Python

ESSENTIAL REFERENCE

David M. Beazley

Third Edition

Python

ESSENTIAL REFERENCE

Third Edition

David Beazley

DEVELOPER'S LIBRARY

Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240 USA

Python Essential Reference, Third Edition

International Standard Book Number: 0-672-32862-3

Library of Congress Catalog Card Number: 200593277

Printed in the United States of America

First Printing: February 2006

09 08 07 06 4 3 2 1

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Acquisitions Editor
Jenny Watson

Development Editor
Scott Meyers

Managing Editor
Charlotte Clapp

Project Editor
Andy Beaster

Copy Editor
Bart Reed

Proofreader
Paula Lowell

Indexer
David Beazley

Technical Editor
Timothy Boronczyk

Publishing Coordinator
Vanessa Evans

Book Designer
Gary Adair

Page Layout
Michelle Mitchell

❖

This book is dedicated to "The Plan."

❖

Contents at a Glance

Table of Contents

About the Author

David M. Beazley is a long-time Python enthusiast, having been involved with the Python community since 1996. He is probably best known for his work on SWIG, a popular software package for integrating C/C++ programs with other programming languages, including Python, Perl, Ruby, Tcl, and Java. He has also written a number of other programming tools, including PLY, a Python implementation of lex and yacc. Dave spent seven years working in the Theoretical Physics Division at Los Alamos National Laboratory, where he helped pioneer the use of Python with massively parallel supercomputers. After that, Dave went off to work as an evil professor, where he briefly enjoyed tormenting college students with a variety of insane programming projects. However, he has since seen the error of his ways and is now working as a professional musician and occasional software consultant in Chicago. He can be contacted at http://www.dabeaz.com.

Acknowledgments

This book would not be possible without the support of many people. First, I'd like to thank Timothy Boronczyk for his feedback regarding the third edition. I'd also like to acknowledge past technical reviewers Paul DuBois, Mats Wichmann, David Ascher, and Tim Bell for their valuable comments and advice that made earlier editions a success. Guido van Rossum, Jeremy Hylton, Fred Drake, Roger Masse, and Barry Warsaw also provided tremendous assistance with the first edition while hosting me for a few weeks back in the hot summer of 1999. Last, but not least, this book would not be possible without all of the feedback I received from readers. There are far too many people to list individually, but I have done my best to incorporate your suggestions for making the book even better. I'd also like to thank all the folks at Sams Publishing and Pearson Education for their continued commitment to the project and assistance. Jenny Watson, Scott Meyers, Andy Beaster, and Bart Reed all helped out to get this edition out the door in good shape. A special thanks is in order for Robin Drake, whose tremendous effort in editing previous editions made the third edition possible. Finally, I'd like to offer a special acknowledgment to my musical partners in crime: Jim Trompeter, David Bloom, Thomas Mucha, Trent Harris, Matt Mayes, Marc Piane, and Alex Alvarado. They had absolutely nothing whatsoever to do with Python or this book, but they put up with me when I was spending far too much time working on the book and not enough time working on interesting new bebop lines. Thanks guys.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail:	opensource@samspublishing.com
Mail:	Mark Taber Associate Publisher Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.samspublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

THIS BOOK IS INTENDED TO BE A CONCISE REFERENCE to the Python programming language. Although an experienced programmer will probably be able to learn Python from this book, it's not intended to be an extended tutorial or a treatise on how to program. Rather, the goal is to present the core Python language, the contents of the Python library, and the Python extension API in a manner that's accurate and concise. This book assumes that the reader has prior programming experience with Python or another language such as C or Java. In a addition, a general familiarity with systems programming topics (for example, basic operating system concepts and network programming) may be useful in understanding certain parts of the library reference.

Python is freely available for download at http://www.python.org. Versions are available for almost every operating system, including UNIX, Windows, Macintosh, and Java. In addition, the Python website includes links to documentation, how-to guides, and a wide assortment of third-party software.

The contents of this book are based on Python 2.4. However, readers should be aware that Python is a constantly evolving language. Most of the topics described herein are likely to be applicable to future versions of Python 2.x. In addition, much of the material is applicable to earlier releases. To a lesser extent, the topics in this book also apply to alternative Python implementations such as Jython (an implementation of Python in Java) and IronPython (an implementation of Python for .NET). However, those implementations are not the primary focus.

Just as Python is an evolving language, the third edition of *Python Essential Reference* has evolved to make use of new language features and new library modules. In fact, since the publication of the second edition, Python has undergone a dramatic transformation involving significant changes to core parts of the language. In addition, a wide variety of new and interesting features have been added. Rather than discussing these changes as a mere afterthought, the entire text has been updated to reflect the modern state of Python programming. Although no distinction is given to new features, detailed descriptions of language changes can be found at http://www.python.org.

Finally, it should be noted that Python already includes thousands of pages of useful documentation. The contents of this book are largely based on that documentation, but with a number of key differences. First, this reference presents most of the same information in a much more compact form, with different examples, and alternative descriptions of many topics. Second, a significant number of topics in the library reference have been expanded to include outside reference material. This is especially true for low-level system and networking modules in which effective use of a module normally relies on a myriad of options listed in manuals and outside references. In addition, in order to produce a more concise reference, a number of deprecated and relatively obscure library modules have been omitted. Finally, this reference doesn't attempt to cover large frameworks such as Tkinter, XML, and the COM extensions, as these topics are beyond the scope of this book and are described in books of their own.

In writing this book, it has been my goal to produce a reference containing virtually everything I have needed to use Python and its large collection of modules. Although this is by no means a gentle introduction to the Python language, I hope that you find the contents of this book to be a useful addition to your programming reference library for many years to come. I welcome your comments.

David Beazley
Chicago, Illinois
November 27, 2005

I

The Python Language

This page intentionally left blank

1
A Tutorial Introduction

THIS CHAPTER PROVIDES A QUICK INTRODUCTION to Python. The goal is to illustrate Python's essential features without getting too bogged down in special rules or details. To do this, the chapter briefly covers basic concepts such as variables, expressions, control flow, functions, classes, and input/output. This chapter is not intended to provide comprehensive coverage, nor does it cover all of Python's more advanced features. However, experienced programmers should be able to extrapolate from the material in this chapter to create more advanced programs. Beginners are encouraged to try a few examples to get a feel for the language.

Running Python

Python programs are executed by an interpreter. On most machines, the interpreter can be started by simply typing **`python`**. However, many different programming environments for Python are currently available (for example, ActivePython, PythonWin, IDLE, and PythonIDE). In this case, Python is started by launching the appropriate application. When the interpreter starts, a prompt appears at which you can start typing programs into a simple read-evaluation loop. For example, in the following output, the interpreter displays its copyright message and presents the user with the `>>>` prompt, at which the user types the familiar "Hello World" command:

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>>
```

Programs can also be placed in a file such as the following:

```
# helloworld.py
print "Hello World"
```

Python source files are ordinary text files and normally have a `.py` suffix. The `#` character denotes a comment that extends to the end of the line.

To execute the `helloworld.py` file, you provide the filename to the interpreter as follows:

```
% python helloworld.py
Hello World
%
```

On Windows, Python programs can be started by double-clicking a `.py` file or typing the name of the program into the "run" command on the Windows "Start" menu. This launches the interpreter and runs the program in a console window. In this case, the console window disappears immediately after the program completes its execution (often before you can read its output). To prevent this problem, you should use an integrated development environment such as PythonWin. An alternative approach is to launch the program using a `.bat` file containing a statement such as `python -i helloworld.py` that instructs the interpreter to enter interactive mode after program execution.

Within the interpreter, the `execfile()` function runs a program, as in the following example:

```
>>> execfile("helloworld.py")
Hello World
```

On UNIX, you can also invoke Python using `#!` in a shell script:

```
#!/usr/local/bin/python
print "Hello World"
```

The interpreter runs until it reaches the end of the input file. If it's running interactively, you can exit the interpreter by typing the EOF (end of file) character or by selecting Exit from a pull-down menu. On UNIX, EOF is Ctrl+D; on Windows, it's Ctrl+Z. A program can also exit by calling the `sys.exit()` function or raising the `SystemExit` exception. For example:

```
>>> import sys
>>> sys.exit()
```

or

```
>>> raise SystemExit
```

Variables and Arithmetic Expressions

The program in Listing 1.1 shows the use of variables and expressions by performing a simple compound-interest calculation.

Listing 1.1 **Simple Compound-Interest Calculation**

```
principal = 1000        # Initial amount
rate = 0.05             # Interest rate
numyears = 5            # Number of years
year = 1
while year <= numyears:
        principal = principal*(1+rate)
        print year, principal
        year += 1
```

The output of this program is the following table:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

Python is a dynamically typed language in which names can represent values of different types during the execution of a program. In fact, the names used in a program are really just labels for various quantities and objects. The assignment operator simply creates an association between a name and a value. This is different from C, for example, in which a name represents a fixed size and location in memory into which results are placed. The dynamic behavior of Python can be seen in Listing 1.1 with the `principal` variable. Initially, it's assigned to an integer value. However, later in the program it's reassigned as follows:

```
principal = principal*(1+rate)
```

This statement evaluates the expression and reassociates the name `principal` with the result. When this occurs, the original binding of `principal` to the integer `1000` is lost. Furthermore, the result of the assignment may change the type of the variable. In this case, the type of `principal` changes from an integer to a floating-point number because `rate` is a floating-point number.

A newline terminates each individual statement. You also can use a semicolon to separate statements, as shown here:

```
principal = 1000; rate = 0.05; numyears = 5;
```

The `while` statement tests the conditional expression that immediately follows. If the tested statement is true, the body of the `while` statement executes. The condition is then retested and the body executed again until the condition becomes false. The body of the loop is denoted by indentation; the three statements following `while` in Listing 1.1 execute on each iteration. Python doesn't specify the amount of required indentation, as long as it's consistent within a block.

One problem with the program in Listing 1.1 is that the output isn't very pretty. To make it better, you could right-align the columns and limit the precision of `principal` to two digits by modifying `print` to use a format string, like this:

```
print "%3d  %0.2f" % (year, principal)
```

Now the output of the program looks like this:

```
1  1050.00
2  1102.50
3  1157.63
4  1215.51
5  1276.28
```

Format strings contain ordinary text and special formatting-character sequences such as `"%d"`, `"%s"`, and `"%f"`. These sequences specify the formatting of a particular type of data such as an integer, string, or floating-point number, respectively. The special-character sequences can also contain modifiers that specify a width and precision. For example, `"%3d"` formats an integer right-aligned in a column of width 3, and `"%0.2f"` formats a floating-point number so that only two digits appear after the decimal point. The behavior of format strings is almost identical to the C `sprintf()` function and is described in detail in Chapter 4, "Operators and Expressions."

Conditionals

The `if` and `else` statements can perform simple tests. Here's an example:

```
# Compute the maximum (z) of a and b
if a < b:
       z = b
```

```
else:
        z = a
```

The bodies of the `if` and `else` clauses are denoted by indentation. The `else` clause is optional.

To create an empty clause, use the `pass` statement as follows:

```
if a < b:
        pass      # Do nothing
else:
        z = a
```

You can form Boolean expressions by using the `or`, `and`, and `not` keywords:

```
if b >= a and b <= c:
        print "b is between a and c"
if not (b < a or b > c):
        print "b is still between a and c"
```

To handle multiple-test cases, use the `elif` statement, like this:

```
if a == '+':
        op = PLUS
elif a == '-':
        op = MINUS
elif a == '*':
        op = MULTIPLY
else:
        raise RuntimeError, "Unknown operator"
```

To denote truth values, you can use the Boolean values `True` and `False`. Here's an example:

```
if c in '0123456789':
    isdigit = True
else:
    isdigit = False
```

File Input and Output

The following program opens a file and reads its contents line by line:

```
f = open("foo.txt")        # Returns a file object
line = f.readline()        # Invokes readline() method on file
while line:
        print line,        # trailing ',' omits newline character
        line = f.readline()
f.close()
```

The `open()` function returns a new file object. By invoking methods on this object, you can perform various file operations. The `readline()` method reads a single line of input, including the terminating newline. An empty string is returned at the end of the file.

In the example, the program is simply looping over all the lines in the file `foo.txt`. Whenever a program loops over a collection of data like this (for instance input lines, numbers, strings, and so on), it is commonly known as "iteration." Because iteration is such a common operation, Python provides a number of shortcuts for simplifying the process. For instance, the same program can be written much more succinctly as follows:

```
for line in open("foo.txt"):
print line,
```

To make the output of a program go to a file, you can supply a file to the `print` statement using `>>`, as shown in the following example:

```
f = open("out","w")     # Open file for writing
while year <= numyears:
        principal = principal*(1+rate)
        print >>f,"%3d   %0.2f" % (year,principal)
        year += 1
f.close()
```

In addition, file objects support a `write()` method that can be used to write raw data. For example, the `print` statement in the previous example could have been written this way:

```
f.write("%3d   %0.2f\n" % (year,principal))
```

Although these examples have worked with files, the same techniques apply to the standard output and input streams of the interpreter. For example, if you wanted to read user input interactively, you can read from the file `sys.stdin`. If you want to write data to the screen, you can write to `sys.stdout`, which is the same file used to output data produced by the `print` statement. For example:

```
import sys
sys.stdout.write("Enter your name :")
name = sys.stdin.readline()
```

The preceding code can also be shortened to the following:

```
name = raw_input("Enter your name :")
```

Strings

To create string literals, enclose them in single, double, or triple quotes as follows:

```
a = "Hello World"
b = 'Python is groovy'
c = """What is footnote 5?"""
```

The same type of quote used to start a string must be used to terminate it. Triple-quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line. Triple-quoted strings are useful when the contents of a string literal span multiple lines of text such as the following:

```
print '''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
'''
```

Strings are sequences of characters indexed by integers, starting at zero. To extract a single character, use the indexing operator `s[i]` like this:

```
a = "Hello World"
b = a[4]                # b = 'o'
```

To extract a substring, use the slicing operator `s[i:j]`. This extracts all elements from `s` whose index `k` is in the range `i <= k < j`. If either index is omitted, the beginning or end of the string is assumed, respectively:

```
c = a[:5]              # c = "Hello"
d = a[6:]               # d = "World"
e = a[3:8]              # e = "lo Wo"
```

Strings are concatenated with the plus (+) operator:

```
g = a + " This is a test"
```

Other data types can be converted into a string by using either the `str()` or `repr()` function or backquotes (`` ` ``), which are a shortcut notation for `repr()`. For example:

```
s = "The value of x is " + str(x)
s = "The value of y is " + repr(y)
s = "The value of y is " + `y`
```

In many cases, `str()` and `repr()` return identical results. However, there are subtle differences in semantics that are described in later chapters.

Lists

Lists are sequences of arbitrary objects. You create a list as follows:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

```
a = names[2]            # Returns the third item of the list, "Ann"
names[0] = "Jeff"       # Changes the first item to "Jeff"
```

To append new items to the end of a list, use the `append()` method:

```
names.append("Kate")
```

To insert an item in the list, use the `insert()` method:

```
names.insert(2, "Sydney")
```

You can extract or reassign a portion of a list by using the slicing operator:

```
b = names[0:2]                     # Returns [ "Jeff", "Mark" ]
c = names[2:]                      # Returns [ "Sydney", "Ann", "Phil", "Kate" ]
names[1] = 'Jeff'                  # Replace the 2nd item in names with 'Jeff'
names[0:2] = ['Dave','Mark','Jeff'] # Replace the first two items of
                                   # the list with the list on the right.
```

Use the plus (+) operator to concatenate lists:

```
a = [1,2,3] + [4,5]     # Result is [1,2,3,4,5]
```

Lists can contain any kind of Python object, including other lists, as in the following example:

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

Nested lists are accessed as follows:

```
a[1]            # Returns "Dave"
a[3][2]         # Returns 9
a[3][3][1]      # Returns 101
```

The program in Listing 1.2 illustrates a few more advanced features of lists by reading a list of numbers from a file specified on the command line and outputting the minimum and maximum values.

Listing 1.2 **Advanced List Features**

```
import sys                       # Load the sys module
if len(sys.argv) != 2            # Check number of command line arguments :
     print "Please supply a filename"
     raise SystemExit
f = open(sys.argv[1])            # Filename on the command line
svalues = f.readlines()          # Read all lines into a list
f.close()

# Convert all of the input values from strings to floats
fvalues = [float(s) for s in svalues]

# Print min and max values
print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

The first line of this program uses the `import` statement to load the `sys` module from the Python library. This module is being loaded in order to obtain command-line arguments.

The `open()` method uses a filename that has been supplied as a command-line option and stored in the list `sys.argv`. The `readlines()` method reads all the input lines into a list of strings.

The expression `[float(s) for s in svalues]` constructs a new list by looping over all the strings in the list `svalues` and applying the function `float()` to each element. This particularly powerful method of constructing a list is known as a *list comprehension*.

After the input lines have been converted into a list of floating-point numbers, the built-in `min()` and `max()` functions compute the minimum and maximum values.

Tuples

Closely related to lists is the tuple data type. You create tuples by enclosing a group of values in parentheses, like this:

```
a = (1,4,5,-9,10)
b = (7,)                                # Singleton (note extra ,)
person = (first_name, last_name, phone)
```

Sometimes Python recognizes that a tuple is intended, even if the parentheses are missing:

```
a = 1,4,5,-9,10
b = 7,
person = first_name, last_name, phone
```

Tuples support most of the same operations as lists, such as indexing, slicing, and concatenation. The only difference is that you cannot modify the contents of a tuple after creation (that is, you cannot modify individual elements or append new elements to a tuple).

Sets

A set is used to contain an unordered collection of objects. To create a set, use the `set()` function and supply a sequence of items such as follows:

```
s = set([3,5,9,10])       # Create a set of numbers
t = set("Hello")          # Create a set of characters
```

Unlike lists and tuples, sets are unordered and cannot be indexed in the same way. More over, the elements of a set are never duplicated. For example, if you print the value of `t` from the preceding code, you get the following:

```
>>> print t
set(['H', 'e', 'l', 'o'])
```

Notice that only one `'l'` appears.

Sets support a standard collection of set operations, including union, intersection, difference, and symmetric difference. For example:

```
a = t | s          # Union of t and s
b = t & s          # Intersection of t and s
c = t - s          # Set difference (items in t, but not in s)
d = t ^ s          # Symmetric difference (items in t or s, but not both)
```

New items can be added to a set using `add()` or `update()`:

```
t.add('x')
s.update([10,37,42])
```

An item can be removed using `remove()`:

```
t.remove('H')
```

Dictionaries

A *dictionary* is an associative array or hash table that contains objects indexed by keys. You create a dictionary by enclosing the values in curly braces (`{ }`) like this:

```
a = {
        "username" : "beazley",
        "home" : "/home/beazley",
        "uid" : 500
     }
```

To access members of a dictionary, use the key-indexing operator as follows:

```
u = a["username"]
d = a["home"]
```

Inserting or modifying objects works like this:

```
a["username"] = "pxl"
a["home"] = "/home/pxl"
a["shell"] = "/usr/bin/tcsh"
```

Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples. Some objects, including lists and dictionaries, cannot be used as keys, because their contents are allowed to change.

Dictionary membership is tested with the `has_key()` method, as in the following example:

```
if a.has_key("username"):
     username = a["username"]
else:
     username = "unknown user"
```

This particular sequence of steps can also be performed more compactly as follows:

```
username = a.get("username", "unknown user")
```

To obtain a list of dictionary keys, use the `keys()` method:

```
k = a.keys()            # k = ["username","home","uid","shell"]
```

Use the `del` statement to remove an element of a dictionary:

```
del a["username"]
```

Iteration and Looping

The simple loop shown earlier used the `while` statement. The other looping construct is the `for` statement, which is used to iterate over a collection of items. Iteration is one of Python's most rich features. However, the most common form of iteration is to simply loop over all the members of a sequence such as a string, list, or tuple. Here's an example:

```
for i in range(1,10):
        print "2 to the %d power is %d" % (i, 2**i)
```

The `range(i,j)` function constructs a list of integers with values from `i` to `j-1`. If the starting value is omitted, it's taken to be zero. An optional stride can also be given as a third argument. For example:

```
a = range(5)         # a = [0,1,2,3,4]
b = range(1,8)       # b = [1,2,3,4,5,6,7]
c = range(0,14,3)    # c = [0,3,6,9,12]
d = range(8,1,-1)    # d = [8,7,6,5,4,3,2]
```

The `range()` function works by constructing a list and populating it with values according to the starting, ending, and stride values. For large ranges, this process is expensive in terms of both memory and runtime performance. To avoid this, you can use the `xrange()` function, as shown here:

```
for i in xrange(1,10):
        print "2 to the %d power is %d" % (i, 2**i)

a = xrange(100000000)       # a = [0,1,2, ..., 99999999]
b = xrange(0,100000000,5)   # b = [0,5,10, ...,99999995]
```

Rather than creating a sequence populated with values, the sequence returned by `xrange()` computes its values from the starting, ending, and stride values whenever it's accessed.

The `for` statement is not limited to sequences of integers and can be used to iterate over many kinds of objects, including strings, lists, and dictionaries. For example:

```
a = "Hello World"
# Print out the characters in a
for c in a:
        print c

b = ["Dave","Mark","Ann","Phil"]
```

```
# Print out the members of a list
for name in b:
        print name

c = { 'a' : 3, 'name': 'Dave', 'x': 7.5 }
# Print out all of the members of a dictionary
for key in c:
        print key, c[key]
```

In addition, the `for` statement can be applied to any object that supports a special iteration protocol. In an earlier example, iteration was used to loop over all the lines in a file:

```
for line in open("foo.txt"):
print line,
```

This works because files provide special iteration methods that work as follows:

```
>>> i = f.__iter__()    # Return an iterator object
>>> i.next()            # Return first line
>>> i.next()            # Return next line
... continues ...
>>> i.next()            # No more data
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Underneath the covers, the `for` statement relies on these methods to iterate over lines in the file.

Instead of iterating over a collection of items such as the elements of a list, it is also possible to iterate over an object that knows how to generate items on demand. This sort of object is called a *generator* and is defined using a function. For example, if you wanted to iterate over the Fibonacci numbers, you could do this:

```
# Generate fibonacci numbers
def fibonacci(max):
    s = 1
    t = 1
    while s < max:
        yield s          # Produce a value
        w = s + t
        s = t
        t = w
    return

# Print fibonacci numbers less than 1000
for n in fibonacci(1000):
    print n
```

In this case, the `yield` statement produces a value used in iteration. When the next value is requested, the function resumes execution right after `yield`. Iteration stops when the generator function returns. More details about iterators and generators can be found in Chapter 6, "Functions and Functional Programming."

Functions

You use the `def` statement to create a function, as shown in the following example:

```
def remainder(a,b):
        q = a // b       # // is truncating division.
        r = a - q*b
        return r
```

To invoke a function, simply use the name of the function followed by its arguments enclosed in parentheses, such as `result = remainder(37,15)`.You can use a tuple to return multiple values from a function, as shown here:

```
def divide(a,b):
        q = a // b        # If a and b are integers, q is integer
        r = a - q*b
        return (q,r)
```

When returning multiple values in a tuple, it's often useful to invoke the function as follows:

```
quotient, remainder = divide(1456,33)
```

To assign a default value to a parameter, use assignment:

```
def connect(hostname,port,timeout=300):
    # Function body
```

When default values are given in a function definition, they can be omitted from subsequent function calls. When omitted, the argument will simply take on the default value. For example:

```
connect('www.python.org', 80)
```

You also can invoke functions by using keyword arguments and supplying the arguments in arbitrary order. However, this requires you to know the names of the arguments in the function definition. For example:

```
connect(port=80,hostname="www.python.org")
```

When variables are created or assigned inside a function, their scope is local. That is, the variable is only defined inside the body of the function and is destroyed when the function returns. To modify the value of a global variable from inside a function, use the `global` statement as follows:

```
a = 4.5
...
def foo():
        global a
        a = 8.8             # Changes the global variable a
```

Classes

The `class` statement is used to define new types of objects and for object-oriented programming. For example, the following class defines a simple stack with `push()`, `pop()`, and `length()` operations:

```
class Stack(object):
        def __init__(self):           # Initialize the stack
                self.stack = [ ]
        def push(self,object):
                self.stack.append(object)
        def pop(self):
                return self.stack.pop()
        def length(self):
                return len(self.stack)
```

In the first line of the class definition, the statement `class Stack(object)` declares `Stack` to be an `object`. The use of parentheses is how Python specifies inheritance—in this case, `Stack` inherits from `object`, which is the root of all Python types. Inside the class definition, methods are defined using the `def` statement. The first argument in each method always refers to the object itself. By convention, `self` is the name used for this argument. All operations involving the attributes of an object must explicitly refer to the `self` variable. Methods with leading and trailing double underscores are special methods. For example, `__init__` is used to initialize an object after it's created.

To use a class, write code such as the following:

```
s = Stack()              # Create a stack
s.push("Dave")           # Push some things onto it
s.push(42)
s.push([3,4,5])
x = s.pop()              # x gets [3,4,5]
y = s.pop()              # y gets 42
del s                    # Destroy s
```

In this example, an entirely new object was created to implement the stack. However, a stack is almost identical to the built-in list object. Therefore, an alternative approach would be to inherit from `list` and add an extra method:

```
class Stack(list):
        # Add push() method for stack interface
        # Note: lists already provide a pop() method.
        def push(self,object):
                self.append(object)
```

Normally, all of the methods defined within a class apply only to instances of that class (that is, the objects that are created). However, different kinds of methods can be defined, such as static methods familiar to C++ and Java programmers. For example:

```
class EventHandler(object):
        @staticmethod
        def dispatcherThread():
               while (1):
                        # Wait for requests
                     ...

EventHandler.dispatcherThread()          # Call method as a function
```

In this case, `@staticmethod` declares the method that follows to be a static method. `@staticmethod` is actually an example of using an object known as a decorator—a topic that is discussed further in the chapter on functions and functional programming.

Exceptions

If an error occurs in your program, an exception is raised and an error message such as the following appears:

```
Traceback (innermost last):
 File "<interactive input>", line 42, in foo.py
NameError: a
```

The error message indicates the type of error that occurred, along with its location. Normally, errors cause a program to terminate. However, you can catch and handle exceptions using the `try` and `except` statements, like this:

```
try:
    f = open("file.txt","r")
except IOError, e:
    print e
```

If an `IOError` occurs, details concerning the cause of the error are placed in `e` and control passes to the code in the `except` block. If some other kind of exception is raised, it's passed to the enclosing code block (if any). If no errors occur, the code in the `except` block is ignored. When an exception is handled, program execution resumes with the statement that immediately follows the `except` block. The program does not return back to the location where the exception occurred.

The `raise` statement is used to signal an exception. When raising an exception, you can use one of the built-in exceptions, like this:

```
raise RuntimeError, "Unrecoverable error"
```

Or you can create your own exceptions, as described in the section "Defining New Exceptions" in Chapter 5, "Control Flow."

Modules

As your programs grow in size, you'll probably want to break them into multiple files for easier maintenance. To do this, Python allows you to put definitions in a file and use them as a module that can be imported into other programs and scripts. To create a module, put the relevant statements and definitions into a file that has the same name as the module. (Note that the file must have a `.py` suffix.) Here's an example:

```
# file :  div.py
def divide(a,b):
    q = a//b        # If a and b are integers, q is an integer
    r = a - q*b
    return (q,r)
```

To use your module in other programs, you can use the `import` statement:

```
import div
a, b = div.divide(2305, 29)
```

The `import` statement creates a new namespace that contains all the objects defined in the module. To access this namespace, simply use the name of the module as a prefix, as in `div.divide()` in the preceding example.

If you want to import a module using a different name, supply the `import` statement with an optional `as` qualifier, as follows:

```
import div as foo
a,b = foo.divide(2305,29)
```

To import specific definitions into the current namespace, use the `from` statement:

```
from div import divide
a,b = divide(2305,29)       # No longer need the div prefix
```

To load all of a module's contents into the current namespace, you can also use the following:

```
from div import *
```

Finally, the `dir()` function lists the contents of a module and is a useful tool for interactive experimentation, because it can be used to provide a list of available functions and variables:

```
>>> import string
>>> dir(string)
['__builtins__', '__doc__', '__file__', '__name__', '_idmap',
 '_idmapL', '_lower', '_swapcase', '_upper', 'atof', 'atof_error',
 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
...
>>>
```

Getting Help

When working with Python, you have several sources of quickly available information. First, when Python is running in interactive mode, you can use the `help()` command to get information about built-in modules and other aspects of Python. Simply type **`help()`** by itself for general information or **`help('`*`modulename`*`')`** for information about a specific module. The `help()` command can also be used to return information about specific functions if you supply a function name.

Most Python functions have documentation strings that describe their usage. To print the doc string, simply print the `__doc__` attribute. Here's an example:

```
>>> print issubclass.__doc__
issubclass(C, B) -> bool

Return whether class C is a subclass (i.e., a derived class) of class B.
When using a tuple as the second argument issubclass(X, (A, B, ...)),
is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).
>>>
```

Last, but not least, most Python installations also include the command `pydoc`, which can be used to return documentation about Python modules. Simply type **`pydoc`** ***`topic`*** at a command prompt (for example, in the Unix command shell).

2

Lexical Conventions and Syntax

THIS CHAPTER DESCRIBES THE SYNTACTIC AND LEXICAL CONVENTIONS of a Python program. Topics include line structure, grouping of statements, reserved words, literals, operators, tokens, and source code encoding. In addition, the use of Unicode string literals is described in detail.

Line Structure and Indentation

Each statement in a program is terminated with a newline. Long statements can span multiple lines by using the line-continuation character (\), as shown in the following example:

```
a = math.cos(3*(x-n)) + \
    math.sin(3*(y-n))
```

You don't need the line-continuation character when the definition of a triple-quoted string, list, tuple, or dictionary spans multiple lines. More generally, any part of a program enclosed in parentheses `(...)`, brackets `[...]`, braces `{...}`, or triple quotes can span multiple lines without use of the line-continuation character because they denote the start and end of a definition.

Indentation is used to denote different blocks of code, such as the bodies of functions, conditionals, loops, and classes. The amount of indentation used for the first statement of a block is arbitrary, but the indentation of the entire block must be consistent. For example:

```
if a:
   statement1     # Consistent indentation
   statement2
else:
   statement3
     statement4   # Inconsistent indentation (error)
```

If the body of a function, conditional, loop, or class is short and contains only a few statements, they can be placed on the same line, like this:

```
if a:  statement1
else:  statement2
```

To denote an empty body or block, use the `pass` statement. For example:

```
if a:
   pass
else:
   statements
```

Although tabs can be used for indentation, this practice is discouraged. The use of spaces is universally preferred (and encouraged) by the Python programming community. When tab characters are encountered, they're converted into the number of spaces required to move to the next column that's a multiple of 8 (for example, a tab appearing in column 11 inserts enough spaces to move to column 16). Running Python with the `-t` option prints warning messages when tabs and spaces are mixed inconsistently within the same program block. The `-tt` option turns these warning messages into `TabError` exceptions.

To place more than one statement on a line, separate the statements with a semicolon (`;`). A line containing a single statement can also be terminated by a semicolon, although this is unnecessary and considered poor style.

The `#` character denotes a comment that extends to the end of the line. A `#` appearing inside a quoted string doesn't start a comment, however.

Finally, the interpreter ignores all blank lines except when running in interactive mode. In this case, a blank line signals the end of input when typing a statement that spans multiple lines.

Identifiers and Reserved Words

An *identifier* is a name used to identify variables, functions, classes, modules, and other objects. Identifiers can include letters, numbers, and the underscore character (_), but must always start with a nonnumeric character. Letters are currently confined to the characters A–Z and a–z in the ISO-Latin character set. Because identifiers are case sensitive, `FOO` is different from `foo`. Special symbols such as `$`, `%`, and `@` are not allowed in identifiers. In addition, words such as `if`, `else`, and `for` are reserved and cannot be used as identifier names. The following list shows all the reserved words:

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

Identifiers starting or ending with underscores often have special meanings. For example, identifiers starting with a single underscore such as `_foo` are not imported by the `from module import *` statement. Identifiers with leading and trailing double underscores such as `__init__` are reserved for special methods, and identifiers with leading double underscores such as `__bar` are used to implement private class members, as described in Chapter 7, "Classes and Object-Oriented Programming." General-purpose use of similar identifiers should be avoided.

Literals

There are five built-in numeric types:

- Booleans
- Integers
- Long integers
- Floating-point numbers
- Complex numbers

The identifiers `True` and `False` are interpreted as Boolean values with the integer values of 0 and 1, respectively. A number such as `1234` is interpreted as a decimal integer. To specify an octal or hexadecimal integer, precede the value with `0` or `0x`, respectively (for example, `0644` or `0x100fea8`). Long integers are typically written with a trailing `l` (ell) or `L` character, as in `1234567890L`. Unlike integers, which are limited by machine precision, long integers can be of any length (up to the maximum memory of the machine). Although the trailing `L` is used to denote long integers, it may be omitted. In this case, a large integer value will automatically be converted into a long integer if it exceeds the precision of the standard integer type. Numbers such as `123.34` and `1.2334e+02` are interpreted as floating-point numbers. An integer or floating-point number with a trailing `j` or `J`, such as `12.34J`, is a complex number. You can create complex numbers with real and imaginary parts by adding a real number and an imaginary number, as in `1.2 + 12.34J`.

Python currently supports two types of string literals:

- 8-bit character data (ASCII)
- Unicode (16-bit-wide character data)

The most commonly used string type is 8-bit character data, because of its use in representing characters from the ASCII or ISO-Latin character set as well as representing raw binary data as a sequence of bytes. By default, 8-bit string literals are defined by enclosing text in single (`'`), double (`"`), or triple (`'''` or `"""`) quotes. You must use the same type of quote to start and terminate a string. The backslash (\) character is used to escape special characters such as newlines, the backslash itself, quotes, and nonprinting characters. Table 2.1 shows the accepted escape codes. Unrecognized escape sequences are left in the string unmodified and include the leading backslash. Furthermore, it's legal for strings to contain embedded null bytes and binary data. Triple-quoted strings can span multiple lines and include unescaped newlines and quotes.

Table 2.1 **Standard Character Escape Codes**

Character	Description
`\`	Newline continuation
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote

Table 2.1 **Continued**

Character	**Description**
`\a`	Bell
`\b`	Backspace
`\e`	Escape
`\0`	Null
`\n`	Line feed
`\v`	Vertical tab
`\t`	Horizontal tab
`\r`	Carriage return
`\f`	Form feed
`\OOO`	Octal value (`\000` to `\377`)
`\xhh`	Hexadecimal value (`\x00` to `\xff`)

Unicode strings are used to represent multibyte international character sets and allow for 65,536 unique characters. Unicode string literals are defined by preceding an ordinary string literal with a `u` or `U`, such as in `u"hello"`. In Unicode, each character is internally represented by a 16-bit integer value. For the purposes of notation, this value is written as `U+`*`XXXX`*, where *`XXXX`* is a four-digit hexadecimal number. (Note that this notation is only a convention used to describe Unicode characters and is not Python syntax.) For example, `U+0068` is the Unicode character for the letter *h* in the Latin-1 character set. When Unicode string literals are defined, standard characters and escape codes are directly mapped as Unicode ordinals in the range [U+0000, U+00FF]. For example, the string `"hello\n"` is mapped to the sequence of ASCII values 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0a, whereas the Unicode string `u"hello\n"` is mapped to the sequence U+0068, U+0065, U+006C, U+006C, U+006F, U+000A. Arbitrary Unicode characters are defined using the `\u`*`XXXX`* escape sequence. This sequence can only appear inside a Unicode string literal and must always specify a four-digit hexadecimal value. For example:

```
s = u"\u0068\u0065\u006c\u006c\u006f\u000a"
```

In older versions of Python, the `\x`*`XXXX`* escape sequence could be used to define Unicode characters. Although this is still allowed, the `\u`*`XXXX`* sequence should be used instead. In addition, the `\OOO` octal escape sequence can be used to define Unicode characters in the range [U+0000, U+01FF]. If you know the standard Unicode name for a character (consult http://www.unicode.org/charts for reference), it can be included using the special `\N{`*`character name`*`}` escape sequence. For example:

```
s = u"M\N{LATIN SMALL LETTER U WITH DIAERESIS}ller"
```

Unicode string literals should not be defined using a sequence of raw bytes that correspond to a multibyte Unicode data encoding such as UTF-8 or UTF-16. For example, writing a raw UTF-8 encoded string such as `u'M\303\274ller'` produces the seven-character Unicode sequence U+004D, U+00C3, U+00BC, U+006C, U+006C, U+0065, U+0072, which is probably not what you want. This is because in UTF-8, the multibyte sequence `\303\274` is supposed to represent the single character U+00FC, not the two characters U+00C3 and U+00BC. However, Python programs can specify a source code encoding that allows UTF-8, UTF-16, and other encoded strings to appear directly in the source code. This is described in the "Source Code Encoding" section at the end of this chapter. For more details about Unicode encodings, see Chapter 3, "Types and Objects" Chapter 4, "Operators and Expressions," and Chapter 9, "Input and Output."

Optionally, you can precede a string with an `r` or `R`, such as in `r'\n\"'`. These strings are known as *raw strings* because all their backslash characters are left intact—that is, the string literally contains the enclosed text, including the backslashes. Raw strings cannot end in a single backslash, such as `r"\"`. When raw Unicode strings are defined, `\uXXXX` escape sequences are still interpreted as Unicode characters, provided that the number of preceding \ characters is odd. For instance, `ur"\u1234"` defines a raw Unicode string with the character U+1234, whereas `ur"\\u1234"` defines a seven-character Unicode string in which the first two characters are slashes and the remaining five characters are the literal `"u1234"`. Also, when defining raw Unicode string literals the "`r`" must appear after the "`u`" as shown.

Adjacent strings (separated by whitespace or a newline) such as `"hello" 'world'` are concatenated to form a single string: `"helloworld"`. String concatenation works with any mix of ordinary, raw, and Unicode strings. However, whenever one of the strings is Unicode, the final result is always coerced to Unicode. Therefore, `"hello" u"world"` is the same as `u"hello" + u"world"`. In addition, due to subtle implementation aspects of Unicode, writing `"s1" u"s2"` may produce a result that's different from writing `u"s1s2"`. The details of this coercion process are described further in Chapter 4.

If Python is run with the `-U` command-line option, all string literals are interpreted as Unicode.

Values enclosed in square brackets `[...]`, parentheses `(...)`, and braces `{...}` denote lists, tuples, and dictionaries, respectively, as in the following example:

```
a = [ 1, 3.4, 'hello' ]          # A list
b = ( 10, 20, 30 )               # A tuple
c = { 'a': 3, 'b':42 }           # A dictionary
```

Operators, Delimiters, and Special Symbols

The following operators are recognized:

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

The following tokens serve as delimiters for expressions, lists, dictionaries, and various parts of a statement:

```
(    )    [    ]    {    }    ,    :    .    `    =    ;
```

For example, the equal (=) character serves as a delimiter between the name and value of an assignment, whereas the comma (`,`) character is used to delimit arguments to a function, elements in lists and tuples, and so on. The period (`.`) is also used in floating-point numbers and in the ellipsis (`...`) used in extended slicing operations.

Finally, the following special symbols are also used:

```
'    "    #    \    @
```

The characters `$` and `?` have no meaning in Python and cannot appear in a program except inside a quoted string literal.

Documentation Strings

If the first statement of a module, class, or function definition is a string, that string becomes a documentation string for the associated object, as in the following example:

```
def fact(n):
    "This function computes a factorial"
    if (n <= 1): return 1
    else: return n*fact(n-1)
```

Code-browsing and documentation-generation tools sometimes use documentation strings. The strings are accessible in the `__doc__` attribute of an object, as shown here:

```
>>> print fact.__doc__
This function computes a factorial
>>>
```

The indentation of the documentation string must be consistent with all the other statements in a definition.

Decorators

Any function or method may be preceded by a special symbol known as a decorator, the purpose of which is to modify the behavior of the definition that follows. Decorators are denoted with the `@` symbol and must be placed on a separate line immediately before the corresponding function or method. For example:

```
class Foo(object):
    @staticmethod
    def bar():
         pass
```

More than one decorator can be used, but each one must be on a separate line. For example:

```
@foo
@bar
def spam():
    pass
```

More information about decorators can be found in Chapter 6, "Functions and Functional Programming," and Chapter 7, "Classes and Object-Oriented Programming."

Source Code Encoding

Python source programs are normally written in standard 7-bit ASCII. However, users working in Unicode environments may find this awkward—especially if they must write a lot of string literals.

It is possible to write Python source code in a different encoding by including a special comment in the first or second line of a Python program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

name = u'M\303\274ller'  # String in quotes is directly encoded in UTF-8.
```

When the special `coding:` comment is supplied, Unicode string literals may be specified directly in the specified encoding (using a Unicode-aware editor program for instance). However, other elements of Python, including identifier names and reserved words, are still restricted to ASCII characters.

This page intentionally left blank

3

Types and Objects

ALL THE DATA STORED IN A PYTHON program is built around the concept of an *object*. Objects include fundamental data types such as numbers, strings, lists, and dictionaries. It's also possible to create user-defined objects in the form of classes or extension types. This chapter describes the Python object model and provides an overview of the built-in data types. Chapter 4, "Operators and Expressions," further describes operators and expressions.

Terminology

Every piece of data stored in a program is an object. Each object has an identity, a type, and a value.

For example, when you write `a = 42`, an integer object is created with the value of `42`. You can view the *identity* of an object as a pointer to its location in memory. `a` is a name that refers to this specific location.

The *type* of an object (which is itself a special kind of object) describes the internal representation of the object as well as the methods and operations that it supports. When an object of a particular type is created, that object is sometimes called an *instance* of that type. After an object is created, its identity and type cannot be changed. If an object's value can be modified, the object is said to be *mutable*. If the value cannot be modified, the object is said to be *immutable*. An object that contains references to other objects is said to be a *container* or *collection*.

In addition to holding a value, many objects define a number of data attributes and methods. An *attribute* is a property or value associated with an object. A *method* is a function that performs some sort of operation on an object when the method is invoked. Attributes and methods are accessed using the dot (`.`) operator, as shown in the following example:

```
a = 3 + 4j          # Create a complex number
r = a.real          # Get the real part (an attribute)

b = [1, 2, 3]       # Create a list
b.append(7)         # Add a new element using the append method
```

Object Identity and Type

The built-in function `id()` returns the identity of an object as an integer. This integer usually corresponds to the object's location in memory, although this is specific to the

Python implementation and the platform being used. The `is` operator compares the identity of two objects. The built-in function `type()` returns the type of an object. For example:

```
# Compare two objects
def compare(a,b):
    print 'The identity of a is ', id(a)
    print 'The identity of b is ', id(b)
    if a is b:
        print 'a and b are the same object'
    if a == b:
        print 'a and b have the same value'
    if type(a) is type(b):
        print 'a and b have the same type'
```

The type of an object is itself an object. This type object is uniquely defined and is always the same for all instances of a given type. Therefore, the type can be compared using the `is` operator. All type objects are assigned names that can be used to perform type checking. Most of these names are built-ins, such as `list`, `dict`, and `file`. For example:

```
if type(s) is list:
    print 'Is a list'

if type(f) is file:
    print 'Is a file'
```

However, some type names are only available in the `types` module. For example:

```
import types
if type(s) is types.NoneType:
    print "is None"
```

Because types can be specialized by defining classes, a better way to check types is to use the built-in `isinstance(`*`object, type`*`)` function. For example:

```
if isinstance(s,list):
    print 'Is a list'

if isinstance(f,file):
    print 'Is a file'

if isinstance(n,types.NoneType):
    print "is None"
```

The `isinstance()` function also works with user-defined classes. Therefore, it is a generic, and preferred, way to check the type of any Python object.

Reference Counting and Garbage Collection

All objects are reference-counted. An object's reference count is increased whenever it's assigned to a new name or placed in a container such as a list, tuple, or dictionary, as shown here:

```
a = 3.4      # Creates an object '3.4'
b = a        # Increases reference count on '3.4'
c = []
c.append(b)  # Increases reference count on '3.4'
```

This example creates a single object containing the value `3.4`. `a` is merely a name that refers to the newly created object. When `b` is assigned `a`, `b` becomes a new name for the same object, and the object's reference count increases. Likewise, when you place `b` into a list, the object's reference count increases again. Throughout the example, only one object contains `3.4`. All other operations are simply creating new references to the object.

An object's reference count is decreased by the `del` statement or whenever a reference goes out of scope (or is reassigned). For example:

```
del a      # Decrease reference count of 3.4
b = 7.8    # Decrease reference count of 3.4
c[0]=2.0   # Decrease reference count of 3.4
```

When an object's reference count reaches zero, it is garbage-collected. However, in some cases a circular dependency may exist among a collection of objects that are no longer in use. For example:

```
a = { }
b = { }
a['b'] = b      # a contains reference to b
b['a'] = a      # b contains reference to a
del a
del b
```

In this example, the `del` statements decrease the reference count of `a` and `b` and destroy the names used to refer to the underlying objects. However, because each object contains a reference to the other, the reference count doesn't drop to zero and the objects remain allocated (resulting in a memory leak). To address this problem, the interpreter periodically executes a cycle-detector that searches for cycles of inaccessible objects and deletes them. The cycle-detection algorithm can be fine-tuned and controlled using functions in the `gc` module.

References and Copies

When a program makes an assignment such as `a = b`, a new reference to `b` is created. For immutable objects such as numbers and strings, this assignment effectively creates a copy of `b`. However, the behavior is quite different for mutable objects such as lists and dictionaries. For example:

```
b = [1,2,3,4]
a = b                # a is a reference to b
a[2] = -100          # Change an element in 'a'
print b              # Produces '[1, 2, -100, 4]'
```

Because `a` and `b` refer to the same object in this example, a change made to one of the variables is reflected in the other. To avoid this, you have to create a copy of an object rather than a new reference.

Two types of copy operations are applied to container objects such as lists and dictionaries: a shallow copy and a deep copy. A *shallow copy* creates a new object, but populates it with references to the items contained in the original object. For example:

```
b = [ 1, 2, [3,4] ]
a = b[:]              # Create a shallow copy of b.
a.append(100)         # Append element to a.
print b               # Produces '[1,2, [3,4]]'. b unchanged.
a[2][0] = -100        # Modify an element of a.
print b               # Produces '[1,2, [-100,4]]'.
```

In this case, a and b are separate list objects, but the elements they contain are shared. Therefore, a modification to one of the elements of a also modifies an element of b, as shown.

A *deep copy* creates a new object and recursively copies all the objects it contains. There is no built-in function to create deep copies of objects. However, the copy.deepcopy() function in the standard library can be used, as shown in the following example:

```
import copy
b = [1, 2, [3, 4] ]
a = copy.deepcopy(b)

a[2] = -100

print a     # produces [1,2, -100, 4]

print b     # produces [1,2,3,4]
```

Built-in Types

Approximately two dozen types are built into the Python interpreter and grouped into a few major categories, as shown in Table 3.1. The Type Name column in the table lists the name that can be used to check for that type using isinstance() and other type-related functions. Types include familiar objects such as numbers and sequences. Others are used during program execution and are of little practical use to most programmers. The next few sections describe the most commonly used built-in types.

Table 3.1 **Built-in Python Types**

Type Category	Type Name	Description
None	types.NoneType	The null object None
Numbers	int	Integer
	long	Arbitrary-precision integer
	float	Floating point
	complex	Complex number
	bool	Boolean (True or False)
Sequences	str	Character string
	unicode	Unicode character string
	basestring	Abstract base type for all strings
	list	List
	tuple	Tuple
	xrange	Returned by xrange()
Mapping	dict	Dictionary
Sets	set	Mutable set
	frozenset	Immutable set

Table 3.1 **Continued**

Type Category	Type Name	Description
Callable	`types.BuiltinFunctionType`	Built-in functions
	`types.BuiltinMethodType`	Built-in methods
	`type`	Type of built-in types and classes
	`object`	Ancestor of all types and classes
	`types.FunctionType`	User-defined function
	`types.InstanceType`	Class object instance
	`types.MethodType`	Bound class method
	`types.UnboundMethodType`	Unbound class method
Modules	`types.ModuleType`	Module
Classes	`object`	Ancestor of all types and classes
Types	`type`	Type of built-in types and classes
Files	`file`	File
Internal	`types.CodeType`	Byte-compiled code
	`types.FrameType`	Execution frame
	`types.GeneratorType`	Generator object
	`types.TracebackType`	Stacks traceback of an exception
	`types.SliceType`	Generated by extended slices
	`types.EllipsisType`	Used in extended slices
Classic Classes	`types.ClassType`	Old-style class definition
	`types.InstanceType`	Old-style class instance

Note that `object` and `type` appear twice in Table 3.1 because classes and types are both callable. The types listed for "Classic Classes" refer to an obsolete, but still supported object-oriented interface. More details about this can be found later in this chapter and in Chapter 7, "Classes and Object-Oriented Programming."

The `None` Type

The `None` type denotes a null object (an object with no value). Python provides exactly one null object, which is written as `None` in a program. This object is returned by functions that don't explicitly return a value. None is frequently used as the default value of optional arguments, so that the function can detect whether the caller has actually passed a value for that argument. `None` has no attributes and evaluates to `False` in Boolean expressions.

Numeric Types

Python uses five numeric types: Booleans, integers, long integers, floating-point numbers, and complex numbers. Except for Booleans, all numeric objects are signed. All numeric types are immutable.

Booleans are represented by two values: `True` and `False`. The names `True` and `False` are respectively mapped to the numerical values of 1 and 0.

Integers represent whole numbers in the range of –2147483648 to 2147483647 (the range may be larger on some machines). Internally, integers are stored as 2's complement binary values, in 32 or more bits. Long integers represent whole numbers of unlimited range (limited only by available memory). Although there are two integer types, Python tries to make the distinction seamless. Most functions and operators that expect integers work with any integer type. Moreover, if the result of a numerical operation exceeds the allowed range of integer values, the result is transparently promoted to a long integer (although in certain cases, an `OverflowError` exception may be raised instead).

Floating-point numbers are represented using the native double-precision (64-bit) representation of floating-point numbers on the machine. Normally this is IEEE 754, which provides approximately 17 digits of precision and an exponent in the range of –308 to 308. This is the same as the `double` type in C. Python doesn't support 32-bit single-precision floating-point numbers. If space and precision are an issue in your program, consider using Numerical Python (http://numpy.sourceforge.net).

Complex numbers are represented as a pair of floating-point numbers. The real and imaginary parts of a complex number *z* are available in *z*`.real` and *z*`.imag`.

Sequence Types

Sequences represent ordered sets of objects indexed by nonnegative integers and include strings, Unicode strings, lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Table 3.2 shows the operators and methods that you can apply to all sequence types. Element *i* of sequence *s* is selected using the indexing operator *s*[*i*], and subsequences are selected using the slicing operator *s*[*i*:*j*] or extended slicing operator *s*[*i*:*j*:*stride*] (these operations are described in Chapter 4). The length of any sequence is returned using the built-in `len(`*s*`)` function. You can find the minimum and maximum values of a sequence by using the built-in `min(`*s*`)` and `max(`*s*`)` functions. However, these functions only work for sequences in which the elements can be ordered (typically numbers and strings).

Table 3.3 shows the additional operators that can be applied to mutable sequences such as lists.

Table 3.2 **Operations and Methods Applicable to All Sequences**

Item	Description
s[*i*]	Returns element *i* of a sequence
s[*i*:*j*]	Returns a slice
s[*i*:*j*:*stride*]	Returns an extended slice
`len(`*s*`)`	Number of elements in *s*

Table 3.2 **Continued**

Item	Description
`min(s)`	Minimum value in *s*
`max(s)`	Maximum value in *s*

Table 3.3 **Operations Applicable to Mutable Sequences**

Item	Description
`s[i] = v`	Item assignment
`s[i:j] = t`	Slice assignment
`s[i:j:stride] = t`	Extended slice assignment
`del s[i]`	Item deletion
`del s[i:j]`	Slice deletion
`del s[i:j:stride]`	Extended slice deletion

Additionally, lists support the methods shown in Table 3.4. The built-in function `list(s)` converts any iterable type to a list. If *s* is already a list, this function constructs a new list that's a shallow copy of *s*. The `s.append(x)` method appends a new element, *x*, to the end of the list. The `s.index(x)` method searches the list for the first occurrence of *x*. If no such element is found, a `ValueError` exception is raised. Similarly, the `s.remove(x)` method removes the first occurrence of *x* from the list. The `s.extend(t)` method extends the list *s* by appending the elements in sequence *t*. The `s.sort()` method sorts the elements of a list and optionally accepts a comparison function, key function, and reverse flag. The comparison function should take two arguments and return negative, zero, or positive, depending on whether the first argument is smaller, equal to, or larger than the second argument, respectively. The key function is a function that is applied to each element prior to comparison during sorting. Specifying a key function is useful if you want to perform special kinds of sorting operations, such as sorting a list of strings, but with case insensitivity. The `s.reverse()` method reverses the order of the items in the list. Both the `sort()` and `reverse()` methods operate on the list elements in place and return `None`.

Table 3.4 **List Methods**

Method	Description
`list(s)`	Converts *s* to a list.
`s.append(x)`	Appends a new element, *x*, to the end of *s*.
`s.extend(t)`	Appends a new list, *t*, to the end of *s*.
`s.count(x)`	Counts occurrences of *x* in *s*.
`s.index(x [,start [,stop]])`	Returns the smallest *i* where `s[i] ==x`. *start* and *stop* optionally specify the starting and ending index for the search.
`s.insert(i,x)`	Inserts *x* at index *i*.

Table 3.4 **Continued**

Method	Description
`s.pop([i])`	Returns the element *i* and removes it from the list. If *i* is omitted, the last element is returned.
`s.remove(x)`	Searches for *x* and removes it from *s*.
`s.reverse()`	Reverses items of *s* in place.
`s.sort([cmpfunc [, keyf [, reverse]]])`	Sorts items of *s* in place. *cmpfunc* is a comparison function. *keyf* is a key function. *reverse* is a flag that sorts the list in reverse order.

Python provides two string object types. Standard strings are sequences of bytes containing 8-bit data. They may contain binary data and embedded NULL bytes. Unicode strings are sequences of 16-bit characters encoded in a format known as UCS-2. This allows for 65,536 unique character values. Although the latest Unicode standard supports up to 1 million unique character values, these extra characters are not supported by Python by default. Instead, they must be encoded as a special two-character (4-byte) sequence known as a *surrogate pair*—the interpretation of which is up to the application. Python does not check data for Unicode compliance or the proper use of surrogates. As an optional feature, Python may be built to store Unicode strings using 32-bit integers (UCS-4). When enabled, this allows Python to represent the entire range of Unicode values from U+000000 to U+110000. All Unicode-related functions are adjusted accordingly.

Both standard and Unicode strings support the methods shown in Table 3.5. Although these methods operate on string instances, none of these methods actually modifies the underlying string data. Thus, methods such as `s.capitalize()`, `s.center()`, and `s.expandtabs()` always return a new string as opposed to modifying the string *s*. Character tests such as `s.isalnum()` and `s.isupper()` return `True` or `False` if all the characters in the string *s* satisfy the test. Furthermore, these tests always return `False` if the length of the string is zero. The `s.find()`, `s.index()`, `s.rfind()`, and `s.rindex()` methods are used to search *s* for a substring. All these functions return an integer index to the substring in *s*. In addition, the `find()` method returns `-1` if the substring isn't found, whereas the `index()` method raises a `ValueError` exception. Many of the string methods accept optional *start* and *end* parameters, which are integer values specifying the starting and ending indices in *s*. In most cases, these values may given negative values, in which case the index is taken from the end of the string. The `s.translate()` method is used to perform character substitutions. The `s.encode()` and `s.decode()` methods are used to transform the string data to and from a specified character encoding. As input it accepts an encoding name such as `'ascii'`, `'utf-8'`, or `'utf-16'`. This method is most commonly used to convert Unicode strings into a data encoding suitable for I/O operations and is described further in Chapter 9, "Input and Output." More details about string methods can be found in the documentation for the `string` module.

Table 3.5 **String Methods**

Method	**Description**
`s.capitalize()`	Capitalizes the first character.
`s.center(width [, pad])`	Centers the string in a field of length *`width`*. *`pad`* is a padding character.
`s.count(sub [,start [,end]])`	Counts occurrences of the specified substring *`sub`*.
`s.decode([encoding [,errors]])`	Decodes a string and returns a Unicode string.
`s.encode([encoding [,errors]])`	Returns an encoded version of the string.
`s.endswith(suffix [,start [,end]])`	Checks the end of the string for a suffix.
`s.expandtabs([tabsize])`	Replaces tabs with spaces.
`s.find(sub [, start [,end]])`	Finds the first occurrence of the specified substring *`sub`*.
`s.index(sub [, start [,end]])`	Finds the first occurrence or error in the specified substring *`sub`*.
`s.isalnum()`	Checks whether all characters are alphanumeric.
`s.isalpha()`	Checks whether all characters are alphabetic.
`s.isdigit()`	Checks whether all characters are digits.
`s.islower()`	Checks whether all characters are lowercase.
`s.isspace()`	Checks whether all characters are whitespace.
`s.istitle()`	Checks whether the string is a title-cased string (first letter of each word capitalized).
`s.isupper()`	Checks whether all characters are uppercase.
`s.join(t)`	Joins the strings *`s`* and *`t`*.
`s.ljust(width [, fill])`	Left-aligns *`s`* in a string of size *`width`*.
`s.lower()`	Converts to lowercase.
`s.lstrip([chrs])`	Removes leading whitespace or characters supplied in *`chrs`*.
`s.replace(old, new [,maxreplace])`	Replaces the substring.
`s.rfind(sub [,start [,end]])`	Finds the last occurrence of a substring.
`s.rindex(sub [,start [,end]])`	Finds the last occurrence or raises an error.
`s.rjust(width [, fill])`	Right-aligns *`s`* in a string of length *`width`*.
`s.rsplit([sep [,maxsplit]])`	Splits a string from the end of the string using *`sep`* as a delimiter. *`maxsplit`* is the maximum number of splits to perform. If *`maxsplit`* is omitted, the result is identical to the `split()` method.
`s.rstrip([chrs])`	Removes trailing whitespace or characters supplied in *`chrs`*.

Table 3.5 **Continued**

Method	Description
`s.split([sep [,maxsplit]])`	Splits a string using *sep* as a delimiter. *maxsplit* is the maximum number of splits to perform.
`s.splitlines([keepends])`	Splits a string into a list of lines. If *keepends* is 1, trailing newlines are preserved.
`s.startswith(prefix [,start [,end]])`	Checks whether a string starts with *prefix*.
`s.strip([chrs])`	Removes leading and trailing whitespace or characters supplied in *chrs*.
`s.swapcase()`	Converts uppercase to lowercase, and vice versa.
`s.title()`	Returns a title-cased version of the string.
`s.translate(table [,deletechars])`	Translates a string using a character translation table *table*, removing characters in *deletechars*.
`s.upper()`	Converts a string to uppercase.
`s.zill(width)`	Pads a string with zeros on the left up to the specified *width*.

Because there are two different string types, Python provides an abstract type, `basestring`, that can be used to test if an object is any kind of string. Here's an example:

```
if isinstance(s,basestring):
    print "is some kind of string"
```

The built-in function `range([i,]j [,stride])` constructs a list and populates it with integers *k* such that `i <= k < j`. The first index, *i*, and the *stride* are optional and have default values of `0` and `1`, respectively. The built-in `xrange([i,] j [,stride])` function performs a similar operation, but returns an immutable sequence of type `xrange`. Rather than storing all the values in a list, this sequence calculates its values whenever it's accessed. Consequently, it's much more memory-efficient when working with large sequences of integers. However, the `xrange` type is much more limited than its list counterpart. For example, none of the standard slicing operations are supported. This limits the utility of `xrange` to only a few applications such as iterating in simple loops. The `xrange` type provides a single method, `s.tolist()`, that converts its values to a list.

Mapping Types

A *mapping object* represents an arbitrary collection of objects that are indexed by another collection of nearly arbitrary key values. Unlike a sequence, a mapping object is unordered and can be indexed by numbers, strings, and other objects. Mappings are mutable.

Dictionaries are the only built-in mapping type and are Python's version of a hash table or associative array. You can use any immutable object as a dictionary key value (strings, numbers, tuples, and so on). Lists, dictionaries, and tuples containing mutable objects cannot be used as keys (the dictionary type requires key values to remain constant).

To select an item in a mapping object, use the key index operator `m[k]`, where *k* is a key value. If the key is not found, a `KeyError` exception is raised. The `len(m)` function returns the number of items contained in a mapping object. Table 3.6 lists the methods and operations.

Table 3.6 **Methods and Operations for Dictionaries**

Item	Description
`len(m)`	Returns the number of items in *m*.
`m[k]`	Returns the item of *m* with key *k*.
`m[k]=x`	Sets `m[k]` to *x*.
`del m[k]`	Removes `m[k]` from *m*.
`m.clear()`	Removes all items from *m*.
`m.copy()`	Makes a shallow copy of *m*.
`m.has_key(k)`	Returns `True` if *m* has key *k*; otherwise, returns `False`.
`m.items()`	Returns a list of `(key, value)` pairs.
`m.iteritems()`	Returns an iterator that produces `(key, value)` pairs.
`m.iterkeys()`	Returns an iterator that produces dictionary keys.
`m.itervalues()`	Returns an iterator that produces dictionary values.
`m.keys()`	Returns a list of key values.
`m.update(b)`	Adds all objects from *b* to *m*.
`m.values()`	Returns a list of all values in *m*.
`m.get(k [,v])`	Returns `m[k]` if found; otherwise, returns *v*.
`m.setdefault(k [, v])`	Returns `m[k]` if found; otherwise, returns *v* and sets `m[k] = v`.
`m.pop(k [,default])`	Returns `m[k]` if found and removes it from *m*; otherwise, returns *`default`* if supplied or raises `KeyError` if not.
`m.popitem()`	Removes a random `(key, value)` pair from *m* and returns it as a tuple.

The `m.clear()` method removes all items. The `m.copy()` method makes a shallow copy of the items contained in a mapping object and places them in a new mapping object. The `m.items()` method returns a list containing `(key, value)` pairs. The `m.keys()` method returns a list with all the key values, and the `m.values()` method returns a list with all the objects. The `m.update(b)` method updates the current mapping object by inserting all the `(key, value)` pairs found in the mapping object *b*. The `m.get(k [,v])` method retrieves an object, but allows for an optional default value, *v*, that's returned if no such object exists. The `m.setdefault(k [,v])` method is similar to `m.get()`, except that in addition to returning *v* if no object exists, it sets `m[k] = v`. If *v* is omitted, it defaults to `None`. The `m.pop()` method returns an item from a dictionary and removes it at the same time. The `m.popitem()` method is used to iteratively destroy the contents of a dictionary. The `m.iteritems()`, `m.iterkeys()`, and `m.itervalues()` methods return iterators that allow looping over all the dictionary items, keys, or values, respectively.

Set Types

A *set* is an unordered collection of unique items. Unlike sequences, sets provide no indexing or slicing operations. They are also unlike dictionaries in that there are no key values associated with the objects. In addition, the items placed into a set must be immutable. Two different set types are available: `set` is a mutable set, and `frozenset` is an immutable set. Both kinds of sets are created using a pair of built-in functions:

```
s = set([1,5,10,15])
f = frozenset(['a',37,'hello'])
```

Both `set()` and `frozenset()` populate the set by iterating over the supplied argument. Both kinds of sets provide the methods outlined in Table 3.7

Table 3.7 **Methods and Operations for Set Types**

Item	Description
`len(`*`s`*`)`	Return number of items in *`s`*.
`s``.copy()`	Makes a shallow copy of *`s`*.
`s``.difference(`*`t`*`)`	Set difference. Returns all the items in *`s`*, but not in *`t`*.
`s``.intersection(`*`t`*`)`	Intersection. Returns all the items that are both in *`s`* and in *`t`*.
`s``.issubbset(`*`t`*`)`	Returns `True` if *`s`* is a subset of *`t`*.
`s``.issuperset(`*`t`*`)`	Returns `True` if *`s`* is a superset of *`t`*.
`s``.symmetric_difference(`*`t`*`)`	Symmetric difference. Returns all the items that are in *`s`* or *`t`*, but not in both sets.
`s``.union(`*`t`*`)`	Union. Returns all items in *`s`* or *`t`*.

The *`s`*`.difference(`*`t`*`)`, *`s`*`.intersection(`*`t`*`)`, *`s`*`.symmetric_difference(`*`t`*`)`, and *`s`*`.union(`*`t`*`)` methods provide the standard mathematical operations on sets. The returned value has the same type as *`s`* (`set` or `frozenset`). The parameter *`t`* can be any Python object that supports iteration. This includes sets, lists, tuples, and strings. These set operations are also available as mathematical operators, as described further in Chapter 4.

Mutable sets (`set`) additionally provide the methods outlined in Table 3.8.

Table 3.8 **Methods for Mutable Set Types**

Item	Description
`s``.add(`*`item`*`)`	Adds *`item`* to *`s`*. Has no effect if *`item`* is already in *`s`*.
`s``.clear()`	Removes all items from *`s`*.
`s``.difference_update(`*`t`*`)`	Removes all the items from *`s`* that are also in *`t`*.
`s``.discard(`*`item`*`)`	Removes *`item`* from *`s`*. If *`item`* is not a member of *`s`*, nothing happens.

Table 3.8 **Continued**

Item	Description
`s.intersection_update(t)`	Computes the intersection of *s* and *t* and leaves the result in *s*.
`s.pop()`	Returns an arbitrary set element and removes it from *s*.
`s.remove(item)`	Removes *item* from *s*. If *item* is not a member, `KeyError` is raised.
`s.symmetric_difference_update(t)`	Computes the symmetric difference of *s* and *t* and leaves the result in *s*.
`s.update(t)`	Adds all the items in *t* to *s*. *t* may be another set, a sequence, or any object that supports iteration.

All these operations modify the set *s* in place. The parameter *t* can be any object that supports iteration.

Callable Types

Callable types represent objects that support the function call operation. There are several flavors of objects with this property, including user-defined functions, built-in functions, instance methods, and classes.

User-defined functions are callable objects created at the module level by using the `def` statement, at the class level by defining a static method, or with the `lambda` operator. Here's an example:

```
def foo(x,y):
    return x+y

class A(object):
    @staticmethod
    def foo(x,y):
        return x+y

bar = lambda x,y: x + y
```

A user-defined function *f* has the following attributes:

Attribute(s)	Description
`f.__doc__` or `f.func_doc`	Documentation string
`f.__name__` or `f.func_name`	Function name
`f.__dict__` or `f.func_dict`	Dictionary containing function attributes
`f.func_code`	Byte-compiled code
`f.func_defaults`	Tuple containing the default arguments
`f.func_globals`	Dictionary defining the global namespace
`f.func_closure`	Tuple containing data related to nested scopes

Methods are functions that operate only on instances of an object. Two types of methods—instance methods and class methods—are defined inside a class definition, as shown here:

```
class Foo(object):
          def __init__(self):
            self.items = [ ]
          def update(self, x):
            self.items.append(x)
          @classmethod
          def whatami(cls):
            return cls
```

An instance method is a method that operates on an instance of an object. The instance is passed to the method as the first argument, which is called `self` by convention. Here's an example:

```
f = Foo()
f.update(2)       # update() method is applied to the object f
```

A class method operates on the class itself. The class object is passed to a class method in the first argument, `cls`. Here's an example:

```
Foo.whatami()   # Operates on the class Foo
f.whatami()     # Operates on the class of f (Foo)
```

A bound method object is a method that is associated with a specific object instance. Here's an example:

```
a = f.update        # a is a method bound to f
b = Foo.whatami     # b is method bound to Foo (classmethod)
```

In this example, the objects `a` and `b` can be called just like a function. When invoked, they will automatically apply to the underlying object to which they were bound. Here's an example:

```
a(4)        # Calls f.update(4)
b()         # Calls Foo.whatami()
```

Bound and unbound methods are no more than a thin wrapper around an ordinary function object. The following attributes are defined for method objects:

Attribute	**Description**
`m.__doc__`	Documentation string
`m.__name__`	Method name
`m.im_class`	Class in which this method was defined
`m.im_func`	Function object implementing the method
`m.im_self`	Instance associated with the method (`None` if unbound)

So far, this discussion has focused on functions and methods, but class objects (described shortly) are also callable. When a class is called, a new class instance is created. In addition, if the class defines an `__init__()` method, it's called to initialize the newly created instance.

An object instance is also callable if it defines a special method, `__call__()`. If this method is defined for an instance, *x*, then `x(args)` invokes the method `x.__call__(args)`.

The final types of callable objects are built-in functions and methods, which correspond to code written in extension modules and are usually written in C or C++. The following attributes are available for built-in methods:

Attribute	**Description**
`b.__doc__`	Documentation string
`b.__name__`	Function/method name
`b.__self__`	Instance associated with the method

For built-in functions such as `len()`, `__self__` is set to `None`, indicating that the function isn't bound to any specific object. For built-in methods such as `x.append()`, where *x* is a list object, `__self__` is set to *x*.

Finally, it is important to note that all functions and methods are first-class objects in Python. That is, function and method objects can be freely used like any other type. For example, they can be passed as arguments, placed in lists and dictionaries, and so forth.

Classes and Types

When you define a class, the class definition normally produces an object of type `type`. Here's an example:

```
>>> class Foo(object):
...     pass
...
>>> type(Foo)
<type 'type'>
```

When an object instance is created, the type of the instance is the class that defined it. Here's an example:

```
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
```

More details about the object-oriented interface can be found in Chapter 7. However, there are a few attributes of types and instances that may be useful. If *t* is a type or class, then the attribute `t.__name__` contains the name of the type. The attributes `t.__bases__` contains a tuple of base classes. If *o* is an object instance, the attribute `o.__class__` contains a reference to its corresponding class and the attribute `o.__dict__` is a dictionary used to hold the object's attributes.

Modules

The *module* type is a container that holds objects loaded with the `import` statement. When the statement `import foo` appears in a program, for example, the name `foo` is assigned to the corresponding module object. Modules define a namespace that's implemented using a dictionary accessible in the attribute `__dict__`. Whenever an attribute of a module is referenced (using the dot operator), it's translated into a dictionary lookup. For example, `m.x` is equivalent to `m.__dict__["x"]`. Likewise, assignment to an attribute such as `m.x = y` is equivalent to `m.__dict__["x"] = y`. The following attributes are available:

Attribute	Description
`m.__dict__`	Dictionary associated with the module
`m.__doc__`	Module documentation string
`m.__name__`	Name of the module
`m.__file__`	File from which the module was loaded
`m.__path__`	Fully qualified package name, defined when the module object refers to a package

Files

The file object represents an open file and is returned by the built-in `open()` function (as well as a number of functions in the standard library). The methods on this type include common I/O operations such as `read()` and `write()`. However, because I/O is covered in detail in Chapter 9, readers should consult that chapter for more details.

Internal Types

A number of objects used by the interpreter are exposed to the user. These include traceback objects, code objects, frame objects, generator objects, slice objects, and the Ellipsis object. It is rarely necessary to manipulate these objects directly. However, their attributes are provided in the following sections for completeness.

Code Objects

Code objects represent raw byte-compiled executable code, or bytecode, and are typically returned by the built-in `compile()` function. Code objects are similar to functions except that they don't contain any context related to the namespace in which the code was defined, nor do code objects store information about default argument values. A code object, *c*, has the following read-only attributes:

Attribute	Description
`c.co_name`	Function name.
`c.co_argcount`	Number of positional arguments (including default values).
`c.co_nlocals`	Number of local variables used by the function.
`c.co_varnames`	Tuple containing names of local variables.
`c.co_cellvars`	Tuple containing names of variables referenced by nested functions.
`c.co_freevars`	Tuple containing names of free variables used by nested functions.
`c.co_code`	String representing raw bytecode.
`c.co_consts`	Tuple containing the literals used by the bytecode.
`c.co_names`	Tuple containing names used by the bytecode.
`c.co_filename`	Name of the file in which the code was compiled.
`c.co_firstlineno`	First line number of the function.
`c.co_lnotab`	String encoding bytecode offsets to line numbers.

Attribute	Description
`c.co_stacksize`	Required stack size (including local variables).
`c.co_flags`	Integer containing interpreter flags. Bit 2 is set if the function uses a variable number of positional arguments using `"*args"`. Bit 3 is set if the function allows arbitrary keyword arguments using `"**kwargs"`. All other bits are reserved.

Frame Objects

Frame objects are used to represent execution frames and most frequently occur in traceback objects (described next). A frame object, `f`, has the following read-only attributes:

Attribute	Description
`f.f_back`	Previous stack frame (toward the caller).
`f.f_code`	Code object being executed.
`f.f_locals`	Dictionary used for local variables.
`f.f_globals`	Dictionary used for global variables.
`f.f_builtins`	Dictionary used for built-in names.
`f.f_restricted`	Set to `1` if executing in restricted execution mode.
`f.f_lineno`	Line number.
`f.f_lasti`	Current instruction. This is an index into the bytecode string of `f_code`.

The following attributes can be modified (and are used by debuggers and other tools):

Attribute	Description
`f.f_trace`	Function called at the start of each source code line
`f.f_exc_type`	Most recent exception type
`f.f_exc_value`	Most recent exception value
`f.f_exc_traceback`	Most recent exception traceback

Traceback Objects

Traceback objects are created when an exception occurs and contains stack trace information. When an exception handler is entered, the stack trace can be retrieved using the `sys.exc_info()` function. The following read-only attributes are available in traceback objects:

Attribute	Description
`t.tb_next`	Next level in the stack trace (toward the execution frame where the exception occurred)
`t.tb_frame`	Execution frame object of the current level
`t.tb_line`	Line number where the exception occurred
`t.tb_lasti`	Instruction being executed in the current level

Generator Objects

Generator objects are created when a generator function is invoked (see Chapter 6, "Functions and Functional Programming"). A generator function is defined whenever a function makes use of the special `yield` keyword. The generator object serves as both an iterator and a container for information about the generator function itself. The following attributes and methods are available:

Attribute	Description
g`.gi_frame`	Execution frame of the generator function.
g`.gi_running`	Integer indicating whether or not the generator function is currently running.
g`.next()`	Execute the function until the next yield statement and return the value.

Slice Objects

Slice objects are used to represent slices given in extended slice syntax, such as *a*`[`*i*`:`*j*`:`*stride*`]`, *a*`[`*i*`:`*j*`, `*n*`:`*m*`]`, or *a*`[..., `*i*`:`*j*`]`. Slice objects are also created using the built-in `slice([`*i*`,] `*j*` [,`*stride*`])` function. The following read-only attributes are available:

Attribute	Description
s`.start`	Lower bound of the slice; `None` if omitted
s`.stop`	Upper bound of the slice; `None` if omitted
s`.step`	Stride of the slice; `None` if omitted

Slice objects also provide a single method, *s*`.indices(`*length*`)`. This function takes a length and returns a tuple `(`*start*`,`*stop*`,`*stride*`)` that indicates how the slice would be applied to a sequence of that length. For example:

```
s = slice(10,20)    #  Slice object represents [10:20]
s.indices(100)      #  Returns (10,20,1) --> [10:20]
s.indices(15)       #  Returns (10,15,1) --> [10:15]
```

Ellipsis Object

The Ellipsis object is used to indicate the presence of an ellipsis (`...`) in a slice. There is a single object of this type, accessed through the built-in name `Ellipsis`. It has no attributes and evaluates as `True`. None of Python's built-in types makes use of `Ellipsis`, but it may be used in third-party applications.

Classic Classes

In versions of Python prior to version 2.2, classes and objects were implemented using an entirely different mechanism that is now deprecated. For backward compatibility, however, these classes, called *classic classes* or *old-style classes*, are still supported.

The reason that classic classes are deprecated is due to their interaction with the Python type system. Classic classes do not define new data types, nor is it possible to specialize any of the built-in types such as lists or dictionaries. To overcome this limitation, Python 2.2 unified types and classes while introducing a different implementation of user-defined classes.

A classic class is created whenever an object *does not* inherit (directly or indirectly) from `object`. For example:

```
# A modern class
class Foo(object):
     pass

# A classic class.  Note: Does not inherit from object
class Bar:
     pass
```

Classic classes are implemented using a dictionary that contains all the objects defined within the class and defines a namespace. References to class attributes such as `c.x` are translated into a dictionary lookup, `c.__dict__["x"]`. If an attribute isn't found in this dictionary, the search continues in the list of base classes. This search is depth first in the order that base classes were specified in the class definition. An attribute assignment such as `c.y = 5` always updates the `__dict__` attribute of `c`, not the dictionaries of any base class.

The following attributes are defined by class objects:

Attribute	**Description**
`c.__dict__`	Dictionary associated with the class
`c.__doc__`	Class documentation string
`c.__name__`	Name of the class
`c.__module__`	Name of the module in which the class was defined
`c.__bases__`	Tuple containing base classes

A *class instance* is an object created by calling a class object. Each instance has its own local namespace that's implemented as a dictionary. This dictionary and the associated class object have the following attributes:

Attribute	**Description**
`x.__dict__`	Dictionary associated with an instance
`x.__class__`	Class to which an instance belongs

When the attribute of an object is referenced, such as in `x.a`, the interpreter first searches in the local dictionary for `x.__dict__["a"]`. If it doesn't find the name locally, the search continues by performing a lookup on the class defined in the `__class__` attribute. If no match is found, the search continues with base classes, as described earlier. If still no match is found and the object's class defines a `__getattr__()` method, it's used to perform the lookup. The assignment of attributes such as `x.a = 4` always updates `x.__dict__`, not the dictionaries of classes or base classes.

Special Methods

All the built-in data types implement a collection of special object methods. The names of special methods are always preceded and followed by double underscores (`__`). These methods are automatically triggered by the interpreter as a program executes. For example, the operation `x + y` is mapped to an internal method, `x.__add__(y)`, and an indexing operation, `x[k]`, is mapped to `x.__getitem__(k)`. The behavior of each data type depends entirely on the set of special methods that it implements.

User-defined classes can define new objects that behave like the built-in types simply by supplying an appropriate subset of the special methods described in this section. In addition, built-in types such as lists and dictionaries can be specialized (via inheritance) by redefining some of the special methods.

Object Creation, Destruction, and Representation

The methods in Table 3.9 create, initialize, destroy, and represent objects. `__new__()` is a static method that is called to create an instance (although this method is rarely redefined). The `__init__()` method initializes the attributes of an object and is called immediately after an object has been newly created. The `__del__()` method is invoked when an object is about to be destroyed. This method is invoked only when an object is no longer in use. It's important to note that the statement `del` *`x`* only decrements an object's reference count and doesn't necessarily result in a call to this function. Further details about these methods can be found in Chapter 7.

Table 3.9 **Special Methods for Object Creation, Destruction, and Representation**

Method	Description
`__new__(`*`cls`*` [,*`*`args`*` [,**`*`kwargs`*`]])`	A static method called to create a new instance
`__init__(`*`self`*` [,*`*`args`*` [,**`*`kwargs`*`]])`	Called to initialize a new instance
`__del__(`*`self`*`)`	Called to destroy an instance
`__repr__(`*`self`*`)`	Creates a full string representation of an object
`__str__(`*`self`*`)`	Creates an informal string representation
`__cmp__(`*`self,other`*`)`	Compares two objects and returns negative, zero, or positive
`__hash__(`*`self`*`)`	Computes a 32-bit hash index
`__nonzero__(`*`self`*`)`	Returns `0` or `1` for truth-value testing
`__unicode__(self)`	Creates a Unicode string representation

The `__new__()` and `__init__()` methods are used to create and initialize new instances. When an object is created by calling `A(`*`args`*`)`, it is translated into the following steps:

```
x = A.__new__(A,args)
is isinstance(x,A): x.__init__(args)
```

The `__repr__()` and `__str__()` methods create string representations of an object. The `__repr__()` method normally returns an expression string that can be evaluated to re-create the object. This method is invoked by the built-in `repr()` function and by the backquotes operator (`` ` ``). For example:

```
a = [2,3,4,5]       # Create a list
s = repr(a)         # s = '[2, 3, 4, 5]'
                    # Note : could have also used s = `a`
b = eval(s)         # Turns s back into a list
```

If a string expression cannot be created, the convention is for `__repr__()` to return a string of the form `<...`*`message`*`...>`, as shown here:

```
f = open("foo")
a = repr(f)       # a = "<open file 'foo', mode 'r' at dc030>"
```

The `__str__()` method is called by the built-in `str()` function and by the `print` statement. It differs from `__repr__()` in that the string it returns can be more concise and informative to the user. If this method is undefined, the `__repr__()` method is invoked.

The `__cmp__(`*`self`*`,`*`other`*`)` method is used by all the comparison operators. It returns a negative number if *`self`*` < `*`other`*, zero if *`self`*` == `*`other`*, and positive if *`self`*` > `*`other`*. If this method is undefined for an object, the object will be compared by object identity. In addition, an object may define an alternative set of comparison functions for each of the relational operators. These are known as rich comparisons and are described shortly. The `__nonzero__()` method is used for truth-value testing and should return `0` or `1` (or `True` or `False`). If undefined, the `__len__()` method is invoked to determine truth.

Finally, the `__hash__()` method computes an integer hash key used in dictionary operations (the hash value can also be returned using the built-in function `hash()`). The value returned should be identical for two objects that compare as equal. Furthermore, mutable objects should not define this method; any changes to an object will alter the hash value and make it impossible to locate an object on subsequent dictionary lookups. An object should not define a `__hash__()` method without also defining `__cmp__()`.

Attribute Access

The methods in Table 3.10 read, write, and delete the attributes of an object using the dot (`.`) operator and the `del` operator, respectively.

Table 3.10 **Special Methods for Attribute Access**

Method	Description
`__getattribute__(`*`self`*`,`*`name`*`)`	Returns the attribute *`self.name`*.
`__getattr__(`*`self`*`, `*`name`*`)`	Returns the attribute *`self.name`* if not found through normal attribute lookup.
`__setattr__(`*`self`*`, `*`name`*`, `*`value`*`)`	Sets the attribute *`self.name`*` = `*`value`*. Overrides the default mechanism.
`__delattr__(`*`self`*`, `*`name`*`)`	Deletes the attribute *`self.name`*.

An example will illustrate:

```
class Foo(object):
     def __init__(self):
          self.x = 37

f = Foo()

a = f.x       # Invokes __getattribute__(f,"x")
b = f.y       # Invokes __getattribute__(f,"y") --> Not found
              # Then invokes __getattr__(f,"y")
```

```
f.x = 42        # Invokes __setattr__(f,"x",42)
f.y = 93        # Invokes __setattr__(f,"y",93)

del f.y         # Invokes __delattr__(f,"y")
```

Whenever an attribute is accessed, the `__getattribute__()` method is always invoked. If the attribute is located, it is returned. Otherwise, the `__getattr__()` method is invoked. The default behavior of `__getattr__()` is to raise an `AttributeError` exception. The `__setattr__()` method is always invoked when setting an attribute, and the `__delattr__()` method is always invoked when deleting an attribute.

A subtle aspect of attribute access concerns a special kind of attribute known as a *descriptor*. A descriptor is an object that implements one or more of the methods in Table 3.11.

Table 3.11 **Special Methods for Descriptor Attributes**

Method	Description
`__get__(`*`self`*`,`*`instance`*`,`*`owner`*`)`	Returns an attribute value or raises `AttributeError`
`__set__(`*`self`*`,`*`instance`*`,`*`value`*`)`	Sets the attribute to *`value`*
`__delete__(`*`self`*`,`*`instance`*`)`	Deletes the attribute

Essentially, a descriptor attribute knows how to compute, set, and delete its own value whenever it is accessed. Typically, it is used to provide advanced features of classes such as static methods and properties. For example:

```
class SimpleProperty(object):
    def __init__(self,fget,fset):
        self.fget = fget
        self.fset = fset
    def __get__(self,instance,cls):
        return self.fget(instance)        # Calls instance.fget()
    def __set__(self,instance,value)
        return self.fset(instance,value)  # Calls instance.fset(value)

class Circle(object):
     def __init__(self,radius):
          self.radius = radius
     def getArea(self):
          return math.pi*self.radius**2
     def setArea(self):
          self.radius = math.sqrt(area/math.pi)
     area = SimpleProperty(getArea,setArea)
```

In this example, the class `SimpleProperty` defines a descriptor in which two functions, *`fget`* and *`fset`*, are supplied by the user to get and set the value of an attribute (note that a more advanced version of this is already provided using the `property()` function described in Chapter 7). In the `Circle` class that follows, these functions are used to create a descriptor attribute called `area`. In subsequent code, the `area` attribute is accessed transparently.

```
c = Circle(10)
a = c.area          # Implicitly calls c.getArea()
c.area = 10.0       # Implicitly calls c.setArea(10.0)
```

Underneath the covers, access to the attribute `c.area` is being translated into an operation such as `Circle.__dict__['area'].__get__(c,Circle)`.

It is important to emphasize that descriptors can only be created at the class level. It is not legal to create descriptors on a per-instance basis by defining descriptor objects inside `__init__()` and other methods.

Sequence and Mapping Methods

The methods in Table 3.12 are used by objects that want to emulate sequence and mapping objects.

Table 3.12 **Methods for Sequences and Mappings**

Method	Description
`__len__(self)`	Returns the length of `self`
`__getitem__(self, key)`	Returns `self[key]`
`__setitem__(self, key, value)`	Sets `self[key] = value`
`__delitem__(self, key)`	Deletes `self[key]`
`__getslice__(self,i,j)`	Returns `self[i:j]`
`__setslice__(self,i,j,s)`	Sets `self[i:j] = s`
`__delslice__(self,i,j)`	Deletes `self[i:j]`
`__contains__(self,obj)`	Returns `True` if *obj* is in `self`; otherwise, returns False

Here's an example:

```
a = [1,2,3,4,5,6]
len(a)               # __len__(a)
x = a[2]             # __getitem__(a,2)
a[1] = 7             # __setitem__(a,1,7)
del a[2]             # __delitem__(a,2)
x = a[1:5]           # __getslice__(a,1,5)
a[1:3] = [10,11,12]  # __setslice__(a,1,3,[10,11,12])
del a[1:4]           # __delslice__(a,1,4)
```

The `__len__` method is called by the built-in `len()` function to return a nonnegative length. This function also determines truth values unless the `__nonzero__()` method has also been defined.

For manipulating individual items, the `__getitem__()` method can return an item by key value. The key can be any Python object, but is typically an integer for sequences. The `__setitem__()` method assigns a value to an element. The `__delitem__()` method is invoked whenever the `del` operation is applied to a single element.

The slicing methods support the slicing operator `s[i:j]`. The `__getslice__()` method returns a slice, which is normally the same type of sequence as the original object. The indices *i* and *j* must be integers, but their interpretation is up to the method. Missing values for *i* and *j* are replaced with `0` and `sys.maxint`, respectively. The `__setslice__()` method assigns values to a slice. Similarly, `__delslice__()` deletes all the elements in a slice.

The `__contains__()` method is used to implement the `in` operator.

In addition to implementing the methods just described, sequences and mappings implement a number of mathematical methods, including `__add__()`, `__radd__()`, `__mul__()`, and `__rmul__()` to support concatenation and sequence replication. These methods are described shortly.

Finally, Python supports an extended slicing operation that's useful for working with multidimensional data structures such as matrices and arrays. Syntactically, you specify an extended slice as follows:

```
a = m[0:100:10]          # Strided slice (stride=10)
b = m[1:10, 3:20]        # Multidimensional slice
c = m[0:100:10, 50:75:5] # Multiple dimensions with strides
m[0:5, 5:10] = n         # extended slice assignment
del m[:10, 15:]          # extended slice deletion
```

The general format for each dimension of an extended slice is `i:j[:stride]`, where *`stride`* is optional. As with ordinary slices, you can omit the starting or ending values for each part of a slice. In addition, a special object known as the `Ellipsis` and written as `...` is available to denote any number of trailing or leading dimensions in an extended slice:

```
a = m[..., 10:20]    # extended slice access with Ellipsis
m[10:20, ...] = n
```

When using extended slices, the `__getitem__()`, `__setitem__()`, and `__delitem__()` methods implement access, modification, and deletion, respectively. However, instead of an integer, the value passed to these methods is a tuple containing one or more slice objects and at most one instance of the `Ellipsis` type. For example,

```
a = m[0:10, 0:100:5, ...]
```

invokes `__getitem__()` as follows:

```
a = __getitem__(m, (slice(0,10,None), slice(0,100,5), Ellipsis))
```

Python strings, tuples, and lists currently provide some support for extended slices, which is described in Chapter 4. Special-purpose extensions to Python, especially those with a scientific flavor, may provide new types and objects with advanced support for extended slicing operations.

Iteration

If an object, *`obj`*, supports iteration, it must provide a method, *`obj`*`.__iter__()`, that returns an iterator object. The iterator object *`iter`*, in turn, must implement a single method, *`iter`*`.next()`, that returns the next object or raises `StopIteration` to signal the end of iteration. Both of these methods are used by the implementation of the `for` statement as well as other operations that implicitly perform iteration. For example, the statement `for x in s` is carried out by performing steps equivalent to the following:

```
_iter = s.__iter__()
while 1:
    try:
         x = _iter.next()
    except StopIteration:
         break
    # Do statements in body of for loop
    ...
```

Mathematical Operations

Table 3.13 lists special methods that objects must implement to emulate numbers. Mathematical operations are always evaluated from left to right; when an expression such as `x + y` appears, the interpreter tries to invoke the method `x.__add__(y)`. The special methods beginning with *r* support operations with reversed operands. These are invoked only if the left operand doesn't implement the specified operation. For example, if *x* in `x + y` doesn't support the `__add__()` method, the interpreter tries to invoke the method `y.__radd__(x)`.

Table 3.13 **Methods for Mathematical Operations**

Method	Result
`__add__(self,other)`	`self + other`
`__sub__(self,other)`	`self - other`
`__mul__(self,other)`	`self * other`
`__div__(self,other)`	`self / other`
`__truediv__(self,other)`	`self / other` (future)
`__floordiv__(self,other)`	`self // other`
`__mod__(self,other)`	`self % other`
`__divmod__(self,other)`	`divmod(self,other)`
`__pow__(self,other [,modulo])`	`self ** other`, `pow(self, other, modulo)`
`__lshift__(self,other)`	`self << other`
`__rshift__(self,other)`	`self >> other`
`__and__(self,other)`	`self & other`
`__or__(self,other)`	`self | other`
`__xor__(self,other)`	`self ^ other`
`__radd__(self,other)`	`other + self`
`__rsub__(self,other)`	`other - self`
`__rmul__(self,other)`	`other * self`
`__rdiv__(self,other)`	`other / self`
`__rtruediv__(self,other)`	`other / self` (future)
`__rfloordiv__(self,other)`	`other // self`
`__rmod__(self,other)`	`other % self`
`__rdivmod__(self,other)`	`divmod(other,self)`
`__rpow__(self,other)`	`other ** self`
`__rlshift__(self,other)`	`other << self`
`__rrshift__(self,other)`	`other >> self`
`__rand__(self,other)`	`other & self`
`__ror__(self,other)`	`other | self`
`__rxor__(self,other)`	`other ^ self`
`__iadd__(self,other)`	`self += other`

Table 3.13 **Continued**

Method	Result
`__isub__(self,other)`	`self -= other`
`__imul__(self,other)`	`self *= other`
`__idiv__(self,other)`	`self /= other`
`__itruediv__(self,other)`	`self /= other` (future)
`__ifloordiv__(self,other)`	`self //= other`
`__imod__(self,other)`	`self %= other`
`__ipow__(self,other)`	`self **= other`
`__iand__(self,other)`	`self &= other`
`__ior__(self,other)`	`self |= other`
`__ixor__(self,other)`	`self ^= other`
`__ilshift__(self,other)`	`self <<= other`
`__irshift__(self,other)`	`self >>= other`
`__neg__(self)`	`-self`
`__pos__(self)`	`+self`
`__abs__(self)`	`abs(self)`
`__invert__(self)`	`~self`
`__int__(self)`	`int(self)`
`__long__(self)`	`long(self)`
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`
`__oct__(self)`	`oct(self)`
`__hex__(self)`	`hex(self)`
`__coerce__(self,other)`	`Type coercion`

The methods `__iadd__()`, `__isub__()`, and so forth are used to support in-place arithmetic operators such as `a+=b` and `a-=b` (also known as *augmented assignment*). A distinction is made between these operators and the standard arithmetic methods because the implementation of the in-place operators might be able to provide certain customizations such as performance optimizations. For instance, if the `self` parameter is not shared, it might be possible to modify its value in place without having to allocate a newly created object for the result.

The three flavors of division operators, `__div__()`, `__truediv__()`, and `__floordiv__()`, are used to implement true division (/) and truncating division (//) operations. The separation of division into two types of operators is a relatively recent change to Python that was started in Python 2.2, but which has far-reaching effects. As of this writing, the default behavior of Python is to map the `/` operator to `__div__()`. In the future, it will be remapped to `__truediv__()`. This latter behavior can currently be enabled as an optional feature by including the statement `from __future__ import division` in a program.

The conversion methods `__int__()`, `__long__()`, `__float__()`, and `__complex__()` convert an object into one of the four built-in numerical types. The `__oct__()` and `__hex__()` methods return strings representing the octal and hexadecimal values of an object, respectively.

The `__coerce__(`*`x`*`,`*`y`*`)` method is used in conjunction with mixed-mode numerical arithmetic. This method returns either a 2-tuple containing the values of *`x`* and *`y`* converted to a common numerical type, or `NotImplemented` (or `None`) if no such conversion is possible. To evaluate the operation *`x op y`*, where *`op`* is an operation such as `+`, the following rules are applied, in order:

1. If *`x`* has a `__coerce__()` method, replace *`x`* and *`y`* with the values returned by *`x`*`.__coerce__(`*`y`*`)`. If `None` is returned, skip to step 3.
2. If *`x`* has a method `__`*`op`*`__()`, return *`x`*`.__`*`op`*`__(`*`y`*`)`. Otherwise, restore *`x`* and *`y`* to their original values and continue.
3. If *`y`* has a `__coerce__()` method, replace *`x`* and *`y`* with the values returned by *`y`*`.__coerce__(`*`x`*`)`. If `None` is returned, raise an exception.
4. If *`y`* has a method `__`*`rop`*`__()`, return *`y`*`.__`*`rop`*`__(`*`x`*`)`. Otherwise, raise an exception.

Although strings define a few arithmetic operations, the `__coerce__()` method is not used in mixed-string operations involving standard and Unicode strings.

The interpreter supports only a limited number of mixed-type operations involving the built-in types, in particular the following:

- If *`x`* is a string, *`x`* `%` *`y`* invokes the string-formatting operation, regardless of the type of `y`.
- If *`x`* is a sequence, *`x`* `+` *`y`* invokes sequence concatenation.
- If either *`x`* or *`y`* is a sequence and the other operand is an integer, *`x`* `*` *`y`* invokes sequence repetition.

Comparison Operations

Table 3.14 lists special methods that objects can implement to provide individualized versions of the relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`). These are known as rich comparisons. Each of these functions takes two arguments and is allowed to return any kind of object, including a Boolean value, a list, or any other Python type. For instance, a numerical package might use this to perform an element-wise comparison of two matrices, returning a matrix with the results. If a comparison can't be made, these functions may also raise an exception.

Table 3.14 **Methods for Comparisons**

Method	Result
`__lt__(`*`self`*`,`*`other`*`)`	*`self`* `<` *`other`*
`__le__(`*`self`*`,`*`other`*`)`	*`self`* `<=` *`other`*
`__gt__(`*`self`*`,`*`other`*`)`	*`self`* `>` *`other`*
`__ge__(`*`self`*`,`*`other`*`)`	*`self`* `>=` *`other`*
`__eq__(`*`self`*`,`*`other`*`)`	*`self`* `==` *`other`*
`__ne__(`*`self`*`,`*`other`*`)`	*`self`* `!=` *`other`*

Callable Objects

Finally, an object can emulate a function by providing the `__call__(self [,*args [, **kwargs]])` method. If an object, *x*, provides this method, it can be invoked like a function. That is, `x(arg1, arg2, ...)` invokes `x.__call__(self, arg1, arg2, ...)`.

Performance Considerations

The execution of a Python program is mostly a sequence of function calls involving the special methods described in the earlier section "Special Methods." If you find that a program runs slowly, you should first check to see if you're using the most efficient algorithm. After that, considerable performance gains can be made simply by understanding Python's object model and trying to eliminate the number of special method calls that occur during execution.

For example, you might try to minimize the number of name lookups on modules and classes. For example, consider the following code:

```
import math
d= 0.0
for i in xrange(1000000):
     d = d + math.sqrt(i)
```

In this case, each iteration of the loop involves two name lookups. First, the `math` module is located in the global namespace; then it's searched for a function object named `sqrt`. Now consider the following modification:

```
from math import sqrt
d = 0.0
for i in xrange(1000000):
     d = d + sqrt(i)
```

In this case, one name lookup is eliminated from the inner loop, resulting in a considerable speedup.

Unnecessary method calls can also be eliminated by making careful use of temporary values and avoiding unnecessary lookups in sequences and dictionaries. For example, consider the following two classes:

```
class Point(object):
    def __init__(self,x,y,z):
          self.x = x
          self.y = y
          self.z = z

class Poly(object):
    def __init__(self):
          self.pts = [ ]
    def addpoint(self,pt):
          self.pts.append(pt)
    def perimeter(self):
          d = 0.0
          self.pts.append(self.pts[0])     # Temporarily close the polygon
          for i in xrange(len(self.pts)-1):
               d2 = (self.pts[i+1].x - self.pts[i].x)**2 + \
                    (self.pts[i+1].y - self.pts[i].y)**2 + \
                    (self.pts[i+1].z - self.pts[i].z)**2
               d = d + math.sqrt(d2)
          self.pts.pop()                   # Restore original list of points
          return d
```

In the `perimeter()` method, each occurrence of `self.pts[i]` involves two special-method lookups—one involving a dictionary and another involving a sequence. You can reduce the number of lookups by rewriting the method as follows:

```
class Poly(object):
      ...
      def perimeter(self):
          d = 0.0
          pts = self.pts
          pts.append(pts[0])
          for i in xrange(len(pts)-1):
                p1 = pts[i+1]
                p2 = pts[i]
                d2 = (p1.x - p2.x)**2 + \
                     (p1.y - p2.y)**2 + \
                     (p1.z - p2.z)**2
                d = d + math.sqrt(d2)
          pts.pop()
          return d
```

Although the performance gains made by such modifications are often modest (15%–20%), an understanding of the underlying object model and the manner in which special methods are invoked can result in faster programs. Of course, if performance is extremely critical, you often can export functionality to a Python extension module written in C or C++.

This page intentionally left blank

4

Operators and Expressions

THIS CHAPTER DESCRIBES PYTHON'S BUILT-IN OPERATORS as well as the precedence rules used in the evaluation of expressions.

Operations on Numbers

The following operations can be applied to all numeric types:

Operation	**Description**
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Truncating division
`x ** y`	Power (x^y)
`x % y`	Modulo (`x mod y`)
`-x`	Unary minus
`+x`	Unary plus

The truncating division operator (also known as *floor division*) truncates the result to an integer and works with both integers and floating-point numbers. As of this writing, the true division operator (`/`) also truncates the result to an integer if the operands are integers. Therefore, `7/4` is `1`, not `1.75`. However, this behavior is scheduled to change in a future version of Python, so you will need to be careful. The modulo operator returns the remainder of the division `x // y`. For example, `7 % 4` is `3`. For floating-point numbers, the modulo operator returns the floating-point remainder of `x // y`, which is `x - (x // y) * y`. For complex numbers, the modulo (`%`) and truncating division operators (`//`) are invalid.

The following shifting and bitwise logical operators can only be applied to integers and long integers:

Operation	**Description**
`x << y`	Left shift
`x >> y`	Right shift
`x & y`	Bitwise AND

Operation	Description
`x | y`	Bitwise OR
`x ^ y`	Bitwise XOR (exclusive OR)
`~x`	Bitwise negation

The bitwise operators assume that integers are represented in a 2's complement binary representation. For long integers, the bitwise operators operate as if the sign bit is infinitely extended to the left. Some care is required if you are working with raw bit-patterns that are intended to map to native integers on the hardware. This is because Python does not truncate the bits or allow values to overflow—instead, a result is promoted to a long integer.

In addition, you can apply the following built-in functions to all the numerical types:

Function	Description
`abs(x)`	Absolute value
`divmod(x,y)`	Returns `(x // y, x % y)`
`pow(x,y [,modulo])`	Returns `(x ** y) % modulo`
`round(x, [n])`	Rounds to the nearest multiple of `10`$^{-n}$ (floating-point numbers only)

The `abs()` function returns the absolute value of a number. The `divmod()` function returns the quotient and remainder of a division operation. The `pow()` function can be used in place of the `**` operator, but also supports the ternary power-modulo function (often used in cryptographic algorithms). The `round()` function rounds a floating-point number, *x*, to the nearest multiple of 10 to the power minus *n*. If *n* is omitted, it's set to `0`. If *x* is equally close to two multiples, rounding is performed away from zero (for example, `0.5` is rounded to `1` and `-0.5` is rounded to `-1`).

When working with integers, the result of an expression is automatically promoted to a long integer if it exceeds the precision available in the integer type. In addition, the Boolean values `True` and `False` can be used anywhere in an expression and have the values `1` and `0`, respectively.

The following comparison operators have the standard mathematical interpretation and return a Boolean value of `True` for true, `False` for false:

Operation	Description
`x < y`	Less than
`x > y`	Greater than
`x == y`	Equal to
`x != y`	Not equal to (same as `<>`)
`x >= y`	Greater than or equal to
`x <= y`	Less than or equal to

Comparisons can be chained together, such as in `w < x < y < z`. Such expressions are evaluated as `w < x and x < y and y < z`. Expressions such as `x < y > z` are legal, but are likely to confuse anyone else reading the code (it's important to note that no comparison is made between *x* and *z* in such an expression). Comparisons other than equality involving complex numbers are undefined and result in a `TypeError`.

Operations involving numbers are valid only if the operands are of the same type. If the types differ, a coercion operation is performed to convert one of the types to the other, as follows:

1. If either operand is a complex number, the other operand is converted to a complex number.
2. If either operand is a floating-point number, the other is converted to a float.
3. If either operand is a long integer, the other is converted to a long integer.
4. Otherwise, both numbers must be integers and no conversion is performed.

Operations on Sequences

The following operators can be applied to sequence types, including strings, lists, and tuples:

Operation	**Description**
`s + r`	Concatenation
`s * n`, `n * s`	Makes *n* copies of *s*, where *n* is an integer
`s % d`	String formatting (strings only)
`s[i]`	Indexing
`s[i:j]`	Slicing
`s[i:j:stride]`	Extended slicing
`x in s`, `x not in s`	Membership
`for x in s:`	Iteration
`len(s)`	Length
`min(s)`	Minimum item
`max(s)`	Maximum item

The `+` operator concatenates two sequences of the same type. The `s * n` operator makes *n* copies of a sequence. However, these are shallow copies that replicate elements by reference only. For example, consider the following code:

```
a = [3,4,5]        # A list
b = [a]            # A list containing a
c = 4*b            # Make four copies of b

# Now modify a
a[0] = -7

# Look at c
print c
```

The output of this program is the following:

```
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
```

In this case, a reference to the list `a` was placed in the list `b`. When `b` was replicated, four additional references to `a` were created. Finally, when `a` was modified, this change was propagated to all the other "copies" of `a`. This behavior of sequence multiplication is often unexpected and not the intent of the programmer. One way to work around the problem is to manually construct the replicated sequence by duplicating the contents of a. For example:

```
a = [ 3, 4, 5 ]
c = [a[:] for j in range(4)]  # [:] makes a copy of a list
```

The copy module in the standard library can also be used to make copies of objects.

The indexing operator *s*[*n*] returns the *n*th object from a sequence in which *s*[0] is the first object. Negative indices can be used to fetch characters from the end of a sequence. For example, *s*[-1] returns the last item. Otherwise, attempts to access elements that are out of range result in an IndexError exception.

The slicing operator *s*[*i*:*j*] extracts a subsequence from *s* consisting of the elements with index *k*, where *i* <= *k* < *j*. Both *i* and *j* must be integers or long integers. If the starting or ending index is omitted, the beginning or end of the sequence is assumed, respectively. Negative indices are allowed and assumed to be relative to the end of the sequence. If *i* or *j* is out of range, they're assumed to refer to the beginning or end of a sequence, depending on whether their value refers to an element before the first item or after the last item, respectively.

The slicing operator may be given an optional stride, *s*[*i*:*j*:*stride*], that causes the slice to skip elements. However, the behavior is somewhat more subtle. If a stride is supplied, *i* is the starting index, *j* is the ending index, and the produced subsequence is the elements *s*[*i*], *s*[*i*+*stride*], s[*i*+2**stride*], and so forth until index *j* is reached (which is not included). The stride may also be negative. If the starting index *i* is omitted, it is set to the beginning of the sequence if *stride* is positive or the end of the sequence if *stride* is negative. If the ending index *j* is omitted, it is set to the end of the sequence if *stride* is positive or the beginning of the sequence if *stride* is negative. Here are some examples:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

b = a[::2]          # b = [0, 2, 4, 6, 8 ]
c = a[::-2]         # c = [9, 7, 5, 3, 1 ]
d = a[0:5:2]        # d = [0,2]
e = a[5:0:-2]       # e = [5,3,1]
f = a[:5:1]         # f = [0,1,2,3,4]
g = a[:5:-1]        # g = [9,8,7,6]
h = a[5::1]         # h = [5,6,7,8,9]
i = a[5::-1]        # i = [5,4,3,2,1,0]
j = a[5:0:-1]       # j = [5,4,3,2,1]
```

The *x* in *s* operator tests to see whether the object *x* is in the sequence *s* and returns True or False. Similarly, the *x* not in *s* operator tests whether *x* is not in the sequence *s*. For strings, the in and not in operators accept subtrings. For example, 'hello' in 'hello world' produces True.

The for *x* in *s* operator iterates over all the elements of a sequence and is described further in Chapter 5, "Control Flow." len(*s*) returns the number of elements in a sequence. min(*s*) and max(*s*) return the minimum and maximum values of a sequence, respectively, although the result may only make sense if the elements can be ordered with respect to the < operator (for example, it would make little sense to find the maximum value of a list of file objects).

Strings and tuples are immutable and cannot be modified after creation. Lists can be modified with the following operators:

Operation	Description
`s[i] = x`	Index assignment
`s[i:j] = r`	Slice assignment
`s[i:j:stride] = r`	Extended slice assignment
`del s[i]`	Deletes an element
`del s[i:j]`	Deletes a slice
`del s[i:j:stride]`	Deletes an extended slice

The `s[i] = x` operator changes element *i* of a list to refer to object *x*, increasing the reference count of *x*. Negative indices are relative to the end of the list and attempts to assign a value to an out-of-range index result in an `IndexError` exception. The slicing assignment operator `s[i:j] = r` replaces elements *k*, where `i <= k < j`, with elements from sequence *r*. Indices may have the same values as for slicing and are adjusted to the beginning or end of the list if they're out of range. If necessary, the sequence *s* is expanded or reduced to accommodate all the elements in *r*. Here's an example:

```
a = [1,2,3,4,5]
a[1] = 6            # a = [1,6,3,4,5]
a[2:4] = [10,11]    # a = [1,6,10,11,5]
a[3:4] = [-1,-2,-3] # a = [1,6,10,-1,-2,-3,5]
a[2:] = [0]         # a = [1,6,0]
```

Slicing assignment may be supplied with an optional stride argument. However, the behavior is somewhat more restricted in that the argument on the right side must have exactly the same number of elements as the slice that's being replaced. Here's an example:

```
a = [1,2,3,4,5]
a[1::2] = [10,11]     # a = [1,10,3,11,5]
a[1::2] = [30,40,50]  # ValueError.  Only two elements in slice on left
```

The `del s[i]` operator removes element *i* from a list and decrements its reference count. `del s[i:j]` removes all the elements in a slice. A stride may also be supplied, as in `del s[i:j:stride]`.

Sequences are compared using the operators `<`, `>`, `<=`, `>=`, `==`, and `!=`. When comparing two sequences, the first elements of each sequence are compared. If they differ, this determines the result. If they're the same, the comparison moves to the second element of each sequence. This process continues until two different elements are found or no more elements exist in either of the sequences. If the end of both sequences is reached, the sequences are considered equal. If *a* is a subsequence of *b*, then `a < b`. Strings are compared using lexicographical ordering. Each character is assigned a unique index determined by the machine's character set (such as ASCII or Unicode). A character is less than another character if its index is less.

The modulo operator (`s % d`) produces a formatted string, given a format string, *s*, and a collection of objects in a tuple or mapping object (dictionary). The string `s` may be a standard or Unicode string. The behavior of this operator is similar to the C `sprintf()` function. The format string contains two types of objects: ordinary characters (which are left unmodified) and conversion specifiers, each of which is replaced with a formatted string representing an element of the associated tuple or mapping. If *d* is a tuple, the number of conversion specifiers must exactly match the number of objects in *d*. If *d* is a mapping, each conversion specifier must be associated with a valid

key name in the mapping (using parentheses, as described shortly). Each conversion specifier
starts with the `%` character and ends with one of the conversion characters shown in Table 4.1.

Table 4.1 **String Formatting Conversions**

Character	**Output Format**
`d,i`	Decimal integer or long integer.
`u`	Unsigned integer or long integer.
`o`	Octal integer or long integer.
`x`	Hexadecimal integer or long integer.
`X`	Hexadecimal integer (uppercase letters).
`f`	Floating point as `[-]`*`m.dddddd`*.
`e`	Floating point as `[-]`*`m.dddddde±xx`*.
`E`	Floating point as `[-]`*`m.ddddddE±xx`*.
`g,G`	Use `%e` or `%E` for exponents less than –4 or greater than the precision; otherwise use `%f`.
`s`	String or any object. The formatting code uses `str()` to generate strings.
`r`	Produces the same string as produced by `repr()`.
`c`	Single character.
`%`	Literal `%`.

Between the `%` character and the conversion character, the following modifiers may appear, in this order:

1. A key name in parentheses, which selects a specific item out of the mapping object. If no such element exists, a `KeyError` exception is raised.
2. One or more of the following:
   - `-` sign, indicating left alignment. By default, values are right-aligned.
   - `+` sign, indicating that the numeric sign should be included (even if positive).
   - `0`, indicating a zero fill.
3. A number specifying the minimum field width. The converted value will be printed in a field at least this wide and padded on the left (or right if the – flag is given) to make up the field width.
4. A period separating the field width from a precision.
5. A number specifying the maximum number of characters to be printed from a string, the number of digits following the decimal point in a floating-point number, or the minimum number of digits for an integer.

In addition, the asterisk (`*`) character may be used in place of a number in any width field. If present, the width will be read from the next item in the tuple.

The following code illustrates a few examples:

```
a = 42
b = 13.142783
c = "hello"
d = {'x':13, 'y':1.54321, 'z':'world'}
e = 5628398123741234L

print 'a is %d' % a            #  "a is 42"
print '%10d %f' % (a,b)        #  "        42 13.142783"
print '%+010d %E' % (a,b)      #  "+000000042 1.314278E+01"
print '%(x)-10d  %(y)0.3g' % d  #  "13         1.54"
print '%0.4s %s' % (c, d['z'])  #  "hell world"
print '%*.*f' % (5,3,b)        #  "13.143"
print 'e = %d' % e             #  "e = 5628398123741234"
```

Operations on Dictionaries

Dictionaries provide a mapping between names and objects. You can apply the following operations to dictionaries:

Operation	Description
`x = d[k]`	Indexing by key
`d[k] = x`	Assignment by key
`del d[k]`	Deletes an item by key
`len(d)`	Number of items in the dictionary

Key values can be any immutable object, such as strings, numbers, and tuples. In addition, dictionary keys can be specified as a comma-separated list of values, like this:

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

In this case, the key values represent a tuple, making the preceding assignments identical to the following:

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

Operations on Sets

The `set` and `frozenset` type support a number of common set operations:

Operation	Description
`s | t`	Union of *s* and *t*
`s & t`	Intersection of *s* and *t*
`s - t`	Set difference
`s ^ t`	Symmetric difference
`len(s)`	Number of items in the set
`max(s)`	Maximum value
`min(s)`	Minimum value

Augmented Assignment

Python provides the following set of augmented assignment operators:

Operation	Description
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`
`x >>= y`	`x = x >> y`
`x <<= y`	`x = x << y`

These operators can be used anywhere that ordinary assignment is used. For example:

```
a = 3
b = [1,2]
c = "Hello %s %s"
a += 1                         # a = 4
b[1] += 10                     # b = [1, 12]
c %= ("Monty", "Python")       # c = "Hello Monty Python"
```

Augmented assignment doesn't violate mutability or perform in-place modification of objects. Therefore, writing `x += y` creates an entirely new object `x` with the value `x + y`. User-defined classes can redefine the augmented assignment operators using the special methods described in Chapter 3, "Types and Objects."

The Attribute (.) Operator

The dot (`.`) operator is used to access the attributes of an object. For example:

```
foo.x = 3
print foo.y
a = foo.bar(3,4,5)
```

More than one dot operator can appear in a single expression, such as in `foo.y.a.b`. The dot operator can also be applied to the intermediate results of functions, as in `a = foo.bar(3,4,5).spam`.

Type Conversion

Sometimes it's necessary to perform conversions between the built-in types. To convert between types you simply use the type name as a function. In addition, several built-in functions are supplied to perform special kinds of conversions. All of these functions return a new object representing the converted value.

Function	**Description**
`int(x [,base])`	Converts *x* to an integer. *base* specifies the base if *x* is a string.
`long(x [,base] )`	Converts *x* to a long integer. *base* specifies the base if *x* is a string.
`float(x)`	Converts *x* to a floating-point number.
`complex(real [,imag])`	Creates a complex number.
`str(x)`	Converts object *x* to a string representation.
`repr(x)`	Converts object *x* to an expression string.
`eval(str)`	Evaluates a string and returns an object.
`tuple(s)`	Converts *s* to a tuple.
`list(s)`	Converts *s* to a list.
`set(s)`	Converts *s* to a set.
`dict(d)`	Creates a dictionary. *d* must be a sequence of `(key,value)` tuples.
`frozenset(s)`	Converts *s* to a frozen set.
`chr(x)`	Converts an integer to a character.
`unichr(x)`	Converts an integer to a Unicode character.
`ord(x)`	Converts a single character to its integer value.
`hex(x)`	Converts an integer to a hexadecimal string.
`oct(x)`	Converts an integer to an octal string.

You also can write the `repr(x)` function using backquotes as `` `x` ``. Note that the `str()` and `repr()` functions may return different results. `repr()` typically creates an expression string that can be evaluated with `eval()` to re-create the object. On the other hand, `str()` produces a concise or nicely formatted representation of the object (and is used by the `print` statement). The `ord()` function returns the integer ordinal value for a standard or Unicode character. The `chr()` and `unichr()` functions convert integers back into standard or Unicode characters, respectively.

To convert strings back into numbers and other objects, use the `int()`, `long()`, and `float()` functions. The `eval()` function can also convert a string containing a valid expression to an object. Here's an example:

```
a = int("34")             # a = 34
b = long("0xfe76214", 16) # b = 266822164L (0xfe76214L)
b = float("3.1415926")    # b = 3.1415926
c = eval("3, 5, 6")       # c = (3,5,6)
```

In functions that create containers (`list()`, `tuple()`, `set()`, and so on), the argument may be any object that supports iteration that is used to generate all the items used to populate the object that's being created.

Unicode Strings

The use of standard strings and Unicode strings in the same program presents a number of subtle complications. This is because such strings may be used in a variety of operations, including string concatenation, comparisons, dictionary key lookups, and as arguments to built-in functions.

To convert a standard string, `s`, to a Unicode string, the built-in `unicode(s [, encoding [,errors]])` function is used. To convert a Unicode string, `u`, to a standard string, the string method `u.encode([encoding [, errors]])` is used. Both of these conversion operators require the use of a special encoding rule that specifies how 16-bit Unicode character values are mapped to a sequence of 8-bit characters in standard strings, and vice versa. The encoding parameter is specified as a string and is one of the following values:

Value	**Description**
`'ascii'`	7-bit ASCII
`'latin-1'` or `'iso-8859-1'`	ISO 8859-1 Latin-1
`'utf-8'`	8-bit variable-length encoding
`'utf-16'`	16-bit variable-length encoding (may be little or big endian)
`'utf-16-le'`	UTF-16, little endian encoding
`'utf-16-be'`	UTF-16, big endian encoding
`'unicode-escape'`	Same format as Unicode literals `u"string"`
`'raw-unicode-escape'`	Same format as raw Unicode literals `ur"string"`

The default encoding is set in the `site` module and can be queried using `sys.getdefaultencoding()`. In most cases, the default encoding is `'ascii'`, which means that ASCII characters with values in the range [0x00,0x7f] are directly mapped to Unicode characters in the range [U+0000, U+007F]. Details about the other encodings can be found in Chapter 9, "Input and Output."

When string values are being converted, a `UnicodeError` exception may be raised if a character that can't be converted is encountered. For instance, if the encoding rule is `'ascii'`, a Unicode character such as U+1F28 can't be converted because its value is too large. Similarly, the string `"\xfc"` can't be converted to Unicode because it contains a character outside the range of valid ASCII character values. The `errors` parameter determines how encoding errors are handled. It's a string with one of the following values:

Value	**Description**
`'strict'`	Raises a `UnicodeError` exception for decoding errors.
`'ignore'`	Ignores invalid characters.
`'replace'`	Replaces invalid characters with a replacement character (`U+FFFD` in Unicode, `'?'` in standard strings).
`'backslashreplace'`	Replaces invalid characters with a Python character escape sequence. For example, the character U+1234 is replaced by `'\u1234'`.
`'xmlcharrefreplace'`	Replaces invalid characters with an XML character reference. For example, the character U+1234 is replaced by `'&#4660;'`.

The default error handling is `'strict'`.

When standard strings and Unicode strings are mixed in an expression, standard strings are automatically coerced to Unicode using the built-in `unicode()` function. For example:

```
s = "hello"
t = u"world"
w = s + t          # w = unicode(s) + t
```

When Unicode strings are used in string methods that return new strings (as described in Chapter 3), the result is always coerced to Unicode. Here's an example:

```
a = "Hello World"
b = a.replace("World", u"Bob")  # Produces u"Hello Bob"
```

Furthermore, even if zero replacements are made and the result is identical to the original string, the final result is still a Unicode string.

If a Unicode string is used as the format string with the `%` operator, all the arguments are first coerced to Unicode and then put together according to the given format rules. If a Unicode object is passed as one of the arguments to the `%` operator, the entire result is coerced to Unicode at the point at which the Unicode object is expanded. For example:

```
c = "%s %s" % ("Hello", u"World") # c = "Hello " + u"World"
d = u"%s %s" % ("Hello", "World") # d = u"Hello " + u"World"
```

When applied to Unicode strings, the `str()` and `repr()` functions automatically coerce the value back to a standard string. For Unicode string *u*, `str(`*u*`)` produces the value *u*`.encode()` and `repr(`*u*`)` produces `u"%s" % repr(`*u*`.encode('unicode-escape'))`.

In addition, most library and built-in functions that only operate with standard strings will automatically coerce Unicode strings to a standard string using the default encoding. If such a coercion is not possible, a `UnicodeError` exception is raised.

Standard and Unicode strings can be compared. In this case, standard strings are coerced to Unicode using the default encoding before any comparison is made. This coercion also occurs whenever comparisons are made during list and dictionary operations. For example, `'`*x*`' in [u'`*x*`', u'`*y*`', u'`*z*`']` coerces `'`*x*`'` to Unicode and returns `True`. For character containment tests such as `'W' in u'Hello World'`, the character `'W'` is coerced to Unicode before the test.

When computing hash values with the `hash()` function, standard strings and Unicode strings produce identical values, provided that the Unicode string only contains characters in the range [U+0000, U+007F]. This allows standard strings and Unicode strings to be used interchangeably as dictionary keys, provided that the Unicode strings are confined to ASCII characters. For example:

```
a = { }
a[u"foo"] = 1234
print a["foo"]        # Prints 1234
```

However, it should be noted that this dictionary key behavior may not hold if the default encoding is ever changed to something other than `'ascii'` or if Unicode strings contain non-ASCII characters. For example, if `'utf-8'` is used as a default character encoding, it's possible to produce pathological examples in which strings compare as equal, but have different hash values. For example:

```
a = u"M\u00fcller"        # Unicode string
b = "M\303\274ller"       # utf-8 encoded version of a
print a == b              # Prints '1', true
print hash(a)==hash(b)    # Prints '0', false
```

Boolean Expressions and Truth Values

The `and`, `or`, and `not` keywords can form Boolean expressions. The behavior of these operators is as follows:

Operator	Description
`x` `or` *`y`*	If *`x`* is false, return *`y`*; otherwise, return *`x`*.
`x` `and` *`y`*	If *`x`* is false, return *`x`*; otherwise, return *`y`*.
`not` *`x`*	If *`x`* is false, return *`1`*; otherwise, return *`0`*.

When you use an expression to determine a true or false value, `True`, any nonzero number, nonempty string, list, tuple, or dictionary is taken to be true. `False`, zero, `None`, and empty lists, tuples, and dictionaries evaluate as false. Boolean expressions are evaluated from left to right and consume the right operand only if it's needed to determine the final value. For example, `a and b` evaluates `b` only if `a` is true.

Object Equality and Identity

The equality operator (*`x`* `==` *`y`*) tests the values of *`x`* and *`y`* for equality. In the case of lists and tuples, all the elements are compared and evaluated as true if they're of equal value. For dictionaries, a true value is returned only if *`x`* and *`y`* have the same set of keys and all the objects with the same key have equal values. Two sets are equal if they have the same elements, which are compared using equality (==).

The identity operators (*`x`* `is` *`y`* and *`x`* `is not` *`y`*) test two objects to see whether they refer to the same object in memory. In general, it may be the case that *`x`* `==` *`y`*, but *`x`* `is not` *`y`*.

Comparison between objects of noncompatible types, such as a file and a floating-point number, may be allowed, but the outcome is arbitrary and may not make any sense. In addition, comparison between incompatible types may result in an exception.

Order of Evaluation

Table 4.2 lists the order of operation (precedence rules) for Python operators. All operators except the power (`**`) operator are evaluated from left to right and are listed in the table from highest to lowest precedence. That is, operators listed first in the table are evaluated before operators listed later. (Note that operators included together within subsections, such as *`x`* `*` *`y`*, *`x`* `/` *`y`*, *`x`* `//` *`y`*, and *`x`* `%` *`y`*, have equal precedence.)

Table 4.2 **Order of Evaluation (Highest to Lowest)**

Operator	Name
`(...), [...], {...}`	Tuple, list, and dictionary creation
`` `...` ``	String conversion

Table 4.2 **Continued**

Operator	Name
`s[i], s[i:j]`	Indexing and slicing
`s.attr`	Attributes
`f(...)`	Function calls
`+x, -x, ~x`	Unary operators
`x ** y`	Power (right associative)
`x * y, x / y, x // y, x % y`	Multiplication, division, floor division, modulo
`x + y, x - y`	Addition, subtraction
`x << y, x >> y`	Bit-shifting
`x & y`	Bitwise and
`x ^ y`	Bitwise exclusive or
`x | y`	Bitwise or
`x < y, x <= y,` `x > y, x >= y,` `x == y, x != y` `x <> y` `x is y, x is not y` `x in s, x not in s`	Comparison, identity, and sequence membership tests
`not x`	Logical negation
`x and y`	Logical and
`x or y`	Logical or
`lambda args: expr`	Anonymous function

This page intentionally left blank

5 Control Flow

THIS CHAPTER DESCRIBES STATEMENTS RELATED TO the control flow of a program. Topics include conditionals, iteration, and exceptions.

Conditionals

The `if`, `else`, and `elif` statements control conditional code execution. The general format of a conditional statement is as follows:

```
if expression:
    statements
elif expression:
    statements
elif expression:
    statements
...
else:
    statements
```

If no action is to be taken, you can omit both the `else` and `elif` clauses of a conditional. Use the `pass` statement if no statements exist for a particular clause:

```
if expression:
    pass            # Do nothing
else:
    statements
```

Loops and Iteration

You implement loops using the `for` and `while` statements. For example:

```
while expression:
    statements

for i in s:
    statements
```

The `while` statement executes statements until the associated expression evaluates to false. The `for` statement iterates over all the elements of `s` until no more elements are available. The `for` statement works with any object that supports iteration. This obviously includes the built-in sequence types such as lists, tuples, and strings, but also any object that implements the iterator protocol.

An object, s, supports iteration if it can be used with the following code, which mirrors the implementation of the for statement:

```
it = s.__iter__()            # Get an iterator for s
while 1:
     try:
         i = it.next()        # Get next item
     except StopIteration:    # No more items
         break
     # Perform operations on i
     ...
```

If the elements used in iteration are tuples of identical size, you can use the following variation of the for statement:

```
for x,y,z in s:
    statements
```

In this case, *s* must contain or produce tuples, each with three elements. On each iteration, the contents of the variables *x*, *y*, and *z* are assigned the contents of the corresponding tuple.

When looping, it is sometimes useful to keep track of a numerical index in addition to the data values. For example:

```
i = 0
for x in s:
    print i, x
    i += 1

# An alternative
for i in range(len(s)):
    print s[i]
```

Python provides a built-in function, enumerate(), that can be used for this purpose:

```
for i,x in enumerate(s):
      print i,x
```

enumerate(*s*) creates an iterator that simply returns (0, *s*[0]), (1, *s*[1]), (2, *s*[2]), and so on.

To break out of a loop, use the break statement. For example, the following function reads lines of text from the user until an empty line of text is entered:

```
while 1:
   cmd = raw_input('Enter command > ')
   if not cmd:
      break           # No input, stop loop
   # process the command
    ...
```

To jump to the next iteration of a loop (skipping the remainder of the loop body), use the continue statement. This statement tends to be used less often, but is sometimes useful when the process of reversing a test and indenting another level would make the program too deeply nested or unnecessarily complicated. As an example, the following loop prints only the nonnegative elements of a list:

```
for a in s:
    if a < 0:
       continue       # Skip negative elements
    print a
```

The break and continue statements apply only to the innermost loop being executed. If it's necessary to break out of a deeply nested loop structure, you can use an exception. Python doesn't provide a "goto" statement.

You can also attach the else statement to loop constructs, as in the following example:

```
# while-else
while i < 10:
     do something
     i = i + 1
else:
     print 'Done'

# for-else
for a in s:
     if a == 'Foo':
          break
else:
     print 'Not found!'
```

The else clause of a loop executes only if the loop runs to completion. This either occurs immediately (if the loop wouldn't execute at all) or after the last iteration. On the other hand, if the loop is terminated early using the break statement, the else clause is skipped.

Exceptions

Exceptions indicate errors and break out of the normal control flow of a program. An exception is raised using the raise statement. The general format of the raise statement is raise *Exception* [, *value*] where *Exception* is the exception type and *value* is an optional value giving specific details about the exception. For example:

```
raise RuntimeError, "Unrecoverable Error"
```

If the raise statement is used without any arguments, the last exception generated is raised again (although this works only while handling a previously raised exception).

To catch an exception, use the try and except statements, as shown here:

```
try:
    f = open('foo')
except IOError, e:
    print "Unable to open 'foo': ", e
```

When an exception occurs, the interpreter stops executing statements in the try block and looks for an except clause that matches the exception that has occurred. If one is found, control is passed to the first statement in the except clause. After the except clause is executed, control continues with the first statement that appears after the try-except block. Otherwise, the exception is propagated up to the block of code in which the try statement appeared. This code may itself be enclosed in a try-except that can handle the exception. If an exception works its way up to the top level of a program without being caught, the interpreter aborts with an error message. If desired, uncaught exceptions can also be passed to a user-defined function, sys.excepthook(), as described in Chapter 13, "Python Runtime Service."

The optional second argument to the except statement is the name of a variable in which the argument supplied to the raise statement is placed if an exception occurs.

Exception handlers can examine this value to find out more about the cause of the exception.

Multiple exception-handling blocks are specified using multiple `except` clauses, as in the following example:

```
try:
   do something
except IOError, e:
   # Handle I/O error
   ...
except TypeError, e:
   # Handle Type error
   ...
except NameError, e:
   # Handle Name error
   ...
```

A single handler can catch multiple exception types like this:

```
try:
   do something
except (IOError, TypeError, NameError), e:
   # Handle I/O, Type, or Name errors
   ...
```

To ignore an exception, use the `pass` statement as follows:

```
try:
   do something
except IOError:
   pass              # Do nothing (oh well).
```

To catch all exceptions, omit the exception name and value:

```
try:
   do something
except:
   print 'An error occurred'
```

The `try` statement also supports an `else` clause, which must follow the last `except` clause. This code is executed if the code in the `try` block doesn't raise an exception. Here's an example:

```
try:
   f = open('foo', 'r')
except IOError:
   print 'Unable to open foo'
else:
   data = f.read()
   f.close()
```

The `finally` statement defines a cleanup action for code contained in a `try` block. For example:

```
f = open('foo','r')
try:
   # Do some stuff
   ...
finally:
   f.close()
   print "File closed regardless of what happened."
```

The `finally` clause isn't used to catch errors. Rather, it's used to provide code that must always be executed, regardless of whether an error occurs. If no exception is raised, the code in the `finally` clause is executed immediately after the code in the `try` block. If an exception occurs, control is first passed to the first statement of the `finally` clause. After this code has executed, the exception is re-raised to be caught by another exception handler. The `finally` and `except` statements cannot appear together within a single `try` statement.

Python defines the built-in exceptions listed in Table 5.1. (For specific details about these exceptions, see Chapter 11.)

Table 5.1 **Built-in Exceptions**

Exception	**Description**
`Exception`	The root of all exceptions
`SystemExit`	Generated by `sys.exit()`
`StopIteration`	Raised to stop iteration
`StandardError`	Base for all built-in exceptions
`ArithmeticError`	Base for arithmetic exceptions
`FloatingPointError`	Failure of a floating-point operation
`OverflowError`	Arithmetic overflow
`ZeroDivisionError`	Division or modulus operation with 0
`AssertionError`	Raised by the `assert` statement
`AttributeError`	Raised when an attribute name is invalid
`EnvironmentError`	Errors that occur externally to Python
`IOError`	I/O or file-related error
`OSError`	Operating system error
`EOFError`	Raised when the end of the file is reached
`ImportError`	Failure of the `import` statement
`KeyboardInterrupt`	Generated by the interrupt key (usually Ctrl+C)
`LookupError`	Indexing and key errors
`IndexError`	Out-of-range sequence offset
`KeyError`	Nonexistent dictionary key
`MemoryError`	Out of memory
`NameError`	Failure to find a local or global name
`UnboundLocalError`	Unbound local variable
`ReferenceError`	Weak reference used after referent destroyed
`RuntimeError`	A generic catchall error
`NotImplementedError`	Unimplemented feature

Table 5.1 **Continued**

Exception	Description
`SyntaxError`	Parsing error
`IndentationError`	Indentation error
`TabError`	Inconsistent tab usage (generated with `-tt` option)
`SystemError`	Nonfatal system error in the interpreter
`TypeError`	Passing an inappropriate type to an operation
`ValueError`	Invalid type
`UnicodeError`	Unicode error
`UnicodeDecodeError`	Unicode decoding error
`UnicodeEncodeError`	Unicode encoding error
`UnicodeTranslateError`	Unicode translation error

Exceptions are organized into a hierarchy as shown in the table. All the exceptions in a particular group can be caught by specifying the group name in an `except` clause. For example:

```
try:
     statements
except LookupError:     # Catch IndexError or KeyError
     statements
```

or

```
try:
     statements
except StandardError:   # Catch any built-in exception
     statements
```

Defining New Exceptions

All the built-in exceptions are defined in terms of classes. To create a new exception, create a new class definition that inherits from `exceptions.Exception`, such as the following:

```
import exceptions
# Exception class
class NetworkError(exceptions.Exception):
     def __init__(self,args=None):
         self.args = args
```

The name `args` should be used as shown. This allows the value used in the `raise` statement to be properly printed in tracebacks and other diagnostics. In other words,

```
raise NetworkError, "Cannot find host."
```

creates an instance of `NetworkError` using the following call:

```
NetworkError("Cannot find host.")
```

The object that is created will print itself as "`NetworkError: Cannot find host.`" If you use a name other than the `self.args` name or don't store the argument, this feature won't work correctly.

When an exception is raised, the optional value supplied in the `raise` statement is used as the argument to the exception's class constructor. If the constructor for an exception requires more than one argument, it can be raised in two ways:

```
import exceptions
# Exception class
class NetworkError(exceptions.Exception):
     def __init__(self,errno,msg):
         self.args = (errno, msg)
         self.errno = errno
         self.errmsg = msg

# Raises an exception (multiple arguments)
def error2():
     raise NetworkError(1, 'Host not found')

# Raises an exception (multiple arguments supplied as a tuple)
def error3():
     raise NetworkError, (1, 'Host not found')
```

Exceptions can be organized into a hierarchy using inheritance. For instance, the `NetworkError` exception defined earlier could serve as a base class for a variety of more specific errors. For example:

```
class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

def error3():
    raise HostnameError

def error4():
    raise TimeoutError

try:
    error3()
except NetworkError:
    import sys
    print sys.exc_type    # Prints exception type
```

In this case, the `except NetworkError` statement catches any exception derived from `NetworkError`. To find the specific type of error that was raised, examine the variable `sys.exc_type`. Similarly, the `sys.exc_value` variable contains the value of the last exception. Alternatively, the `sys.exc_info()` function can be used to retrieve exception information in a manner that doesn't rely on global variables and is thread-safe.

Assertions and `__debug__`

The `assert` statement can introduce debugging code into a program. The general form of `assert` is

```
assert test [, data]
```

where *`test`* is an expression that should evaluate to true or false. If *`test`* evaluates to false, `assert` raises an `AssertionError` exception with the optional *`data`* supplied to the `assert` statement. For example:

```
def write_data(file,data):
    assert file, "write_data: file is None!"
    ...
```

Assertions are not checked when Python runs in optimized mode (specified with the -O option).

In addition to assert, Python provides the built-in read-only variable __debug__, which is set to 1 unless the interpreter is running in optimized mode (specified with the -O option). Programs can examine this variable as needed—possibly running extra error-checking procedures if set.

The assert statement should not be used for code that must be executed to make the program correct, because it won't be executed if Python is run in optimized mode. In particular, it's an error to use assert to check user input. Instead, assert statements are used to check things that should always be true; if one is violated, it represents a bug in the program, not an error by the user.

For example, if the function write_data(), shown previously, were intended for use by an end user, the assert statement should be replaced by a conventional if statement and the desired error-handling.

6

Functions and Functional Programming

MOST SUBSTANTIAL PROGRAMS ARE BROKEN UP into functions for better modularity and ease of maintenance. Python makes it easy to define functions, but borrows a number of ideas from functional programming languages that simplify certain tasks. This chapter describes functions, anonymous functions, generators, and functional programming features, as well as the `eval()` and `execfile()` functions and the `exec` statement. It also describes list comprehensions, a powerful list-construction technique.

Functions

Functions are defined with the `def` statement:

```
def add(x,y):
    return x+y
```

You invoke a function by writing the function name followed by a tuple of function arguments, such as `a = add(3,4)`. The order and number of arguments must match those given in the function definition. If a mismatch exists, a `TypeError` exception is raised.

You can attach default arguments to function parameters by assigning values in the function definition. For example:

```
def foo(x,y,z = 42):
```

When a function defines a parameter with a default value, that parameter and all the parameters that follow are optional. If values are not assigned to all the optional parameters in the function definition, a `SyntaxError` exception is raised.

Default parameter values are always set to the objects that were supplied as values when the function was defined. For example:

```
a = 10
def foo(x = a):
    print x

a = 5               # Reassign 'a'.
foo()               # Prints '10' (default value not changed)
```

However, the use of mutable objects as default values may lead to unintended behavior:

```
a = [10]
def foo(x = a):
    print x
a.append(20)
foo()              # Prints '[10, 20]'
```

A function can accept a variable number of parameters if an asterisk (*) is added to the last parameter name:

```
def fprintf(file, fmt, *args):
    file.write(fmt % args)

# Use fprintf. args gets (42,"hello world", 3.45)
fprintf(out,"%d %s %f", 42, "hello world", 3.45)
```

In this case, all the remaining arguments are placed into the *args* variable as a tuple. To pass the tuple *args* to another function as if they were parameters, the `*args` syntax can be used as follows:

```
def printf(fmt, *args):
        # Call another function and pass along args
        fprintf(sys.stdout,fmt, *args)
```

You can also pass function arguments by explicitly naming each parameter and specifying a value, as follows:

```
def foo(w,x,y,z):
    print w,x,y,z

# Keyword invocation
foo(x=3, y=22, w='hello', z=[1,2])
```

With keyword arguments, the order of the parameters doesn't matter. However, unless you're using default values, you must explicitly name all the function parameters. If you omit any of the required parameters or if the name of a keyword doesn't match any of the parameter names in the function definition, a `TypeError` exception is raised.

Positional arguments and keyword arguments can appear in the same function call, provided that all the positional arguments appear first, values are provided for all non-optional arguments, and no argument value is defined more than once. For example:

```
foo('hello',3, z=[1,2], y=22)
foo(3,22, w='hello', z=[1,2])    # TypeError. Multiple values for w
```

If the last argument of a function definition begins with `**`, all the additional keyword arguments (those that don't match any of the parameter names) are placed in a dictionary and passed to the function. For example:

```
def spam(**parms):
    print "You supplied the following args:"
    for k in parms.keys():
        print "%s = %s" % (k, parms[k])

spam(x=3, a="hello", foobar=(2,3))
```

You can combine extra keyword arguments with variable-length argument lists, as long as the `**` parameter appears last:

```
# Accept variable number of positional or keyword arguments
def spam(x, *args, **keywords):
    print x, args, keywords
```

Keywords arguments can also be passed to another function using the `**keywords` syntax:

```
def callfunc(func, *args, **kwargs):
    print args
    print kwargs
    func(*args,**kwargs)
```

Finally, functions can have arbitrary attributes attached to them. For example:

```
def foo():
    print "Hello world"

foo.secure = 1
foo.private = 1
```

Function attributes are stored in a dictionary that is available as the `__dict__` attribute of a function.

The primary use of function attributes is in specialized applications such as parser generators and network applications that would like to attach additional information to a function. They may also be set by the function itself to hold information that carries through to the next invocation of the function.

Parameter Passing and Return Values

When a function is invoked, its parameters are passed by reference. If a mutable object (such as a list or dictionary) is passed to a function where it's then modified, those changes will be reflected in the caller. For example:

```
a = [1,2,3,4,5]
def foo(x):
    x[3] = -55    # Modify an element of x

foo(a)            # Pass a
print a           # Produces [1,2,3,-55,5]
```

The `return` statement returns a value from a function. If no value is specified or you omit the `return` statement, the `None` object is returned. To return multiple values, place them in a tuple:

```
def factor(a):
    d = 2
    while (d <= (a/2)):
        if ((a/d)*d == a):
              return ((a/d),d)
        d = d + 1
    return (a,1)
```

Multiple return values returned in a tuple can be assigned to individual variables:

```
x,y = factor(1243)    # Return values placed in x and y.
```

or

```
(x,y) = factor(1243)  # Alternate version. Same behavior.
```