, `yield` is not a statement, but an expression, so it has a value. When a generator is resumed by calling its method `next`, the corresponding `yield`'s value is `None`. To pass a value *x* into some generator *g* (so that *g* receives *x* as the value of the `yield` on which it's suspended), instead of calling *g*`.next()`, the caller calls *g*`.send(`*x*`)` (calling *g*`.send(None)` is just like calling *g*`.next()`). Also, a bare `yield` without arguments, in Python 2.5, becomes legal, and equivalent to `yield None`.

Other Python 2.5 enhancements to generators have to do with exceptions, and are covered in "Generator enhancements" on page 126.

Recursion

Python supports recursion (i.e., a Python function can call itself), but there is a limit to how deep the recursion can be. By default, Python interrupts recursion and raises a `RecursionLimitExceeded` exception (covered in "Standard Exception Classes" on page 130) when it detects that the stack of recursive calls has gone over a depth of 1,000. You can change the recursion limit with function `setrecursionlimit` of module `sys`, covered in `setrecursionlimit` on page 171.

However, changing the recursion limit does not give you unlimited recursion; the absolute maximum limit depends on the platform on which your program is running, particularly on the underlying operating system and C runtime library, but it's typically a few thousand levels. If recursive calls get too deep, your program crashes. Such runaway recursion, after a call to `setrecursionlimit` that exceeds the platform's capabilities, is one of the very few ways a Python program can crash—really crash, hard, without the usual safety net of Python's exception mechanisms. Therefore, be wary of trying to fix a program that is getting `RecursionLimitExceeded` exceptions by raising the recursion limit too high with `setrecursionlimit`. Most often, you'd be better advised to look for ways to remove the recursion or, more specifically, limit the depth of recursion that your program needs.

Readers who are familiar with Lisp, Scheme, or functional-programming languages must in particular be aware that Python does *not* implement the optimization of "tail-call elimination," which is so important in these languages. In Python, any call, recursive or not, has the same cost in terms of both time and memory space, dependent only on the number of arguments: the cost does not change, whether the call is a "tail-call" (meaning that the call is the last operation that the caller executes) or any other, nontail call.