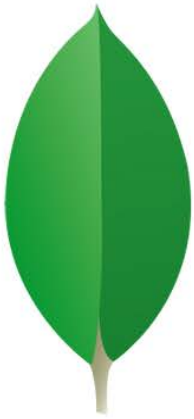
Book for programming large systems

Hero.Nguyen.1905@Gmail.com

python



mongoDB

Developer's Library

CÔNG TY CP TRUYỀN THÔNG
VÀ MÁY TÍNH HÀ NỘI

CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập – Tự do – Hạnh phúc
ଓ★ଓ
Hà Nội, ngày 07 tháng 02 năm 2012

TUYỂN DỤNG NHÂN SỰ VÀ THỰC TẬP SINH

A. TUYỂN DỤNG NHÂN SỰ

Do nhu cầu mở rộng quy mô hoạt động của công ty, chúng tôi cần tuyển thêm nhân sự, cụ thể như sau:

I. Lập trình viên PHP

Số lượng: 6

Loại hình công việc: Toàn thời gian hoặc bán thời gian

1. Công việc
   - Lập trình Web ứng dụng
   - Tối ưu và kiện toàn bảo mật, tốc độ thực thi của hệ thống
2. Yêu cầu
   - Thành thạo lập trình PHP, HTML5, CSS3, JavaScript
   - Đã có kinh nghiệm lập trình PHP qua các dự án thực tế

II. Lập trình viên Python hoặc Perl, Ruby

Số lượng: 6

Loại hình công việc: Toàn thời gian hoặc bán thời gian

1. Công việc:
   - Lập trình hệ thống với python
   - Liên quan nhiều đến thu thập dữ liệu, xử lý dữ liệu lớn, text processing…
2. Yêu cầu
   - Tư duy lập trình tốt – đặc biệt lập trình hướng đối tượng
   - Đã có kinh nghiệm lập trình python hoặc perl, ruby …

III. Lập trình viên Java

Số lượng: 6

Loại hình công việc: Toàn thời gian hoặc bán thời gian

1. Công việc:
   - Lập trình client – server
   - Phát triển hệ thống trên nền tảng sẵn có
2. Yêu cầu
   - Thành thạo 1 trong 3 nền tảng J2SE, J2EE, J2ME
   - Có kiến thức về Spring, Hibernate, ANT, Tomcat
   - Có kinh nghiệm về Oracle và/hoặc DB2 là một lợi thế

IV. Chuyên viên hệ thống viễn thông

Số lượng: 5

Loại hình công việc: Toàn thời gian hoặc bán thời gian

1. Công việc:
   - Phát triển hệ thống Unified Communication trên nền NGN, IMS
   - Triển khai hệ thống tại khách hàng
2. Yêu cầu
   - Có kiến thức về mạng tương đương CCNA
   - Có kiến thức về SIP 2.0
   - Có kiến thức về lập trình là một lợi thế

V. Yêu cầu chung

- Hiểu biết về Linux
- Có kinh nghiệm trong các dự án thực tế
- Trình bày được về một/ một vài sản phẩm do bản thân xây dựng hoặc tham gia chiếm tỷ trọng lớn.
- Đọc hiểu tốt tài liệu tiếng Anh, giao tiếp được là một lợi thế
- Năng động, sáng tạo, nhiệt tình

VI. Quyền lợi được hưởng

- Mức lương: Theo thoả thuận + Thưởng dự án
- BHXH, BHYT và các chế độ khác theo quy định

B. TUYỂN DỤNG THỰC TẬP SINH

Công ty duy trì chương trình tuyển dụng thực tập sinh về kỹ thuật bao gồm lập trình viên hệ thống và chuyên viên hệ thống mạng viễn thông.

I. Mô tả chương trình thực tập

- Hình thức thực tập theo nhóm dự án 2 – 3 người
- Các nhóm thực tập sinh được đào tạo theo chương trình với các yêu cầu cụ thể dựa trên dự án thực tiễn
- Mỗi kỳ thực tập kéo dài 3 tháng
- Dựa trên phần công việc dự án được giao, sẽ có phụ cấp cho nhóm thực tập sinh
- Các thực tập sinh xuất sắc sẽ có cơ hội vào làm việc chính thức tại công ty

II. Yêu cầu tối thiểu

- Kỹ năng tin học thành thục, tiếng Anh chuyên ngành, kết quả học tập tại trường tốt
- Có kiến thức về lập trình, cơ sở dữ liệu, nắm tốt về một trong các ngôn ngữ lập trình chính
- Tư duy kiến trúc và thiết kế tốt, cẩn mẫn, sáng tạo

C. LIÊN HỆ

Công ty CP Truyền thông và Máy tính Hà Nội
Địa chỉ: Số 90/259 phố Vọng, Quận Hai Bà Trưng, Hà Nội
Điện thoại: 04 35 400 800 – 19006638

PHP và Python, Perl, Ruby: HungNA@thoaidoanhnghiep.com – Y!M:Squall_nah – 0123 63 69 229
Java và những lĩnh vực khác: KhoaNV@thoaidoanhnghiep.com – khoa_informath - 0904 824 242

GIÁM ĐỐC

Contents

Part 1: Python

Contents at a Glance

Table of Contents

1

A Tutorial Introduction

This chapter provides a quick introduction to Python. The goal is to illustrate most of 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, generators, classes, and input/output. This chapter is not intended to provide comprehensive coverage. 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. If you are new to Python and using Python 3, you might want to follow this chapter using Python 2.6 instead. Virtually all the major concepts apply to both versions, but there are a small number of critical syntax changes in Python 3—mostly related to printing and I/O—that might break many of the examples shown in this section. Please refer to Appendix A, "Python 3," for further details.

Running Python

Python programs are executed by an interpreter. Usually, the interpreter is started by simply typing **python** into a command shell. However, there are many different implementations of the interpreter and Python development environments (for example, Jython, IronPython, IDLE, ActivePython, Wing IDE, pydev, etc.), so you should consult the documentation for startup details. 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 2.6rc2 (r26rc2:66504, Sep 19 2008, 08:50:24)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>>
```

Note

If you try the preceding example and it fails with a `SyntaxError`, you are probably using Python 3. If this is the case, you can continue to follow along with this chapter, but be aware that the `print` statement turned into a function in Python 3. Simply add parentheses around the items to be printed in the examples that follow. For instance:

```
>>> print("Hello World")
Hello World
>>>
```

Putting parentheses around the item to be printed also works in Python 2 as long as you are printing just a single item. However, it's not a syntax that you commonly see in existing Python code. In later chapters, this syntax is sometimes used in examples in which the primary focus is a feature not directly related to printing, but where the example is supposed to work with both Python 2 and 3.

Python's interactive mode is one of its most useful features. In the interactive shell, you can type any valid statement or sequence of statements and immediately view the results. Many people, including the author, even use interactive Python as their desktop calculator. For example:

```
>>> 6000 + 4523.50 + 134.12
10657.620000000001
>>> _ + 8192.32
18849.940000000002
>>>
```

When you use Python interactively, the special variable _ holds the result of the last operation. This can be useful if you want to save or use the result of the last operation in subsequent statements. However, it's important to stress that this variable is only defined when working interactively.

If you want to create a program that you can run repeatedly, put statements 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. However, be aware that the console window will disappear immediately after the program completes its execution (often before you can read its output). For debugging, it is better to run the program within a Python development tool such as IDLE.

On UNIX, you can use `#!` on the first line of the program, like this:

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

The interpreter runs statements 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 pull-down menu of a Python IDE. On UNIX, EOF is Ctrl+D; on Windows, it's Ctrl+Z. A program can request to exit by raising the `SystemExit` exception.

```
>>> 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     # Reminder: print(year, principal) in Python 3
    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 where variable names are bound to different values, possibly of varying types, during program execution. The assignment operator simply creates an association between a name and a value. Although each value has an associated type such as an integer or string, variable names are untyped and can be made to refer to any type of data during execution. This is different from C, for example, in which a name represents a fixed type, size, and location in memory into which a value is stored. 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. Although the original value of `principal` was an integer `1000`, the new value is now a floating-point number (`rate` is defined as a float, so the value of the above expression is also a float). Thus, the apparent "type" of `principal` dynamically changes from an integer to a float in the middle of the program. However, to be precise, it's not the type of `principal` that has changed, but rather the value to which the `principal` name refers.

A newline terminates each statement. However, you can use a semicolon to separate statements on the same line, 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. Because 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. However, it is most common (and generally recommended) to use four spaces per indentation level.

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. There are several ways to achieve this formatting. The most widely used approach is to use the string formatting operator (`%`) like this:

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

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 `printf()` function and is described in detail in Chapter 4, "Operators and Expressions."

A more modern approach to string formatting is to format each part individually using the `format()` function. For example:

```
print format(year,"3d"),format(principal,"0.2f")
print(format(year,"3d"),format(principal,"0.2f")) # Python 3
```

`format()` uses format specifiers that are similar to those used with the traditional string formatting operator (`%`). For example, `"3d"` formats an integer right-aligned in a column of width 3, and `"0.2f"` formats a float-point number to have two digits of accuracy. Strings also have a `format()` method that can be used to format many values at once. For example:

```
print "{0:3d} {1:0.2f}".format(year,principal)
print("{0:3d} {1:0.2f}".format(year,principal)) # Python 3
```

In this example, the number before the colon in `"{0:3d}"` and `"{1:0.2f}"` refers to the associated argument passed to the `format()` method and the part after the colon is the format specifier.

Conditionals

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

```
if a < b:
    print "Computer says Yes"
else:
    print "Computer says No"
```

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:
    print "Computer says No"
```

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

```
if product == "game" and type == "pirate memory" \
                    and not (age < 4 or age > 8):
    print "I'll take it!"
```

> **Note**
>
> Writing complex test cases commonly results in statements that involve an annoyingly long line of code. To improve readability, you can continue any statement to the next line by using a backslash (`\`) at the end of a line as shown. If you do this, the normal indentation rules don't apply to the next line, so you are free to format the continued lines as you wish.

Python does not have a special `switch` or `case` statement for testing values. To handle multiple-test cases, use the `elif` statement, like this:

```
if suffix == ".htm":
    content = "text/html"
elif suffix == ".jpg":
    content = "image/jpeg"
elif suffix == ".png":
    content = "image/png"
else:
    raise RuntimeError("Unknown content type")
```

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

```
if 'spam' in s:
    has_spam = True
else:
    has_spam = False
```

All relational operators such as `<` and `>` return `True` or `False` as results. The `in` operator used in this example is commonly used to check whether a value is contained inside of another object such as a string, list, or dictionary. It also returns `True` or `False`, so the preceding example could be shortened to this:

```
has_spam = 'spam' in s
```

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
    # print(line,end='')        # Use in Python 3
    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. The 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, etc.), it is commonly known as *iteration*. Because iteration is such a common operation, Python provides a dedicated statement, `for`, that is used to iterate over items. 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()
```

The `>>` syntax only works in Python 2. If you are using Python 3, change the `print` statement to the following:

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

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()
```

In Python 2, this code can also be shortened to the following:

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

In Python 3, the `raw_input()` function is called `input()`, but it works in exactly the same manner.

Strings

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

```
a = "Hello World"
b = 'Python is groovy'
c = """Computer says 'No'"""
```

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 stored as 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 characters 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"
```

Python never implicitly interprets the contents of a string as numerical data (i.e., as in other languages such as Perl or PHP). For example, + always concatenates strings:

```
x = "37"
y = "42"
z = x + y     # z = "3742" (String Concatenation)
```

To perform mathematical calculations, strings first have to be converted into a numeric value using a function such as `int()` or `float()`. For example:

```
z = int(x) + int(y)   # z = 79  (Integer +)
```

Non-string values can be converted into a string representation by using the `str()`, `repr()`, or `format()` function. Here's an example:

```
s = "The value of x is " + str(x)
s = "The value of x is " + repr(x)
s = "The value of x is " + format(x,"4d")
```

Although `str()` and `repr()` both create strings, their output is usually slightly different. `str()` produces the output that you get when you use the `print` statement, whereas `repr()` creates a string that you type into a program to exactly represent the value of an object. For example:

```
>>> x = 3.4
>>> str(x)
'3.4'
>>> repr(x)
'3.3999999999999999'
>>>
```

The inexact representation of 3.4 in the previous example is not a bug in Python. It is an artifact of double-precision floating-point numbers, which by their design can not exactly represent base-10 decimals on the underlying computer hardware.

The `format()` function is used to convert a value to a string with a specific formatting applied. For example:

```
>>> format(x,"0.5f")
'3.40000'
>>>
```

Lists

Lists are sequences of arbitrary objects. You create a list by enclosing values in square brackets, 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("Paula")
```

To insert an item into the middle of a list, use the `insert()` method:

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

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 [ "Thomas", "Ann", "Phil", "Paula" ]
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]
```

An empty list is created in one of two ways:

```
names = []        # An empty list
names = list()    # An empty list
```

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]
```

Items contained in nested lists are accessed by applying more than one indexing operation, 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(1)
f = open(sys.argv[1])           # Filename on the command line
lines = f.readlines()           # Read all lines into a list
f.close()

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

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

The expression `[float(line) for line in lines]` constructs a new list by looping over all the strings in the list `lines` and applying the function `float()` to each element. This particularly powerful method of constructing a list is known as a *list comprehension*. Because the lines in a file can also be read using a `for` loop, the program can be shortened by converting values using a single statement like this:

```
fvalues = [float(line) for line in open(sys.argv[1])]
```

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

To create simple data structures, you can pack a collection of values together into a single object using a *tuple*. You create a tuple by enclosing a group of values in parentheses like this:

```
stock = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
person = (first_name, last_name, phone)
```

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

```
stock = 'GOOG', 100, 490.10
address = 'www.python.org',80
person = first_name, last_name, phone
```

For completeness, 0- and 1-element tuples can be defined, but have special syntax:

```
a = ()          # 0-tuple (empty tuple)
b = (item,)     # 1-tuple (note the trailing comma)
c = item,       # 1-tuple (note the trailing comma)
```

The values in a tuple can be extracted by numerical index just like a list. However, it is more common to unpack tuples into a set of variables like this:

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

Although tuples support most of the same operations as lists (such as indexing, slicing, and concatenation), the contents of a tuple cannot be modified after creation (that is, you cannot replace, delete, or append new elements to an existing tuple). This reflects the fact that a tuple is best viewed as a single object consisting of several parts, not as a collection of distinct objects to which you might insert or remove items.

Because there is so much overlap between tuples and lists, some programmers are inclined to ignore tuples altogether and simply use lists because they seem to be more flexible. Although this works, it wastes memory if your program is going to create a large number of small lists (that is, each containing fewer than a dozen items). This is because lists slightly overallocate memory to optimize the performance of operations that add new items. Because tuples are immutable, they use a more compact representation where there is no extra space.

Tuples and lists are often used together to represent data. For example, this program shows how you might read a file consisting of different columns of data separated by commas:

```
# File containing lines of the form "name,shares,price"
filename = "portfolio.csv"
portfolio = []
for line in open(filename):
    fields = line.split(",")       # Split each line into a list
    name   = fields[0]             # Extract and convert individual fields
    shares = int(fields[1])
    price  = float(fields[2])
    stock  = (name,shares,price)   # Create a tuple (name, shares, price)
    portfolio.append(stock)        # Append to list of records
```

The `split()` method of strings splits a string into a list of fields separated by the given delimiter character. The resulting `portfolio` data structure created by this program

looks like a two-dimension array of rows and columns. Each row is represented by a tuple and can be accessed as follows:

```
>>> portfolio[0]
('GOOG', 100, 490.10)
>>> portfolio[1]
('MSFT', 50, 54.23)
>>>
```

Individual items of data can be accessed like this:

```
>>> portfolio[1][1]
50
>>> portfolio[1][2]
54.23
>>>
```

Here's an easy way to loop over all of the records and expand fields into a set of variables:

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

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 unique characters
```

Unlike lists and tuples, sets are unordered and cannot be indexed by numbers. Moreover, the elements of a set are never duplicated. For example, if you inspect the value of `t` from the preceding code, you get the following:

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

Notice that only one `'l'` appears.

Sets support a standard collection of operations, including union, intersection, difference, and symmetric difference. Here's an 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')             # Add a single item
s.update([10,37,42])   # Adds multiple items to s
```

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:

```
stock = {
        "name"   : "GOOG",
        "shares" : 100,
        "price"  : 490.10
     }
```

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

```
name  = stock["name"]
value = stock["shares"] * shares["price"]
```

Inserting or modifying objects works like this:

```
stock["shares"] = 75
stock["date"]   = "June 7, 2007"
```

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 can change.

A dictionary is a useful way to define an object that consists of named fields as shown previously. However, dictionaries are also used as a container for performing fast lookups on unordered data. For example, here's a dictionary of stock prices:

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM"  : 91.50,
    "MSFT" : 52.13
}
```

An empty dictionary is created in one of two ways:

```
prices = {}     # An empty dict
prices = dict() # An empty dict
```

Dictionary membership is tested with the `in` operator, as in the following example:

```
if "SCOX" in prices:
    p = prices["SCOX"]
else:
    p = 0.0
```

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

```
p = prices.get("SCOX",0.0)
```

To obtain a list of dictionary keys, convert a dictionary to a list:

```
syms = list(prices)         # syms = ["AAPL", "MSFT", "IBM", "GOOG"]
```

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

```
del prices["MSFT"]
```

Dictionaries are probably the most finely tuned data type in the Python interpreter. So, if you are merely trying to store and work with data in your program, you are almost always better off using a dictionary than trying to come up with some kind of custom data structure on your own.

Iteration and Looping

The most widely used looping construct is the `for` statement, which is used to iterate over a collection of items. Iteration is one of Python's richest 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 n in [1,2,3,4,5,6,7,8,9]:
    print "2 to the %d power is %d" % (n, 2**n)
```

In this example, the variable `n` will be assigned successive items from the list `[1,2,3,4,...,9]` on each iteration. Because looping over ranges of integers is quite common, the following shortcut is often used for that purpose:

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

The `range(`*i*`,`*j* `[,`*stride*`])` function creates an object that represents a range of integers with values *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. Here's an 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
```

One caution with `range()` is that in Python 2, the value it creates is a fully populated list with all of the integer values. For extremely large ranges, this can inadvertently consume all available memory. Therefore, in older Python code, you will see programmers using an alternative function `xrange()`. For example:

```
for i in xrange(100000000):    # i = 0,1,2,...,99999999
    statements
```

The object created by `xrange()` computes the values it represents on demand when lookups are requested. For this reason, it is the preferred way to represent extremely large ranges of integer values. In Python 3, the `xrange()` function has been renamed to `range()` and the functionality of the old `range()` function has been removed.

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

```
a = "Hello World"
# Print out the individual 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 = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# Print out all of the members of a dictionary
for key in c:
    print key, c[key]

# Print all of the lines in a file
f = open("foo.txt")


for line in f:
    print line,
```

The `for` loop is one of Python's most powerful language features because you can create custom iterator objects and generator functions that supply it with sequences of values. More details about iterators and generators can be found later in this chapter and 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, you can easily unpack the result into separate variables like this:

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

To assign a default value to a function 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. Here's an 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. Here's an 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:

```
count = 0
...
def foo():
    global count
    count += 1             # Changes the global variable count
```

Generators

Instead of returning a single value, a function can generate an entire sequence of results if it uses the `yield` statement. For example:

```
def countdown(n):
    print "Counting down!"
    while n > 0:
         yield n       # Generate a value (n)
         n -= 1
```

Any function that uses `yield` is known as a *generator*. Calling a generator function creates an object that produces a sequence of results through successive calls to a `next()` method (or `__next__()` in Python 3). For example:

```
>>> c = countdown(5)
>>> c.next()
Counting down!
5
>>> c.next()
4
>>> c.next()
3
>>>
```

The `next()` call makes a generator function run until it reaches the next `yield` statement. At this point, the value passed to `yield` is returned by `next()`, and the function suspends execution. The function resumes execution on the statement following `yield` when `next()` is called again. This process continues until the function returns.

Normally you would not manually call `next()` as shown. Instead, you hook it up to a `for` loop like this:

```
>>> for i in countdown(5):
...     print i,
Counting down!
5 4 3 2 1
>>>
```

Generators are an extremely powerful way of writing programs based on processing pipelines, streams, or data flow. For example, the following generator function mimics the behavior of the UNIX `tail -f` command that's commonly used to monitor log files:

```
# tail a file (like tail -f)
import time
def tail(f):
    f.seek(0,2)      # Move to EOF
    while True:
        line = f.readline()     # Try reading a new line of text
        if not line:            # If nothing, sleep briefly and try again
             time.sleep(0.1)
             continue
        yield line
```

Here's a generator that looks for a specific substring in a sequence of lines:

```
def grep(lines, searchtext):
    for line in lines:
        if searchtext in line: yield line
```

Here's an example of hooking both of these generators together to create a simple processing pipeline:

```
# A python implementation of Unix "tail -f | grep python"
wwwlog  = tail(open("access-log"))
pylines = grep(wwwlog,"python")
for line in pylines:
    print line,
```

A subtle aspect of generators is that they are often mixed together with other iterable objects such as lists or files. Specifically, when you write a statement such as `for` *`item`* `in` *`s`*, *`s`* could represent a list of items, the lines of a file, the result of a generator function, or any number of other objects that support iteration. The fact that you can just plug different objects in for *`s`* can be a powerful tool for creating extensible programs.

Coroutines

Normally, functions operate on a single set of input arguments. However, a function can also be written to operate as a task that processes a sequence of inputs sent to it. This type of function is known as a *coroutine* and is created by using the `yield` statement as an expression `(yield)` as shown in this example:

```
def print_matches(matchtext):
    print "Looking for", matchtext
    while True:
         line = (yield)       # Get a line of text
         if matchtext in line:
             print line
```

To use this function, you first call it, advance it to the first `(yield)`, and then start sending data to it using `send()`. For example:

```
>>> matcher = print_matches("python")
>>> matcher.next()     # Advance to the first (yield)
Looking for python
>>> matcher.send("Hello World")
>>> matcher.send("python is cool")
python is cool
>>> matcher.send("yow!")
>>> matcher.close()    # Done with the matcher function call
>>>
```

A coroutine is suspended until a value is sent to it using `send()`. When this happens, that value is returned by the `(yield)` expression inside the coroutine and is processed by the statements that follow. Processing continues until the next `(yield)` expression is encountered—at which point the function suspends. This continues until the coroutine function returns or `close()` is called on it as shown in the previous example.

Coroutines are useful when writing concurrent programs based on producer-consumer problems where one part of a program is producing data to be consumed by another part of the program. In this model, a coroutine represents a consumer of data. Here is an example of using generators and coroutines together:

```
# A set of matcher coroutines
matchers = [
    print_matches("python"),
    print_matches("guido"),
    print_matches("jython")
]

# Prep all of the matchers by calling next()
for m in matchers: m.next()

# Feed an active log file into all matchers.  Note for this to work,
   # a web server must be actively writing data to the log.
wwwlog  = tail(open("access-log"))
for line in wwwlog:
    for m in matchers:
        m.send(line)       # Send data into each matcher coroutine
```

Further details about coroutines can be found in Chapter 6.

Objects and Classes

All values used in a program are objects. An *object* consists of internal data and methods that perform various kinds of operations involving that data. You have already used objects and methods when working with the built-in types such as strings and lists. For example:

```
items = [37, 42]    # Create a list object
items.append(73)    # Call the append() method
```

The `dir()` function lists the methods available on an object and is a useful tool for interactive experimentation. For example:

```
>>> items = [37, 42]
>>> dir(items)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
...
 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
>>>
```

When inspecting objects, you will see familiar methods such as `append()` and `insert()` listed. However, you will also see special methods that always begin and end with a double underscore. These methods implement various language operations. For example, the `__add__()` method implements the `+` operator:

```
>>> items.__add__([73,101])
[37, 42, 73, 101]
>>>
```

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 like a function
```

In this case, `@staticmethod` declares the method that follows to be a static method. `@staticmethod` is an example of using an *decorator*, a topic that is discussed further in Chapter 6.

Exceptions

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

```
Traceback (most recent call last):
 File "foo.py", line 12, in <module>
IOError: [Errno 2] No such file or directory: 'file.txt'
```

The traceback 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 `try` and `except` statements, like this:

```
try:
    f = open("file.txt","r")
except IOError as 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 last except block. The program does not return 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("Computer says no")
```

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

Proper management of system resources such as locks, files, and network connections is often a tricky problem when combined with exception handling. To simplify such programming, you can use the with statement with certain kinds of objects. Here is an example of writing code that uses a mutex lock:

```
import threading
message_lock = threading.Lock()
...
with message_lock:
    messages.add(newmessage)
```

In this example, the message_lock object is automatically acquired when the with statement executes. When execution leaves the context of the with block, the lock is automatically released. This management takes place regardless of what happens inside the with block. For example, if an exception occurs, the lock is released when control leaves the context of the block.

The with statement is normally only compatible with objects related to system resources or the execution environment such as files, connections, and locks. However, user-defined objects can define their own custom processing. This is covered in more detail in the "Context Management Protocol" section of Chapter 3, "Types and Objects."

Modules

As your programs grow in size, you will 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 and executes all the statements in the associated .py file within that namespace. To access the contents of the namespace after import, 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 *
```

As with objects, the dir() function lists the contents of a module and is a useful tool for interactive experimentation:

```
>>> 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 system command prompt.

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.

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 clearly 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. Here's an 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 single statement, it 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. Here's an 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.

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

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.

Numeric Literals

There are four types of built-in numeric literals:

- Booleans
- Integers

- Floating-point numbers
- Complex numbers

The identifiers `True` and `False` are interpreted as Boolean values with the integer values of 1 and 0, respectively. A number such as `1234` is interpreted as a decimal integer. To specify an integer using octal, hexadecimal, or binary notation, precede the value with `0`, `0x`, or `0b`, respectively (for example, `0644`, `0x100fea8`, or `0b11101010`).

Integers in Python can have an arbitrary number of digits, so if you want to specify a really large integer, just write out all of the digits, as in `1234567890123456789`0. However, when inspecting values and looking at old Python code, you might see large numbers written with a trailing `l` (lowercase *L*) or `L` character, as in `1234567890123456789`0`L`. This trailing `L` is related to the fact that Python internally represents integers as either a fixed-precision machine integer or an arbitrary precision long integer type depending on the magnitude of the value. In older versions of Python, you could explicitly choose to use either type and would add the trailing `L` to explicitly indicate the long type. Today, this distinction is unnecessary and is actively discouraged. So, if you want a large integer value, just write it without the `L`.

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 an imaginary 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`.

String Literals

String literals are used to specify a sequence of characters and are defined by enclosing text in single (`'`), double (`"`), or triple (`'''` or `"""`) quotes. There is no semantic difference between quoting styles other than the requirement that you use the same type of quote to start and terminate a string. Single- and double-quoted strings must be defined on a single line, whereas triple-quoted strings can span multiple lines and include all of the enclosed formatting (that is, newlines, tabs, spaces, and so on). Adjacent strings (separated by white space, newline, or a line-continuation character) such as `"hello" 'world'` are concatenated to form a single string `"helloworld"`.

Within string literals, 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.

Table 2.1 **Standard Character Escape Codes**

Character	Description
`\`	Newline continuation
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	Bell
`\b`	Backspace
`\e`	Escape
`\0`	Null

Table 2.1 **Continued**

Character	Description
`\n`	Line feed
`\v`	Vertical tab
`\t`	Horizontal tab
`\r`	Carriage return
`\f`	Form feed
`\OOO`	Octal value (`\000` to `\377`)
`\u`*xxxx*	Unicode character (`\u0000` to `\uffff`)
`\U`*xxxxxxxx*	Unicode character (`\U00000000` to `\Uffffffff`)
`\N{`*charname*`}`	Unicode character name
`\x`*hh*	Hexadecimal value (`\x00` to `\xff`)

The escape codes `\OOO` and `\x` are used to embed characters into a string literal that can't be easily typed (that is, control codes, nonprinting characters, symbols, international characters, and so on). For these escape codes, you have to specify an integer value corresponding to a character value. For example, if you wanted to write a string literal for the word "Jalapeño", you might write it as `"Jalape\xf1o"` where `\xf1` is the character code for ñ.

In Python 2 string literals correspond to 8-bit character or byte-oriented data. A serious limitation of these strings is that they do not fully support international character sets and Unicode. To address this limitation, Python 2 uses a separate string type for Unicode data. To write a Unicode string literal, you prefix the first quote with the letter "u". For example:

```
s = u"Jalape\u00f1o"
```

In Python 3, this prefix character is unnecessary (and is actually a syntax error) as all strings are already Unicode. Python 2 will emulate this behavior if you run the interpreter with the `-U` option (in which case all string literals will be treated as Unicode and the `u` prefix can be omitted).

Regardless of which Python version you are using, the escape codes of `\u`, `\U`, and `\N` in Table 2.1 are used to insert arbitrary characters into a Unicode literal. Every Unicode character has an assigned *code point*, which is typically denoted in Unicode charts as `U+`*XXXX* where *XXXX* is a sequence of four or more hexadecimal digits. (Note that this notation is not Python syntax but is often used by authors when describing Unicode characters.) For example, the character ñ has a code point of `U+00F1`. The `\u` escape code is used to insert Unicode characters with code points in the range `U+0000` to `U+FFFF` (for example, `\u00f1`). The `\U` escape code is used to insert characters in the range `U+10000` and above (for example, `\U00012345`). One subtle caution concerning the `\U` escape code is that Unicode characters with code points above `U+10000` usually get decomposed into a pair of characters known as a *surrogate pair*. This has to do with the internal representation of Unicode strings and is covered in more detail in Chapter 3, "Types and Objects."

Unicode characters also have a descriptive name. If you know the name, you can use the `\N{`*character name*`}` escape sequence. For example:

```
s = u"Jalape\N{LATIN SMALL LETTER N WITH TILDE}o"
```

For an authoritative reference on code points and character names, consult http://www.unicode.org/charts.

Optionally, you can precede a string literal with an `r` or `R`, such as in `r'\d'`. 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. The main use of raw strings is to specify literals where the backslash character has some significance. Examples might include the specification of regular expression patterns with the `re` module or specifying a filename on a Windows machine (for example, `r'c:\newdata\tests'`).

Raw strings cannot end in a single backslash, such as `r"\"`. Within raw strings, `\u`*XXXX* 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 single character U+1234, whereas `ur"\\u1234"` defines a seven-character string in which the first two characters are slashes and the remaining five characters are the literal `"u1234"`. Also, in Python 2.2, the `r` must appear after the `u` in raw Unicode strings as shown. In Python 3.0, the `u` prefix is unnecessary.

String literals should not be defined using a sequence of raw bytes that correspond to a data encoding such as UTF-8 or UTF-16. For example, directly writing a raw UTF-8 encoded string such as `'Jalape\xc3\xb1o'` simply produces a nine-character string U+004A, U+0061, U+006C, U+0061, U+0070, U+0065, U+00C3, U+00B1, U+006F, which is probably not what you intended. This is because in UTF-8, the multibyte sequence `\xc3\xb1` is supposed to represent the single character U+00F1, not the two characters U+00C3 and U+00B1. To specify an encoded byte string as a literal, prefix the first quote with a `"b"` as in `b"Jalape\xc3\xb1o"`. When defined, this literally creates a string of single bytes. From this representation, it is possible to create a normal string by decoding the value of the byte literal with its `decode()` method. More details about this are covered in Chapter 3 and Chapter 4, "Operators and Expressions."

The use of byte literals is quite rare in most programs because this syntax did not appear until Python 2.6, and in that version there is no difference between a byte literal and a normal string. In Python 3, however, byte literals are mapped to a new `bytes` datatype that behaves differently than a normal string (see Appendix A, "Python 3").

Containers

Values enclosed in square brackets `[...]`, parentheses `(...)`, and braces `{...}` denote a collection of objects contained in a list, tuple, and dictionary, 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
```

List, tuple, and dictionary literals can span multiple lines without using the line-continuation character (\). In addition, a trailing comma is allowed on the last item. For example:

```
a = [ 1,
      3.4,
      'hello',
    ]
```

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. In addition, a documentation string cannot be computed or assigned from a variable as an expression. The documentation string always has to be a string literal enclosed in quotes.

Decorators

Function, method, or class definitions 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, method, or class. Here's an example:

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

More than one decorator can be used, but each one must be on a separate line. Here's an 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 with international characters.

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

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

s = "Jalapeño"  # String in quotes is directly encoded in UTF-8.
```

When the special `coding:` comment is supplied, string literals may be typed in directly using a Unicode-aware editor. However, other elements of Python, including identifier names and reserved words, should still be restricted to ASCII characters.

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. However, it's also possible to create user-defined objects in the form of classes. In addition, most objects related to program structure and the internal operation of the interpreter are also exposed. This chapter describes the inner workings of the Python object model and provides an overview of the built-in data types. Chapter 4, "Operators and Expressions," further describes operators and expressions. Chapter 7, "Classes and Object-Oriented Programming," describes how to create user-defined objects.

Terminology

Every piece of data stored in a program is an object. Each object has an identity, a type (which is also known as its class), 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, also known as the object's *class*, 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 instance 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*.

Most objects are characterized by a number of data attributes and methods. An *attribute* is a value associated with an object. A *method* is a function that performs some sort of operation on an object when the method is invoked as a function. 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 no such interpretation of the identity should be made. The `is` operator compares the identity of two objects. The built-in function `type()` returns the type of an object. Here's an example of different ways you might compare two objects:

```
# Compare two objects
def compare(a,b):
    if a is b:
        # a and b are the same object
        statements
    if a == b:
        # a and b have the same value
        statements
    if type(a) is type(b):
        # a and b have the same type
        statements
```

The type of an object is itself an object known as the object's class. This 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`. Here's an example:

```
if type(s) is list:
    s.append(item)

if type(d) is dict:
    d.update(t)
```

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

```
if isinstance(s,list):
    s.append(item)

if isinstance(d,dict):
    d.update(t)
```

Because the `isinstance()` function is aware of inheritance, it is the preferred way to check the type of any Python object.

Although type checks can be added to a program, type checking is often not as useful as you might imagine. For one, excessive checking severely affects performance. Second, programs don't always define objects that neatly fit into an inheritance hierarchy. For instance, if the purpose of the preceding `isinstance(s,list)` statement is to test whether `s` is "list-like," it wouldn't work with objects that had the same programming interface as a list but didn't directly inherit from the built-in `list` type. Another option for adding type-checking to a program is to define abstract base classes. This is described in Chapter 7.

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 = 37       # Creates an object with value 37
b = a        # Increases reference count on 37
c = []
c.append(b)  # Increases reference count on 37
```

This example creates a single object containing the value `37`. `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 `37`. 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). Here's an example:

```
del a      # Decrease reference count of 37
b = 42     # Decrease reference count of 37
c[0] = 2.0 # Decrease reference count of 37
```

The current reference count of an object can be obtained using the `sys.getrefcount()` function. For example:

```
>>> a = 37
>>> import sys
>>> sys.getrefcount(a)
7
>>>
```

In many cases, the reference count is much higher than you might guess. For immutable data such as numbers and strings, the interpreter aggressively shares objects between different parts of the program in order to conserve memory.

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. Here's an 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 runs periodically as the interpreter allocates more and more memory during execution. The exact behavior can be fine-tuned and controlled using functions in the `gc` module (see Chapter 13, "Python Runtime Services").

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. Here's an example:

```
>>> a = [1,2,3,4]
>>> b = a                 # b is a reference to a
>>> b is a
True
```

```
>>> b[2] = -100          # Change an element in b
>>> a                    # Notice how a also changed
[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. Here's an example:

```
>>> a = [ 1, 2, [3,4] ]
>>> b = list(a)          # Create a shallow copy of a.
>>> b is a
False
>>> b.append(100)        # Append element to b.
>>> b
[1, 2, [3, 4], 100]
>>> a                    # Notice that a is unchanged
[1, 2, [3, 4]]
>>> b[2][0] = -100       # Modify an element inside b
>>> b
[1, 2, [-100, 4], 100]
>>> a                    # Notice the change inside a
[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 operation 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
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> b
[1, 2, [-100, 4]]
>>> a                  # Notice that a is unchanged
[1, 2, [3, 4]]
>>>
```

First-Class Objects

All objects in Python are said to be "first class." This means that all objects that can be named by an identifier have equal status. It also means that all objects that can be named can be treated as data. For example, here is a simple dictionary containing two values:

```
items = {
    'number' : 42
    'text' : "Hello World"
}
```

The first-class nature of objects can be seen by adding some more unusual items to this dictionary. Here are some examples:

```
items["func"]  = abs             # Add the abs() function
import math
items["mod"]   = math            # Add a module
items["error"] = ValueError      # Add an exception type
nums = [1,2,3,4]
items["append"] = nums.append    # Add a method of another object
```

In this example, the `items` dictionary contains a function, a module, an exception, and a method of another object. If you want, you can use dictionary lookups on `items` in place of the original names and the code will still work. For example:

```
>>> items["func"](-45)          # Executes abs(-45)
45
>>> items["mod"].sqrt(4)        # Executes math.sqrt(4)
2.0
>>> try:
...     x = int("a lot")
... except items["error"] as e:   # Same as except ValueError as e
...     print("Couldn't convert")
...
Couldn't convert
>>> items["append"](100)        # Executes nums.append(100)
>>> nums
[1, 2, 3, 4, 100]
>>>
```

The fact that everything in Python is first-class is often not fully appreciated by new programmers. However, it can be used to write very compact and flexible code. For example, suppose you had a line of text such as `"GOOG,100,490.10"` and you wanted to convert it into a list of fields with appropriate type-conversion. Here's a clever way that you might do it by creating a list of types (which are first-class objects) and executing a few simple list processing operations:

```
>>> line = "GOOG,100,490.10"
>>> field_types = [str, int, float]
>>> raw_fields = line.split(',')
>>> fields = [ty(val) for ty,val in zip(field_types,raw_fields)]
>>> fields
['GOOG', 100, 490.10000000000002]
>>>
```

Built-in Types for Representing Data

There are approximately a dozen built-in data types that are used to represent most of the data used in programs. These are grouped into a few major categories as shown in Table 3.1. The Type Name column in the table lists the name or expression that you can use to check for that type using `isinstance()` and other type-related functions. Certain types are only available in Python 2 and have been indicated as such (in Python 3, they have been deprecated or merged into one of the other types).

Table 3.1 **Built-In Types for Data Representation**

Type Category	Type Name	Description
None	`type(None)`	The null object `None`
Numbers	`int`	Integer
	`long`	Arbitrary-precision integer (Python 2 only)
	`float`	Floating point
	`complex`	Complex number
	`bool`	Boolean (`True` or `False`)
Sequences	`str`	Character string
	`unicode`	Unicode character string (Python 2 only)
	`list`	List
	`tuple`	Tuple
	`xrange`	A range of integers created by `xrange()` (In Python 3, it is called `range`.)
Mapping	`dict`	Dictionary
Sets	`set`	Mutable set
	`frozenset`	Immutable set

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). 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 (in fact, in Python 3, the two types have been unified into a single integer type). Thus, although you will sometimes see references to long integers in existing Python code, this is mostly an implementation detail that can be ignored—just use the integer type for all integer operations. The one exception is in code that performs explicit type checking for integer values. In Python 2, the expression `isinstance(x, int)` will return `False` if `x` is an integer that has been promoted to a `long`.

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 precise control over the space and precision of numbers is an issue in your program, consider using the numpy extension (which can be found at 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`. The method `z.conjugate()` calculates the complex conjugate of `z` (the conjugate of *a*+*b*j is *a*-*b*j).

Numeric types have a number of properties and methods that are meant to simplify operations involving mixed arithmetic. For simplified compatibility with rational numbers (found in the `fractions` module), integers have the properties `x.numerator` and `x.denominator`. An integer or floating-point number `y` has the properties `y.real` and `y.imag` as well as the method `y.conjugate()` for compatibility with complex numbers. A floating-point number `y` can be converted into a pair of integers representing a fraction using `y.as_integer_ratio()`. The method `y.is_integer()` tests if a floating-point number `y` represents an integer value. Methods `y.hex()` and `y.fromhex()` can be used to work with floating-point numbers using their low-level binary representation.

Several additional numeric types are defined in library modules. The `decimal` module provides support for generalized base-10 decimal arithmetic. The `fractions` module adds a rational number type. These modules are covered in Chapter 14, "Mathematics."

Sequence Types

Sequences represent ordered sets of objects indexed by non-negative integers and include 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.

Operations Common to All Sequences

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). `sum(s)` sums items in `s` but only works for numeric data.

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

Table 3.2 **Continued**

Item	Description
`len(s)`	Number of elements in *s*
`min(s)`	Minimum value in *s*
`max(s)`	Maximum value in *s*
`sum(s [,initial])`	Sum of items in *s*
`all(s)`	Checks whether all items in *s* are `True`.
`any(s)`	Checks whether any item in *s* is `True`.

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

Lists

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 or raises `ValueError` if no such item exists. 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 key function and reverse flag, both of which must be specified as keyword arguments. The key function is a function that is applied to each element prior to comparison during sorting. If given, this function should take a single item as input and return the value that will be used to perform the comparison while 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*.

Table 3.4 **Continued**

Method	Description
`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*.
`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([key [, reverse]])`	Sorts items of *s* in place. *key* is a key function. *reverse* is a flag that sorts the list in reverse order. *key* and *reverse* should always be specified as keyword arguments.

Strings

Python 2 provides two string object types. Byte strings are sequences of bytes containing 8-bit data. They may contain binary data and embedded NULL bytes. Unicode strings are sequences of unencoded Unicode characters, which are internally represented by 16-bit integers. This allows for 65,536 unique character values. Although the Unicode standard supports up to 1 million unique character values, these extra characters are not supported by Python by default. Instead, they are encoded as a special two-character (4-byte) sequence known as a *surrogate pair*—the interpretation of which is up to the application. As an optional feature, Python may be built to store Unicode characters using 32-bit integers. 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.

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. The `s.replace()` method is used to replace a substring with replacement text. It is important to emphasize that all of these methods only work with simple substrings. Regular expression pattern matching and searching is handled by functions in the `re` library module.

The `s.split()` and `s.rsplit()` methods split a string into a list of fields separated by a delimiter. The `s.partition()` and `s.rpartition()` methods search for a separator substring and partition *s* into three parts corresponding to text before the separator, the separator itself, and text after the separator.

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 be given negative values, in which case the index is taken from the end of the string.

The `s.translate()` method is used to perform advanced character substitutions such as quickly stripping all control characters out of a string. As an argument, it accepts a translation table containing a one-to-one mapping of characters in the original string to characters in the result. For 8-bit strings, the translation table is a 256-character string. For Unicode, the translation table can be any sequence object *s* where `s[n]` returns an integer character code or Unicode character corresponding to the Unicode character with integer value *n*.

The `s.encode()` and `s.decode()` methods are used to transform string data to and from a specified character encoding. As input, these accept an encoding name such as `'ascii'`, `'utf-8'`, or `'utf-16'`. These methods are most commonly used to convert Unicode strings into a data encoding suitable for I/O operations and are described further in Chapter 9, "Input and Output." Be aware that in Python 3, the `encode()` method is only available on strings, and the `decode()` method is only available on the bytes datatype.

The `s.format()` method is used to perform string formatting. As arguments, it accepts any combination of positional and keyword arguments. Placeholders in *s* denoted by {*item*} are replaced by the appropriate argument. Positional arguments can be referenced using placeholders such as {`0`} and {`1`}. Keyword arguments are referenced using a placeholder with a name such as {*name*}. Here is an example:

```
>>> a = "Your name is {0} and your age is {age}"
>>> a.format("Mike", age=40)
'Your name is Mike and your age is 40'
>>>
```

Within the special format strings, the {*item*} placeholders can also include simple index and attribute lookup. A placeholder of {*item*`[n]`} where *n* is a number performs a sequence lookup on *item*. A placeholder of {*item*`[`*key*`]`} where *key* is a non-numeric string performs a dictionary lookup of *item*`["`*key*`"]`. A placeholder of {*item.attr*} refers to attribute *attr* of *item*. Further details on the `format()` method can be found in the "String Formatting" section of Chapter 4.

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 (byte strings only).
`s.encode([encoding [,errors]])`	Returns an encoded version of the string (unicode strings only).
`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* or returns -1.

Table 3.5 **Continued**

Method	Description
`s.format(*args, **kwargs)`	Formats *s*.
`s.index(sub [, start [,end]])`	Finds the first occurrence of the specified substring *sub* or raises an error.
`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 in sequence *t* with *s* as a separator.
`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.partition(sep)`	Partitions a string based on a separator string *sep*. Returns a tuple `(head,sep,tail)` or `(s, "","")` if *sep* isn't found.
`s.replace(old, new [,maxreplace])`	Replaces a 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.rpartition(sep)`	Partitions *s* based on a separator *sep*, but searches from the end of the string.
`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*.
`s.split([sep [,maxsplit]])`	Splits a string using *sep* as a delimiter. *maxsplit* is the maximum number of splits to perform.

Table 3.5 **Continued**

`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 white-space 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.zfill(width)`	Pads a string with zeros on the left up to the specified *width*.

`xrange()` Objects

The built-in function `xrange([i,]j [,stride])` creates an object that represents a range of 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. An `xrange` object calculates its values whenever it's accessed and although an `xrange` object looks like a sequence, it is actually somewhat limited. 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.

It should be noted that in Python 3, `xrange()` has been renamed to `range()`. However, it operates in exactly the same manner as described here.

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*.
`k in m`	Returns `True` if *k* is a key in *m*.
`m.clear()`	Removes all items from *m*.
`m.copy()`	Makes a copy of *m*.
`m.fromkeys(s [,value])`	Create a new dictionary with keys from sequence *s* and values all set to *value*.
`m.get(k [,v])`	Returns `m[k]` if found; otherwise, returns *v*.
`m.has_key(k)`	Returns `True` if *m* has key *k*; otherwise, returns `False`. (Deprecated, use the `in` operator instead. Python 2 only)
`m.items()`	Returns a sequence of `(key,value)` pairs.
`m.keys()`	Returns a sequence of key values.
`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.
`m.setdefault(k [, v])`	Returns `m[k]` if found; otherwise, returns *v* and sets `m[k] = v`.
`m.update(b)`	Adds all objects from *b* to *m*.
`m.values()`	Returns a sequence of all values in *m*.

Most of the methods in Table 3.6 are used to manipulate or retrieve the contents of a dictionary. The `m.clear()` method removes all items. 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 key 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.copy()` method makes a shallow copy of the items contained in a mapping object and places them in a new mapping object. The `m.fromkeys(s [,value])` method creates a new mapping with keys all taken from a sequence *s*. The type of the resulting mapping will be the same as *m*. The value associated with all of these keys is set to `None` unless an alternative value is given with the optional *value* parameter. The `fromkeys()` method is defined as a class method, so an alternative way to invoke it would be to use the class name such as `dict.fromkeys()`.

The `m.items()` method returns a sequence containing `(key,value)` pairs. The `m.keys()` method returns a sequence with all the key values, and the `m.values()` method returns a sequence with all the values. For these methods, you should assume that the only safe operation that can be performed on the result is iteration. In Python 2 the result is a list, but in Python 3 the result is an iterator that iterates over the current contents of the mapping. If you write code that simply assumes it is an iterator, it will be generally compatible with both versions of Python. If you need to store the result of these methods as data, make a copy by storing it in a list. For example, `items = list(m.items())`. If you simply want a list of all keys, use `keys = list(m)`.

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. 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)`	Returns the number of items in *s*.
`s.copy()`	Makes a 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.isdisjoint(t)`	Returns `True` if *s* and *t* have no items in common.
`s.issubset(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*.

Table 3.8 **Continued**

Item	Description
`s.discard(item)`	Removes *item* from *s*. If *item* is not a member of *s*, nothing happens.
`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.

Built-in Types for Representing Program Structure

In Python, functions, classes, and modules are all objects that can be manipulated as data. Table 3.9 shows types that are used to represent various elements of a program itself.

Table 3.9 **Built-in Python Types for Program Structure**

Type Category	Type Name	Description
Callable	`types.BuiltinFunctionType`	Built-in function or method
	`type`	Type of built-in types and classes
	`object`	Ancestor of all types and classes
	`types.FunctionType`	User-defined function
	`types.MethodType`	Class method
Modules	`types.ModuleType`	Module
Classes	`object`	Ancestor of all types and classes
Types	`type`	Type of built-in types and classes

Note that `object` and `type` appear twice in Table 3.9 because classes and types are both callable as a function.

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

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

```
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__`	Documentation string
`f.__name__`	Function name
`f.__dict__`	Dictionary containing function attributes
`f.__code__`	Byte-compiled code
`f.__defaults__`	Tuple containing the default arguments
`f.__globals__`	Dictionary defining the global namespace
`f.__closure__`	Tuple containing data related to nested scopes

In older versions of Python 2, many of the preceding attributes had names such as `func_code`, `func_defaults`, and so on. The attribute names listed are compatible with Python 2.6 and Python 3.

Methods

Methods are functions that are defined inside a class definition. There are three common types of methods—instance methods, class methods, and static methods:

```
class Foo(object):
    def instance_method(self,arg):
        statements
    @classmethod
    def class_method(cls,arg):
        statements
    @staticmethod
    def static_method(arg):
        statements
```

An *instance method* is a method that operates on an instance belonging to a given class. The instance is passed to the method as the first argument, which is called `self` by convention. A *class method* operates on the class itself as an object. The class object is passed to a class method in the first argument, `cls`. A *static method* is a just a function that happens to be packaged inside a class. It does not receive an instance or a class object as a first argument.

Both instance and class methods are represented by a special object of type `types.MethodType`. However, understanding this special type requires a careful understanding of how object attribute lookup (`.`) works. The process of looking something up on an object (`.`) is always a separate operation from that of making a function call. When you invoke a method, both operations occur, but as distinct steps. This example illustrates the process of invoking `f.instance_method(arg)` on an instance of `Foo` in the preceding listing:

```
f = Foo()                  # Create an instance
meth = f.instance_method   # Lookup the method and notice the lack of ()
meth(37)                   # Now call the method
```

In this example, `meth` is known as a *bound method*. A bound method is a callable object that wraps both a function (the method) and an associated instance. When you call a bound method, the instance is passed to the method as the first parameter (self). Thus, `meth` in the example can be viewed as a method call that is primed and ready to go but which has not been invoked using the function call operator ().

Method lookup can also occur on the class itself. For example:

```
umeth = Foo.instance_method   # Lookup instance_method on Foo
umeth(f,37)                   # Call it, but explicitly supply self
```

In this example, `umeth` is known as an *unbound method*. An unbound method is a callable object that wraps the method function, but which expects an instance of the proper type to be passed as the first argument. In the example, we have passed `f`, a an instance of `Foo`, as the first argument. If you pass the wrong kind of object, you get a `TypeError`. For example:

```
>>> umeth("hello",5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'instance_method' requires a 'Foo' object but received a
'str'
>>>
```

For user-defined classes, bound and unbound methods are both represented as an object of type `types.MethodType`, which is nothing 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.__class__`	Class in which this method was defined
`m.__func__`	Function object implementing the method
`m.__self__`	Instance associated with the method (`None` if unbound)

One subtle feature of Python 3 is that unbound methods are no longer wrapped by a `types.MethodType` object. If you access `Foo.instance_method` as shown in earlier examples, you simply obtain the raw function object that implements the method. Moreover, you'll find that there is no longer any type checking on the `self` parameter.

Built-in Functions and Methods

The object `types.BuiltinFunctionType` is used to represent functions and methods implemented in C and 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 (if bound)

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`.

Classes and Instances as Callables

Class objects and instances also operate as callable objects. A class object is created by the `class` statement and is called as a function in order to create new instances. In this case, the arguments to the function are passed to the `__init__()` method of the class in order to initialize the newly created instance. An instance can emulate a function 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)`.

Classes, Types, and Instances

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'>
```

The following table shows commonly used attributes of a type object `t`:

Attribute	Description
`t.__doc__`	Documentation string
`t.__name__`	Class name
`t.__bases__`	Tuple of base classes
`t.__dict__`	Dictionary holding class methods and variables
`t.__module__`	Module name in which the class is defined
`t.__abstractmethods__`	Set of abstract method names (may be undefined if there aren't any)

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'>
```

The following table shows special attributes of an instance `i`:

Attribute	Description
`i.__class__`	Class to which the instance belongs
`i.__dict__`	Dictionary holding instance data

The `__dict__` attribute is normally where all of the data associated with an instance is stored. When you make assignments such as `i.attr = value`, the value is stored here. However, if a user-defined class uses `__slots__`, a more efficient internal representation is used and instances will not have a `__dict__` attribute. More details on objects and the organization of the Python object system can be found in Chapter 7.

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, only defined when the module object refers to a package

Built-in Types for Interpreter Internals

A number of objects used by the internals of the interpreter are exposed to the user. These include traceback objects, code objects, frame objects, generator objects, slice objects, and the `Ellipsis` as shown in Table 3.10. It is relatively rare for programs to manipulate these objects directly, but they may be of practical use to tool-builders and framework designers.

Table 3.10 **Built-in Python Types for Interpreter Internals**

Type Name	Description
`types.CodeType`	Byte-compiled code
`types.FrameType`	Execution frame
`types.GeneratorType`	Generator object
`types.TracebackType`	Stack traceback of an exception
`slice`	Generated by extended slices
`Ellipsis`	Used in extended slices

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.

Attribute	Description
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.
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_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 (Python 2 only)
f.f_exc_value	Most recent exception value (Python 2 only)
f.f_exc_traceback	Most recent exception traceback (Python 2 only)

Traceback Objects

Traceback objects are created when an exception occurs and contain 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_lineno	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_code	Code object for the generator function.
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 (this method is called __next__ in Python 3).
g.send(*value*)	Sends a value to a generator. The passed value is returned by the yield expression in the generator that executes until the next yield expression is encountered. send() returns the value passed to yield in this expression.
g.close()	Closes a generator by raising a GeneratorExit exception in the generator function. This method executes automatically when a generator object is garbage-collected.
g.throw(*exc* [,*exc_value* [,*exc_tb*]])	Raises an exception in a generator at the point of the current yield statement. *exc* is the exception type, *exc_value* is the exception value, and *exc_tb* is an optional traceback. If the resulting exception is caught and handled, returns the value passed to the next yield statement.

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. Here's an 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 an index lookup []. 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 make use of Ellipsis, but it may be useful if you are trying to build advanced functionality into the indexing operator [] on your own objects. The following code shows how an Ellipsis gets created and passed into the indexing operator:

```
class Example(object):
    def __getitem__(self,index):
        print(index)
e = Example()
e[3, ..., 4]       # Calls e.__getitem__((3, Ellipsis, 4))
```

Object Behavior and Special Methods

Objects in Python are generally classified according to their behaviors and the features that they implement. For example, all of the sequence types such as strings, lists, and tuples are grouped together merely because they all happen to support a common set of sequence operations such as *s*[*n*], len(*s*), etc. All basic interpreter operations are implemented through 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.

The next few sections describe the special methods associated with different categories of interpreter features.

Object Creation and Destruction

The methods in Table 3.11 create, initialize, and destroy instances. __new__() is a class method that is called to create an instance. 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 3.11 **Special Methods for Object Creation and Destruction**

Method	Description
__new__(*cls* [,*args* [,**kwargs]])	A class method called to create a new instance
__init__(*self* [,*args* [,**kwargs]])	Called to initialize a new instance
__del__(*self*)	Called when an instance is being destroyed

The __new__() and __init__() methods are used together 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)
```

In user-defined objects, it is rare to define __new__() or __del__(). __new__() is usually only defined in metaclasses or in user-defined objects that happen to inherit from one of the immutable types (integers, strings, tuples, and so on). __del__() is only defined in situations in which there is some kind of critical resource management issue, such as releasing a lock or shutting down a connection.

Object String Representation

The methods in Table 3.12 are used to create various string representations of an object.

Table 3.12 **Special Methods for Object Representation**

Method	Description
__format__(*self*, *format_spec*)	Creates a formatted representation
__repr__(*self*)	Creates a string representation of an object
__str__(*self*)	Creates a simple string representation

The __repr__() and __str__() methods create simple string representations of an object. The __repr__() method normally returns an expression string that can be evaluated to re-create the object. This is also the method responsible for creating the output of values you see when inspecting variables in the interactive interpreter. This method is invoked by the built-in repr() function. Here's an example of using repr() and eval() together:

```
a = [2,3,4,5]       # Create a list
s = repr(a)         # s = '[2, 3, 4, 5]'
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 functions related to printing. 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 `__format__()` method is called by the `format()` function or the `format()` method of strings. The *format_spec* argument is a string containing the format specification. This string is the same as the *format_spec* argument to `format()`. For example:

```
format(x,"spec")            # Calls x.__format__("spec")
"x is {0:spec}".format(x)   # Calls x.__format__("spec")
```

The syntax of the format specification is arbitrary and can be customized on an object-by-object basis. However, a standard syntax is described in Chapter 4.

Object Comparison and Ordering

Table 3.13 shows methods that can be used to perform simple tests on an object. The `__bool__()` method is used for truth-value testing and should return `True` or `False`. If undefined, the `__len__()` method is a fallback that is invoked to determine truth. The `__hash__()` method is defined on objects that want to work as keys in a dictionary. The value returned is an integer that 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.

Table 3.13 **Special Methods for Object Testing and Hashing**

Method	Description
`__bool__(self)`	Returns `False` or `True` for truth-value testing
`__hash__(self)`	Computes an integer hash index

Objects can implement one or more of the relational operators (<, >, <=, >=, ==, !=). Each of these methods 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 shows the special methods for comparison operators.

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*

Table 3.14 **Continued**

Method	Result
`__eq__(self,other)`	*self == other*
`__ne__(self,other)`	*self != other*

It is not necessary for an object to implement all of the operations in Table 3.14. However, if you want to be able to compare objects using == or use an object as a dictionary key, the `__eq__()` method should be defined. If you want to be able to sort objects or use functions such as `min()` or `max()`, then `__lt__()` must be minimally defined.

Type Checking

The methods in Table 3.15 can be used to redefine the behavior of the type checking functions `isinstance()` and `issubclass()`. The most common application of these methods is in defining abstract base classes and interfaces, as described in Chapter 7.

Table 3.15 **Methods for Type Checking**

Method	Result
`__instancecheck__(cls,object)`	`isinstance(object, cls)`
`__subclasscheck__(cls, sub)`	`issubclass(sub, cls)`

Attribute Access

The methods in Table 3.16 read, write, and delete the attributes of an object using the dot (`.`) operator and the `del` operator, respectively.

Table 3.16 **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 or raise `AttributeError`.
`__setattr__(self, name, value)`	Sets the attribute *self.name = value*. Overrides the default mechanism.
`__delattr__(self, name)`	Deletes the attribute *self.name*.

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.

Attribute Wrapping and Descriptors

A subtle aspect of attribute manipulation is that sometimes the attributes of an object are wrapped with an extra layer of logic that interact with the get, set, and delete operations described in the previous section. This kind of wrapping is accomplished by creating a *descriptor* object that implements one or more of the methods in Table 3.17. Keep in mind that descriptions are optional and rarely need to be defined.

Table 3.17 **Special Methods for Descriptor Object**

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

The `__get__()`, `__set__()`, and `__delete__()` methods of a descriptor are meant to interact with the default implementation of `__getattribute__()`, `__setattr__()`, and `__delattr__()` methods on classes and types. This interaction occurs if you place an instance of a descriptor object in the body of a user-defined class. In this case, all access to the descriptor attribute will implicitly invoke the appropriate method on the descriptor object itself. Typically, descriptors are used to implement the low-level functionality of the object system including bound and unbound methods, class methods, static methods, and properties. Further examples appear in Chapter 7.

Sequence and Mapping Methods

The methods in Table 3.18 are used by objects that want to emulate sequence and mapping objects.

Table 3.18 **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]*
`__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)              # a.__len__()
x = a[2]            # x = a.__getitem__(2)
a[1] = 7            # a.__setitem__(1,7)
del a[2]            # a.__delitem__(2)
5 in a              # a.__contains__(5)
```

The `__len__` method is called by the built-in `len()` function to return a nonnegative length. This function also determines truth values unless the `__bool__()` 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 `__contains__()` method is used to implement the `in` operator.

The slicing operations such as `x = s[i:j]` are also implemented using `__getitem__()`, `__setitem__()`, and `__delitem__()`. However, for slices, a special `slice` object is passed as the key. This object has attributes that describe the range of the slice being requested. For example:

```
a = [1,2,3,4,5,6]
x = a[1:5]          # x = a.__getitem__(slice(1,5,None))
a[1:3] = [10,11,12] # a.__setitem__(slice(1,3,None), [10,11,12])
del a[1:4]          # a.__delitem__(slice(1,4,None))
```

The slicing features of Python are actually more powerful than many programmers realize. For example, the following variations of extended slicing are all supported and might be useful for working with multidimensional data structures such as matrices and arrays:

```
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, the ellipsis (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 a combination of `slice` or `Ellipsis` objects. For example,

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

invokes `__getitem__()` as follows:

```
a = m.__getitem__((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()` (or *iter*`.__next__()` in Python 3), 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()(#_iter.__next__() in Python 3)
    except StopIteration:
        break
    # Do statements in body of for loop
    ...
```

Mathematical Operations

Table 3.19 lists special methods that objects must implement to emulate numbers. Mathematical operations are always evaluated from left to right according the precedence rules described in Chapter 4; 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 3.19 **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* (Python 2 only)
`__truediv__(self,other)`	*self / other* (Python 3)
`__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* (Python 2 only)
`__rtruediv__(self,other)`	*other / self* (Python 3)
`__rfloordiv__(self,other)`	*other // self*
`__rmod__(self,other)`	*other % self*
`__rdivmod__(self,other)`	`divmod(other,self)`

Table 3.19 **Continued**

Method	Result
`__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*
`__isub__(self,other)`	*self -= other*
`__imul__(self,other)`	*self *= other*
`__idiv__(self,other)`	*self /= other* (Python 2 only)
`__itruediv__(self,other)`	*self /= other* (Python 3)
`__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)` (Python 2 only)
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`

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, the value of an object could be modified 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 reasons why there are three operations deal with a change in the semantics of integer division that started in Python 2.2 but became the default behavior in Python 3. In Python 2, the default behavior of Python is to map the `/` operator to `__div__()`. For integers, this operation truncates the result to an integer. In Python 3, division is mapped to `__truediv__()` and for integers, a float is returned. This latter behavior can be enabled in Python 2 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. These methods are invoked by explicit type conversions such as `int()` and `float()`. However, these methods are not used to implicitly coerce types in mathematical operations. For example, the expression `3 + x` produces a `TypeError` even if *x* is a user-defined object that defines `__int__()` for integer conversion.

Callable Interface

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, ...)`. Objects that emulate functions can be useful for creating functors or proxies. Here is a simple example:

```
class DistanceFrom(object):
    def __init__(self,origin):
        self.origin = origin
    def __call__(self, x):
        return abs(x - self.origin)

nums = [1, 37, 42, 101, 13, 9, -20]
nums.sort(key=DistanceFrom(10))         # Sort by distance from 10
```

In this example, the `DistanceFrom` class creates instances that emulate a single-argument function. These can be used in place of a normal function—for instance, in the call to `sort()` in the example.

Context Management Protocol

The `with` statement allows a sequence of statements to execute under the control of another object known as a *context manager*. The general syntax is as follows:

```
with context [ as var]:
    statements
```

The *context* object shown here is expected to implement the methods shown in Table 3.20. The `__enter__()` method is invoked when the `with` statement executes. The value returned by this method is placed into the variable specified with the optional `as` *var* specifier. The `__exit__()` method is called as soon as control-flow leaves from the block of statements associated with the `with` statement. As arguments, `__exit__()` receives the current exception type, value, and traceback if an exception has been raised. If no errors are being handled, all three values are set to `None`.

Table 3.20 **Special Methods for Context Managers**

Method	Description
`__enter__(self)`	Called when entering a new context. The return value is placed in the variable listed with the `as` specifier to the `with` statement.

Table 3.20 **Continued**

Method	Description
`__exit__(self, type, value, tb)`	Called when leaving a context. If an exception occurred, *type*, *value*, and *tb* have the exception type, value, and traceback information. The primary use of the context management interface is to allow for simplified resource control on objects involving system state such as open files, network connections, and locks. By implementing this interface, an object can safely clean up resources when execution leaves a context in which an object is being used. Further details are found in Chapter 5, "Program Structure and Control Flow."

Object Inspection and `dir()`

The `dir()` function is commonly used to inspect objects. An object can supply the list of names returned by `dir()` by implementing `__dir__(self)`. Defining this makes it easier to hide the internal details of objects that you don't want a user to directly access. However, keep in mind that a user can still inspect the underlying `__dict__` attribute of instances and classes to see everything that is defined.

4 Operators and Expressions

This chapter describes Python's built-in operators, expressions, and evaluation rules. Although much of this chapter describes Python's built-in types, user-defined objects can easily redefine any of the operators to provide their own behavior.

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. In Python 2, 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 changes in Python 3, where division produces a floating-point result. 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 be applied only to integers:

Operation	Description
`x << y`	Left shift
`x >> y`	Right shift
`x & y`	Bitwise and
`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 and that 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, the result will grow arbitrarily large in magnitude.

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 and is only valid on non-complex numbers. 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, Python 2 rounds to the nearest multiple away from zero (for example, `0.5` is rounded to `1.0` and `-0.5` is rounded to `-1.0`). One caution here is that Python 3 rounds equally close values to the nearest even multiple (for example, 0.5 is rounded to 0.0, and 1.5 is rounded to 2.0). This is a subtle portability issue for mathematical programs being ported to Python 3.

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
`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 reading the code (it's important to note that no comparison is made between *x* and *z* in such an expression). Comparisons involving complex numbers are undefined and result in a `TypeError`.

Operations involving numbers are valid only if the operands are of the same type. For built-in numbers, 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. Otherwise, both numbers must be integers and no conversion is performed.

For user-defined objects, the behavior of expressions involving mixed operands depends on the implementation of the object. As a general rule, the interpreter does not try to perform any kind of implicit type conversion.

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
`v1,v2…, vn = s`	Variable unpacking
`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
`all(s)`	Returns `True` if all items in *s* are true.
`any(s)`	Returns `True` if any item in *s* is true.
`len(s)`	Length
`min(s)`	Minimum item in *s*
`max(s)`	Maximum item in *s*
`sum(s [, initial])`	Sum of items with an optional initial value

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]
>>> b = [a]
>>> c = 4*b
>>> c
[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7
>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

Notice how the change to `a` modified every element of the list `c`. 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`. Here's an example:

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

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

All sequences can be unpacked into a sequence of variable names. For example:

```
items = [ 3, 4, 5 ]
x,y,z = items       # x = 3, y = 4, z = 5

letters = "abc"
x,y,z = letters       # x = 'a', y = 'b', z = 'c'

datetime = ((5, 19, 2008), (10, 30, "am"))
(month,day,year),(hour,minute,am_pm) = datetime
```

When unpacking values into variables, the number of variables must exactly match the number of items in the sequence. In addition, the structure of the variables must match that of the sequence. For example, the last line of the example unpacks values into six variables, organized into two 3-tuples, which is the structure of the sequence on the right. Unpacking sequences into variables works with any kind of sequence, including those created by iterators and generators.

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`. It is important to note that the `in` operator does not support wildcards or any kind of pattern matching. For this, you need to use a library module such as the `re` module for regular expression patterns.

The `for x in s` operator iterates over all the elements of a sequence and is described further in Chapter 5, "Program Structure and 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). `sum(s)` sums all of the items in `s` but usually works only if the items represent numbers. An optional initial value can be given to `sum()`. The type of this value usually determines the result. For example, if you used `sum(items, decimal.Decimal(0))`, the result would be a `Decimal` object (see more about the `decimal` module in Chapter 14, "Mathematics").

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 element `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 numerical index determined by the character set (such as ASCII or Unicode). A character is less than another character if its index is less. One caution concerning character ordering is that the preceding simple comparison operators are not related to the character ordering rules associated with locale or language settings. Thus, you would not use these operations to order strings according to the standard conventions of a foreign language (see the `unicodedata` and `locale` modules for more information).

Another caution, this time involving strings. Python has two types of string data: byte strings and Unicode strings. Byte strings differ from their Unicode counterpart in that they are usually assumed to be encoded, whereas Unicode strings represent raw unencoded character values. Because of this, you should never mix byte strings and Unicode together in expressions or comparisons (such as using `+` to concatenate a byte string and Unicode string or using `==` to compare mixed strings). In Python 3, mixing string types results in a `TypeError` exception, but Python 2 attempts to perform an implicit promotion of byte strings to Unicode. This aspect of Python 2 is widely considered to be a design mistake and is often a source of unanticipated exceptions and inexplicable program behavior. So, to keep your head from exploding, don't mix string types in sequence operations.

String Formatting

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) `d`. 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 = 5628398123741234

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

When used with a dictionary, the string formatting operator `%` is often used to mimic the string interpolation feature often found in scripting languages (e.g., expansion of `$var` symbols in strings). For example, if you have a dictionary of values, you can expand those values into fields within a formatted string as follows:

```
stock = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10 }

r = "%(shares)d of %(name)s at %(price)0.2f" % stock
# r = "100 shares of GOOG at 490.10"
```

The following code shows how to expand the values of currently defined variables within a string. The `vars()` function returns a dictionary containing all of the variables defined at the point at which `vars()` is called.

```
name = "Elwood"
age  = 41
r = "%(name)s is %(age)s years old" % vars()
```

Advanced String Formatting

A more advanced form of string formatting is available using the `s.format(*args, *kwargs)` method on strings. This method collects an arbitrary collection of positional and keyword arguments and substitutes their values into placeholders embedded in `s`. A placeholder of the form `'{n}'`, where `n` is a number, gets replaced by positional argument `n` supplied to `format()`. A placeholder of the form `'{name}'` gets replaced by keyword argument `name` supplied to format. Use `'{{'` to output a single `'{'` and `'}}'` to output a single `'}'`. For example:

```
r = "{0} {1} {2}".format('GOOG',100,490.10)
r = "{name} {shares} {price}".format(name='GOOG',shares=100,price=490.10)
r = "Hello {0}, your age is {age}".format("Elwood",age=47)
r = "Use {{ and }} to output single curly braces".format()
```

With each placeholder, you can additionally perform both indexing and attribute lookups. For example, in `'{name[n]}'` where `n` is an integer, a sequence lookup is performed and in `'{name[key]}'` where `key` is a non-numeric string, a dictionary lookup of the form `name['key']` is performed. In `'{name.attr}'`, an attribute lookup is performed. Here are some examples:

```
stock = { 'name' : 'GOOG',
      'shares' : 100,
      'price' : 490.10 }
r = "{0[name]} {0[shares]} {0[price]}".format(stock)

x = 3 + 4j
r = "{0.real} {0.imag}".format(x)
```

In these expansions, you are only allowed to use names. Arbitrary expressions, method calls, and other operations are not supported.

You can optionally specify a format specifier that gives more precise control over the output. This is supplied by adding an optional format specifier to each placeholder using a colon (:), as in `'{place:format_spec}'`. By using this specifier, you can specify column widths, decimal places, and alignment. Here is an example:

```
r = "{name:8} {shares:8d} {price:8.2f}".format
(name="GOOG",shares=100,price=490.10)
```

The general format of a specifier is `[[fill[align]][sign][0][width][.precision][type]` where each part enclosed in `[]` is optional. The *width* specifier specifies the minimum field width to use, and the *align* specifier is one of `'<'`, `'>'`, or `'^'` for left, right, and centered alignment within the field. An optional fill character *fill* is used to pad the space. For example:

```
name = "Elwood"
r = "{0:<10}".format(name)      # r = 'Elwood    '
r = "{0:>10}".format(name)      # r = '    Elwood'
r = "{0:^10}".format(name)      # r = '  Elwood  '
r = "{0:=^10}".format(name)     # r = '==Elwood=='
```

The *type* specifier indicates the type of data. Table 4.2 lists the supported format codes. If not supplied, the default format code is `'s'` for strings, `'d'` for integers, and `'f'` for floats.

Table 4.2 **Advanced String Formatting Type Specifier Codes**

Character	Output Format
d	Decimal integer or long integer.
b	Binary integer or long integer.
o	Octal integer or long integer.
x	Hexadecimal integer or long integer.
X	Hexadecimal integer (uppercase letters).
f,F	Floating point as `[-]m.dddddd`.
e	Floating point as `[-]m.ddddde±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`.
n	Same as `g` except that the current locale setting determines the decimal point character.
%	Multiplies a number by 100 and displays it using `f` format followed by a `%` sign.
s	String or any object. The formatting code uses `str()` to generate strings.
c	Single character.

The *sign* part of a format specifier is one of `'+'`, `'-'`, or `' '`. A `'+'` indicates that a leading sign should be used on all numbers. `'-'` is the default and only adds a sign character for negative numbers. A `' '` adds a leading space to positive numbers. The *precision* part of the specifier supplies the number of digits of accuracy to use for decimals. If a leading `'0'` is added to the field width for numbers, numeric values are padded with leading 0s to fill the space. Here are some examples of formatting different kinds of numbers:

```
x = 42
r = '{0:10d}'.format(x)         # r = '        42'
r = '{0:10x}'.format(x)         # r = '        2a'
r = '{0:10b}'.format(x)         # r = '    101010'
r = '{0:010b}'.format(x)        # r = '0000101010'

y = 3.1415926
r = '{0:10.2f}'.format(y)       # r = '      3.14'


r = '{0:10.2e}'.format(y)       # r = '  3.14e+00'
r = '{0:+10.2f}'.format(y)      # r = '     +3.14'
r = '{0:+010.2f}'.format(y)     # r = '+000003.14'
r = '{0:+10.2%}'.format(y)      # r = '  +314.16%'
```

Parts of a format specifier can optionally be supplied by other fields supplied to the format function. They are accessed using the same syntax as normal fields in a format string. For example:

```
y = 3.1415926
r = '{0:{width}.{precision}f}'.format(y,width=10,precision=3)
r = '{0:{1}.{2}f}'.format(y,10,3)
```

This nesting of fields can only be one level deep and can only occur in the format specifier portion. In addition, the nested values cannot have any additional format specifiers of their own.

One caution on format specifiers is that objects can define their own custom set of specifiers. Underneath the covers, advanced string formatting invokes the special method `__format__(self, format_spec)` on each field value. Thus, the capabilities of the `format()` operation are open-ended and depend on the objects to which it is applied. For example, dates, times, and other kinds of objects may define their own format codes.

In certain cases, you may want to simply format the `str()` or `repr()` representation of an object, bypassing the functionality implemented by its `__format__()` method. To do this, you can add the `'!s'` or `'!r'` modifier before the format specifier. For example:

```
name = "Guido"
r = '{0!r:^20}'.format(name)     # r = "      'Guido'       "
```

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
`k in d`	Tests for the existence of a 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

The result of union, intersection, and difference operations will have the same type as the left-most operand. For example, if *s* is a `frozenset`, the result will be a `frozenset` even if `t` is a `set`.

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. Here's an 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. Here's an 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`.

User-defined classes can redefine or customize the behavior of (.). More details are found in Chapter 3 and Chapter 7, "Classes and Object-Oriented Programming."

The Function Call () Operator

The `f(args)` operator is used to make a function call on *f*. Each argument to a function is an expression. Prior to calling the function, all of the argument expressions are fully evaluated from left to right. This is sometimes known as *applicative order evaluation*.

It is possible to partially evaluate function arguments using the `partial()` function in the `functools` module. For example:

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

from functools import partial
f = partial(foo,1,2)  # Supply values to x and y arguments of foo
f(3)                  # Calls foo(1,2,3), result is 6
```

The `partial()` function evaluates some of the arguments to a function and returns an object that you can call to supply the remaining arguments at a later point. In the previous example, the variable `f` represents a partially evaluated function where the first two arguments have already been calculated. You merely need to supply the last remaining argument value for the function to execute. Partial evaluation of function arguments is closely related to a process known as *currying*, a mechanism by which a function taking multiple arguments such as `f(x,y)` is decomposed into a series of functions each taking only one argument (for example, you partially evaluate `f` by fixing `x` to get a new function to which you give values of `y` to produce a result).

Conversion Functions

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.
`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.

Function	Description
`repr(x)`	Converts object *x* to an expression string.
`format(x [,format_spec])`	Converts object *x* to a formatted 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 (Python 2 only).
`ord(x)`	Converts a single character to its integer value.
`hex(x)`	Converts an integer to a hexadecimal string.
`bin(x)`	Converts an integer to a binary string.
`oct(x)`	Converts an integer to an octal string.

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 `format(x, [format_spec])` function produces the same output as that produced by the advanced string formatting operations but applied to a single object *x*. As input, it accepts an optional *format_spec*, which is a string containing the formatting code. The `ord()` function returns the integer ordinal value of a character. For Unicode, this value will be the integer code point. The `chr()` and `unichr()` functions convert integers back into characters.

To convert strings back into numbers, use the `int()`, `float()`, and `complex()` 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 used to generate all the items used to populate the object that's being created.

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. This is sometimes known as *"short-circuit" evaluation*.

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. It may also result in an exception depending on the type.

Order of Evaluation

Table 4.3 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 4.3 **Order of Evaluation (Highest to Lowest)**

Operator	Name
`(...), [...], {...}`	Tuple, list, and dictionary creation
`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,`	Comparison, identity, and sequence membership tests
`x > y, x >= y,`	
`x == y, x != y`	
`x is y, x is not y`	
`x in s, x not in s`	
`not x`	Logical negation
`x and y`	Logical and
`x or y`	Logical or
`lambda args: expr`	Anonymous function

The order of evaluation is not determined by the types of *x* and *y* in Table 4.3. So, even though user-defined objects can redefine individual operators, it is not possible to customize the underlying evaluation order, precedence, and associativity rules.

Conditional Expressions

A common programming pattern is that of conditionally assigning a value based on the result of an expression. For example:

```
if a <= b:
   minvalue = a
else:
   minvalue = b
```

This code can be shortened using a *conditional expression*. For example:

```
minvalue = a if a <=b else b
```

In such expressions, the condition in the middle is evaluated first. The expression to the left of the `if` is then evaluated if the result is `True`. Otherwise, the expression after the `else` is evaluated.

Conditional expressions should probably be used sparingly because they can lead to confusion (especially if they are nested or mixed with other complicated expressions). However, one particularly useful application is in list comprehensions and generator expressions. For example:

```
values = [1, 100, 45, 23, 73, 37, 69 ]
clamped = [x if x < 50 else 50 for x in values]
print(clamped)      # [1, 50, 45, 23, 50, 37, 50]
```

5

Program Structure and Control Flow

This chapter covers the details of program structure and control flow. Topics include conditionals, iteration, exceptions, and context managers.

Program Structure and Execution

Python programs are structured as a sequence of statements. All language features, including variable assignment, function definitions, classes, and module imports, are statements that have equal status with all other statements. In fact, there are no "special" statements, and every statement can be placed anywhere in a program. For example, this code defines two different versions of a function:

```
if debug:
    def square(x):
        if not isinstance(x,float):
            raise TypeError("Expected a float")
        return x * x
else:
    def square(x):
        return x * x
```

When loading source files, the interpreter always executes every statement in order until there are no more statements to execute. This execution model applies both to files you simply run as the main program and to library files that are loaded via `import`.

Conditional Execution

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. Here's an 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 (Use __next__ in Python 3)
     except StopIteration:     # No more items
         break
         # Perform operations on i
     ...
```

In the statement `for i in s`, the variable *`i`* is known as the *iteration variable*. On each iteration of the loop, it receives a new value from *`s`*. The scope of the iteration variable is not private to the `for` statement. If a previously defined variable has the same name, that value will be overwritten. Moreover, the iteration variable retains the last value after the loop has completed.

If the elements used in iteration are sequences of identical size, you can unpack their values into individual iteration variables using a statement such as the following:

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

In this example, *`s`* must contain or produce sequences, each with three elements. On each iteration, the contents of the variables *`x`*, *`y`*, and *`z`* are assigned the items of the corresponding sequence. Although it is most common to see this used when *`s`* is a sequence of tuples, unpacking works if the items in *`s`* are any kind of sequence including lists, generators, and strings.

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

```
i = 0
for x in s:


    statements
    i += 1
```

Python provides a built-in function, `enumerate()`, that can be used to simplify this code:

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

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

Another common looping problem concerns iterating in parallel over two or more sequences—for example, writing a loop where you want to take items from different sequences on each iteration as follows:

```
# s and t are two sequences
i = 0
while i < len(s) and i < len(t):
    x = s[i]     # Take an item from s
    y = t[i]     # Take an item from t
    statements
    i += 1
```

This code can be simplified using the `zip()` function. For example:

```
# s and t are two sequences
for x,y in zip(s,t):
    statements
```

`zip(`*`s`*`,`*`t`*`)` combines sequences *`s`* and *`t`* into a sequence of tuples `(`*`s`*`[0],`*`t`*`[0]), (`*`s`*`[1],`*`t`*`[1]), (`*`s`*`[2], `*`t`*`[2])`, and so forth, stopping with the shortest of the sequences *`s`* and *`t`* should they be of unequal length. One caution with `zip()` is that in Python 2, it fully consumes both *`s`* and *`t`*, creating a list of tuples. For generators and sequences containing a large amount of data, this may not be what you want. The function `itertools.izip()` achieves the same effect as `zip()` but generates the zipped values one at a time rather than creating a large list of tuples. In Python 3, the `zip()` function also generates values in this manner.

To break out of a loop, use the `break` statement. For example, this code reads lines of text from a file until an empty line of text is encountered:

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break          # A blank line, stop reading
    # process the stripped line
    ...
```

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 skips all of the blank lines in a file:

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        continue      # Skip the blank line
    # process the stripped line
    ...
```

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:

```
# for-else
for line in open("foo.txt"):
     stripped = line.strip()
     if not stripped:
         break
     # process the stripped line
     ...
else:
    raise RuntimeError("Missing section separator")
```

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.

The primary use case for the looping `else` clause is in code that iterates over data but which needs to set or check some kind of flag or condition if the loop breaks prematurely. For example, if you didn't use `else`, the previous code might have to be rewritten with a flag variable as follows:

```
found_separator = False
for line in open("foo.txt"):
     stripped = line.strip()
     if not stripped:
         found_separator = True
         break
     # process the stripped line
     ...
if not found_separator:
    raise RuntimeError("Missing section separator")
```

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. Here's an example:

```
raise RuntimeError("Unrecoverable Error")
```

If the `raise` statement is used by itself, 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 as e:
    statements
```

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 Services."

The optional `as` *`var`* modifier to the `except` statement supplies the name of a variable in which an instance of the exception type 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. For example, you can use `isinstance()` to check the exception type. One caution on the syntax: In previous versions of Python, the `except` statement was written as `except` *`ExcType, var`* where the exception type and variable were separated by a comma (`,`). In Python 2.6, this syntax still works, but it is deprecated. In new code, use the `as` *`var`* syntax because it is required in Python 3.

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

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

A single handler can catch multiple exception types like this:

```
try:
   do something
except (IOError, TypeError, NameError) as 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 except those related to program exit, use `Exception` like this:

```
try:
   do something
except Exception as e:
   error_log.write('An error occurred : %s\n' % e)
```

When catching all exceptions, you should take care to report accurate error information to the user. For example, in the previous code, an error message and the associated exception value is being logged. If you don't include any information about the exception value, it can make it very difficult to debug code that is failing for reasons that you don't expect.

All exceptions can be caught using `except` with no exception type as follows:

```
try:
   do something
except:
   error_log.write('An error occurred\n')
```

Correct use of this form of `except` is a lot trickier than it looks and should probably be avoided. For instance, this code would also catch keyboard interrupts and requests for program exit—things that you may not want to catch.

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 as e:
   error_log.write('Unable to open foo : %s\n' % e)
else:
   data = f.read()
   f.close()
```

The `finally` statement defines a cleanup action for code contained in a `try` block. Here's an example:

```
f = open('foo','r')
try:
   # Do some stuff
   ...
finally:
   f.close()
   # 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.

Built-in Exceptions

Python defines the built-in exceptions listed in Table 5.1.

Table 5.1 **Built-in Exceptions**

Exception	Description
`BaseException`	The root of all exceptions.
`GeneratorExit`	Raised by `.close()` method on a generator.
`KeyboardInterrupt`	Generated by the interrupt key (usually Ctrl+C).
`SystemExit`	Program exit/termination.
`Exception`	Base class for all non-exiting exceptions.
`StopIteration`	Raised to stop iteration.
`StandardError`	Base for all built-in exceptions (Python 2 only). In Python 3, all exceptions below are grouped under `Exception`.
`ArithmeticError`	Base for arithmetic exceptions.
`FloatingPointError`	Failure of a floating-point operation.
`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.
`LookupError`	Indexing and key errors.
`IndexError`	Out-of-range sequence index.
`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.
`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. Here's an example:

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

or

```
try:
     statements
except Exception:   # Catch any program-related exception
     statements
```

At the top of the exception hierarchy, the exceptions are grouped according to whether or not the exceptions are related to program exit. For example, the `SystemExit` and `KeyboardInterrupt` exceptions are not grouped under `Exception` because programs that want to catch all program-related errors usually don't want to also capture program termination by accident.

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 `Exception`, such as the following:

```
class NetworkError(Exception): pass
```

To use your new exception, use it with the `raise` statement as follows:

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

When raising an exception, the optional values supplied with the `raise` statement are used as the arguments to the exception's class constructor. Most of the time, this is simply a string indicating some kind of error message. However, user-defined exceptions can be written to take one or more exception values as shown in this example:

```
class DeviceError(Exception):
     def __init__(self,errno,msg):
         self.args = (errno, msg)
         self.errno = errno
         self.errmsg = msg

# Raises an exception (multiple arguments)
raise DeviceError(1, 'Not Responding')
```

When you create a custom exception class that redefines `__init__()`, it is important to assign a tuple containing the arguments to `__init__()` to the attribute `self.args` as shown. This attribute is used when printing exception traceback messages. If you leave it undefined, users won't be able to see any useful information about the exception when an error occurs.

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. Here's an example:

```
class HostnameError(NetworkError): pass
class TimeoutError(NetworkError): pass

def error1():
    raise HostnameError("Unknown host")

def error2():
    raise TimeoutError("Timed out")

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # Perform special actions for this kind of error
        ...
```

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 type of the execution value with `type()`. Alternatively, the `sys.exc_info()` function can be used to retrieve information about the last raised exception.

Context Managers and the `with` Statement

Proper management of system resources such as files, locks, and connections is often a tricky problem when combined with exceptions. For example, a raised exception can cause control flow to bypass statements responsible for releasing critical resources such as a lock.

The `with` statement allows a series of statements to execute inside a runtime context that is controlled by an object that serves as a context manager. Here is an example:

```
with open("debuglog","a") as f:
     f.write("Debugging\n")
     statements
     f.write("Done\n")

import threading
lock = threading.Lock()
with lock:
     # Critical section
     statements
     # End critical section
```

In the first example, the `with` statement automatically causes the opened file to be closed when control-flow leaves the block of statements that follows. In the second example, the `with` statement automatically acquires and releases a lock when control enters and leaves the block of statements that follows.

The `with` *`obj`* statement allows the object *`obj`* to manage what happens when control-flow enters and exits the associated block of statements that follows. When the `with` *`obj`* statement executes, it executes the method *`obj`*`.__enter__()` to signal that a new context is being entered. When control flow leaves the context, the method *`obj`*`.__exit__(`*`type,value,traceback`*`)` executes. If no exception has been raised, the three arguments to `__exit__()` are all set to `None`. Otherwise, they contain the type, value, and traceback associated with the exception that has caused control-flow to leave the context. The `__exit__()` method returns `True` or `False` to indicate whether the raised exception was handled or not (if `False` is returned, any exceptions raised are propagated out of the context).

The `with` *obj* statement accepts an optional `as` *var* specifier. If given, the value returned by *obj*`.__enter__()` is placed into *var*. It is important to emphasize that *obj* is not necessarily the value assigned to *var*.

The `with` statement only works with objects that support the context management protocol (the `__enter__()` and `__exit__()` methods). User-defined classes can implement these methods to define their own customized context-management. Here is a simple example:

```
class ListTransaction(object):
    def __init__(self,thelist):
        self.thelist = thelist
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy
    def __exit__(self,type,value,tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

This class allows one to make a sequence of modifications to an existing list. However, the modifications only take effect if no exceptions occur. Otherwise, the original list is left unmodified. For example:

```
items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)       # Produces [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("We're hosed!")
except RuntimeError:
    pass
print(items)   # Produces [1,2,3,4,5]
```

The `contextlib` module allows custom context managers to be more easily implemented by placing a wrapper around a generator function. Here is an example:

```
from contextlib import contextmanager
@contextmanager
def ListTransaction(thelist):
    workingcopy = list(thelist)
    yield workingcopy
    # Modify the original list only if no errors
    thelist[:] = workingcopy
```

In this example, the value passed to `yield` is used as the return value from `__enter__()`. When the `__exit__()` method gets invoked, execution resumes after the `yield`. If an exception gets raised in the context, it shows up as an exception in the generator function. If desired, an exception could be caught, but in this case, exceptions will simply propagate out of the generator to be handled elsewhere.

Assertions and `__debug__`

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

```
assert test [, msg]
```

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 message *msg* supplied to the `assert` statement. Here's an example:

```
def write_data(file,data):
    assert file, "write_data: file not defined!"
    ...
```

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 (specified with the `-O` option to the interpreter). 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.

In addition to `assert`, Python provides the built-in read-only variable `__debug__`, which is set to `True` 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 underlying implementation of the `__debug__` variable is optimized in the interpreter so that the extra control-flow logic of the `if` statement itself is not actually included. If Python is running in its normal mode, the statements under the `if __debug__` statement are just inlined into the program without the `if` statement itself. In optimized mode, the `if __debug__` statement and all associated statements are completely removed from the program.

The use of `assert` and `__debug__` allow for efficient dual-mode development of a program. For example, in debug mode, you can liberally instrument your code with assertions and debug checks to verify correct operation. In optimized mode, all of these extra checks get stripped, resulting in no extra performance penalty.

Functions and Functional Programming

Substantial programs are broken up into functions for better modularity and ease of maintenance. Python makes it easy to define functions but also incorporates a surprising number of features from functional programming languages. This chapter describes functions, scoping rules, closures, decorators, generators, coroutines, and other functional programming features. In addition, list comprehensions and generator expressions are described—both of which are powerful tools for declarative-style programming and data processing.

Functions

Functions are defined with the `def` statement:

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

The body of a function is simply a sequence of statements that execute when the function is called. 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 split(line,delimiter=','):
    statements
```

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. Here's an example:

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

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

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

```
def foo(x, items=[]):
    items.append(x)
    return items
foo(1)       # returns [1]
foo(2)       # returns [1, 2]
foo(3)       # returns [1, 2, 3]
```

Notice how the default argument retains modifications made from previous invocations. To prevent this, it is better to use `None` and add a check as follows:

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

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 a tuple *args* to a function as if they were parameters, the `*args` syntax can be used in a function call as follows:

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

Function arguments can also be supplied by explicitly naming each parameter and specifying a value. These are known as *keyword arguments*. Here is an example:

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

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

With keyword arguments, the order of the parameters doesn't matter. However, unless there are default values, you must explicitly name all of the required 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. Also, since any Python function can be called using the keyword calling style, it is generally a good idea to define functions with descriptive argument names.

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. Here's an 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 other parameter names) are placed in a dictionary and passed to the function. This can be a useful way to write functions that accept a large number of potentially open-ended configuration options that would be too unwieldy to list as parameters. Here's an example:

```
def make_table(data, **parms):
    # Get configuration parameters from parms (a dict)
    fgcolor = parms.pop("fgcolor","black")
    bgcolor = parms.pop("bgcolor","white")
    width = parms.pop("width",None)
    ...
    # No more options
    if parms:
         raise TypeError("Unsupported configuration options %s" % list(parms))

make_table(items, fgcolor="black", bgcolor="white", border=1,
                  borderstyle="grooved", cellpadding=10,
                  width=400)
```

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(*args, **kwargs):
    # args is a tuple of positional args
    # kwargs is dictionary of keyword args
    ...
```

Keyword arguments can also be passed to another function using the **kwargs syntax:

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

This use of *args and **kwargs is commonly used to write wrappers and proxies for other functions. For example, the callfunc() accepts any combination of arguments and simply passes them through to func().

Parameter Passing and Return Values

When a function is invoked, the function parameters are simply names that refer to the passed input objects. The underlying semantics of parameter passing doesn't neatly fit into any single style, such as "pass by value" or "pass by reference," that you might know about from other programming languages. For example, if you pass an immutable value, the argument effectively looks like it was passed by value. However, 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 original object. Here's an example:

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i,x in enumerate(items):
        items[i] = x * x     # Modify items in-place

square(a)      # Changes a to [1, 4, 9, 16, 25]
```

Functions that mutate their input values or change the state of other parts of the program behind the scenes like this are said to have *side effects*. As a general rule, this is a programming style that is best avoided because such functions can become a source of subtle programming errors as programs grow in size and complexity (for example, it's not obvious from reading a function call if a function has side effects). Such functions interact poorly with programs involving threads and concurrency because side effects typically need to be protected by locks.

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.
```

Scoping Rules

Each time a function executes, a new local namespace is created. This namespace represents a local environment that contains the names of the function parameters, as well as the names of variables that are assigned inside the function body. When resolving names, the interpreter first searches the local namespace. If no match exists, it searches the global namespace. The global namespace for a function is always the module in which the function was defined. If the interpreter finds no match in the global namespace, it makes a final check in the built-in namespace. If this fails, a NameError exception is raised.

One peculiarity of namespaces is the manipulation of global variables within a function. For example, consider the following code:

```
a = 42
def foo():
    a = 13
foo()
# a is still 42
```

When this code executes, a returns its value of 42, despite the appearance that we might be modifying the variable a inside the function foo. When variables are assigned inside a function, they're always bound to the function's local namespace; as a result, the variable a in the function body refers to an entirely new object containing the value 13, not the outer variable. To alter this behavior, use the global statement. global simply declares names as belonging to the global namespace, and it's necessary only when global variables will be modified. It can be placed anywhere in a function body and used repeatedly. Here's an example:

```
a = 42
b = 37
def foo():
    global a       # 'a' is in global namespace
    a = 13
    b = 0
foo()
# a is now 13. b is still 37.
```

Python supports nested function definitions. Here's an example:

```
def countdown(start):
   n = start
   def display():            # Nested function definition
       print('T-minus %d' % n)
   while n > 0:
       display()
       n -= 1
```

Variables in nested functions are bound using *lexical scoping*. That is, names are resolved by first checking the local scope and then all enclosing scopes of outer function definitions from the innermost scope to the outermost scope. If no match is found, the global and built-in namespaces are checked as before. Although names in enclosing scopes are accessible, Python 2 only allows variables to be reassigned in the innermost scope (local variables) and the global namespace (using global). Therefore, an inner function can't reassign the value of a local variable defined in an outer function. For example, this code does not work:

```
def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       n -= 1             # Fails in Python 2
   while n > 0:
        display()
        decrement()
```

In Python 2, you can work around this by placing values you want to change in a list or dictionary. In Python 3, you can declare n as nonlocal as follows:

```
def countdown(start):
   n = start
   def display():
       print('T-minus %d' % n)
   def decrement():
       nonlocal n    # Bind to outer n (Python 3 only)
       n -= 1
   while n > 0:
        display()
        decrement()
```

The nonlocal declaration does not bind a name to local variables defined inside arbitrary functions further down on the current call-stack (that is, *dynamic scope*). So, if you're coming to Python from Perl, nonlocal is not the same as declaring a Perl local variable.

If a local variable is used before it's assigned a value, an UnboundLocalError exception is raised. Here's an example that illustrates one scenario of how this might occur:

```
i = 0
def foo():
    i = i + 1      # Results in UnboundLocalError exception
    print(i)
```

In this function, the variable i is defined as a local variable (because it is being assigned inside the function and there is no global statement). However, the assignment i = i + 1 tries to read the value of i before its local value has been first assigned. Even though there is a global variable i in this example, it is not used to supply a value here. Variables are determined to be either local or global at the time of function definition and cannot suddenly change scope in the middle of a function. For example, in the preceding code, it is not the case that the i in the expression i + 1 refers to the global variable i, whereas the i in print(i) refers to the local variable i created in the previous statement.

Functions as Objects and Closures

Functions are first-class objects in Python. This means that they can be passed as arguments to other functions, placed in data structures, and returned by a function as a result. Here is an example of a function that accepts another function as input and calls it:

```
# foo.py
def callf(func):
   return func()
```

Here is an example of using the above function:

```
>>> import foo
>>> def helloworld():
...     return 'Hello World'
...
>>> foo.callf(helloworld)     # Pass a function as an argument
'Hello World'
>>>
```

When a function is handled as data, it implicitly carries information related to the surrounding environment where the function was defined. This affects how free variables in the function are bound. As an example, consider this modified version foo.py that now contains a variable definition:

```
# foo.py
x = 42
def callf(func):
   return func()
```

Now, observe the behavior of this example:

```
>>> import foo
>>> x = 37
>>> def helloworld():
...     return "Hello World. x is %d" % x
...
>>> foo.callf(helloworld)     # Pass a function as an argument
'Hello World. x is 37'
>>>
```

In this example, notice how the function `helloworld()` uses the value of `x` that's defined in the same environment as where `helloworld()` was defined. Thus, even though there is also an `x` defined in `foo.py` and that's where `helloworld()` is actually being called, that value of `x` is not the one that's used when `helloworld()` executes.

When the statements that make up a function are packaged together with the environment in which they execute, the resulting object is known as a *closure*. The behavior of the previous example is explained by the fact that all functions have a `__globals__` attribute that points to the global namespace in which the function was defined. This always corresponds to the enclosing module in which a function was defined. For the previous example, you get the following:

```
>>> helloworld.__globals__
{'__builtins__': <module '__builtin__' (built-in)>,
 'helloworld': <function helloworld at 0x7bb30>,
 'x': 37, '__name__': '__main__', '__doc__': None
 'foo': <module 'foo' from 'foo.py'>}
>>>
```

When nested functions are used, closures capture the entire environment needed for the inner function to execute. Here is an example:

```
import foo
def bar():
    x = 13
    def helloworld():
        return "Hello World. x is %d" % x
    foo.callf(helloworld)        # returns 'Hello World, x is 13'
```

Closures and nested functions are especially useful if you want to write code based on the concept of lazy or delayed evaluation. Here is another example:

```
from urllib import urlopen
# from urllib.request import urlopen (Python 3)
def page(url):
    def get():
        return urlopen(url).read()
    return get
```

In this example, the `page()` function doesn't actually carry out any interesting computation. Instead, it merely creates and returns a function `get()` that will fetch the contents of a web page when it is called. Thus, the computation carried out in `get()` is actually delayed until some later point in a program when `get()` is evaluated. For example:

```
>>> python = page("http://www.python.org")
>>> jython = page("http://www.jython.org")
>>> python
<function get at 0x95d5f0>
>>> jython
<function get at 0x9735f0>
>>> pydata = python()          # Fetches http://www.python.org
>>> jydata = jython()          # Fetches http://www.jython.org
>>>
```

In this example, the two variables `python` and `jython` are actually two different versions of the `get()` function. Even though the `page()` function that created these values is no longer executing, both `get()` functions implicitly carry the values of the outer variables that were defined when the `get()` function was created. Thus, when `get()` executes, it calls `urlopen(url)` with the value of `url` that was originally supplied to `page()`. With a little inspection, you can view the contents of variables that are carried along in a closure. For example:

```
>>> python.__closure__
(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents
'http://www.python.org'
>>> jython.__closure__[0].cell_contents
'http://www.jython.org'
>>>
```

A closure can be a highly efficient way to preserve state across a series of function calls. For example, consider this code that runs a simple counter:

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next

# Example use
next = countdown(10)
while True:
    v = next()          # Get the next value
    if not v: break
```

In this code, a closure is being used to store the internal counter value `n`. The inner function `next()` updates and returns the previous value of this counter variable each time it is called. Programmers not familiar with closures might be inclined to implement similar functionality using a class such as this:

```
class Countdown(object):
    def __init__(self,n):
        self.n = n
    def next(self):
        r = self.n
        self.n -= 1
        return r

# Example use
c = Countdown(10)
while True:
    v = c.next()          # Get the next value
    if not v: break
```

However, if you increase the starting value of the countdown and perform a simple timing benchmark, you will find that that the version using closures runs much faster (almost a 50% speedup when tested on the author's machine).

The fact that closures capture the environment of inner functions also make them useful for applications where you want to wrap existing functions in order to add extra capabilities. This is described next.

Decorators

A *decorator* is a function whose primary purpose is to wrap another function or class. The primary purpose of this wrapping is to transparently alter or enhance the behavior of the object being wrapped. Syntactically, decorators are denoted using the special `@` symbol as follows:

```
@trace
def square(x):
     return x*x
```

The preceding code is shorthand for the following:

```
def square(x):
    return x*x
square = trace(square)
```

In the example, a function `square()` is defined. However, immediately after its definition, the function object itself is passed to the function `trace()`, which returns an object that replaces the original `square`. Now, let's consider an implementation of `trace` that will clarify how this might be useful:

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s returned %s\n" % (func.__name, r))
            return r
        return callf
    else:
        return func
```

In this code, `trace()` creates a wrapper function that writes some debugging output and then calls the original function object. Thus, if you call `square()`, you will see the output of the `write()` methods in the wrapper. The function `callf` that is returned from `trace()` is a closure that serves as a replacement for the original function. A final interesting aspect of the implementation is that the tracing feature itself is only enabled through the use of a global variable `enable_tracing` as shown. If set to `False`, the `trace()` decorator simply returns the original function unmodified. Thus, when tracing is disabled, there is no added performance penalty associated with using the decorator.

When decorators are used, they must appear on their own line immediately prior to a function or class definition. More than one decorator can also be applied. Here's an example:

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

In this case, the decorators are applied in the order listed. The result is the same as this:

```
def grok(x):
    pass
grok = foo(bar(spam(grok)))
```

A decorator can also accept arguments. Here's an example:

```
@eventhandler('BUTTON')
def handle_button(msg):
     ...
@eventhandler('RESET')
def handle_reset(msg):
     ...
```

If arguments are supplied, the semantics of the decorator are as follows:

```
def handle_button(msg):
    ...
temp = eventhandler('BUTTON')        # Call decorator with supplied arguments
handle_button = temp(handle_button)  # Call the function returned by the decorator
```

In this case, the decorator function only accepts the arguments supplied with the `@` specifier. It then returns a function that is called with the function as an argument. Here's an example:

```
# Event handler decorator
event_handlers = { }
def eventhandler(event):
     def register_function(f):
          event_handlers[event] = f
          return f
     return register_function
```

Decorators can also be applied to class definitions. For example:

```
@foo
class Bar(object):
    def __init__(self,x):
        self.x = x
    def spam(self):
        statements
```

For class decorators, you should always have the decorator function return a class object as a result. Code that expects to work with the original class definition may want to reference members of the class directly such as `Bar.spam`. This won't work correctly if the decorator function `foo()` returns a function.

Decorators can interact strangely with other aspects of functions such as recursion, documentation strings, and function attributes. These issues are described later in this chapter.

Generators and `yield`

If a function uses the `yield` keyword, it defines an object known as a *generator*. A generator is a function that produces a sequence of values for use in iteration. Here's an example:

```
def countdown(n):
     print("Counting down from %d" % n)
     while n > 0:
          yield n
          n -= 1
      return
```

If you call this function, you will find that none of its code starts executing. For example:

```
>>> c = countdown(10)
>>>
```

Instead, a generator object is returned. The generator object, in turn, executes the function whenever `next()` is called (or `__next__()` in Python 3). Here's an example:

```
>>> c.next()            # Use c.__next__() in Python 3
Counting down from 10
10
>>> c.next()
9
```

When `next()` is invoked, the generator function executes statements until it reaches a `yield` statement. The `yield` statement produces a result at which point execution of the function stops until `next()` is invoked again. Execution then resumes with the statement following `yield`.

You normally don't call `next()` directly on a generator but use it with the `for` statement, `sum()`, or some other operation that consumes a sequence. For example:

```
for n in countdown(10):
     statements
a = sum(countdown(10))
```

A generator function signals completion by returning or raising `StopIteration`, at which point iteration stops. It is never legal for a generator to return a value other than `None` upon completion.

A subtle problem with generators concerns the case where a generator function is only partially consumed. For example, consider this code:

```
for n in countdown(10):
     if n == 2: break
     statements
```

In this example, the `for` loop aborts by calling `break`, and the associated generator never runs to full completion. To handle this case, generator objects have a method `close()` that is used to signal a shutdown. When a generator is no longer used or deleted, `close()` is called. Normally it is not necessary to call `close()`, but you can also call it manually as shown here:

```
>>> c = countdown(10)
>>> c.next()
Counting down from 10
10
>>> c.next()
9
>>> c.close()
>>> c.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Inside the generator function, `close()` is signaled by a `GeneratorExit` exception occurring on the `yield` statement. You can optionally catch this exception to perform cleanup actions.

```
def countdown(n):
    print("Counting down from %d" % n)
    try:
         while n > 0:
              yield n
              n = n - 1
    except GeneratorExit:
         print("Only made it to %d" % n)
```

Although it is possible to catch `GeneratorExit`, it is illegal for a generator function to handle the exception and produce another output value using `yield`. Moreover, if a program is currently iterating on generator, you should not call `close()` asynchronously on that generator from a separate thread of execution or from a signal handler.

Coroutines and `yield` Expressions

Inside a function, the `yield` statement can also be used as an expression that appears on the right side of an assignment operator. For example:

```
def receiver():
    print("Ready to receive")
    while True:
          n = (yield)
          print("Got %s" % n)
```

A function that uses `yield` in this manner is known as a *coroutine*, and it executes in response to values being sent to it. Its behavior is also very similar to a generator. For example:

```
>>> r = receiver()
>>> r.next()  # Advance to first yield (r.__next__() in Python 3)
Ready to receive
>>> r.send(1)
Got 1
>>> r.send(2)
Got 2
>>> r.send("Hello")
Got Hello
>>>
```

In this example, the initial call to `next()` is necessary so that the coroutine executes statements leading to the first `yield` expression. At this point, the coroutine suspends, waiting for a value to be sent to it using the `send()` method of the associated generator object `r`. The value passed to `send()` is returned by the `(yield)` expression in the coroutine. Upon receiving a value, a coroutine executes statements until the next `yield` statement is encountered.

The requirement of first calling `next()` on a coroutine is easily overlooked and a common source of errors. Therefore, it is recommended that coroutines be wrapped with a decorator that automatically takes care of this step.

```
def coroutine(func):
    def start(*args,**kwargs):
        g = func(*args,**kwargs)
        g.next()
        return g
    return start
```

Using this decorator, you would write and use coroutines using:

```
@coroutine
def receiver():
    print("Ready to receive")
    while True:
          n = (yield)
          print("Got %s" % n)
# Example use
r = receiver()
r.send("Hello World")       # Note : No initial .next() needed
```

A coroutine will typically run indefinitely unless it is explicitly shut down or it exits on its own. To close the stream of input values, use the `close()` method like this:

```
>>> r.close()
>>> r.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Once closed, a `StopIteration` exception will be raised if further values are sent to a coroutine. The `close()` operation raises `GeneratorExit` inside the coroutine as described in the previous section on generators. For example:

```
def receiver():
    print("Ready to receive")
    try:
        while True:
            n = (yield)
            print("Got %s" % n)
    except GeneratorExit:
        print("Receiver done")
```

Exceptions can be raised inside a coroutine using the `throw(`*`exctype`*` [, `*`value`*` [, `*`tb`*`]])` method where *`exctype`* is an exception type, *`value`* is the exception value, and *`tb`* is a traceback object. For example:

```
>>> r.throw(RuntimeError,"You're hosed!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in receiver
RuntimeError: You're hosed!
```

Exceptions raised in this manner will originate at the currently executing `yield` statement in the coroutine. A coroutine can elect to catch exceptions and handle them as appropriate. It is not safe to use `throw()` as an asynchronous signal to a coroutine—it should never be invoked from a separate execution thread or in a signal handler.

A coroutine may simultaneously receive and emit return values using `yield` if values are supplied in the `yield` expression. Here is an example that illustrates this:

```
def line_splitter(delimiter=None):
    print("Ready to split")
    result = None
    while True:
        line = (yield result)
        result = line.split(delimiter)
```

In this case, we use the coroutine in the same way as before. However, now calls to `send()` also produce a result. For example:

```
>>> s = line_splitter(",")
>>> s.next()
Ready to split
>>> s.send("A,B,C")
['A', 'B', 'C' ]
>>> s.send("100,200,300")
['100', '200', '300']
>>>
```

Understanding the sequencing of this example is critical. The first `next()` call advances the coroutine to `(yield result)`, which returns `None`, the initial value of `result`. On subsequent `send()` calls, the received value is placed in `line` and split into `result`. The value returned by `send()` is the value passed to the next `yield` statement encountered. In other words, the value returned by `send()` comes from the next `yield` expression, not the one responsible for receiving the value passed by `send()`.

If a coroutine returns values, some care is required if exceptions raised with `throw()` are being handled. If you raise an exception in a coroutine using `throw()`, the value passed to the next `yield` in the coroutine will be returned as the result of `throw()`. If you need this value and forget to save it, it will be lost.

Using Generators and Coroutines

At first glance, it might not be obvious how to use generators and coroutines for practical problems. However, generators and coroutines can be particularly effective when applied to certain kinds of programming problems in systems, networking, and distributed computation. For example, generator functions are useful if you want to set up a processing pipeline, similar in nature to using a pipe in the UNIX shell. One example of this appeared in the Introduction. Here is another example involving a set of generator functions related to finding, opening, reading, and processing files:

```
import os
import fnmatch

def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
          for name in filelist:
              if fnmatch.fnmatch(name, pattern):
                  yield os.path.join(path,name)

import gzip, bz2
def opener(filenames):
    for name in filenames:
        if name.endswith(".gz"):  f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        yield f

def cat(filelist):
    for f in filelist:
        for line in f:
            yield line

def grep(pattern, lines):
    for line in lines:
        if pattern in line:
            yield line
```

Here is an example of using these functions to set up a processing pipeline:

```
wwwlogs = find("www","access-log*")
files   = opener(wwwlogs)
lines   = cat(files)
pylines = grep("python", lines)
for line in pylines:
    sys.stdout.write(line)
```

In this example, the program is processing all lines in all `"access-log*"` files found within all subdirectories of a top-level directory `"www"`. Each `"access-log"` is tested for file compression and opened using an appropriate file opener. Lines are concatenated together and processed through a filter that is looking for a substring `"python"`. The entire program is being driven by the `for` statement at the end. Each iteration of this loop pulls a new value through the pipeline and consumes it. Moreover, the implementation is highly memory-efficient because no temporary lists or other large data structures are ever created.

Coroutines can be used to write programs based on data-flow processing. Programs organized in this way look like inverted pipelines. Instead of pulling values through a sequence of generator functions using a `for` loop, you send values into a collection of linked coroutines. Here is an example of coroutine functions written to mimic the generator functions shown previously:

```
import os
import fnmatch

@coroutine
def find_files(target):
    while True:
        topdir, pattern = (yield)
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name,pattern):
                    target.send(os.path.join(path,name))

import gzip, bz2
@coroutine
def opener(target):
    while True:
        name = (yield)
        if name.endswith(".gz"):  f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        target.send(f)

@coroutine
def cat(target):
    while True:
        f = (yield)
        for line in f:
            target.send(line)

@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)

@coroutine
def printer():
    while True:
        line = (yield)
        sys.stdout.write(line)
```

Here is how you would link these coroutines to create a dataflow processing pipeline:

```
finder = find_files(opener(cat(grep("python",printer()))))

# Now, send a value
finder.send(("www","access-log*"))
finder.send(("otherwww","access-log*"))
```

In this example, each coroutine sends data to another coroutine specified in the `target` argument to each coroutine. Unlike the generator example, execution is entirely driven by pushing data into the first coroutine `find_files()`. This coroutine, in turn, pushes data to the next stage. A critical aspect of this example is that the coroutine pipeline remains active indefinitely or until `close()` is explicitly called on it. Because of this, a program can continue to feed data into a coroutine for as long as necessary—for example, the two repeated calls to `send()` shown in the example.

Coroutines can be used to implement a form of concurrency. For example, a centralized task manager or event loop can schedule and send data into a large collection of hundreds or even thousands of coroutines that carry out various processing tasks. The fact that input data is "sent" to a coroutine also means that coroutines can often be easily mixed with programs that use message queues and message passing to communicate between program components. Further information on this can be found in Chapter 20, "Threads."

List Comprehensions

A common operation involving functions is that of applying a function to all of the items of a list, creating a new list with the results. For example:

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)
```

Because this type of operation is so common, it is has been turned into an operator known as a *list comprehension*. Here is a simple example:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

The general syntax for a list comprehension is as follows:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]
```

This syntax is roughly equivalent to the following code:

```
s = []
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                for itemN in iterableN:
                    if conditionN: s.append(expression)
```

To illustrate, here are some more examples:

```
a = [-3,5,2,-10,7,8]
b = 'abc'

c = [2*s for s in a]           # c = [-6,10,4,-20,14,16]
d = [s for s in a if s >= 0]   # d = [5,2,7,8]
e = [(x,y) for x in a          # e = [(5,'a'),(5,'b'),(5,'c'),
           for y in b          #      (2,'a'),(2,'b'),(2,'c'),
           if x > 0 ]          #      (7,'a'),(7,'b'),(7,'c'),
                               #      (8,'a'),(8,'b'),(8,'c')]

f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x+y*y)        # f = [2.23606, 5.0, 7.81024]
     for x,y in f]
```

The sequences supplied to a list comprehension don't have to be the same length because they're iterated over their contents using a nested set of `for` loops, as previously shown. The resulting list contains successive values of expressions. The `if` clause is optional; however, if it's used, *`expression`* is evaluated and added to the result only if *`condition`* is true.

If a list comprehension is used to construct a list of tuples, the tuple values must be enclosed in parentheses. For example, `[(x,y) for x in a for y in b]` is legal syntax, whereas `[x,y for x in a for y in b]` is not.

Finally, it is important to note that in Python 2, the iteration variables defined within a list comprehension are evaluated within the current scope and remain defined after the list comprehension has executed. For example, in `[x for x in a]`, the iteration variable `x` overwrites any previously defined value of `x` and is set to the value of the last item in `a` after the resulting list is created. Fortunately, this is not the case in Python 3 where the iteration variable remains private.

Generator Expressions

A *generator expression* is an object that carries out the same computation as a list comprehension, but which iteratively produces the result. The syntax is the same as for list comprehensions except that you use parentheses instead of square brackets. Here's an example:

```
(expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN)
```

Unlike a list comprehension, a generator expression does not actually create a list or immediately evaluate the expression inside the parentheses. Instead, it creates a generator object that produces the values on demand via iteration. Here's an example:

```
>>> a = [1, 2, 3, 4]
>>> b = (10*i for i in a)
>>> b
<generator object at 0x590a8>
>>> b.next()
10
>>> b.next()
20
...
```

The difference between list and generator expressions is important, but subtle. With a list comprehension, Python actually creates a list that contains the resulting data. With a generator expression, Python creates a generator that merely knows how to produce data on demand. In certain applications, this can greatly improve performance and memory use. Here's an example:

```
# Read a file
f = open("data.txt")                        # Open a file
lines = (t.strip() for t in f)              # Read lines, strip
                                            # trailing/leading whitespace
comments = (t for t in lines if t[0] == '#') # All comments
for c in comments:
     print(c)
```

In this example, the generator expression that extracts lines and strips whitespace does not actually read the entire file into memory. The same is true of the expression that extracts comments. Instead, the lines of the file are actually read when the program starts iterating in the `for` loop that follows. During this iteration, the lines of the file are produced upon demand and filtered accordingly. In fact, at no time will the entire file be loaded into memory during this process. Therefore, this would be a highly efficient way to extract comments from a gigabyte-sized Python source file.

Unlike a list comprehension, a generator expression does not create an object that works like a sequence. It can't be indexed, and none of the usual list operations will work (for example, `append()`). However, a generator expression can be converted into a list using the built-in `list()` function:

```
clist = list(comments)
```

Declarative Programming

List comprehensions and generator expressions are strongly tied to operations found in declarative languages. In fact, the origin of these features is loosely derived from ideas in mathematical set theory. For example, when you write a statement such as `[x*x for x in a if x > 0]`, it's somewhat similar to specifying a set such as $\{ x^2 \mid x \in a, x > 0 \}$.

Instead of writing programs that manually iterate over data, you can use these declarative features to structure programs as a series of computations that simply operate on all of the data all at once. For example, suppose you had a file "portfolio.txt" containing stock portfolio data like this:

```
AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
```

Here is a declarative-style program that calculates the total cost by summing up the second column multiplied by the third column:

```
lines  = open("portfolio.txt")
fields = (line.split() for line in lines)
print(sum(float(f[1]) * float(f[2]) for f in fields))
```

In this program, we really aren't concerned with the mechanics of looping line-by-line over the file. Instead, we just declare a sequence of calculations to perform on all of the data. Not only does this approach result in highly compact code, but it also tends to run faster than this more traditional version:

```
total = 0
for line in open("portfolio.txt"):
    fields = line.split()
    total += float(fields[1]) * float(fields[2])
print(total)
```

The declarative programming style is somewhat tied to the kinds of operations a programmer might perform in a UNIX shell. For instance, the preceding example using generator expressions is similar to the following one-line `awk` command:

```
% awk '{ total += $2 * $3} END { print total }' portfolio.txt
44671.2
%
```

The declarative style of list comprehensions and generator expressions can also be used to mimic the behavior of SQL `select` statements, commonly used when processing databases. For example, consider these examples that work on data that has been read in a list of dictionaries:

```
fields = (line.split() for line in open("portfolio.txt"))
portfolio = [ {'name'   : f[0],
               'shares' : int(f[1]),
               'price'  : float(f[2]) }
              for f in fields]

# Some queries
msft = [s for s in portfolio if s['name'] == 'MSFT']
large_holdings = [s for s in portfolio
                        if s['shares']*s['price'] >= 10000]
```

In fact, if you are using a module related to database access (see Chapter 17), you can often use list comprehensions and database queries together all at once. For example:

```
sum(shares*cost for shares,cost in
         cursor.execute("select shares, cost from portfolio")
             if shares*cost >= 10000)
```

The `lambda` Operator

Anonymous functions in the form of an expression can be created using the `lambda` statement:

```
lambda args : expression
```

`args` is a comma-separated list of arguments, and *`expression`* is an expression involving those arguments. Here's an example:

```
a = lambda x,y : x+y
r = a(2,3)              # r gets 5
```

The code defined with `lambda` must be a valid expression. Multiple statements and other non-expression statements, such as `for` and `while`, cannot appear in a `lambda` statement. `lambda` expressions follow the same scoping rules as functions.

The primary use of `lambda` is in specifying short callback functions. For example, if you wanted to sort a list of names with case-insensitivity, you might write this:

```
names.sort(key=lambda n: n.lower())
```

Recursion

Recursive functions are easily defined. For example:

```
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)
```

However, be aware that there is a limit on the depth of recursive function calls. The function `sys.getrecursionlimit()` returns the current maximum recursion depth, and the function `sys.setrecursionlimit()` can be used to change the value. The default value is `1000`. Although it is possible to increase the value, programs are still limited by the stack size limits enforced by the host operating system. When the recursion depth is exceeded, a `RuntimeError` exception is raised. Python does not perform tail-recursion optimization that you often find in functional languages such as Scheme.

Recursion does not work as you might expect in generator functions and coroutines. For example, this code prints all items in a nested collection of lists:

```
def flatten(lists):
    for s in lists:
        if isinstance(s,list):
            flatten(s)
        else:
            print(s)

items = [[1,2,3],[4,5,[5,6]],[7,8,9]]
flatten(items)     # Prints 1 2 3 4 5 6 7 8 9
```

However, if you change the `print` operation to a `yield`, it no longer works. This is because the recursive call to `flatten()` merely creates a new generator object without actually iterating over it. Here's a recursive generator version that works:

```
def genflatten(lists):
    for s in lists:
        if isinstance(s,list):
            for item in genflatten(s):
                yield item
        else:
            yield item
```

Care should also be taken when mixing recursive functions and decorators. If a decorator is applied to a recursive function, all inner recursive calls now get routed through the decorated version. For example:

```
@locked
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)  # Calls the wrapped version of factorial
```

If the purpose of the decorator was related to some kind of system management such as synchronization or locking, recursion is something probably best avoided.

Documentation Strings

It is common practice for the first statement of function to be a documentation string describing its usage. For example:

```
def factorial(n):
    """Computes n factorial. For example:

       >>> factorial(6)
       120
       >>>
    """
    if n <= 1: return 1
    else: return n*factorial(n-1)
```

The documentation string is stored in the `__doc__` attribute of the function that is commonly used by IDEs to provide interactive help.

If you are using decorators, be aware that wrapping a function with a decorator can break the help features associated with documentation strings. For example, consider this code:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
@wrap
def factorial(n):
    """Computes n factorial."""
    ...
```

If a user requests help on this version of `factorial()`, he will get a rather cryptic explanation:

```
>>> help(factorial)
Help on function call in module __main__:

call(*args, **kwargs)
(END)
>>>
```

To fix this, write decorator functions so that they propagate the function name and documentation string. For example:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    return call
```

Because this is a common problem, the `functools` module provides a function `wraps` that can automatically copy these attributes. Not surprisingly, it is also a decorator:

```
from functools import wraps
def wrap(func):
    @wraps(func)
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
```

The `@wraps(`*`func`*`)` decorator, defined in `functools`, propagates attributes from *`func`* to the wrapper function that is being defined.

Function Attributes

Functions can have arbitrary attributes attached to them. Here's an example:

```
def foo():
    statements

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 highly specialized applications such as parser generators and application frameworks that would like to attach additional information to function objects.

As with documentation strings, care should be given if mixing function attributes with decorators. If a function is wrapped by a decorator, access to the attributes will actually take place on the decorator function, not the original implementation. This may or may not be what you want depending on the application. To propagate already defined function attributes to a decorator function, use the following template or the `functools.wraps()` decorator as shown in the previous section:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    call.__dict__.update(func.__dict__)
    return call
```

`eval()`, `exec()`, and `compile()`

The `eval(str [,globals [,locals]])` function executes an expression string and returns the result. Here's an example:

```
a = eval('3*math.sin(3.5+x) + 7.2')
```

Similarly, the `exec(str [, globals [, locals]])` function executes a string containing arbitrary Python code. The code supplied to `exec()` is executed as if the code actually appeared in place of the `exec` operation. Here's an example:

```
a = [3, 5, 10, 13]
exec("for i in a: print(i)")
```

One caution with `exec` is that in Python 2, `exec` is actually defined as a statement. Thus, in legacy code, you might see statements invoking `exec` without the surrounding parentheses, such as `exec "for i in a: print i"`. Although this still works in Python 2.6, it breaks in Python 3. Modern programs should use `exec()` as a function.

Both of these functions execute within the namespace of the caller (which is used to resolve any symbols that appear within a string or file). Optionally, `eval()` and `exec()` can accept one or two mapping objects that serve as the global and local namespaces for the code to be executed, respectively. Here's an example:

```
globals = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
          }
locals = { }

# Execute using the above dictionaries as the global and local namespace
a = eval("3 * x + 4 * y", globals, locals)
exec("for b in birds: print(b)", globals, locals)
```

If you omit one or both namespaces, the current values of the global and local namespaces are used. Also, due to issues related to nested scopes, the use of `exec()` inside of a function body may result in a `SyntaxError` exception if that function also contains nested function definitions or uses the `lambda` operator.

When a string is passed to `exec()` or `eval()` the parser first compiles it into bytecode. Because this process is expensive, it may be better to precompile the code and reuse the bytecode on subsequent calls if the code will be executed multiple times.

The `compile(str,filename,kind)` function compiles a string into bytecode in which *`str`* is a string containing the code to be compiled and *`filename`* is the file in which the string is defined (for use in traceback generation). The *`kind`* argument specifies the type of code being compiled—`'single'` for a single statement, `'exec'` for a set of statements, or `'eval'` for an expression. The code object returned by the `compile()` function can also be passed to the `eval()` function and `exec()` statement. Here's an example:

```
s = "for i in range(0,10): print(i)"
c = compile(s,'','exec')      # Compile into a code object
exec(c)                       # Execute it

s2 = "3 * x + 4 * y"
c2 = compile(s2, '', 'eval')  # Compile into an expression
result = eval(c2)             # Execute it
```

7

Classes and Object-Oriented Programming

Classes are the mechanism used to create new kinds of objects. This chapter covers the details of classes, but is not intended to be an in-depth reference on object-oriented programming and design. It's assumed that the reader has some prior experience with data structures and object-oriented programming in other languages such as C or Java. (Chapter 3, "Types and Objects," contains additional information about the terminology and internal implementation of objects.)

The `class` Statement

A *class* defines a set of attributes that are associated with, and shared by, a collection of objects known as *instances*. A class is most commonly a collection of functions (known as *methods*), variables (which are known as *class variables*), and computed attributes (which are known as *properties*).

A class is defined using the `class` statement. The body of a class contains a series of statements that execute during class definition. Here's an example:

```
class Account(object):
     num_accounts = 0
     def __init__(self,name,balance):
         self.name = name
         self.balance = balance
         Account.num_accounts += 1
     def __del__(self):
         Account.num_accounts -= 1
     def deposit(self,amt):
         self.balance = self.balance + amt
     def withdraw(self,amt):
         self.balance = self.balance - amt
     def inquiry(self):
         return self.balance
```

The values created during the execution of the class body are placed into a class object that serves as a namespace much like a module. For example, the members of the `Account` class are accessed as follows:

```
Account.num_accounts
Account.__init__
Account.__del__
Account.deposit
Account.withdraw
Account.inquiry
```

It's important to note that a `class` statement by itself doesn't create any instances of the class (for example, no accounts are actually created in the preceding example). Rather, a class merely sets up the attributes that will be common to all the instances that will be created later. In this sense, you might think of it as a blueprint.

The functions defined inside a class are known as *instance methods*. An instance method is a function that operates on an instance of the class, which is passed as the first argument. By convention, this argument is called `self`, although any legal identifier name can be used. In the preceding example, `deposit()`, `withdraw()`, and `inquiry()` are examples of instance methods.

Class variables such as `num_accounts` are values that are shared among all instances of a class (that is, they're not individually assigned to each instance). In this case, it's a variable that's keeping track of how many `Account` instances are in existence.

Class Instances

Instances of a class are created by calling a class object as a function. This creates a new instance that is then passed to the `__init__()` method of the class. The arguments to `__init__()` consist of the newly created instance `self` along with the arguments supplied when calling the class object. For example:

```
# Create a few accounts
a = Account("Guido", 1000.00)  # Invokes Account.__init__(a,"Guido",1000.00)
b = Account("Bill", 10.00)
```

Inside `__init__()`, attributes are saved in the instance by assigning to `self`. For example, `self.name = name` is saving a `name` attribute in the instance. Once the newly created instance has been returned to the user, these attributes as well as attributes of the class are accessed using the dot (.) operator as follows:

```
a.deposit(100.00)        # Calls Account.deposit(a,100.00)
b.withdraw(50.00)        # Calls Account.withdraw(b,50.00)
name = a.name            # Get account name
```

The dot (.) operator is responsible for attribute binding. When you access an attribute, the resulting value may come from several different places. For example, `a.name` in the previous example returns the `name` attribute of the instance `a`. However, `a.deposit` returns the `deposit` attribute (a method) of the `Account` class. When you access an attribute, the instance is checked first and if nothing is known, the search moves to the instance's class instead. This is the underlying mechanism by which a class shares its attributes with all of its instances.

Scoping Rules

Although classes define a namespace, classes do not create a scope for names used inside the bodies of methods. Therefore, when you're implementing a class, references to attributes and methods must be fully qualified. For example, in methods you always reference attributes of the instance through `self`. Thus, in the example you use `self.balance`, not `balance`. This also applies if you want to call a method from another method, as shown in the following example:

```
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)     # Incorrect! 'bar' generates a NameError
        self.bar()    # This works
        Foo.bar(self) # This also works
```

The lack of scoping in classes is one area where Python differs from C++ or Java. If you have used those languages, the `self` parameter in Python is the same as the `this` pointer. The explicit use of `self` is required because Python does not provide a means to explicitly declare variables (that is, a declaration such as `int x` or `float y` in C). Without this, there is no way to know whether an assignment to a variable in a method is supposed to be a local variable or if it's supposed to be saved as an instance attribute. The explicit use of `self` fixes this—all values stored on `self` are part of the instance and all other assignments are just local variables.

Inheritance

Inheritance is a mechanism for creating a new class that specializes or modifies the behavior of an existing class. The original class is called a *base class* or a *superclass*. The new class is called a *derived class* or a *subclass*. When a class is created via inheritance, it "inherits" the attributes defined by its base classes. However, a derived class may redefine any of these attributes and add new attributes of its own.

Inheritance is specified with a comma-separated list of base-class names in the `class` statement. If there is no logical base class, a class inherits from `object`, as has been shown in prior examples. `object` is a class which is the root of all Python objects and which provides the default implementation of some common methods such as `__str__()`, which creates a string for use in printing.

Inheritance is often used to redefine the behavior of existing methods. As an example, here's a specialized version of `Account` that redefines the `inquiry()` method to periodically overstate the current balance with the hope that someone not paying close attention will overdraw his account and incur a big penalty when making a payment on their subprime mortgage:

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * 1.10   # Note: Patent pending idea
        else:
           return self.balance

c = EvilAccount("George", 1000.00)
c.deposit(10.0)          # Calls Account.deposit(c,10.0)
available = c.inquiry()  # Calls EvilAccount.inquiry(c)
```

In this example, instances of `EvilAccount` are identical to instances of `Account` except for the redefined `inquiry()` method.

Inheritance is implemented with only a slight enhancement of the dot (.) operator. Specifically, if the search for an attribute doesn't find a match in the instance or the instance's class, the search moves on to the base class. This process continues until there are no more base classes to search. In the previous example, this explains why `c.deposit()` calls the implementation of `deposit()` defined in the `Account` class.

A subclass can add new attributes to the instances by defining its own version of `__init__()`. For example, this version of `EvilAccount` adds a new attribute `evilfactor`:

```
class EvilAccount(Account):
    def __init__(self,name,balance,evilfactor):
        Account.__init__(self,name,balance)    # Initialize Account
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * self.evilfactor
        else:
           return self.balance
```

When a derived class defines `__init__()`, the `__init__()` methods of base classes are not automatically invoked. Therefore, it's up to a derived class to perform the proper initialization of the base classes by calling their `__init__()` methods. In the previous example, this is shown in the statement that calls `Account.__init__()`. If a base class does not define `__init__()`, this step can be omitted. If you don't know whether the base class defines `__init__()`, it is always safe to call it without any arguments because there is always a default implementation that simply does nothing.

Occasionally, a derived class will reimplement a method but also want to call the original implementation. To do this, a method can explicitly call the original method in the base class, passing the instance `self` as the first parameter as shown here:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)              # Subtract the "convenience" fee
        EvilAccount.deposit(self,amount) # Now, make deposit
```

A subtlety in this example is that the class `EvilAccount` doesn't actually implement the `deposit()` method. Instead, it is implemented in the `Account` class. Although this code works, it might be confusing to someone reading the code (e.g., was `EvilAccount` supposed to implement `deposit()`?). Therefore, an alternative solution is to use the `super()` function as follows:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)                          # Subtract convenience fee
        super(MoreEvilAccount,self).deposit(amount) # Now, make deposit
```

`super(`*`cls, instance`*`)` returns a special object that lets you perform attribute lookups on the base classes. If you use this, Python will search for an attribute using the normal search rules that would have been used on the base classes. This frees you from hard-coding the exact location of a method and more clearly states your intentions (that is, you want to call the previous implementation without regard for which base class defines it). Unfortunately, the syntax of `super()` leaves much to be desired. If you are using Python 3, you can use the simplified statement `super().deposit(amount)` to carry out the calculation shown in the example. In Python 2, however, you have to use the more verbose version.

Python supports multiple inheritance. This is specified by having a class list multiple base classes. For example, here are a collection of classes:

```
class DepositCharge(object):
    fee = 5.00
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 2.50
    def withdraw_fee(self):
        self.withdraw(self.fee)

# Class using multiple inheritance
class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self,amt):
        self.deposit_fee()
        super(MostEvilAccount,self).deposit(amt)
    def withdraw(self,amt):
        self.withdraw_fee()
        super(MostEvilAcount,self).withdraw(amt)
```

When multiple inheritance is used, attribute resolution becomes considerably more complicated because there are many possible search paths that could be used to bind attributes. To illustrate the possible complexity, consider the following statements:

```
d = MostEvilAccount("Dave",500.00,1.10)
d.deposit_fee()    # Calls DepositCharge.deposit_fee().   Fee is 5.00
d.withdraw_fee()   # Calls WithdrawCharge.withdraw_fee(). Fee is 5.00 ??
```

In this example, methods such as `deposit_fee()` and `withdraw_fee()` are uniquely named and found in their respective base classes. However, the `withdraw_fee()` function doesn't seem to work right because it doesn't actually use the value of `fee` that was initialized in its own class. What has happened is that the attribute `fee` is a class variable defined in two different base classes. One of those values is used, but which one? (Hint: it's `DepositCharge.fee`.)

To find attributes with multiple inheritance, all base classes are ordered in a list from the "most specialized" class to the "least specialized" class. Then, when searching for an attribute, this list is searched in order until the first definition of the attribute is found. In the example, the class `EvilAccount` is more specialized than `Account` because it inherits from `Account`. Similarly, within `MostEvilAccount, DepositCharge` is considered to be more specialized than `WithdrawCharge` because it is listed first in the list of base classes. For any given class, the ordering of base classes can be viewed by printing its `__mro__` attribute. Here's an example:

```
>>> MostEvilAccount.__mro__
(<class '__main__.MostEvilAccount'>,
 <class '__main__.EvilAccount'>,
 <class '__main__.Account'>,
 <class '__main__.DepositCharge'>,
 <class '__main__.WithdrawCharge'>,
 <type 'object'>)
>>>
```

In most cases, this list is based on rules that "make sense." That is, a derived class is always checked before its base classes and if a class has more than one parent, the parents are always checked in the same order as listed in the class definition. However, the precise ordering of base classes is actually quite complex and not based on any sort of "simple" algorithm such as depth-first or breadth-first search. Instead, the ordering is determined according to the C3 linearization algorithm, which is described in the paper "A Monotonic Superclass Linearization for Dylan" (K. Barrett, et al, presented at OOPSLA'96). A subtle aspect of this algorithm is that certain class hierarchies will be rejected by Python with a `TypeError`. Here's an example:

```
class X(object): pass
class Y(X): pass
class Z(X,Y): pass  # TypeError.
                    # Can't create consistent method resolution order__
```

In this case, the method resolution algorithm rejects class `Z` because it can't determine an ordering of the base classes that makes sense. For example, the class `X` appears before class `Y` in the inheritance list, so it must be checked first. However, class `Y` is more specialized because it inherits from `X`. Therefore, if `X` is checked first, it would not be possible to resolve specialized methods in `Y`. In practice, these issues should rarely arise—and if they do, it usually indicates a more serious design problem with a program.

As a general rule, multiple inheritance is something best avoided in most programs. However, it is sometimes used to define what are known as *mixin* classes. A mixin class typically defines a set of methods that are meant to be "mixed in" to other classes in order to add extra functionality (almost like a macro). Typically, the methods in a mixin will assume that other methods are present and will build upon them. The `DepositCharge` and `WithdrawCharge` classes in the earlier example illustrate this. These classes add new methods such as `deposit_fee()` to classes that include them as one of the base classes. However, you would never instantiate `DepositCharge` by itself. In fact, if you did, it wouldn't create an instance that could be used for anything useful (that is, the one defined method wouldn't even execute correctly).

Just as a final note, if you wanted to fix the problematic references to `fee` in this example, the implementation of `deposit_fee()` and `withdraw_fee()` should be changed to refer to the attribute directly using the class name instead of `self` (for example, `DepositChange.fee`).

Polymorphism Dynamic Binding and Duck Typing

Dynamic binding (also sometimes referred to as *polymorphism* when used in the context of inheritance) is the capability to use an instance without regard for its type. It is handled entirely through the attribute lookup process described for inheritance in the preceding section. Whenever an attribute is accessed as *`obj.attr`*, *`attr`* is located by searching within the instance itself, the instance's class definition, and then base classes, in that order. The first match found is returned.

A critical aspect of this binding process is that it is independent of what kind of object *`obj`* is. Thus, if you make a lookup such as *`obj`*`.name`, it will work on any *`obj`* that happens to have a `name` attribute. This behavior is sometimes referred to as *duck typing* in reference to the adage "if it looks like, quacks like, and walks like a duck, then it's a duck."

Python programmers often write programs that rely on this behavior. For example, if you want to make a customized version of an existing object, you can either inherit from it or you can simply create a completely new object that looks and acts like it but is otherwise unrelated. This latter approach is often used to maintain a loose coupling of program components. For example, code may be written to work with any kind of object whatsoever as long as it has a certain set of methods. One of the most common examples is with various "file-like" objects defined in the standard library. Although these objects work like files, they don't inherit from the built-in file object.

Static Methods and Class Methods

In a class definition, all functions are assumed to operate on an instance, which is always passed as the first parameter `self`. However, there are two other common kinds of methods that can be defined.

A *static method* is an ordinary function that just happens to live in the namespace defined by a class. It does not operate on any kind of instance. To define a static method, use the `@staticmethod` decorator as shown here:

```
class Foo(object):
     @staticmethod
     def add(x,y):
         return x + y
```

To call a static method, you just prefix it by the class name. You do not pass it any additional information. For example:

```
x = Foo.add(3,4)       # x  = 7
```

A common use of static methods is in writing classes where you might have many different ways to create new instances. Because there can only be one `__init__()` function, alternative creation functions are often defined as shown here:

```
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
    @staticmethod
    def now():
         t = time.localtime()
         return Date(t.tm_year, t.tm_mon, t.tm_day)
    @staticmethod
    def tomorrow():
         t = time.localtime(time.time()+86400)
         return Date(t.tm_year, t.tm_mon, t.tm_day)

# Example of creating some dates
a = Date(1967, 4, 9)
b = Date.now()       # Calls static method now()
c = Date.tomorrow()  # Calls static method tomorrow()
```

Class methods are methods that operate on the class itself as an object. Defined using the `@classmethod` decorator, a class method is different than an instance method in that the class is passed as the first argument which is named `cls` by convention. For example:

```
class Times(object):
    factor = 1
    @classmethod
    def mul(cls,x):
        return cls.factor*x

class TwoTimes(Times):
    factor = 2

x = TwoTimes.mul(4)      # Calls Times.mul(TwoTimes, 4) -> 8
```

In this example, notice how the class `TwoTimes` is passed to `mul()` as an object. Although this example is esoteric, there are practical, but subtle, uses of class methods. As an example, suppose that you defined a class that inherited from the `Date` class shown previously and customized it slightly:

```
class EuroDate(Date):
    # Modify string conversion to use European dates
    def __str__(self):
        return "%02d/%02d/%4d" % (self.day, self.month, self.year)
```

Because the class inherits from `Date`, it has all of the same features. However, the `now()` and `tomorrow()` methods are slightly broken. For example, if someone calls `EuroDate.now()`, a `Date` object is returned instead of a `EuroDate` object. A class method can fix this:

```
class Date(object):
    ...
    @classmethod
    def now(cls):
         t = time.localtime()
         # Create an object of the appropriate type
         return cls(t.tm_year, t.tm_month, t.tm_day)

class EuroDate(Date):
    ...

a = Date.now()       # Calls Date.now(Date) and returns a Date
b = EuroDate.now()   # Calls Date.now(EuroDate) and returns a EuroDate
```

One caution about static and class methods is that Python does not manage these methods in a separate namespace than the instance methods. As a result, they can be invoked on an instance. For example:

```
a = Date(1967,4,9)
b = d.now()          # Calls Date.now(Date)
```

This is potentially quite confusing because a call to `d.now()` doesn't really have anything to do with the instance `d`. This behavior is one area where the Python object system differs from that found in other OO languages such as Smalltalk and Ruby. In those languages, class methods are strictly separate from instance methods.

Properties

Normally, when you access an attribute of an instance or a class, the associated value that is stored is returned. A *property* is a special kind of attribute that computes its value when accessed. Here is a simple example:

```
class Circle(object):
     def __init__(self,radius):
         self.radius = radius
     # Some additional properties of Circles
     @property
     def area(self):
         return math.pi*self.radius**2
     @property
     def perimeter(self):
         return 2*math.pi*self.radius
```

The resulting `Circle` object behaves as follows:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
>>> c.area = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

In this example, `Circle` instances have an instance variable `c.radius` that is stored. `c.area` and `c.perimeter` are simply computed from that value. The `@property` decorator makes it possible for the method that follows to be accessed as a simple attribute, without the extra `()` that you would normally have to add to call the method. To the user of the object, there is no obvious indication that an attribute is being computed other than the fact that an error message is generated if an attempt is made to redefine the attribute (as shown in the `AttributeError` exception above).

Using properties in this way is related to something known as the *Uniform Access Principle*. Essentially, if you're defining a class, it is always a good idea to make the programming interface to it as uniform as possible. Without properties, certain attributes of an object would be accessed as a simple attribute such as `c.radius` whereas other attributes would be accessed as methods such as `c.area()`. Keeping track of when to add the extra `()` adds unnecessary confusion. A property can fix this.

Python programmers don't often realize that methods themselves are implicitly handled as a kind of property. Consider this class:

```
class Foo(object):
     def __init__(self,name):
         self.name = name
     def spam(self,x):
         print("%s, %s" % (self.name, x)
```

When a user creates an instance such as `f = Foo("Guido")` and then accesses `f.spam`, the original function object `spam` is not returned. Instead, you get something known as a *bound method*, which is an object that represents the method call that will execute when the `()` operator is invoked on it. A bound method is like a partially evaluated function where the `self` parameter has already been filled in, but the additional arguments still need to be supplied by you when you call it using `()`. The creation of this bound method object is silently handled through a property function that executes behind the scenes. When you define static and class methods using `@staticmethod` and `@classmethod`, you are actually specifying the use of a different property function that will handle the access to those methods in a different way. For example, `@staticmethod` simply returns the method function back "as is" without any special wrapping or processing.

Properties can also intercept operations to set and delete an attribute. This is done by attaching additional setter and deleter methods to a property. Here is an example:

```
class Foo(object):
     def __init__(self,name):
         self.__name = name
     @property
     def name(self):
         return self.__name
     @name.setter
     def name(self,value):
         if not isinstance(value,str):
              raise TypeError("Must be a string!")
         self.__name = value
     @name.deleter
     def name(self):
         raise TypeError("Can't delete name")

f = Foo("Guido")
n = f.name          # calls f.name() - get function
f.name = "Monty"    # calls setter name(f,"Monty")
f.name = 45         # calls setter name(f,45) -> TypeError
del f.name          # Calls deleter name(f) -> TypeError
```

In this example, the attribute `name` is first defined as a read-only property using the `@property` decorator and associated method. The `@name.setter` and `@name.deleter` decorators that follow are associating additional methods with the set and deletion operations on the `name` attribute. The names of these methods must exactly match the name of the original property. In these methods, notice that the actual value of the name is stored in an attribute `__name`. The name of the stored attribute does not have to follow any convention, but it has to be different than the property in order to distinguish it from the name of the property itself.

In older code, you will often see properties defined using the `property(`*`getf`*`=None, `*`setf`*`=None, `*`delf`*`=None, `*`doc`*`=None)` function with a set of uniquely named methods for carrying out each operation. For example:

```
class Foo(object):
     def getname(self):
         return self.__name
     def setname(self,value):
         if not isinstance(value,str):
              raise TypeError("Must be a string!")
         self.__name = value
     def delname(self):
         raise TypeError("Can't delete name")
     name = property(getname,setname,delname)
```

This older approach is still supported, but the decorator version tends to lead to classes that are a little more polished. For example, if you use decorators, the `get`, `set`, and `delete` functions aren't also visible as methods.

Descriptors

With properties, access to an attribute is controlled by a series of user-defined `get`, `set`, and `delete` functions. This sort of attribute control can be further generalized through the use of a *descriptor object*. A descriptor is simply an object that represents the value of an attribute. By implementing one or more of the special methods `__get__()`, `__set__()`, and `__delete__()`, it can hook into the attribute access mechanism and can customize those operations. Here is an example:

```
class TypedProperty(object):
     def __init__(self,name,type,default=None):
         self.name = "_" + name
         self.type = type
self.default = default if default else type()
     def __get__(self,instance,cls):
         return getattr(instance,self.name,self.default)
     def __set__(self,instance,value):
         if not isinstance(value,self.type):
             raise TypeError("Must be a %s" % self.type)
         setattr(instance,self.name,value)
     def __delete__(self,instance):
         raise AttributeError("Can't delete attribute")

class Foo(object):
     name = TypedProperty("name",str)
     num  = TypedProperty("num",int,42)
```

In this example, the class `TypedProperty` defines a descriptor where type checking is performed when the attribute is assigned and an error is produced if an attempt is made to delete the attribute. For example:

```
f = Foo()
a = f.name           # Implicitly calls Foo.name.__get__(f,Foo)
f.name = "Guido"     # Calls Foo.name.__set__(f,"Guido")
del f.name           # Calls Foo.name.__delete__(f)
```

Descriptors can only be instantiated at the class level. It is not legal to create descriptors on a per-instance basis by creating descriptor objects inside `__init__()` and other methods. Also, the attribute name used by the class to hold a descriptor takes precedence over attributes stored on instances. In the previous example, this is why the descriptor object takes a name parameter and why the name is changed slightly by inserting a leading underscore. In order for the descriptor to store a value on the instance, it has to pick a name that is different than that being used by the descriptor itself.

Data Encapsulation and Private Attributes

By default, all attributes and methods of a class are "public." This means that they are all accessible without any restrictions. It also implies that everything defined in a base class is inherited and accessible within a derived class. This behavior is often undesirable in object-oriented applications because it exposes the internal implementation of an object and can lead to namespace conflicts between objects defined in a derived class and those defined in a base class.

To fix this problem, all names in a class that start with a double underscore, such as `__Foo`, are automatically mangled to form a new name of the form `_`*`Classname`*`__Foo`. This effectively provides a way for a class to have private attributes and methods because private names used in a derived class won't collide with the same private names used in a base class. Here's an example:

```
class A(object):
     def __init__(self):
         self.__X = 3        # Mangled to self._A__X
     def __spam(self):       # Mangled to _A__spam()
         pass
     def bar(self):
         self.__spam()       # Only calls A.__spam()
```

```
class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37       # Mangled to self._B__X
    def __spam(self):       # Mangled to _B__spam()
        pass
```

Although this scheme provides the illusion of data hiding, there's no strict mechanism in place to actually prevent access to the "private" attributes of a class. In particular, if the name of the class and corresponding private attribute are known, they can be accessed using the mangled name. A class can make these attributes less visible by redefining the `__dir__()` method, which supplies the list of names returned by the `dir()` function that's used to inspect objects.

Although this name mangling might look like an extra processing step, the mangling process actually only occurs once at the time a class is defined. It does not occur during execution of the methods, nor does it add extra overhead to program execution. Also, be aware that name mangling does not occur in functions such as `getattr()`, `hasattr()`, `setattr()`, or `delattr()` where the attribute name is specified as a string. For these functions, you need to explicitly use the mangled name such as *`_Classname__name`* to access the attribute.

It is recommended that private attributes be used when defining mutable attributes via properties. By doing so, you will encourage users to use the property name rather than accessing the underlying instance data directly (which is probably not what you intended if you wrapped it with a property to begin with). An example of this appeared in the previous section.

Giving a method a private name is a technique that a superclass can use to prevent a derived class from redefining and changing the implementation of a method. For example, the `A.bar()` method in the example only calls `A.__spam()`, regardless of the type of `self` or the presence of a different `__spam()` method in a derived class.

Finally, don't confuse the naming of private class attributes with the naming of "private" definitions in a module. A common mistake is to define a class where a single leading underscore is used on attribute names in an effort to hide their values (e.g., `_name`). In modules, this naming convention prevents names from being exported by the `from module import *` statement. However, in classes, this naming convention does not hide the attribute nor does it prevent name clashes that arise if someone inherits from the class and defines a new attribute or method with the same name.

Object Memory Management

When a class is defined, the resulting class is a factory for creating new instances. For example:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius

# Create some Circle instances
c = Circle(4.0)
d = Circle(5.0)
```

The creation of an instance is carried out in two steps using the special method `__new__()`, which creates a new instance, and `__init__()`, which initializes it. For example, the operation `c = Circle(4.0)` performs these steps:

```
c = Circle.__new__(Circle, 4.0)
if isinstance(c,Circle):
    Circle.__init__(c,4.0)
```

The `__new__()` method of a class is something that is rarely defined by user code. If it is defined, it is typically written with the prototype `__new__(cls, *args, **kwargs)` where `args` and `kwargs` are the same arguments that will be passed to `__init__()`. `__new__()` is always a class method that receives the class object as the first parameter. Although `__new__()` creates an instance, it does not automatically call `__init__()`.

If you see `__new__()` defined in a class, it usually means the class is doing one of two things. First, the class might be inheriting from a base class whose instances are immutable. This is common if defining objects that inherit from an immutable built-in type such as an integer, string, or tuple because `__new__()` is the only method that executes prior to the instance being created and is the only place where the value could be modified (in `__init__()`, it would be too late). For example:

```
class Upperstr(str):
    def __new__(cls,value=""):
        return str.__new__(cls, value.upper())

u = Upperstr("hello")     # value is "HELLO"
```

The other major use of `__new__()` is when defining metaclasses. This is described at the end of this chapter.

Once created, instances are managed by reference counting. If the reference count reaches zero, the instance is immediately destroyed. When the instance is about to be destroyed, the interpreter first looks for a `__del__()` method associated with the object and calls it. In practice, it's rarely necessary for a class to define a `__del__()` method. The only exception is when the destruction of an object requires a cleanup action such as closing a file, shutting down a network connection, or releasing other system resources. Even in these cases, it's dangerous to rely on `__del__()` for a clean shutdown because there's no guarantee that this method will be called when the interpreter exits. A better approach may be to define a method such as `close()` that a program can use to explicitly perform a shutdown.

Occasionally, a program will use the `del` statement to delete a reference to an object. If this causes the reference count of the object to reach zero, the `__del__()` method is called. However, in general, the `del` statement doesn't directly call `__del__()`.

A subtle danger involving object destruction is that instances for which `__del__()` is defined cannot be collected by Python's cyclic garbage collector (which is a strong reason not to define `__del__` unless you need to). Programmers coming from languages without automatic garbage collection (e.g., C++) should take care not to adopt a programming style where `__del__()` is unnecessarily defined. Although it is rare to break the garbage collector by defining `__del__()`, there are certain types of programming patterns, especially those involving parent-child relationships or graphs, where this can be a problem. For example, suppose you had an object that was implementing a variant of the "Observer Pattern."

```
class Account(object):
    def __init__(self,name,balance):
        self.name = name
        self.balance = balance
        self.observers = set()
    def __del__(self):
        for ob in self.observers:
            ob.close()
        del self.observers
    def register(self,observer):
        self.observers.add(observer)
    def unregister(self,observer):
        self.observers.remove(observer)
    def notify(self):
        for ob in self.observers:
            ob.update()
    def withdraw(self,amt):
        self.balance -= amt
        self.notify()

class AccountObserver(object):
    def __init__(self, theaccount):
        self.theaccount = theaccount
        theaccount.register(self)
    def __del__(self):
        self.theaccount.unregister(self)
        del self.theaccount
    def update(self):
        print("Balance is %0.2f" % self.theaccount.balance)
    def close(self):
        print("Account no longer in use")

# Example setup
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

In this code, the `Account` class allows a set of `AccountObserver` objects to monitor an `Account` instance by receiving an update whenever the balance changes. To do this, each `Account` keeps a set of the observers and each `AccountObserver` keeps a reference back to the account. Each class has defined `__del__()` in an attempt to provide some sort of cleanup (such as unregistering and so on). However, it just doesn't work. Instead, the classes have created a reference cycle in which the reference count never drops to 0 and there is no cleanup. Not only that, the garbage collector (the `gc` module) won't even clean it up, resulting in a permanent memory leak.

One way to fix the problem shown in this example is for one of the classes to create a weak reference to the other using the `weakref` module. A *weak reference* is a way of creating a reference to an object without increasing its reference count. To work with a weak reference, you have to add an extra bit of functionality to check whether the object being referred to still exists. Here is an example of a modified observer class:

```
import weakref
class AccountObserver(object):
    def __init__(self, theaccount):
        self.accountref = weakref.ref(theaccount)  # Create a weakref
        theaccount.register(self)

    def __del__(self):
        acc = self.accountref()        # Get account
        if acc:                        # Unregister if still exists
            acc.unregister(self)
    def update(self):
        print("Balance is %0.2f" % self.accountref().balance)
    def close(self):
        print("Account no longer in use")

# Example setup
a = Account('Dave',1000.00)
a_ob = AccountObserver(a)
```

In this example, a weak reference `accountref` is created. To access the underlying `Account`, you call it like a function. This either returns the `Account` or `None` if it's no longer around. With this modification, there is no longer a reference cycle. If the `Account` object is destroyed, its `__del__` method runs and observers receive notification. The `gc` module also works properly. More information about the `weakref` module can be found in Chapter 13, "Python Runtime Services."

Object Representation and Attribute Binding

Internally, instances are implemented using a dictionary that's accessible as the instance's `__dict__` attribute. This dictionary contains the data that's unique to each instance. Here's an example:

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'balance': 1100.0, 'name': 'Guido'}
```

New attributes can be added to an instance at any time, like this:

```
a.number = 123456    # Add attribute 'number' to a.__dict__
```

Modifications to an instance are always reflected in the local `__dict__` attribute. Likewise, if you make modifications to `__dict__` directly, those modifications are reflected in the attributes.

Instances are linked back to their class by a special attribute `__class__`. The class itself is also just a thin layer over a dictionary which can be found in its own `__dict__` attribute. The class dictionary is where you find the methods. For example:

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
['__dict__', '__module__', 'inquiry', 'deposit', 'withdraw',
'__del__', 'num_accounts', '__weakref__', '__doc__', '__init__']
>>>
```

Finally, classes are linked to their base classes in a special attribute `__bases__`, which is a tuple of the base classes. This underlying structure is the basis for all of the operations that get, set, and delete the attributes of objects.

Whenever an attribute is set using *`obj.name = value`*, the special method *`obj.__setattr__("name", value)`* is invoked. If an attribute is deleted using `del` *`obj.name`*, the special method *`obj.__delattr__("name")`* is invoked. The default behavior of these methods is to modify or remove values from the local `__dict__` of `obj` unless the requested attribute happens to correspond to a property or descriptor. In

that case, the set and delete operation will be carried out by the set and delete functions associated with the property.

For attribute lookup such as `obj.name`, the special method `obj.__getattrribute__("name")` is invoked. This method carries out the search process for finding the attribute, which normally includes checking for properties, looking in the local `__dict__` attribute, checking the class dictionary, and searching the base classes. If this search process fails, a final attempt to find the attribute is made by trying to invoke the `__getattr__()` method of the class (if defined). If this fails, an `AttributeError` exception is raised.

User-defined classes can implement their own versions of the attribute access functions, if desired. For example:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __getattr__(self,name):
        if name == 'area':
            return math.pi*self.radius**2
        elif name == 'perimeter':
            return 2*math.pi*self.radius
        else:
            return object.__getattr__(self,name)
    def __setattr__(self,name,value):
        if name in ['area','perimeter']:
            raise TypeError("%s is readonly" % name)
        object.__setattr__(self,name,value)
```

A class that reimplements these methods should probably rely upon the default implementation in `object` to carry out the actual work. This is because the default implementation takes care of the more advanced features of classes such as descriptors and properties.

As a general rule, it is relatively uncommon for classes to redefine the attribute access operators. However, one application where they are often used is in writing general-purpose wrappers and proxies to existing objects. By redefining `__getattr__()`, `__setattr__()`, and `__delattr__()`, a proxy can capture attribute access and transparently forward those operations on to another object.

__slots__

A class can restrict the set of legal instance attribute names by defining a special variable called `__slots__`. Here's an example:

```
class Account(object):
    __slots__ = ('name','balance')
    ...
```

When `__slots__` is defined, the attribute names that can be assigned on instances are restricted to the names specified. Otherwise, an `AttributeError` exception is raised. This restriction prevents someone from adding new attributes to existing instances and solves the problem that arises if someone assigns a value to an attribute that they can't spell correctly.

In reality, `__slots__` was never implemented to be a safety feature. Instead, it is actually a performance optimization for both memory and execution speed. Instances of a class that uses `__slots__` no longer use a dictionary for storing instance data. Instead, a much more compact data structure based on an array is used. In programs that create a large number of objects, using `__slots__` can result in a substantial reduction in memory use and execution time.

Be aware that the use of `__slots__` has a tricky interaction with inheritance. If a class inherits from a base class that uses `__slots__`, it also needs to define `__slots__` for storing its own attributes (even if it doesn't add any) to take advantage of the benefits `__slots__` provides. If you forget this, the derived class will run slower and use even more memory than what would have been used if `__slots__` had not been used on *any* of the classes!

The use of `__slots__` can also break code that expects instances to have an underlying `__dict__` attribute. Although this often does not apply to user code, utility libraries and other tools for supporting objects may be programmed to look at `__dict__` for debugging, serializing objects, and other operations.

Finally, the presence of `__slots__` has no effect on the invocation of methods such as `__getattribute__()`, `__getattr__()`, and `__setattr__()` should they be redefined in a class. However, the default behavior of these methods will take `__slots__` into account. In addition, it should be stressed that it is not necessary to add method or property names to `__slots__`, as they are stored in the class, not on a per-instance basis.

Operator Overloading

User-defined objects can be made to work with all of Python's built-in operators by adding implementations of the special methods described in Chapter 3 to a class. For example, if you wanted to add a new kind of number to Python, you could define a class in which special methods such as `__add__()` were defined to make instances work with the standard mathematical operators.

The following example shows how this works by defining a class that implements the complex numbers with some of the standard mathematical operators.

> **Note**
>
> Because Python already provides a complex number type, this class is only provided for the purpose of illustration.

```
class Complex(object):
    def __init__(self,real,imag=0):
        self.real = float(real)
        self.imag = float(imag)
    def __repr__(self):
        return "Complex(%s,%s)" % (self.real, self.imag)
    def __str__(self):
        return "(%g+%gj)" % (self.real, self.imag)
    # self + other
    def __add__(self,other):
        return Complex(self.real + other.real, self.imag + other.imag)
    # self - other
    def __sub__(self,other):
        return Complex(self.real - other.real, self.imag - other.imag)
```

In the example, the `__repr__()` method creates a string that can be evaluated to re-create the object (that is, `"Complex(real,imag)"`). This convention should be followed for all user-defined objects as applicable. On the other hand, the `__str__()` method creates a string that's intended for nice output formatting (this is the string that would be produced by the `print` statement).

The other operators, such as `__add__()` and `__sub__()`, implement mathematical operations. A delicate matter with these operators concerns the order of operands and type coercion. As implemented in the previous example, the `__add__()` and `__sub__()` operators are applied *only* if a complex number appears on the left side of the operator. They do not work if they appear on the right side of the operator and the left-most operand is not a `Complex`. For example:

```
>>> c = Complex(2,3)
>>> c + 4.0
Complex(6.0,3.0)
>>> 4.0 + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Complex'

>>>
```

The operation `c + 4.0` works partly by accident. All of Python's built-in numbers already have `.real` and `.imag` attributes, so they were used in the calculation. If the `other` object did not have these attributes, the implementation would break. If you want your implementation of `Complex` to work with objects missing these attributes, you have to add extra conversion code to extract the needed information (which might depend on the type of the other object).

The operation `4.0 + c` does not work at all because the built-in floating point type doesn't know anything about the `Complex` class. To fix this, you can add reversed-operand methods to `Complex`:

```
class Complex(object):
    ...
    def __radd__(self,other):
        return Complex(other.real + self.real, other.imag + self.imag)
    def __rsub__(self,other):
        return Complex(other.real - self.real, other.imag - self.img)
    ...
```

These methods serve as a fallback. If the operation `4.0 + c` fails, Python tries to execute `c.__radd__(4.0)` first before issuing a `TypeError`.

Older versions of Python have tried various approaches to coerce types in mixed-type operations. For example, you might encounter legacy Python classes that implement a `__coerce__()` method. This is no longer used by Python 2.6 or Python 3. Also, don't be fooled by special methods such as `__int__()`, `__float__()`, or `__complex__()`. Although these methods are called by explicit conversions such as `int(x)` or `float(x)`, they are never called implicitly to perform type conversion in mixed-type arithmetic. So, if you are writing classes where operators must work with mixed types, you have to explicitly handle the type conversion in the implementation of each operator.

Types and Class Membership Tests

When you create an instance of a class, the type of that instance is the class itself. To test for membership in a class, use the built-in function `isinstance(obj,cname)`. This function returns `True` if an object, *`obj`*, belongs to the class *`cname`* or any class derived from *`cname`*. Here's an example:

```
class A(object): pass
class B(A): pass
class C(object): pass

a = A()          # Instance of 'A'
b = B()          # Instance of 'B'
c = C()          # Instance of 'C'

type(a)          # Returns the class object A
isinstance(a,A)  # Returns True
isinstance(b,A)  # Returns True, B derives from A
isinstance(b,C)  # Returns False, C not derived from A
```

Similarly, the built-in function `issubclass(A,B)` returns `True` if the class *`A`* is a subclass of class *`B`*. Here's an example:

```
issubclass(B,A)   # Returns True
issubclass(C,A)   # Returns False
```

A subtle problem with type-checking of objects is that programmers often bypass inheritance and simply create objects that mimic the behavior of another object. As an example, consider these two classes:

```
class Foo(object):
    def spam(self,a,b):
        pass

class FooProxy(object):
    def __init__(self,f):
        self.f = f
    def spam(self,a,b):
        return self.f.spam(a,b)
```

In this example, `FooProxy` is functionally identical to `Foo`. It implements the same methods, and it even uses `Foo` underneath the covers. Yet, in the type system, `FooProxy` is different than `Foo`. For example:

```
f = Foo()           # Create a Foo
g = FooProxy(f)     # Create a FooProxy
isinstance(g, Foo)  # Returns False
```

If a program has been written to explicitly check for a `Foo` using `isinstance()`, then it certainly won't work with a `FooProxy` object. However, this degree of strictness is often not exactly what you want. Instead, it might make more sense to assert that an object can simply be used as `Foo` because it has the same interface. To do this, it is possible to define an object that redefines the behavior of `isinstance()` and `issubclass()` for the purpose of grouping objects together and type-checking. Here is an example:

```
class IClass(object):
    def __init__(self):
        self.implementors = set()
    def register(self,C):
        self.implementors.add(C)
    def __instancecheck__(self,x):
        return self.__subclasscheck__(type(x))
```

```
    def __subclasscheck__(self,sub):
        return any(c in self.implementors for c in sub.mro())

# Now, use the above object
IFoo = IClass()
IFoo.register(Foo)
IFoo.register(FooProxy)
```

In this example, the class `IClass` creates an object that merely groups a collection of other classes together in a set. The `register()` method adds a new class to the set. The special method `__instancecheck__()` is called if anyone performs the operation `isinstance(x, IClass)`. The special method `__subclasscheck__()` is called if the operation `issubclass(C,IClass)` is called.

By using the `IFoo` object and registered implementers, one can now perform type checks such as the following:

```
f = Foo()           # Create a Foo
g = FooProxy(f)     # Create a FooProxy
isinstance(f, IFoo)        # Returns True
isinstance(g, IFoo)        # Returns True
issubclass(FooProxy, IFoo) # Returns True
```

In this example, it's important to emphasize that no strong type-checking is occurring. The `IFoo` object has overloaded the instance checking operations in a way that allows a you to assert that a class belongs to a group. It doesn't assert any information on the actual programming interface, and no other verification actually occurs. In fact, you can simply register any collection of objects you want to group together without regard to how those classes are related to each other. Typically, the grouping of classes is based on some criteria such as all classes implementing the same programming interface. However, no such meaning should be inferred when overloading `__instancecheck__()` or `__subclasscheck__()`. The actual interpretation is left up to the application.

Python provides a more formal mechanism for grouping objects, defining interfaces, and type-checking. This is done by defining an abstract base class, which is defined in the next section.

Abstract Base Classes

In the last section, it was shown that the `isinstance()` and `issubclass()` operations can be overloaded. This can be used to create objects that group similar classes together and to perform various forms of type-checking. *Abstract base classes* build upon this concept and provide a means for organizing objects into a hierarchy, making assertions about required methods, and so forth.

To define an abstract base class, you use the `abc` module. This module defines a metaclass (`ABCMeta`) and a set of decorators (`@abstractmethod` and `@abstractproperty`) that are used as follows:

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:                      # In Python 3, you use the syntax
    __metaclass__ = ABCMeta     # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self,a,b):
        pass
    @abstractproperty



    def name(self):
        pass
```

The definition of an abstract class needs to set its metaclass to `ABCMeta` as shown (also, be aware that the syntax differs between Python 2 and 3). This is required because the implementation of abstract classes relies on a metaclass (described in the next section). Within the abstract class, the `@abstractmethod` and `@abstractproperty` decorators specify that a method or property must be implemented by subclasses of `Foo`.

An abstract class is not meant to be instantiated directly. If you try to create a `Foo` for the previous class, you will get the following error:

```
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
>>>
```

This restriction carries over to derived classes as well. For instance, if you have a class `Bar` that inherits from `Foo` but it doesn't implement one or more of the abstract methods, attempts to create a `Bar` will fail with a similar error. Because of this added checking, abstract classes are useful to programmers who want to make assertions on the methods and properties that must be implemented on subclasses.

Although an abstract class enforces rules about methods and properties that must be implemented, it does not perform conformance checking on arguments or return values. Thus, an abstract class will not check a subclass to see whether a method has used the same arguments as an abstract method. Likewise, an abstract class that requires the definition of a property does not check to see whether the property in a subclass supports the same set of operations (`get`, `set`, and `delete`) of the property specified in a base.

Although an abstract class can not be instantiated, it can define methods and properties for use in subclasses. Moreover, an abstract method in the base can still be called from a subclass. For example, calling `Foo.spam(a,b)` from the subclass is allowed.

Abstract base classes allow preexisting classes to be registered as belonging to that base. This is done using the `register()` method as follows:

```
class Grok(object):
    def spam(self,a,b):
        print("Grok.spam")

Foo.register(Grok)       # Register with Foo abstract base class
```

When a class is registered with an abstract base, type-checking operations involving the abstract base (such as `isinstance()` and `issubclass()`) will return `True` for instances of the registered class. When a class is registered with an abstract class, no checks are made to see whether the class actually implements any of the abstract methods or properties. This registration process only affects type-checking. It does not add extra error checking to the class that is registered.

Unlike many other object-oriented languages, Python's built-in types are organized into a relatively flat hierarchy. For example, if you look at the built-in types such as `int` or `float`, they directly inherit from `object`, the root of all objects, instead of an intermediate base class representing numbers. This makes it clumsy to write programs that want to inspect and manipulate objects based on a generic category such as simply being an instance of a number.

The abstract class mechanism addresses this issue by allowing preexisting objects to be organized into user-definable type hierarchies. Moreover, some library modules aim to organize the built-in types according to different capabilities that they possess. The `collections` module contains abstract base classes for various kinds of operations involving sequences, sets, and dictionaries. The `numbers` module contains abstract base classes related to organizing a hierarchy of numbers. Further details can be found in Chapter 14, "Mathematics," and Chapter 15, "Data Structures, Algorithms, and Utilities."

Metaclasses

When you define a class in Python, the class definition itself becomes an object. Here's an example:

```
class Foo(object): pass
isinstance(Foo,object)        # Returns True
```

If you think about this long enough, you will realize that something had to create the `Foo` object. This creation of the class object is controlled by a special kind of object called a *metaclass*. Simply stated, a metaclass is an object that knows how to create and manage classes.

In the preceding example, the metaclass that is controlling the creation of `Foo` is a class called `type`. In fact, if you display the type of `Foo`, you will find out that it *is* a `type`:

```
>>> type(Foo)
<type 'type'>
```

When a new class is defined with the `class` statement, a number of things happen. First, the body of the class is executed as a series of statements within its own private dictionary. The execution of statements is exactly the same as in normal code with the addition of the name mangling that occurs on private members (names that start with `__`). Finally, the name of the class, the list of base classes, and the dictionary are passed to the constructor of a metaclass to create the corresponding class object. Here is an example of how it works:

```
class_name = "Foo"                  # Name of class
class_parents = (object,)           # Base classes
class_body = """                    # Class body
def __init__(self,x):
    self.x = x
def blah(self):
    print("Hello World")
"""
class_dict = { }
# Execute the body in the local dictionary class_dict
exec(class_body,globals(),class_dict)

# Create the class object Foo
Foo = type(class_name,class_parents,class_dict)
```

The final step of class creation where the metaclass `type()` is invoked can be customized. The choice of what happens in the final step of class definition is controlled in a number of ways. First, the class can explicitly specify its metaclass by either setting a `__metaclass__` class variable (Python 2), or supplying the `metaclass` keyword argument in the tuple of base classes (Python 3).

```
class Foo:                  # In Python 3, use the syntax
    __metaclass__ = type    # class Foo(metaclass=type)
    ...
```

If no metaclass is explicitly specified, the `class` statement examines the first entry in the tuple of base classes (if any). In this case, the metaclass is the same as the type of the first base class. Therefore, when you write

```
class Foo(object): pass
```

`Foo` will be the same type of class as `object`.

If no base classes are specified, the `class` statement checks for the existence of a global variable called `__metaclass__`. If this variable is found, it will be used to create classes. If you set this variable, it will control how classes are created when a simple class statement is used. Here's an example:

```
__metaclass__ = type
class Foo:
    pass
```

Finally, if no `__metaclass__` value can be found anywhere, Python uses the default metaclass. In Python 2, this defaults to `types.ClassType`, which is known as an *old-style class*. This kind of class, deprecated since Python 2.2, corresponds to the original implementation of classes in Python. Although these classes are still supported, they should be avoided in new code and are not covered further here. In Python 3, the default metaclass is simply `type()`.

The primary use of metaclasses is in frameworks that want to assert more control over the definition of user-defined objects. When a custom metaclass is defined, it typically inherits from `type()` and reimplements methods such as `__init__()` or `__new__()`. Here is an example of a metaclass that forces all methods to have a documentation string:

```
class DocMeta(type):
    def __init__(self,name,bases,dict):
        for key, value in dict.items():
            # Skip special and private methods
            if key.startswith("__"): continue
            # Skip anything not callable
            if not hasattr(value,"__call__"): continue
            # Check for a doc-string
            if not getattr(value,"__doc__"):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self,name,bases,dict)
```

In this metaclass, the `__init__()` method has been written to inspect the contents of the class dictionary. It scans the dictionary looking for methods and checking to see whether they all have documentation strings. If not, a `TypeError` exception is generated. Otherwise, the default implementation of `type.__init__()` is called to initialize the class.

To use this metaclass, a class needs to explicitly select it. The most common technique for doing this is to first define a base class such as the following:

```
class Documented:               # In Python 3, use the syntax
    __metaclass__ = DocMeta     # class Documented(metaclass=DocMeta)
```

This base class is then used as the parent for all objects that are to be documented. For example:

```
class Foo(Documented):
    spam(self,a,b):
        "spam does something"
        pass
```

This example illustrates one of the major uses of metaclasses, which is that of inspecting and gathering information about class definitions. The metaclass isn't changing anything about the class that actually gets created but is merely adding some additional checks.

In more advanced metaclass applications, a metaclass can both inspect and alter the contents of a class definition prior to the creation of the class. If alterations are going to be made, you should redefine the `__new__()` method that runs prior to the creation of the class itself. This technique is commonly combined with techniques that wrap attributes with descriptors or properties because it is one way to capture the names being used in the class. As an example, here is a modified version of the `TypedProperty` descriptor that was used in the "Descriptors" section:

```
class TypedProperty(object):
    def __init__(self,type,default=None):
        self.name = None
        self.type = type
        if default: self.default = default
        else:       self.default = type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Can't delete attribute")
```

In this example, the `name` attribute of the descriptor is simply set to `None`. To fill this in, we'll rely on a meta class. For example:

```
class TypedMeta(type):
    def __new__(cls,name,bases,dict):
        slots = []
        for key,value in dict.items():
            if isinstance(value,TypedProperty):
                value.name = "_" + key
                slots.append(value.name)
        dict['__slots__'] = slots
        return type.__new__(cls,name,bases,dict)

# Base class for user-defined objects to use
class Typed:                    # In Python 3, use the syntax
    __metaclass__ = TypedMeta    # class Typed(metaclass=TypedMeta)
```

In this example, the metaclass scans the class dictionary and looks for instances of `TypedProperty`. If found, it sets the `name` attribute and builds a list of names in `slots`. After this is done, a `__slots__` attribute is added to the class dictionary, and the class is constructed by calling the `__new__()` method of the `type()` metaclass. Here is an example of using this new metaclass:

```
class Foo(Typed):
    name = TypedProperty(str)
    num  = TypedProperty(int,42)
```

Although metaclasses make it possible to drastically alter the behavior and semantics of user-defined classes, you should probably resist the urge to use metaclasses in a way that makes classes work wildly different from what is described in the standard Python documentation. Users will be confused if the classes they must write don't adhere to any of the normal coding rules expected for classes.

Class Decorators

In the previous section, it was shown how the process of creating a class can be customized by defining a metaclass. However, sometimes all you want to do is perform some kind of extra processing after a class is defined, such as adding a class to a registry or database. An alternative approach for such problems is to use a class decorator. A *class decorator* is a function that takes a class as input and returns a class as output. For example:

```
registry = { }
def register(cls):
    registry[cls.__clsid__] = cls
    return cls
```

In this example, the register function looks inside a class for a `__clsid__` attribute. If found, it's used to add the class to a dictionary mapping class identifiers to class objects. To use this function, you can use it as a decorator right before the class definition. For example:

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
```

Here, the use of the decorator syntax is mainly one of convenience. An alternative way to accomplish the same thing would have been this:

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
register(Foo)       # Register the class
```

Although it's possible to think of endless diabolical things one might do to a class in a class decorator function, it's probably best to avoid excessive magic such as putting a wrapper around the class or rewriting the class contents.

8

Modules, Packages, and Distribution

Large Python programs are organized into modules and packages. In addition, a large number of modules are included in the Python standard library. This chapter describes the module and package system in more detail. In addition, it provides information on how to install third-party modules and distribute source code.

Modules and the `import` Statement

Any Python source file can be used as a module. For example, consider the following code:

```
# spam.py
a = 37
def foo():
    print("I'm foo and a is %s" % a)
def bar():
    print("I'm bar and I'm calling foo")
    foo()
class Spam(object):
    def grok(self):
        print("I'm Spam.grok")
```

To load this code as a module, use the statement `import spam`. The first time `import` is used to load a module, it does three things:

1. It creates a new namespace that serves as a container for all the objects defined in the corresponding source file. This is the namespace accessed when functions and methods defined within the module use the `global` statement.
2. It executes the code contained in the module within the newly created namespace.
3. It creates a name within the caller that refers to the module namespace. This name matches the name of the module and is used as follows:

```
import spam           # Loads and executes the module 'spam'
x = spam.a            # Accesses a member of module 'spam'
spam.foo()            # Call a function in module 'spam'
s = spam.Spam()       # Create an instance of spam.Spam()
s.grok()
...
```

It is important to emphasize that `import` executes all of the statements in the loaded source file. If a module carries out a computation or produces output in addition to defining variables, functions, and classes, you will see the result. Also, a common confusion with modules concerns the access to classes. Keep in mind that if a file `spam.py` defines a class `Spam`, you must use the name `spam.Spam` to refer to the class.

To import multiple modules, you can supply `import` with a comma-separated list of module names, like this:

```
import socket, os, re
```

The name used to refer to a module can be changed using the `as` qualifier. Here's an example:

```
import spam as sp
import socket as net
sp.foo()
sp.bar()
net.gethostname()
```

When a module is loaded using a different name like this, the new name only applies to the source file or context where the `import` statement appeared. Other program modules can still load the module using its original name.

Changing the name of the imported module can be a useful tool for writing extensible code. For example, suppose you have two modules, `xmlreader.py` and `csvreader.py`, that both define a function `read_data(`*`filename`*`)` for reading some data from a file, but in different input formats. You can write code that selectively picks the reader module like this:

```
if format == 'xml':
    import xmlreader as reader
elif format == 'csv':
    import csvreader as reader
data = reader.read_data(filename)
```

Modules are first class objects in Python. This means that they can be assigned to variables, placed in data structures such as a list, and passed around in a program as a data. For instance, the `reader` variable in the previous example simply refers to the corresponding module object. Underneath the covers, a module object is a layer over a dictionary that is used to hold the contents of the module namespace. This dictionary is available as the `__dict__` of a module, and whenever you look up or change a value in a module, you're working with this dictionary.

The `import` statement can appear at any point in a program. However, the code in each module is loaded and executed only once, regardless of how often you use the `import` statement. Subsequent `import` statements simply bind the module name to the module object already created by the previous import. You can find a dictionary containing all currently loaded modules in the variable `sys.modules`. This dictionary maps module names to module objects. The contents of this dictionary are used to determine whether `import` loads a fresh copy of a module.

Importing Selected Symbols from a Module

The `from` statement is used to load specific definitions within a module into the current namespace. The `from` statement is identical to `import` except that instead of creating a name referring to the newly created module namespace, it places references to one or more of the objects defined in the module into the current namespace:

```
from spam import foo   # Imports spam and puts 'foo' in current namespace
foo()                  # Calls spam.foo()
spam.foo()             # NameError: spam
```

The `from` statement also accepts a comma-separated list of object names. For example:

```
from spam import foo, bar
```

If you have a very long list of names to import, the names can be enclosed in parentheses. This makes it easier to break the `import` statement across multiple lines. Here's an example:

```
from spam import (foo,
                  bar,
                  Spam)
```

In addition, the `as` qualifier can be used to rename specific objects imported with `from`. Here's an example:

```
from spam import Spam as Sp
s = Sp()
```

The asterisk (`*`) wildcard character can also be used to load all the definitions in a module, except those that start with an underscore. Here's an example:

```
from spam import *   # Load all definitions into current namespace
```

The `from` *`module`* `import *` statement may only be used at the top level of a module. In particular, it is illegal to use this form of import inside function bodies due to the way in which it interacts with function scoping rules (e.g., when functions are compiled into internal bytecode, all of the symbols used within the function need to be fully specified).

Modules can more precisely control the set of names imported by `from` *`module`* `import *` by defining the list `__all__`. Here's an example:

```
# module: spam.py
__all__ = [ 'bar', 'Spam' ]  # Names I will export with from spam import *
```

Importing definitions with the `from` form of import does not change their scoping rules. For example, consider this code:

```
from spam import foo
a = 42
foo()      # Prints "I'm foo and a is 37"
```

In this example, the definition of `foo()` in `spam.py` refers to a global variable `a`. When a reference to `foo` is placed into a different namespace, it doesn't change the binding rules for variables within that function. Thus, the global namespace for a function is always the module in which the function was defined, not the namespace into which a function is imported and called. This also applies to function calls. For example, in the following code, the call to `bar()` results in a call to `spam.foo()`, not the redefined `foo()` that appears in the previous code example:

```
from spam import bar
def foo():
    print("I'm a different foo")
bar()      # When bar calls foo(), it calls spam.foo(), not
           # the definition of foo() above
```

Another common confusion with the `from` form of import concerns the behavior of global variables. For example, consider this code:

```
from spam import a, foo      # Import a global variable
a = 42                       # Modify the variable
foo()                        # Prints "I'm foo and a is 37"
print(a)                     # Prints "42"
```

Here, it is important to understand that variable assignment in Python is not a storage operation. That is, the assignment to `a` in the earlier example is not storing a new value in `a`, overwriting the previous value. Instead, a new object containing the value `42` is created and the name `a` is made to refer to it. At this point, `a` is no longer bound to the value in the imported module but to some other object. Because of this behavior, it is not possible to use the `from` statement in a way that makes variables behave similarly as global variables or common blocks in languages such as C or Fortran. If you want to have mutable global program parameters in your program, put them in a module and use the module name explicitly using the `import` statement (that is, use `spam.a` explicitly).

Execution as the Main Program

There are two ways in which a Python source file can execute. The `import` statement executes code in its own namespace as a library module. However, code might also execute as the main program or script. This occurs when you supply the program as the script name to the interpreter:

```
% python spam.py
```

Each module defines a variable, `__name__`, that contains the module name. Programs can examine this variable to determine the module in which they're executing. The top-level module of the interpreter is named `__main__`. Programs specified on the command line or entered interactively run inside the `__main__` module. Sometimes a program may alter its behavior, depending on whether it has been imported as a module or is running in `__main__`. For example, a module may include some testing code that is executed if the module is used as the main program but which is not executed if the module is simply imported by another module. This can be done as follows:

```
# Check if running as a program
if __name__ == '__main__':
     # Yes
     statements
else:
     # No, I must have been imported as a module
     statements
```

It is common practice for source files intended for use as libraries to use this technique for including optional testing or example code. For example, if you're developing a module, you can put code for testing the features of your library inside an `if` statement as shown and simply run Python on your module as the main program to run it. That code won't run for users who import your library.

The Module Search Path

When loading modules, the interpreter searches the list of directories in `sys.path`. The first entry in `sys.path` is typically an empty string `''`, which refers to the current working directory. Other entries in `sys.path` may consist of directory names, `.zip` archive files, and `.egg` files. The order in which entries are listed in `sys.path` determines the search order used when modules are loaded. To add new entries to the search path, simply add them to this list.

Although the path usually contains directory names, zip archive files containing Python modules can also be added to the search path. This can be a convenient way to package a collection of modules as a single file. For example, suppose you created two modules, `foo.py` and `bar.py`, and placed them in a zip file called `mymodules.zip`. The file could be added to the Python search path as follows:

```
import sys
sys.path.append("mymodules.zip")
import foo, bar
```

Specific locations within the directory structure of a zip file can also be used. In addition, zip files can be mixed with regular pathname components. Here's an example:

```
sys.path.append("/tmp/modules.zip/lib/python")
```

In addition to `.zip` files, you can also add `.egg` files to the search path. `.egg` files are packages created by the `setuptools` library. This is a common format encountered when installing third-party Python libraries and extensions. An `.egg` file is actually just a `.zip` file with some extra metadata (e.g., version number, dependencies, etc.) added to it. Thus, you can examine and extract data from an `.egg` file using standard tools for working with `.zip` files.

Despite support for zip file imports, there are some restrictions to be aware of. First, it is only possible import `.py`, `.pyw`, `.pyc`, and `.pyo` files from an archive. Shared libraries and extension modules written in C cannot be loaded directly from archives, although packaging systems such as `setuptools` are sometimes able to provide a workaround (typically by extracting C extensions to a temporary directory and loading modules from it). Moreover, Python will not create `.pyc` and `.pyo` files when `.py` files are loaded from an archive (described next). Thus, it is important to make sure these files are created in advance and placed in the archive in order to avoid poor performance when loading modules.

Module Loading and Compilation

So far, this chapter has presented modules as files containing pure Python code. However, modules loaded with `import` really fall into four general categories:

- Code written in Python (`.py` files)
- C or C++ extensions that have been compiled into shared libraries or DLLs
- Packages containing a collection of modules
- Built-in modules written in C and linked into the Python interpreter

When looking for a module (for example, `foo`), the interpreter searches each of the directories in `sys.path` for the following files (listed in search order):

1. A directory, `foo`, defining a package
2. `foo.pyd`, `foo.so`, `foomodule.so`, or `foomodule.dll` (compiled extensions)
3. `foo.pyo` (only if the `-O` or `-OO` option has been used)
4. `foo.pyc`
5. `foo.py` (on Windows, Python also checks for `.pyw` files.)

Packages are described shortly; compiled extensions are described in Chapter 26, "Extending and Embedding Python." For `.py` files, when a module is first imported, it's compiled into bytecode and written back to disk as a `.pyc` file. On subsequent imports, the interpreter loads this precompiled bytecode unless the modification date of the `.py` file is more recent (in which case, the `.pyc` file is regenerated). `.pyo` files are used in conjunction with the interpreter's `-O` option. These files contain bytecode stripped of line numbers, assertions, and other debugging information. As a result, they're somewhat smaller and allow the interpreter to run slightly faster. If the `-OO` option is specified instead of `-O`, documentation strings are also stripped from the file. This removal of documentation strings occurs only when `.pyo` files are created—not when they're loaded. If none of these files exists in any of the directories in `sys.path`, the interpreter checks whether the name corresponds to a built-in module name. If no match exists, an `ImportError` exception is raised.

The automatic compilation of files into `.pyc` and `.pyo` files occurs only in conjunction with the `import` statement. Programs specified on the command line or standard input don't produce such files. In addition, these files aren't created if the directory containing a module's `.py` file doesn't allow writing (e.g., either due to insufficient permission or if it's part of a zip archive). The `-B` option to the interpreter also disables the generation of these files.

If `.pyc` and `.pyo` files are available, it is not necessary for a corresponding `.py` file to exist. Thus, if you are packaging code and don't wish to include source, you can merely bundle a set of `.pyc` files together. However, be aware that Python has extensive support for introspection and disassembly. Knowledgeable users will still be able to inspect and find out a lot of details about your program even if the source hasn't been provided. Also, be aware that `.pyc` files tend to be version-specific. Thus, a `.pyc` file generated for one version of Python might not work in a future release.

When `import` searches for files, it matches filenames in a case-sensitive manner—even on machines where the underlying file system is case-insensitive, such as on Windows and OS X (such systems are case-preserving, however). Therefore, `import foo` will only import the file `foo.py` and not the file `FOO.PY`. However, as a general rule, you should avoid the use of module names that differ in case only.

Module Reloading and Unloading

Python provides no real support for reloading or unloading of previously imported modules. Although you can remove a module from `sys.modules`, this does not generally unload a module from memory. This is because references to the module object may still exist in other program components that used `import` to load that module. Moreover, if there are instances of classes defined in the module, those instances contain references back to their class object, which in turn holds references to the module in which it was defined.

The fact that module references exist in many places makes it generally impractical to reload a module after making changes to its implementation. For example, if you remove a module from `sys.modules` and use `import` to reload it, this will not retroactively change all of the previous references to the module used in a program. Instead, you'll have one reference to the new module created by the most recent `import` statement and a set of references to the old module created by imports in other parts of the code. This is rarely what you want and never safe to use in any kind of sane production code unless you are able to carefully control the entire execution environment.

Older versions of Python provided a `reload()` function for reloading a module. However, use of this function was never really safe (for all of the aforementioned reasons), and its use was actively discouraged except as a possible debugging aid. Python 3 removes this feature entirely. So, it's best not to rely upon it.

Finally, it should be noted that C/C++ extensions to Python cannot be safely unloaded or reloaded in any way. No support is provided for this, and the underlying operating system may prohibit it anyways. Thus, your only recourse is to restart the Python interpreter process.

Packages

Packages allow a collection of modules to be grouped under a common package name. This technique helps resolve namespace conflicts between module names used in different applications. A package is defined by creating a directory with the same name as the package and creating the file `__init__.py` in that directory. You can then place additional source files, compiled extensions, and subpackages in this directory, as needed. For example, a package might be organized as follows:

```
Graphics/
      __init__.py
      Primitive/
         __init__.py
         lines.py
         fill.py
         text.py
         ...
      Graph2d/
         __init__.py
         plot2d.py
         ...
      Graph3d/
         __init__.py
         plot3d.py
         ...
      Formats/
         __init__.py
         gif.py

         png.py
         tiff.py
         jpeg.py
```

The `import` statement is used to load modules from a package in a number of ways:

- `import Graphics.Primitive.fill`

  This loads the submodule `Graphics.Primitive.fill`. The contents of this module have to be explicitly named, such as `Graphics.Primitive.fill.floodfill(img,x,y,color)`.

- `from Graphics.Primitive import fill`

  This loads the submodule `fill` but makes it available without the package prefix; for example, `fill.floodfill(img,x,y,color)`.

- `from Graphics.Primitive.fill import floodfill`

  This loads the submodule `fill` but makes the `floodfill` function directly accessible; for example, `floodfill(img,x,y,color)`.

Whenever any part of a package is first imported, the code in the file `__init__.py` is executed. Minimally, this file may be empty, but it can also contain code to perform package-specific initializations. All the `__init__.py` files encountered during an `import` are executed. Therefore, the statement `import Graphics.Primitive.fill`, shown earlier, would first execute the `__init__.py` file in the `Graphics` directory and then the `__init__.py` file in the `Primitive` directory.

One peculiar problem with packages is the handling of this statement:

```
from Graphics.Primitive import *
```

A programmer who uses this statement usually wants to import all the submodules associated with a package into the current namespace. However, because filename conventions vary from system to system (especially with regard to case sensitivity), Python cannot accurately determine what modules those might be. As a result, this statement just imports all the names that are defined in the `__init__.py` file in the `Primitive` directory. This behavior can be modified by defining a list, `__all__`, that contains all the module names associated with the package. This list should be defined in the package `__init__.py` file, like this:

```
# Graphics/Primitive/__init__.py
__all__ = ["lines","text","fill"]
```

Now when the user issues a `from Graphics.Primitive import *` statement, all the listed submodules are loaded as expected.

Another subtle problem with packages concerns submodules that want to import other submodules within the same package. For example, suppose the `Graphics.Primitive.fill` module wants to import the `Graphics.Primitive.lines` module. To do this, you can simply use the fully specified named (e.g., `from Graphics.Primitives import lines`) or use a package relative import like this:

```
# fill.py
from . import lines
```

In this example, the `.` used in the statement `from . import lines` refers to the same directory of the calling module. Thus, this statement looks for a module `lines` in the same directory as the file `fill.py`. Great care should be taken to avoid using a statement such as `import module` to import a package submodule. In older versions of Python, it was unclear whether the `import module` statement was referring to a standard library module or a submodule of a package. Older versions of Python would first try to load the module from the same package directory as the submodule where the `import` statement appeared and then move on to standard library modules if no match was found. However, in Python 3, `import` assumes an absolute path and will simply try to load *`module`* from the standard library. A relative import more clearly states your intentions.

Relative imports can also be used to load submodules contained in different directories of the same package. For example, if the module `Graphics.Graph2D.plot2d` wanted to import `Graphics.Primitives.lines`, it could use a statement like this:

```
# plot2d.py
from ..Primitives import lines
```

Here, the `..` moves out one directory level and `Primitives` drops down into a different package directory.

Relative imports can only be specified using the `from module import symbol` form of the import statement. Thus, statements such as `import ..Primitives.lines` or `import .lines` are a syntax error. Also, *`symbol`* has to be a valid identifier. So, a statement such as `from .. import Primitives.lines` is also illegal. Finally, relative imports can only be used within a package; it is illegal to use a relative import to refer to modules that are simply located in a different directory on the filesystem.

Importing a package name alone doesn't import all the submodules contained in the package. For example, the following code doesn't work:

```
import Graphics
Graphics.Primitive.fill.floodfill(img,x,y,color)  # Fails!
```

However, because the `import Graphics` statement executes the `__init__.py` file in the `Graphics` directory, relative imports can be used to load all the submodules automatically, as follows:

```
# Graphics/__init__.py
from . import Primitive, Graph2d, Graph3d

# Graphics/Primitive/__init__.py
from . import lines, fill, text, ...
```

Now the `import Graphics` statement imports all the submodules and makes them available using their fully qualified names. Again, it is important to stress that a package relative import should be used as shown. If you use a simple statement such as `import module`, standard library modules may be loaded instead.

Finally, when Python imports a package, it defines a special variable, `__path__`, which contains a list of directories that are searched when looking for package submodules (`__path__` is a package-specific version of the `sys.path` variable). `__path__` is accessible to the code contained in `__init__.py` files and initially contains a single item with the directory name of the package. If necessary, a package can supply additional directories to the `__path__` list to alter the search path used for finding submodules. This might be useful if the organization of a package on the file system is complicated and doesn't neatly match up with the package hierarchy.

Distributing Python Programs and Libraries

To distribute Python programs to others, you should use the `distutils` module. As preparation, you should first cleanly organize your work into a directory that has a `README` file, supporting documentation, and your source code. Typically, this directory will contain a mix of library modules, packages, and scripts. Modules and packages refer to source files that will be loaded with `import` statements. Scripts are programs that will run as the main program to the interpreter (e.g., running as `python scriptname`). Here is an example of a directory containing Python code:

```
spam/
     README.txt
     Documentation.txt
     libspam.py          # A single library module
     spampkg/            # A package of support modules
         __init__.py
         foo.py
         bar.py
     runspam.py          # A script to run as: python runspam.py
```

You should organize your code so that it works normally when running the Python interpreter in the top-level directory. For example, if you start Python in the `spam` directory, you should be able to import modules, import package components, and run scripts without having to alter any of Python's settings such as the module search path.

After you have organized your code, create a file `setup.py` in the top most directory (`spam` in the previous examples). In this file, put the following code:

```
# setup.py
from distutils.core import setup

setup(name = "spam",
            version = "1.0",
            py_modules = ['libspam'],
            packages = ['spampkg'],
            scripts = ['runspam.py'],
            )
```

In the `setup()` call, the `py_modules` argument is a list of all of the single-file Python modules, `packages` is a list of all package directories, and `scripts` is a list of script files. Any of these arguments may be omitted if your software does not have any matching components (i.e., there are no scripts). `name` is the name of your package, and `version` is the version number as a string.

The call to `setup()` supports a variety of other parameters that supply various metadata about your package. Table 8.1 shows the most common parameters that can be specified. All values are strings except for the `classifiers` parameter, which is a list of strings such as `['Development Status :: 4 - Beta','Programming Language :: Python']` (a full list can be found at http://pypi.python.org).

Table 8.1 **Parameters to `setup()`**

Parameter	Description
`name`	Name of the package (required)
`version`	Version number (required)
`author`	Author's name
`author_email`	Author's email address

Table 8.1 **Continued**

Parameter	Description
`maintainer`	Maintainer's name
`maintainer_email`	Maintainer's email
`url`	Home page for the package
`description`	Short description of the package
`long_description`	Long description of the package
`download_url`	Location where package can be downloaded
`classifiers`	List of string classifiers

Creating a `setup.py` file is enough to create a source distribution of your software. Type the following shell command to make a source distribution:

```
% python setup.py sdist
...
%
```

This creates an archive file such as `spam-1.0.tar.gz` or `spam-1.0.zip` in the directory `spam/dist`. This is the file you would give to others to install your software. To install, a user simply unpacks the archive and performs these steps:

```
% unzip spam-1.0.zip
...
% cd spam-1.0
% python setup.py install
...
%
```

This installs the software into the local Python distribution and makes it available for general use. Modules and packages are normally installed into a directory called `"site-packages"` in the Python library. To find the exact location of this directory, inspect the value of `sys.path`. Scripts are normally installed into the same directory as the Python interpreter on UNIX-based systems or into a `"Scripts"` directory on Windows (found in `"C:\Python26\Scripts"` in a typical installation).

On UNIX, if the first line of a script starts with `#!` and contains the text `"python"`, the installer will rewrite the line to point to the local installation of Python. Thus, if you have written scripts that have been hard-coded to a specific Python location such as `/usr/local/bin/python`, they should still work when installed on other systems where Python is in a different location.

The `setup.py` file has a number of other commands concerning the distribution of software. If you type `'python setup.py bdist'`, a binary distribution is created in which all of the `.py` files have already been precompiled into `.pyc` files and placed into a directory structure that mimics that of the local platform. This kind of distribution is needed only if parts of your application have platform dependencies (for example, if you also have C extensions that need to be compiled). If you run `'python setup.py bdist_wininst'` on a Windows machine, an `.exe` file will be created. When opened, a Windows installer dialog will start, prompting the user for information about where the software should be installed. This kind of distribution also adds entries to the registry, making it easy to uninstall your package at a later date.

The `distutils` module assumes that users already have a Python installation on their machine (downloaded separately). Although it is possible to create software packages where the Python runtime and your software are bundled together into a single binary executable, that is beyond the scope of what can be covered here (look at a third-party module such as `py2exe` or `py2app` for further details). If all you are doing is distributing libraries or simple scripts to people, it is usually unnecessary to package your code with the Python interpreter and runtime as well.

Finally, it should be noted that there are many more options to `distutils` than those covered here. Chapter 26 describes how `distutils` can be used to compile C and C++ extensions.

Although not part of the standard Python distribution, Python software is often distributed in the form of an `.egg` file. This format is created by the popular `setuptools` extension (http://pypi.python.org/pypi/setuptools). To support `setuptools`, you can simply change the first part of your `setup.py` file as follows:

```
# setup.py
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

setup(name = "spam",
      ...
)
```

Installing Third-Party Libraries

The definitive resource for locating third-party libraries and extensions to Python is the *Python Package Index (PyPI)*, which is located at http://pypi.python.org. Installing third-party modules is usually straightforward but can become quite involved for very large packages that also depend on other third-party modules. For the more major extensions, you will often find a platform-native installer that simply steps you through the process using a series of dialog screens. For other modules, you typically unpack the download, look for the `setup.py` file, and type **`python setup.py install`** to install the software.

By default, third-party modules are installed in the `site-packages` directory of the Python standard library. Access to this directory typically requires root or administrator access. If this is not the case, you can type **`python setup.py install --user`** to have the module installed in a per-user library directory. This installs the package in a per-user directory such as `"/Users/beazley/.local/lib/python2.6/site-packages"` on UNIX.

If you want to install the software somewhere else entirely, use the `--prefix` option to `setup.py`. For example, typing `python setup.py install --prefix=/home/beazley/pypackages` installs a module under the directory `/home/beazley/pypackages`. When installing in a nonstandard location, you will probably have to adjust the setting of `sys.path` in order for Python to locate your newly installed modules.

Be aware that many extensions to Python involve C or C++ code. If you have downloaded a source distribution, your system will have to have a C++ compiler installed in order to run the installer. On UNIX, Linux, and OS X, this is usually not an issue. On Windows, it has traditionally been necessary to have a version of Microsoft Visual Studio installed. If you're working on that platform, you're probably better off looking for a precompiled version of your extension.

If you have installed `setuptools`, a script `easy_install` is available to install packages. Simply type **`easy_install`** ***`pkgname`*** to install a specific package. If configured correctly, this will download the appropriate software from PyPI along with any dependencies and install it for you. Of course, your mileage might vary.

If you would like to add your own software to PyPI, simply type `python` **`setup.py register`**. This will upload metadata about the latest version of your software to the index (note that you will have to register a username and password first).

9

Input and Output

This chapter describes the basics of Python input and output (I/O), including command-line options, environment variables, file I/O, Unicode, and how to serialize objects using the `pickle` module.

Reading Command-Line Options

When Python starts, command-line options are placed in the list `sys.argv`. The first element is the name of the program. Subsequent items are the options presented on the command line *after* the program name. The following program shows a minimal prototype of manually processing simple command-line arguments:

```
import sys
if len(sys.argv) != 3:
    sys.stderr.write("Usage : python %s inputfile outputfile\n" % sys.argv[0])
    raise SystemExit(1)
inputfile  = sys.argv[1]
outputfile = sys.argv[2]
```

In this program, `sys.argv[0]` contains the name of the script being executed. Writing an error message to `sys.stderr` and raising `SystemExit` with a non-zero exit code as shown is standard practice for reporting usage errors in command-line tools.

Although you can manually process command options for simple scripts, use the `optparse` module for more complicated command-line handling. Here is a simple example:

```
import optparse
p = optparse.OptionParser()

# An option taking an argument
p.add_option("-o",action="store",dest="outfile")
p.add_option("--output",action="store",dest="outfile")

# An option that sets a boolean flag
p.add_option("-d",action="store_true",dest="debug")
p.add_option("--debug",action="store_true",dest="debug")

# Set default values for selected options
p.set_defaults(debug=False)

# Parse the command line
opts, args = p.parse_args()

# Retrieve the option settings
outfile   = opts.outfile
debugmode = opts.debug
```

In this example, two types of options are added. The first option, `-o` or `--output`, has a required argument. This behavior is selected by specifying `action='store'` in the call to `p.add_option()`. The second option, `-d` or `--debug`, is merely setting a Boolean flag. This is enabled by specifying `action='store_true'` in `p.add_option()`. The `dest` argument to `p.add_option()` selects an attribute name where the argument value will be stored after parsing. The `p.set_defaults()` method sets default values for one or more of the options. The argument names used with this method should match the destination names selected for each option. If no default value is selected, the default value is set to `None`.

The previous program recognizes all of the following command-line styles:

```
% python prog.py -o outfile -d infile1 ... infileN
% python prog.py --output=outfile --debug infile1 ... infileN
% python prog.py -h
% python prog.py --help
```

Parsing is performed using the `p.parse_args()` method. This method returns a 2-tuple `(opts, args)` where `opts` is an object containing the parsed option values and `args` is a list of items on the command line not parsed as options. Option values are retrieved using `opts.dest` where *`dest`* is the destination name used when adding an option. For example, the argument to the `-o` or `--output` argument is placed in `opts.outfile`, whereas `args` is a list of the remaining arguments such as `['infile1', ..., 'infileN']`. The `optparse` module automatically provides a `-h` or `--help` option that lists the available options if requested by the user. Bad options also result in an error message.

This example only shows the simplest use of the `optparse` module. Further details on some of the more advanced options can be found in Chapter 19, "Operating System Services."

Environment Variables

Environment variables are accessed in the dictionary `os.environ`. Here's an example:

```
import os
path   = os.environ["PATH"]
user   = os.environ["USER"]
editor = os.environ["EDITOR"]
... etc ...
```

To modify the environment variables, set the `os.environ` variable. For example:

```
os.environ["FOO"] = "BAR"
```

Modifications to `os.environ` affect both the running program and subprocesses created by Python.

Files and File Objects

The built-in function `open(`*`name`* `[,`*`mode`* `[,`*`bufsize`*`]])` opens and creates a file object, as shown here:

```
f = open("foo")        # Opens "foo" for reading
f = open("foo",'r')    # Opens "foo" for reading (same as above)
f = open("foo",'w')    # Open for writing
```

The file mode is `'r'` for read, `'w'` for write, or `'a'` for append. These file modes assume text-mode and may implicitly perform translation of the newline character `'\n'`. For example, on Windows, writing the character `'\n'` actually outputs the two-character sequence `'\r\n'` (and when reading the file back, `'\r\n'` is translated back into a single `'\n'` character). If you are working with binary data, append a `'b'` to the file mode such as `'rb'` or `'wb'`. This disables newline translation and should be included if you are concerned about portability of code that processes binary data (on UNIX, it is a common mistake to omit the `'b'` because there is no distinction between text and binary files). Also, because of the distinction in modes, you might see text-mode specified as `'rt'`, `'wt'`, or `'at'`, which more clearly expresses your intent.

A file can be opened for in-place updates by supplying a plus (+) character, such as `'r+'` or `'w+'`. When a file is opened for update, you can perform both input and output, as long as all output operations flush their data before any subsequent input operations. If a file is opened using `'w+'` mode, its length is first truncated to zero.

If a file is opened with mode `'U'` or `'rU'`, it provides universal newline support for reading. This feature simplifies cross-platform work by translating different newline encodings (such as `'\n'`, `'\r'`, and `'\r\n'`) to a standard `'\n'` character in the strings returned by various file I/O functions. This can be useful if, for example, you are writing scripts on UNIX systems that must process text files generated by programs on Windows.

The optional *`bufsize`* parameter controls the buffering behavior of the file, where 0 is unbuffered, 1 is line buffered, and a negative number requests the system default. Any other positive number indicates the approximate buffer size in bytes that will be used.

Python 3 adds four additional parameters to the `open()` function, which is called as `open(`*`name`* `[,`*`mode`* `[,`*`bufsize`* `[,` *`encoding`* `[,` *`errors`* `[,` *`newline`* `[,` *`closefd`*`]]]]])`. *`encoding`* is an encoding name such as `'utf-8'` or `'ascii'`. *`errors`* is the error-handling policy to use for encoding errors (see the later sections in this chapter on Unicode for more information). *`newline`* controls the behavior of universal newline mode and is set to `None`, `''`, `'\n'`, `'\r'`, or `'\r\n'`. If set to `None`, any line ending of the form `'\n'`, `'\r'`, or `'\r\n'` is translated into `'\n'`. If set to `''` (the empty string), any of these line endings are recognized as newlines, but left untranslated in the input text. If *`newline`* has any other legal value, that value is what is used to terminate lines. *`closefd`* controls whether the underlying file descriptor is actually closed when the `close()` method is invoked. By default, this is set to `True`.

Table 9.1 shows the methods supported by `file` objects.

Table 9.1 **File Methods**

Method	Description
`f``.read([`*`n`*`])`	Reads at most *`n`* bytes.
`f``.readline([`*`n`*`])`	Reads a single line of input up to *`n`* characters. If *`n`* is omitted, this method reads the entire line.
`f``.readlines([size])`	Reads all the lines and returns a list. *`size`* optionally specifies the approximate number of characters to read on the file before stopping.
`f``.write(`*`s`*`)`	Writes string *`s`*.
`f``.writelines(`*`lines`*`)`	Writes all strings in sequence *`lines`*.
`f``.close()`	Closes the file.
`f``.tell()`	Returns the current file pointer.
`f``.seek(`*`offset`* `[,` *`whence`*`])`	Seeks to a new file position.
`f``.isatty()`	Returns `1` if *`f`* is an interactive terminal.
`f``.flush()`	Flushes the output buffers.
`f``.truncate([`*`size`*`])`	Truncates the file to at most *`size`* bytes.
`f``.fileno()`	Returns an integer file descriptor.
`f``.next()`	Returns the next line or raises `StopIteration`. In Python 3, it is called *`f`*`.__next__()`.

The `read()` method returns the entire file as a string unless an optional *`length`* parameter is given specifying the maximum number of characters. The `readline()` method returns the next line of input, including the terminating newline; the `readlines()` method returns all the input lines as a list of strings. The `readline()` method optionally accepts a maximum line length, *`n`*. If a line longer than *`n`* characters is read, the first *`n`* characters are returned. The remaining line data is not discarded and will be returned on subsequent read operations. The `readlines()` method accepts a size parameter that specifies the approximate number of characters to read before stopping. The actual number of characters read may be larger than this depending on how much data has been buffered.

Both the `readline()` and `readlines()` methods are platform-aware and handle different representations of newlines properly (for example, `'\n'` versus `'\r\n'`). If the file is opened in universal newline mode (`'U'` or `'rU'`), newlines are converted to `'\n'`.

`read()` and `readline()` indicate end-of-file (EOF) by returning an empty string. Thus, the following code shows how you can detect an EOF condition:

```
while True:
    line = f.readline()
    if not line:        # EOF
        break
```

A convenient way to read all lines in a file is to use iteration with a `for` loop. For example:

```
for line in f:        # Iterate over all lines in the file
    # Do something with line
    ...
```

Be aware that in Python 2, the various read operations always return 8-bit strings, regardless of the file mode that was specified (text or binary). In Python 3, these operations return Unicode strings if a file has been opened in text mode and byte strings if the file is opened in binary mode.

The `write()` method writes a string to the file, and the `writelines()` method writes a list of strings to the file. `write()` and `writelines()` do not add newline characters to the output, so all output that you produce should already include all necessary formatting. These methods can write raw-byte strings to a file, but only if the file has been opened in binary mode.

Internally, each file object keeps a file pointer that stores the byte offset at which the next read or write operation will occur. The `tell()` method returns the current value of the file pointer as a long integer. The `seek()` method is used to randomly access parts of a file given an *`offset`* and a placement rule in *`whence`*. If *`whence`* is `0` (the default), `seek()` assumes that *`offset`* is relative to the start of the file; if *`whence`* is `1`, the position is moved relative to the current position; and if *`whence`* is `2`, the offset is taken from the end of the file. `seek()` returns the new value of the file pointer as an integer. It should be noted that the file pointer is associated with the file object returned by `open()` and not the file itself. The same file can be opened more than once in the same program (or in different programs). Each instance of the open file has its own file pointer that can be manipulated independently.

The `fileno()` method returns the integer file descriptor for a file and is sometimes used in low-level I/O operations in certain library modules. For example, the `fcntl` module uses the file descriptor to provide low-level file control operations on UNIX systems.

File objects also have the read-only data attributes shown in Table 9.2.

Table 9.2 **File Object Attributes**

Attribute	Description
`f``.closed`	Boolean value indicates the file state: `False` if the file is open, `True` if closed.
`f``.mode`	The I/O mode for the file.
`f``.name`	Name of the file if created using `open()`. Otherwise, it will be a string indicating the source of the file.
`f``.softspace`	Boolean value indicating whether a space character needs to be printed before another value when using the `print` statement. Classes that emulate files must provide a writable attribute of this name that's initially initialized to zero (Python 2 only).
`f``.newlines`	When a file is opened in universal newline mode, this attribute contains the newline representation actually found in the file. The value is `None` if no newlines have been encountered, a string containing `'\n'`, `'\r'`, or `'\r\n'`, or a tuple containing all the different newline encodings seen.
`f``.encoding`	A string that indicates file encoding, if any (for example, `'latin-1'` or `'utf-8'`). The value is `None` if no encoding is being used.

Standard Input, Output, and Error

The interpreter provides three standard file objects, known as *standard input*, *standard output*, and *standard error*, which are available in the `sys` module as `sys.stdin`, `sys.stdout`, and `sys.stderr`, respectively. `stdin` is a file object corresponding to the stream of input characters supplied to the interpreter. `stdout` is the file object that receives output produced by `print`. `stderr` is a file that receives error messages. More often than not, `stdin` is mapped to the user's keyboard, whereas `stdout` and `stderr` produce text onscreen.

The methods described in the preceding section can be used to perform raw I/O with the user. For example, the following code writes to standard output and reads a line of input from standard input:

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

Alternatively, the built-in function `raw_input(`*`prompt`*`)` can read a line of text from `stdin` and optionally print a prompt:

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

Lines read by `raw_input()` do not include the trailing newline. This is different than reading directly from `sys.stdin` where newlines are included in the input text. In Python 3, `raw_input()` has been renamed to `input()`.

Keyboard interrupts (typically generated by Ctrl+C) result in a `KeyboardInterrupt` exception that can be caught using an exception handler.

If necessary, the values of `sys.stdout`, `sys.stdin`, and `sys.stderr` can be replaced with other file objects, in which case the `print` statement and input functions use the new values. Should it ever be necessary to restore the original value of `sys.stdout`, it should be saved first. The original values of `sys.stdout`, `sys.stdin`, and `sys.stderr` at interpreter startup are also available in `sys.__stdout__`, `sys.__stdin__`, and `sys.__stderr__`, respectively.

Note that in some cases `sys.stdin`, `sys.stdout`, and `sys.stderr` may be altered by the use of an integrated development environment (IDE). For example, when Python is run under IDLE, `sys.stdin` is replaced with an object that behaves like a file but is really an object in the development environment. In this case, certain low-level methods, such as `read()` and `seek()`, may be unavailable.

The `print` Statement

Python 2 uses a special `print` statement to produce output on the file contained in `sys.stdout`. `print` accepts a comma-separated list of objects such as the following:

```
print "The values are", x, y, z
```

For each object, the `str()` function is invoked to produce an output string. These output strings are then joined and separated by a single space to produce the final output string. The output is terminated by a newline unless a trailing comma is supplied to the `print` statement. In this case, the next `print` statement will insert a space before printing more items. The output of this space is controlled by the `softspace` attribute of the file being used for output.

```
print "The values are ", x, y, z, w
# Print the same text, using two print statements
print "The values are ", x, y,   # Omits trailing newline
print z, w                       # A space is printed before z
```

To produce formatted output, use the string-formatting operator (`%`) or the `.format()` method as described in Chapter 4, "Operators and Expressions." Here's an example:

```
print "The values are %d %7.5f %s" % (x,y,z) # Formatted I/O
print "The values are {0:d} {1:7.5f} {2}".format(x,y,z)
```

You can change the destination of the `print` statement by adding the special `>>`*`file`* modifier followed by a comma, where *`file`* is a file object that allows writes. Here's an example:

```
f = open("output","w")
print >>f, "hello world"
...
f.close()
```

The `print()` Function

One of the most significant changes in Python 3 is that `print` is turned into a function. In Python 2.6, it is also possible to use `print` as a function if you include the statement `from __future__ import print_function` in each module where used. The `print()` function works almost exactly the same as the `print` statement described in the previous section.

To print a series of values separated by spaces, just supply them all to `print()` like this:

```
print("The values are", x, y, z)
```

To suppress or change the line ending, use the `end=`*`ending`* keyword argument. For example:

```
print("The values are", x, y, z, end='')   # Suppress the newline
```

To redirect the output to a file, use the `file=`*`outfile`* keyword argument. For example:

```
print("The values are", x, y, z, file=f)   # Redirect to file object f
```

To change the separator character between items, use the `sep=`*`sepchr`* keyword argument. For example:

```
print("The values are", x, y, z, sep=',')   # Put commas between the values
```

Variable Interpolation in Text Output

A common problem when generating output is that of producing large text fragments containing embedded variable substitutions. Many scripting languages such as Perl and PHP allow variables to be inserted into strings using dollar-variable substitutions (that is, `$name`, `$address`, and so on). Python provides no direct equivalent of this feature, but it can be emulated using formatted I/O combined with triple-quoted strings. For example, you could write a short form letter, filling in a `name`, an `item` name, and an `amount`, as shown in the following example:

```
# Note: trailing slash right after """ prevents
# a blank line from appearing as the first line
form = """\
Dear  %(name)s,
Please send back my %(item)s or pay me $%(amount)0.2f.
                                    Sincerely yours,

                                    Joe Python User
"""
print form % { 'name': 'Mr. Bush',
               'item': 'blender',
               'amount': 50.00,
                 }
```

This produces the following output:

```
Dear Mr. Bush,

Please send back my blender or pay me $50.00.

                                    Sincerely yours,

                                    Joe Python User
```

The `format()` method is a more modern alternative that cleans up some of the previous code. For example:

```
form = """\
Dear {name},
Please send back my {item} or pay me {amount:0.2f}.
                                    Sincerely yours,

                                    Joe Python User
"""
print form.format(name='Mr. Bush', item='blender', amount=50.0)
```

For certain kinds of forms, you can also use `Template` strings, as follows:

```
import string
form = string.Template("""\
Dear  $name,
Please send back my $item or pay me $amount.
                                    Sincerely yours,

                                    Joe Python User
""")
print form.substitute({'name': 'Mr. Bush',
                       'item': 'blender',
                       'amount': "%0.2f" % 50.0})
```

In this case, special `$` variables in the string indicate substitutions. The `form.substitute()` method takes a dictionary of replacements and returns a new string. Although the previous approaches are simple, they aren't always the most powerful solutions to text generation. Web frameworks and other large application frameworks tend to provide their own template string engines that support embedded control-flow, variable substitutions, file inclusion, and other advanced features.

Generating Output

Working directly with files is the I/O model most familiar to programmers. However, generator functions can also be used to emit an I/O stream as a sequence of data fragments. To do this, simply use the `yield` statement like you would use a `write()` or `print` statement. Here is an example:

```
def countdown(n):
    while n > 0:
         yield "T-minus %d\n" % n
         n -= 1
    yield "Kaboom!\n"
```

Producing an output stream in this manner provides great flexibility because the production of the output stream is decoupled from the code that actually directs the stream to its intended destination. For example, if you wanted to route the above output to a file *`f`*, you could do this:

```
count = countdown(5)
f.writelines(count)
```

If, instead, you wanted to redirect the output across a socket *`s`*, you could do this:

```
for chunk in count:
    s.sendall(chunk)
```

Or, if you simply wanted to capture all of the output in a string, you could do this:

```
out = "".join(count)
```

More advanced applications can use this approach to implement their own I/O buffering. For example, a generator could be emitting small text fragments, but another function could be collecting the fragments into large buffers to create a larger, more efficient I/O operation:

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
         outf.write("".join(chunks))
         chunks.clear()
         buffered_size = 0
outf.write("".join(chunks)
```

For programs that are routing output to files or network connections, a generator approach can also result in a significant reduction in memory use because the entire output stream can often be generated and processed in small fragments as opposed to being first collected into one large output string or list of strings. This approach to output is sometimes seen when writing programs that interact with the Python Web Services Gateway Interface (WSGI) that's used to communicate between components in certain web frameworks.

Unicode String Handling

A common problem associated with I/O handling is that of dealing with international characters represented as Unicode. If you have a string *`s`* of raw bytes containing an encoded representation of a Unicode string, use the *`s`*`.decode([`*`encoding`*` [,`*`errors`*`]])` method to convert it into a proper Unicode string. To convert a Unicode string, *`u`*, to an encoded byte string, use the string method *`u`*`.encode([`*`encoding`*` [,`*`errors`*`]])`. Both of these conversion operators require the use of a special encoding name that specifies how Unicode character values are mapped to a sequence of 8-bit characters in byte strings, and vice versa. The encoding parameter is specified as a string

and is one of more than a hundred different character encodings. The following values, however, are most common:

Value	Description
`'ascii'`	7-bit ASCII
`'latin-1'` or `'iso-8859-1'`	ISO 8859-1 Latin-1
`'cp1252'`	Windows 1252 encoding
`'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 many 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]`. However, `'utf-8'` is also a very common setting. Technical details concerning common encodings appears in a later section.

When using the *`s`*`.decode()` method, it is always assumed that *`s`* is a string of bytes. In Python 2, this means that *`s`* is a standard string, but in Python 3, *`s`* must be a special `bytes` type. Similarly, the result of *`t`*`.encode()` is always a byte sequence. One caution if you care about portability is that these methods are a little muddled in Python 2. For instance, Python 2 strings have both `decode()` and `encode()` methods, whereas in Python 3, strings only have an `encode()` method and the `bytes` type only has a `decode()` method. To simplify code in Python 2, make sure you only use `encode()` on Unicode strings and `decode()` on byte strings.

When string values are being converted, a `UnicodeError` exception might be raised if a character that can't be converted is encountered. For instance, if you are trying to encode a string into `'ascii'` and it contains a Unicode character such as U+1F28, you will get an encoding error because this character value is too large to be represented in the ASCII character set. The `errors` parameter of the `encode()` and `decode()` methods determines how encoding errors are handled. It's a string with one of the following values:

Value	Description
`'strict'`	Raises a `UnicodeError` exception for encoding and 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'`.

The `'xmlcharrefreplace'` error handling policy is often a useful way to embed international characters into ASCII-encoded text on web pages. For example, if you output the Unicode string `'Jalape\u00f1o'` by encoding it to ASCII with `'xmlcharrefreplace'` handling, browsers will almost always correctly render the output text as "Jalapeño" and not some garbled alternative.

To keep your brain from exploding, encoded byte strings and unencoded strings should never be mixed together in expressions (for example, using + to concatenate). Python 3 prohibits this altogether, but Python 2 will silently go ahead with such operations by automatically promoting byte strings to Unicode according to the default encoding setting. This behavior is often a source of surprising results or inexplicable error messages. Thus, you should carefully try to maintain a strict separation between encoded and unencoded character data in your program.

Unicode I/O

When working with Unicode strings, it is never possible to directly write raw Unicode data to a file. This is due to the fact that Unicode characters are internally represented as multibyte integers and that writing such integers directly to an output stream causes problems related to byte ordering. For example, you would have to arbitrarily decide if the Unicode character `U+HHLL` is to be written in "little endian" format as the byte sequence `LL HH` or in "big endian" format as the byte sequence `HH LL`. Moreover, other tools that process Unicode would have to know which encoding you used.

Because of this problem, the external representation of Unicode strings is always done according to a specific encoding rule that precisely defines how Unicode characters are to be represented as a byte sequence. Thus, to support Unicode I/O, the encoding and decoding concepts described in the previous section are extended to files. The built-in `codecs` module contains a collection of functions for converting byte-oriented data to and from Unicode strings according to a variety of different data-encoding schemes.

Perhaps the most straightforward way to handle Unicode files is to use the `codecs.open(`*`filename`* `[, `*`mode`* `[, `*`encoding`* `[, `*`errors`*`]]])` function, as follows:

```
f = codecs.open('foo.txt','r','utf-8','strict')     # Reading
g = codecs.open('bar.txt','w','utf-8')              # Writing
```

This creates a file object that reads or writes Unicode strings. The encoding parameter specifies the underlying character encoding that will be used to translate data as it is read or written to the file. The *`errors`* parameter determines how errors are handled and is one of `'strict'`, `'ignore'`, `'replace'`, `'backslashreplace'`, or `'xmlcharrefreplace'` as described in the previous section.

If you already have a file object, the `codecs.EncodedFile(`*`file, inputenc`* `[, `*`outputenc`* `[, `*`errors`*`]])` function can be used to place an encoding wrapper around it. Here's an example:

```
f = open("foo.txt","rb")
...
fenc = codecs.EncodedFile(f,'utf-8')
```

In this case, data read from the file will be interpreted according to the encoding supplied in *`inputenc`*. Data written to the file will be interpreted according to the encoding in *`inputenc`* and written according to the encoding in *`outputenc`*. If *`outputenc`* is omitted, it defaults to the same as *`inputenc`*. *`errors`* has the same meaning as described earlier. When putting an `EncodedFile` wrapper around an existing file, make sure that file is in binary mode. Otherwise, newline translation might break the encoding.

When you're working with Unicode files, the data encoding is often embedded in the file itself. For example, XML parsers may look at the first few bytes of the string `'<?xml ...>'` to determine the document encoding. If the first four values are `3C 3F 78 6D` (`'<?xm'`), the encoding is assumed to be UTF-8. If the first four values are `00 3C 00 3F` or `3C 00 3F 00`, the encoding is assumed to be UTF-16 big endian or UTF-16 little endian, respectively. Alternatively, a document encoding may appear in MIME headers or as an attribute of other document elements. Here's an example:

```
<?xml ... encoding="ISO-8859-1" ... ?>
```

Similarly, Unicode files may also include special byte-order markers (BOM) that indicate properties of the character encoding. The Unicode character `U+FEFF` is reserved for this purpose. Typically, the marker is written as the first character in the file. Programs then read this character and look at the arrangement of the bytes to determine encoding (for example, `'\xff\xfe'` for UTF-16-LE or `'\xfe\xff'` UTF-16-BE). Once the encoding is determined, the BOM character is discarded and the remainder of the file is processed. Unfortunately, all of this extra handling of the BOM is not something that happens behind the scenes. You often have to take care of this yourself if your application warrants it.

When the encoding is read from a document, code similar to the following can be used to turn the input file into an encoded stream:

```
f = open("somefile","rb")
# Determine encoding of the file
...
# Put an appropriate encoding wrapper on the file.
# Assumes that the BOM (if any) has already been discarded
# by earlier statements.
fenc = codecs.EncodedFile(f,encoding)
data = fenc.read()
```

Unicode Data Encodings

Table 9.3 lists some of the most commonly used encoders in the `codecs` module.

Table 9.3 **Encoders in the `codecs` Module**

Encoder	Description
`'ascii'`	ASCII encoding
`'latin-1'`, `'iso-8859-1'`	Latin-1 or ISO-8859-1 encoding
`'cp437'`	CP437 encoding
`'cp1252'`	CP1252 encoding
`'utf-8'`	8-bit variable-length encoding
`'utf-16'`	16-bit variable-length encoding
`'utf-16-le'`	UTF-16, but with explicit little endian encoding
`'utf-16-be'`	UTF-16, but with explicit big endian encoding
`'unicode-escape'`	Same format as `u"`*`string`*`"`
`'raw-unicode-escape'`	Same format as `ur"`*`string`*`"`

The following sections describe each of the encoders in more detail.

`'ascii'` Encoding

In `'ascii'` encoding, character values are confined to the ranges `[0x00,0x7f]` and `[U+0000, U+007F]`. Any character outside this range is invalid.

`'iso-8859-1'`, `'latin-1'` Encoding

Characters can be any 8-bit value in the ranges `[0x00,0xff]` and `[U+0000, U+00FF]`. Values in the range `[0x00,0x7f]` correspond to characters from the ASCII character set. Values in the range `[0x80,0xff]` correspond to characters from the ISO-8859-1 or extended ASCII character set. Any characters with values outside the range `[0x00,0xff]` result in an error.

`'cp437'` Encoding

This encoding is similar to `'iso-8859-1'` but is the default encoding used by Python when it runs as a console application on Windows. Certain characters in the range `[x80,0xff]` correspond to special symbols used for rendering menus, windows, and frames in legacy DOS applications.

`'cp1252'` Encoding

This is an encoding that is very similar to `'iso-8859-1'` used on Windows. However, this encoding defines characters in the range `[0x80-0x9f]` that are undefined in `'iso-8859-1'` and which have different code points in Unicode.

`'utf-8'` Encoding

UTF-8 is a variable-length encoding that allows all Unicode characters to be represented. A single byte is used to represent ASCII characters in the range 0–127. All other characters are represented by multibyte sequences of 2 or 3 bytes. The encoding of these bytes is shown here:

Unicode Characters	Byte 0	Byte 1	Byte 2
U+0000 - U+007F	0*nnnnnnn*		
U+007F - U+07FF	110*nnnnn*	10*nnnnnn*	
U+0800 - U+FFFF	1110*nnnn*	10*nnnnnn*	10*nnnnnn*

For 2-byte sequences, the first byte always starts with the bit sequence `110`. For 3-byte sequences, the first byte starts with the bit sequence `1110`. All subsequent data bytes in multibyte sequences start with the bit sequence `10`.

In full generality, the UTF-8 format allows for multibyte sequences of up to 6 bytes. In Python, 4-byte UTF-8 sequences are used to encode a pair of Unicode characters

known as a *surrogate pair*. Both characters have values in the range `[U+D800, U+DFFF]` and are combined to encode a 20-bit character value. The surrogate encoding is as follows: The 4-byte sequence `11110`*nnn* `10`*nnnnnn* `10`*nnmmmm* `10`*mmmmm* is encoded as the pair `U+D800 + N, U+DC00 + M`, where `N` is the upper 10 bits and `M` is the lower 10 bits of the 20-bit character encoded in the 4-byte UTF-8 sequence. Five- and 6-byte UTF-8 sequences (denoted by starting bit sequences of `111110` and `1111110`, respectively) are used to encode character values up to 32 bits in length. These values are not supported by Python and currently result in a `UnicodeError` exception if they appear in an encoded data stream.

UTF-8 encoding has a number of useful properties that allow it to be used by older software. First, the standard ASCII characters are represented in their standard encoding. This means that a UTF-8–encoded ASCII string is indistinguishable from a traditional ASCII string. Second, UTF-8 doesn't introduce embedded NULL bytes for multibyte character sequences. Therefore, existing software based on the C library and programs that expect NULL-terminated 8-bit strings will work with UTF-8 strings. Finally, UTF-8 encoding preserves the lexicographic ordering of strings. That is, if `a` and `b` are Unicode strings and `a < b`, then `a < b` also holds when `a` and `b` are converted to UTF-8. Therefore, sorting algorithms and other ordering algorithms written for 8-bit strings will also work for UTF-8.

`'utf-16'`, `'utf-16-be'`, and `'utf-16-le'` Encoding

UTF-16 is a variable-length 16-bit encoding in which Unicode characters are written as 16-bit values. Unless a byte ordering is specified, big endian encoding is assumed. In addition, a byte-order marker of `U+FEFF` can be used to explicitly specify the byte ordering in a UTF-16 data stream. In big endian encoding, `U+FEFF` is the Unicode character for a zero-width nonbreaking space, whereas the reversed value `U+FFFE` is an illegal Unicode character. Thus, the encoder can use the byte sequence `FE FF` or `FF FE` to determine the byte ordering of a data stream. When reading Unicode data, Python removes the byte-order markers from the final Unicode string.

`'utf-16-be'` encoding explicitly selects UTF-16 big endian encoding. `'utf-16-le'` encoding explicitly selects UTF-16 little ending encoding.

Although there are extensions to UTF-16 to support character values greater than 16 bits, none of these extensions are currently supported.

`'unicode-escape'` and `'raw-unicode-escape'` Encoding

These encoding methods are used to convert Unicode strings to the same format as used in Python Unicode string literals and Unicode raw string literals. Here's an example:

```
s = u'\u14a8\u0345\u2a34'
t = s.encode('unicode-escape')   #t = '\u14a8\u0345\u2a34'
```

Unicode Character Properties

In addition to performing I/O, programs that use Unicode may need to test Unicode characters for various properties such as capitalization, numbers, and whitespace. The `unicodedata` module provides access to a database of character properties. General character properties can be obtained with the `unicodedata.category(`*`c`*`)` function. For example, `unicodedata.category(u"A")` returns `'Lu'`, signifying that the character is an uppercase letter.

Another tricky problem with Unicode strings is that there might be multiple representations of the same Unicode string. For example, the character `U+00F1` (ñ), might be fully composed as a single character `U+00F1` or decomposed into a multicharacter sequence `U+006e U+0303` (n, ˜). If consistent processing of Unicode strings is an issue, use the `unicodedata.normalize()` function to ensure a consistent character representation. For example, `unicodedata.normalize('NFC',` *`s`*`)` will make sure that all characters in *`s`* are fully composed and not represented as a sequence of combining characters.

Further details about the Unicode character database and the `unicodedata` module can be found in Chapter 16, "Strings and Text Handling."

Object Persistence and the `pickle` Module

Finally, it's often necessary to save and restore the contents of an object to a file. One approach to this problem is to write a pair of functions that simply read and write data from a file in a special format. An alternative approach is to use the `pickle` and `shelve` modules.

The `pickle` module serializes an object into a stream of bytes that can be written to a file and later restored. The interface to `pickle` is simple, consisting of a `dump()` and `load()` operation. For example, the following code writes an object to a file:

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj, f)      # Save object on f
f.close()
```

To restore the object, you can use the following code:

```
import pickle
f = open(filename,'rb')
obj = pickle.load(f)   # Restore the object
f.close()
```

A sequence of objects can be saved by issuing a series of `dump()` operations one after the other. To restore these objects, simply use a similar sequence of `load()` operations.

The `shelve` module is similar to `pickle` but saves objects in a dictionary-like database:

```
import shelve
obj = SomeObject()
db = shelve.open("filename")     # Open a shelve
db['key'] = obj                  # Save object in the shelve
...
obj = db['key']                   # Retrieve it
db.close()                        # Close the shelve
```

Although the object created by `shelve` looks like a dictionary, it also has restrictions. First, the keys must be strings. Second, the values stored in a shelf must be compatible with `pickle`. Most Python objects will work, but special-purpose objects such as files and network connections maintain an internal state that cannot be saved and restored in this manner.

The data format used by `pickle` is specific to Python. However, the format has evolved several times over Python versions. The choice of protocol can be selected using an optional protocol parameter to the pickle `dump(`*`obj, file, protocol`*`)` operation. By default, protocol 0 is used. This is the oldest pickle data format that stores objects in a format understood by virtually all Python versions. However, this format is also incompatible with many of Python's more modern features of user-defined classes such as slots. Protocol 1 and 2 use a more efficient binary data representation. To use these alternative protocols, you would perform operations such as the following:

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj,f,2)                          # Save using protocol 2
pickle.dump(obj,f,pickle.HIGHEST_PROTOCOL)    # Use the most modern protocol
f.close()
```

It is not necessary to specify the protocol when restoring an object using `load()`. The underlying protocol is encoded into the file itself.

Similarly, a `shelve` can be opened to save Python objects using an alternative pickle protocol like this:

```
import shelve
db = shelve.open(filename,protocol=2)
...
```

It is not normally necessary for user-defined objects to do anything extra to work with `pickle` or `shelve`. However, the special methods `__getstate__()` and `__setstate__()` can be used to assist the pickling process. The `__getstate__()` method, if defined, will be called to create a value representing the state of an object. The value returned by `__getstate__()` should typically be a string, tuple, list, or dictionary. The `__setstate__()` method receives this value during unpickling and should restore the state of an object from it. Here is an example that shows how these methods could be used with an object involving an underlying network connection. Although the actual connection can't be pickled, the object saves enough information to reestablish it when it's unpickled later:

```
import socket
class Client(object):
    def __init__(self,addr):
        self.server_addr = addr
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(addr)
    def __getstate__(self):
        return self.server_addr
    def __setstate__(self,value):
        self.server_addr = value
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(self.server_addr)
```

Because the data format used by pickle is Python-specific, you would not use this feature as a means for exchanging data between applications written in different programming languages. Moreover, due to security concerns, programs should not process pickled data from untrusted sources (a knowledgeable attacker can manipulate the pickle data format to execute arbitrary system commands during unpickling).

The `pickle` and `shelve` modules have many more customization features and advanced usage options. For more details, consult Chapter 13, "Python Runtime Services."

10

Execution Environment

This chapter describes the environment in which Python programs are executed. The goal is to describe the runtime behavior of the interpreter, including program startup, configuration, and program termination.

Interpreter Options and Environment

The interpreter has a number of options that control its runtime behavior and environment. Options are given to the interpreter on the command line as follows:

```
python [options] [-c cmd | filename | - ] [args]
```

Here's a list of the most common command-line options:

Table 10.1 **Interpreter Command-Line Arguments**

Option	Description
`-3`	Enables warnings about features that are being removed or changed in Python 3.
`-B`	Prevents the creation of `.pyc` or `.pyo` files on import.
`-E`	Ignores environment variables.
`-h`	Prints a list of all available command-line options.
`-i`	Enters interactive mode after program execution.
`-m` *`module`*	Runs library module *`module`* as a script.
`-O`	Optimized mode.
`-OO`	Optimized mode plus removal of documentation strings when creating `.pyo` files.
-Q arg	Specifies the behavior of the division operator in Python 2. One of `-Qold` (the default), `-Qnew`, `-Qwarn`, or `-Qwarnall`.
`-s`	Prevents the addition of the user site directory to `sys.path`.
`-S`	Prevents inclusion of the `site` initialization module.
`-t`	Reports warnings about inconsistent tab usage.
`-tt`	Inconsistent tab usage results in a `TabError` exception.
`-u`	Unbuffered binary `stdout` and `stdin`.
`-U`	Unicode literals. All string literals are handled as Unicode (Python 2 only).
`-v`	Verbose mode. Traces import statements.
`-V`	Prints the version number and exits.
`-x`	Skips the first line of the source program.
`-c` *`cmd`*	Executes *`cmd`* as a string.

The -i option starts an interactive session immediately after a program has finished execution and is useful for debugging. The -m option runs a library module as a script which executes inside the __main__ module prior to the execution of the main script. The -O and -OO options apply some optimization to byte-compiled files and are described in Chapter 8, "Modules, Packages, and Distribution." The -S option omits the site initialization module described in the later section "Site Configuration Files." The -t, -tt, and -v options report additional warnings and debugging information. -x ignores the first line of a program in the event that it's not a valid Python statement (for example, when the first line starts the Python interpreter in a script).

The program name appears after all the interpreter options. If no name is given, or the hyphen (-) character is used as a filename, the interpreter reads the program from standard input. If standard input is an interactive terminal, a banner and prompt are presented. Otherwise, the interpreter opens the specified file and executes its statements until an end-of-file marker is reached. The -c *cmd* option can be used to execute short programs in the form of a command-line option—for example, `python -c "print('hello world')"`.

Command-line options appearing after the program name or hyphen (-) are passed to the program in sys.argv, as described in the section "Reading Options and Environment Variables" in Chapter 9, "Input and Output."

Additionally, the interpreter reads the following environment variables:

Table 10.2 **Interpreter Environment Variables**

Environment Variable	Description
PYTHONPATH	Colon-separated module search path.
PYTHONSTARTUP	File executed on interactive startup.
PYTHONHOME	Location of the Python installation.
PYTHONINSPECT	Implies the -i option.
PYTHONUNBUFFERED	Implies the -u option.
PYTHONIOENCODING	Encoding and error handling for stdin, stdout, and stderr. This is a string of the form "*encoding*[:*errors*]" such as "utf-8" or "utf-8:ignore".
PYTHONDONTWRITEBYTECODE	Implies the -B option
PYTHONOPTIMIZE	Implies the -O option.
PYTHONNOUSERSITE	Implies the -s option.
PYTHONVERBOSE	Implies the -v option.
PYTHONUSERBASE	Root directory for per-user site packages.
PYTHONCASEOK	Indicates to use case-insensitive matching for module names used by import.

PYTHONPATH specifies a module search path that is inserted into the beginning of sys.path, which is described in Chapter 9. PYTHONSTARTUP specifies a file to execute when the interpreter runs in interactive mode. The PYTHONHOME variable is used to set the location of the Python installation but is rarely needed because Python knows how to find its own libraries and the site-packages directory where extensions are normally installed. If a single directory such as /usr/local is given, the interpreter expects to find all files in that location. If two directories are given, such as /usr/local:/usr/local/sparc-solaris-2.6, the interpreter searches for platform-independent files in the first directory and platform-dependent files in the second. PYTHONHOME has no effect if no valid Python installation exists at the specified location.

The PYTHONIOENCODING environment setting might be of interest to users of Python 3 because it sets both the encoding and error handling of the standard I/O streams. This might be important because Python 3 directly outputs Unicode while running the interactive interpreter prompt. This, in turn, can cause unexpected exceptions merely while inspecting data. For example:

```
>>> a = 'Jalape\xf1o'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/lib/python3.0/io.py", line 1486, in write
    b = encoder.encode(s)
  File "/tmp/lib/python3.0/encodings/ascii.py", line 22, in encode
    return codecs.ascii_encode(input, self.errors)[0]
UnicodeEncodeError: 'ascii' codec can't encode character '\xf1' in position 7:
ordinal not in range(128)
>>>
```

To fix this, you can set the environment variable PYTHONIOENCODING to something such as 'ascii:backslashreplace' or 'utf-8'. Now, you will get this:

```
>>> a = 'Jalape\xf1o'
>>> a
'Jalape\xf1o'
>>>
```

On Windows, some of the environment variables such as PYTHONPATH are additionally read from registry entries found in HKEY_LOCAL_MACHINE/Software/Python.

Interactive Sessions

If no program name is given and the standard input to the interpreter is an interactive terminal, Python starts in interactive mode. In this mode, a banner message is printed and the user is presented with a prompt. In addition, the interpreter evaluates the script contained in the PYTHONSTARTUP environment variable (if set). This script is evaluated as if it's part of the input program (that is, it isn't loaded using an import statement). One application of this script might be to read a user configuration file such as .pythonrc.

When interactive input is being accepted, two user prompts appear. The >>> prompt appears at the beginning of a new statement; the ... prompt indicates a statement continuation. Here's an example:

```
>>> for i in range(0,4):
...     print i,
...
0 1 2 3
>>>
```

In customized applications, you can change the prompts by modifying the values of sys.ps1 and sys.ps2.

On some systems, Python may be compiled to use the GNU readline library. If enabled, this library provides command histories, completion, and other additions to Python's interactive mode.

By default, the output of commands issued in interactive mode is generated by printing the output of the built-in repr() function on the result. This can be changed by setting the variable sys.displayhook to a function responsible for displaying results. Here's an example that truncates long results:

```
>>> def my_display(x):
...     r = repr(x)
...     if len(r) > 40: print(r[:40]+"..."+r[-1])
...     else: print(r)
>>> sys.displayhook = my_display
>>> 3+4
7
>>> range(100000)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1...]
>>>
```

Finally, in interactive mode, it is useful to know that the result of the last operation is stored in a special variable (_). This variable can be used to retrieve the result should you need to use it in subsequent operations. Here's an example:

```
>>> 7 + 3
10
>>> _ + 2
12
>>>
```

The setting of the _ variable occurs in the displayhook() function shown previously. If you redefine displayhook(), your replacement function should also set _ if you want to retain that functionality.

Launching Python Applications

In most cases, you'll want programs to start the interpreter automatically, rather than first having to start the interpreter manually. On UNIX, this is done by giving the program execute permission and setting the first line of a program to something like this:

```
#!/usr/bin/env python
# Python code from this point on...
print "Hello world"
...
```

On Windows, double-clicking a .py, .pyw, .wpy, .pyc, or .pyo file automatically launches the interpreter. Normally, programs run in a console window unless they're renamed with a .pyw suffix (in which case the program runs silently). If it's necessary to supply options to the interpreter, Python can also be started from a .bat file. For example, this .bat file simply runs Python on a script and passes any options supplied on the command prompt along to the interpreter:

```
:: foo.bat
:: Runs foo.py script and passes supplied command line options along (if any)
c:\python26\python.exe c:\pythonscripts\foo.py %*
```

Site Configuration Files

A typical Python installation may include a number of third-party modules and packages. To configure these packages, the interpreter first imports the module site. The role of site is to search for package files and to add additional directories to the module search path sys.path. In addition, the site module sets the default encoding for Unicode string conversions.

The site module works by first creating a list of directory names constructed from the values of sys.prefix and sys.exec_prefix as follows:

```
[ sys.prefix,              # Windows only
  sys.exec_prefix,         # Windows only
  sys.prefix + 'lib/pythonvers/site-packages',
  sys.prefix + 'lib/site-python',
  sys.exec_prefix + 'lib/pythonvers/site-packages',
  sys.exec_prefix + 'lib/site-python' ]
```

In addition, if enabled, a user-specific site packages directory may be added to this list (described in the next section).

For each directory in the list, a check is made to see whether the directory exists. If so, it's added to the sys.path variable. Next, a check is made to see whether it contains any path configuration files (files with a .pth suffix). A path configuration file contains a list of directories, zip files, or .egg files relative to the location of the path file that should be added to sys.path. For example:

```
# foo package configuration file 'foo.pth'
foo
bar
```

Each directory in the path configuration file must be listed on a separate line. Comments and blank lines are ignored. When the site module loads the file, it checks to see whether each directory exists. If so, the directory is added to sys.path. Duplicated items are added to the path only once.

After all paths have been added to sys.path, an attempt is made to import a module named sitecustomize. The purpose of this module is to perform any additional (and arbitrary) site customization. If the import of sitecustomize fails with an ImportError, the error is silently ignored. The import of sitecustomize occurs prior to adding any user directories to sys.path. Thus, placing this file in your own directory has no effect.

The site module is also responsible for setting the default Unicode encoding. By default, the encoding is set to 'ascii'. However, the encoding can be changed by placing code in sitecustomize.py that calls sys.setdefaultencoding() with a new encoding such as 'utf-8'. If you're willing to experiment, the source code of site can also be modified to automatically set the encoding based on the machine's locale settings.

Per-user Site Packages

Normally, third-party modules are installed in a way that makes them accessible to all users. However, individual users can install modules and packages in a per-user site directory. On UNIX and Macintosh systems, this directory is found under ~/.local and is named something such as ~/.local/lib/python2.6/site-packages. On Windows systems, this directory is determined by the %APPDATA% environment variable,

which is usually something similar to `C:\Documents and Settings\David Beazley\Application Data`. Within that folder, you will find a `"Python\Python26\site-packages"` directory.

If you are writing your own Python modules and packages that you want to use in a library, they can be placed in the per-user site directory. If you are installing third-party modules, you can manually install them in this directory by supplying the `--user` option to `setup.py`. For example: `python setup.py install --user`.

Enabling Future Features

New language features that affect compatibility with older versions of Python are often disabled when they first appear in a release. To enable these features, the statement `from __future__ import` *feature* can be used. Here's an example:

```
# Enable new division semantics
from __future__ import division
```

When used, this statement should appear as the first statement of a module or program. Moreover, the scope of a `__future__` import is restricted only to the module in which it is used. Thus, importing a future feature does not affect the behavior of Python's library modules or older code that requires the previous behavior of the interpreter to operate correctly.

Currently, the following features have been defined:

Table 10.3 **Feature Names in the `__future__` Module**

Feature Name	Description
`nested_scopes`	Support for nested scopes in functions. First introduced in Python 2.1 and made the default behavior in Python 2.2.
`generators`	Support for generators. First introduced in Python 2.2 and made the default behavior in Python 2.3.
`division`	Modified division semantics where integer division returns a fractional result. For example, 1/4 yields 0.25 instead of 0. First introduced in Python 2.2 and is still an optional feature as of Python 2.6. This is the default behavior in Python 3.0.
`absolute_import`	Modified behavior of package-relative imports. Currently, when a submodule of a package makes an import statement such as import string, it first looks in the current directory of the package and then directories in `sys.path`. However, this makes it impossible to load modules in the standard library if a package happens to use conflicting names. When this feature is enabled, the statement import module is an absolute import. Thus, a statement such as `import string` will always load the string module from the standard library. First introduced in Python 2.5 and still disabled in Python 2.6. It is enabled in Python 3.0.
`with_statement`	Support for context managers and the `with` statement. First introduced in Python 2.5 and enabled by default in Python 2.6.
`print_function`	Use Python 3.0 `print()` function instead of the `print` statement. First introduced in Python 2.6 and enabled by default in Python 3.0.

It should be noted that no feature name is ever deleted from `__future__`. Thus, even if a feature is turned on by default in a later Python version, no existing code that uses that feature name will break.

Program Termination

A program terminates when no more statements exist to execute in the input program, when an uncaught `SystemExit` exception is raised (as generated by `sys.exit()`), or when the interpreter receives a `SIGTERM` or `SIGHUP` signal (on UNIX). On exit, the interpreter decrements the reference count of all objects in all the currently known namespaces (and destroys each namespace as well). If the reference count of an object reaches zero, the object is destroyed and its `__del__()` method is invoked.

It's important to note that in some cases the `__del__()` method might not be invoked at program termination. This can occur if circular references exist between objects (in which case objects may be allocated but accessible from no known namespace). Although Python's garbage collector can reclaim unused circular references during execution, it isn't normally invoked on program termination.

Because there's no guarantee that `__del__()` will be invoked at termination, it may be a good idea to explicitly clean up certain objects, such as open files and network connections. To accomplish this, add specialized cleanup methods (for example, `close()`) to user-defined objects. Another possibility is to write a termination function and register it with the `atexit` module, as follows:

```
import atexit
connection = open_connection("deaddot.com")

def cleanup():
    print "Going away..."
    close_connection(connection)

atexit.register(cleanup)
```

The garbage collector can also be invoked in this manner:

```
import atexit, gc
atexit.register(gc.collect)
```

One final peculiarity about program termination is that the `__del__` method for some objects may try to access global data or methods defined in other modules. Because these objects may already have been destroyed, a `NameError` exception occurs in `__del__`, and you may get an error such as the following:

```
Exception exceptions.NameError: 'c' in <method Bar.__del__
of Bar instance at c0310> ignored
```

If this occurs, it means that `__del__` has aborted prematurely. It also implies that it may have failed in an attempt to perform an important operation (such as cleanly shutting down a server connection). If this is a concern, it's probably a good idea to perform an explicit shutdown step in your code, rather than rely on the interpreter to destroy objects cleanly at program termination. The peculiar `NameError` exception can also be eliminated by declaring default arguments in the declaration of the `__del__()` method:

```
import foo
class Bar(object):
   def __del__(self, foo=foo):
      foo.bar()        # Use something in module foo
```

In some cases, it may be useful to terminate program execution without performing any cleanup actions. This can be accomplished by calling `os._exit(`*status*`)`. This function provides an interface to the low-level `exit()` system call responsible for killing the Python interpreter process. When it's invoked, the program immediately terminates without any further processing or cleanup.

11

Testing, Debugging, Profiling, and Tuning

Unlike programs in languages such as C or Java, Python programs are not processed by a compiler that produces an executable program. In those languages, the compiler is the first line of defense against programming errors—catching mistakes such as calling functions with the wrong number of arguments or assigning improper values to variables (that is, type checking). In Python, however, these kinds of checks do not occur until a program runs. Because of this, you will never really know if your program is correct until you run and test it. Not only that, unless you are able to run your program in a way that executes every possible branch of its internal control-flow, there is always some chance of a hidden error just waiting to strike (fortunately, this usually only happens a few days *after* shipping, however).

To address these kinds of problems, this chapter covers techniques and library modules used to test, debug, and profile Python code. At the end, some strategies for optimizing Python code are discussed.

Documentation Strings and the `doctest` Module

If the first line of a function, class, or module is a string, that string is known as a *documentation string*. The inclusion of documentation strings is considered good style because these strings are used to supply information to Python software development tools. For example, the `help()` command inspects documentation strings, and Python IDEs look at the strings as well. Because programmers tend to view documentation strings while experimenting in the interactive shell, it is common for the strings to include short interactive examples. For example:

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Splits a line of text and optionally performs type conversion.
       For example:

       >>> split('GOOG 100 490.50')
       ['GOOG', '100', '490.50']
       >>> split('GOOG 100 490.50',[str, int, float])
       ['GOOG', 100, 490.5]
       >>>
```

```
    By default, splitting is performed on whitespace, but a different
    delimiter can be selected with the delimiter keyword argument:

    >>> split('GOOG,100,490.50',delimiter=',')
    ['GOOG', '100', '490.50']
    >>>
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

A common problem with writing documentation is keeping the documentation synchronized with the actual implementation of a function. For example, a programmer might modify a function but forget to update the documentation.

To address this problem, use the `doctest` module. `doctest` collects documentation strings, scans them for interactive sessions, and executes them as a series of tests. To use `doctest`, you typically create a separate module for testing. For example, if the previous function is in a file `splitter.py`, you would create a file `testsplitter.py` for testing, as follows:

```
# testsplitter.py
import splitter
import doctest

nfail, ntests = doctest.testmod(splitter)
```

In this code, the call to `doctest.testmod(`*`module`*`)` runs tests on the specified module and returns the number of failures and total number of tests executed. No output is produced if all of the tests pass. Otherwise, you will get a failure report that shows the difference between the expected and received output. If you want to see verbose output of the tests, you can use `testmod(`*`module`*`, verbose=True)`.

As an alternative to creating a separate testing file, library modules can test themselves by including code such as this at the end of the file:

```
...
if __name__ == '__main__':
    # test myself
    import doctest
    doctest.testmod()
```

With this code, documentation tests will run if the file is run as the main program to the interpreter. Otherwise, the tests are ignored if the file is loaded with `import`.

`doctest` expects the output of functions to literally match the exact output you get in the interactive interpreter. As a result, it is quite sensitive to issues of white space and numerical precision. For example, consider this function:

```
def half(x):
    """Halves x.  For example:

    >>> half(6.8)
    3.4
    >>>
    """
    return x/2
```

If you run `doctest` on this function, you will get a failure report such as this:

```
**********************************************************************
File "half.py", line 4, in __main__.half
Failed example:
    half(6.8)
Expected:
    3.4
Got:
    3.3999999999999999
**********************************************************************
```

To fix this, you either need to make the documentation exactly match the output or need to pick a better example in the documentation.

Because using `doctest` is almost trivial, there is almost no excuse for not using it with your own programs. However, keep in mind that `doctest` is not a module you would typically use for exhaustive program testing. Doing so tends to result in excessively long and complicated documentation strings—which defeats the point of producing useful documentation (e.g., a user will probably be annoyed if he asks for help and the documentation lists 50 examples covering all sorts of tricky corner cases). For this kind of testing, you want to use the `unittest` module.

Last, the `doctest` module has a large number of configuration options that concerns various aspects of how testing is performed and how results are reported. Because these options are not required for the most common use of the module, they are not covered here. Consult http://docs.python.org/library/doctest.html for more details.

Unit Testing and the `unittest` Module

For more exhaustive program testing, use the `unittest` module. With unit testing, a developer writes a collection of isolated test cases for each element that makes up a program (for example, individual functions, methods, classes, and modules). These tests are then run to verify correct behavior of the basic building blocks that make up larger programs. As programs grow in size, unit tests for various components can be combined to create large testing frameworks and testing tools. This can greatly simplify the task of verifying correct behavior as well as isolating and fixing problems when they do occur. Use of this module can be illustrated by the code listing in the previous section:

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Splits a line of text and optionally performs type conversion.
       ...
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

If you wanted to write unit tests for testing various aspects of the `split()` function, you would create a separate module `testsplitter.py`, like this:

```
# testsplitter.py
import splitter
import unittest

# Unit tests
class TestSplitFunction(unittest.TestCase):
      def setUp(self):
          # Perform set up actions (if any)
          pass
      def tearDown(self):
          # Perform clean-up actions (if any)
          pass
      def testsimplestring(self):
          r = splitter.split('GOOG 100 490.50')
          self.assertEqual(r,['GOOG','100','490.50'])
      def testtypeconvert(self):
          r = splitter.split('GOOG 100 490.50',[str, int, float])
          self.assertEqual(r,['GOOG', 100, 490.5])
      def testdelimiter(self):
          r = splitter.split('GOOG,100,490.50',delimiter=',')
          self.assertEqual(r,['GOOG','100','490.50'])

# Run the unittests
if __name__ == '__main__':
   unittest.main()
```

To run tests, simply run Python on the file `testsplitter.py`. Here's an example:

```
% python testsplitter.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.014s

OK
```

Basic use of `unittest` involves defining a class that inherits from `unittest.TestCase`. Within this class, individual tests are defined by methods starting with the name `'test'`—for example, `'testsimplestring'`, `'testtypeconvert'`, and so on. (It is important to emphasize that the names are entirely up to you as long as they start with `'test'`.) Within each test, various assertions are used to check for different conditions.

An instance, *t*, of `unittest.TestCase` has the following methods that are used when writing tests and for controlling the testing process:

***t*.setUp()**

Called to perform set-up steps prior to running any of the testing methods.

***t*.tearDown()**

Called to perform clean-up actions after running the tests.

***t*.assert_(*expr* [, *msg*])**
***t*.failUnless(*expr* [, *msg*])**

Signals a test failure if *expr* evaluates as `False`. *msg* is a message string giving an explanation for the failure (if any).

***t*.assertEqual(*x*, *y* [,*msg*])**
***t*.failUnlessEqual(*x*, *y* [, *msg*])**

Signals a test failure if *x* and *y* are not equal to each other. *msg* is a message explaining the failure (if any).

***t*.assertNotEqual(*x*, *y* [, *msg*])**
***t*.failIfEqual(*x*, *y*, [, *msg*])**

Signals a test failure if *x* and *y* are equal to each other. *msg* is a message explaining the failure (if any).

***t*.assertAlmostEqual(*x*, *y* [, *places* [, *msg*]])**
***t*.failUnlessAlmostEqual(*x*, *y*, [, *places* [, *msg*]])**

Signals a test failure if numbers *x* and *y* are not within *places* decimal places of each other. This is checked by computing the difference of *x* and *y* and rounding the result to the given number of places. If the result is zero, *x* and *y* are almost equal. *msg* is a message explaining the failure (if any).

***t*.assertNotAlmostEqual(*x*, *y*, [, *places* [, *msg*]])**
***t*.failIfAlmostEqual(*x*, *y* [, *places* [, *msg*]])**

Signals a test failure if *x* and *y* are not at least *places* decimal places apart. *msg* is a message explaining the failure (if any).

***t*.assertRaises(*exc*, *callable*, ...)**
***t*.failUnlessRaises(*exc*, *callable*, ...)**

Signals a test failure if the callable object *callable* does not raise the exception *exc*. Remaining arguments are passed as arguments to *callable*. Multiple exceptions can be checked by using a tuple of exceptions as *exc*.

***t*.failIf(*expr* [, *msg*])**

Signals a test failure if *expr* evaluates as `True`. *msg* is a message explaining the failure (if any).

***t*.fail([*msg*])**

Signals a test failure. *msg* is a message explaining the failure (if any).

***t*.failureException**

This attribute is set to the last exception value caught in a test. This may be useful if you not only want to check that an exception was raised, but that the exception raises an appropriate value—for example, if you wanted to check the error message generated as part of raising an exception.

It should be noted that the `unittest` module contains a large number of advanced customization options for grouping tests, creating test suites, and controlling the environment in which tests run. These features are not directly related to the process of writing tests for your code (you tend to write testing classes as shown independently of how tests actually get executed). Consult the documentation at http://docs.python.org/library/unittest.html for more information on how to organize tests for larger programs.

The Python Debugger and the `pdb` Module

Python includes a simple command-based debugger which is found in the `pdb` module. The `pdb` module supports post-mortem debugging, inspection of stack frames, breakpoints, single-stepping of source lines, and code evaluation.

There are several functions for invoking the debugger from a program or from the interactive Python shell.

`run(`*`statement`* **`[,` *`globals`* **`[,` *`locals`*`]])`

Executes the string *`statement`* under debugger control. The debugger prompt will appear immediately before any code executes. Typing `'continue'` will force it to run. *`globals`* and *`locals`* define the global and local namespaces, respectively, in which the code runs.

`runeval(`*`expression`* **`[,` *`globals`* **`[,` *`locals`*`]])`

Evaluates the *`expression`* string under debugger control. The debugger prompt will appear before any code executes, so you will need to type `'continue'` to force it to execute as with `run()`. On success, the value of the expression is returned.

`runcall(`*`function`* **`[,` *`argument`*`, ...])`

Calls a function within the debugger. *`function`* is a callable object. Additional arguments are supplied as the arguments to *`function`*. The debugger prompt will appear before any code executes. The return value of the function is returned upon completion.

`set_trace()`

Starts the debugger at the point at which this function is called. This can be used to hard-code a debugger breakpoint into a specific code location.

`post_mortem(`*`traceback`*`)`

Starts post-mortem debugging of a traceback object. *`traceback`* is typically obtained using a function such as `sys.exc_info()`.

`pm()`

Enters post-mortem debugging using the traceback of the last exception.

Of all of the functions for launching the debugger, the `set_trace()` function may be the easiest to use in practice. If you are working on a complicated application but you have detected a problem in one part of it, you can insert a `set_trace()` call into the code and simply run the application. When encountered, this will suspend the program and go directly to the debugger where you can inspect the execution environment. Execution resumes after you leave the debugger.

Debugger Commands

When the debugger starts, it presents a `(Pdb)` prompt such as the following:

```
>>> import pdb
>>> import buggymodule
>>> pdb.run('buggymodule.start()')
> <string>(0)?()
(Pdb)
```

`(Pdb)` is the debugger prompt at which the following commands are recognized. Note that some commands have a short and a long form. In this case, parentheses are used to indicate both forms. For example, `h(elp)` means that either `h` or `help` is acceptable.

`[!]`*`statement`

Executes the (one-line) *`statement`* in the context of the current stack frame. The exclamation point may be omitted, but it must be used to avoid ambiguity if the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a "`global`" command on the same line:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

`a(rgs)`

Prints the argument list of the current function.

`alias [`*`name`* **`[`*`command`*`]]`

Creates an alias called *`name`* that executes *`command`*. Within the *`command`* string, the substrings `'%1'`,`'%2'`, and so forth are replaced by parameters when the alias is typed. `'%*'` is replaced by all parameters. If no command is given, the current alias list is shown. Aliases can be nested and can contain anything that can be legally typed at the `Pdb` prompt. Here's an example:

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
# Print instance variables in self
alias ps pi self
```

`b(reak) [`*`loc`* **`[,` *`condition`*`]]`

Sets a breakpoint at location *`loc`*. *`loc`* either specifies a specific filename and line number or is the name of a function within a module. The following syntax is used:

Setting	Description
`n`	A line number in the current file
`filename``:`*`n`*	A line number in another file
`function`	A function name in the current module
`module.function`	A function name in a module

If *`loc`* is omitted, all the current breakpoints are printed. *`condition`* is an expression that must evaluate to true before the breakpoint is honored. All breakpoints are assigned numbers that are printed as output upon the completion of this command. These numbers are used in several other debugger commands that follow.

`cl(ear) [`*`bpnumber`* **`[`*`bpnumber`* **`...]]`

Clears a list of breakpoint numbers. If breakpoints are specified, all breaks are cleared.

`commands [`*`bpnumber`*`]`

Sets a series of debugger commands to execute automatically when the breakpoint *`bpnumber`* is encountered. When listing the commands to execute, simply type them on the subsequent lines and use `end` to mark the end of the command sequence. If you include the `continue` command, the execution of the program will resume automatically when the breakpoint is encountered. If *`bpnumber`* is omitted, the last breakpoint set is used.

`condition `*`bpnumber`* **`[`*`condition`*`]`

Places a condition on a breakpoint. *`condition`* is an expression that must evaluate to true before the breakpoint is recognized. Omitting the condition clears any previous condition.

`c(ont(inue))`

Continues execution until the next breakpoint is encountered.

`disable [`*`bpnumber`* **`[`*`bpnumber`* **`...]]`

Disables the set of specified breakpoints. Unlike with `clear`, they can be reenabled later.

`d(own)`

Moves the current frame one level down in the stack trace.

`enable [`*`bpnumber`* **`[`*`bpnumber`* **`...]]`

Enables a specified set of breakpoints.

`h(elp) [`*`command`*`]`

Shows the list of available commands. Specifying a command returns help for that command.

`ignore `*`bpnumber`* **`[`*`count`*`]`

Ignores a breakpoint for *`count`* executions.

`j(ump) `*`lineno`

Sets the next line to execute. This can only be used to move between statements in the same execution frame. Moreover, you can't jump into certain statements, such as statements in the middle of a loop.

`l(ist) [`*`first`* **`[,` *`last`*`]]`

Lists source code. Without arguments, this command lists 11 lines around the current line (5 lines before and 5 lines after). With one argument, it lists 11 lines around that line. With two arguments, it lists lines in a given range. If *`last`* is less than *`first`*, it's interpreted as a count.

`n(ext)`

Executes until the next line of the current function. Skips the code contained in function calls.

`p `*`expression`

Evaluates the expression in the current context and prints its value.

`pp `*`expression`

The same as the `p` command, but the result is formatted using the pretty-printing module (`pprint`).

`q(uit)`

Quits from the debugger.

`r(eturn)`

Runs until the current function returns.

`run [`*`args`*`]`

Restarts the program and uses the command-line arguments in *`args`* as the new setting of `sys.argv`. All breakpoints and other debugger settings are preserved.

`s(tep)`

Executes a single source line and stops inside called functions.

`tbreak [`*`loc`* **`[,` *`condition`*`]]`

Sets a temporary breakpoint that's removed after its first hit.

`u(p)`

Moves the current frame one level up in the stack trace.

`unalias `*`name`

Deletes the specified alias.

`until`

Resumes execution until control leaves the current execution frame or until a line number greater than the current line number is reached. For example, if the debugger was stopped at the last line in a loop body, typing **`until`** will execute all of the statements in the loop until the loop is finished.

`w(here)`

Prints a stack trace.

Debugging from the Command Line

An alternative method for running the debugger is to invoke it on the command line. Here's an example:

```
% python -m pdb someprogram.py
```

In this case, the debugger is launched automatically at the beginning of program startup where you are free to set breakpoints and make other configuration changes. To make the program run, simply use the `continue` command. For example, if you wanted to debug the `split()` function from within a program that used it, you might do this:

```
% python -m pdb someprogram.py
> /Users/beazley/Code/someprogram.py(1)<module>()
-> import splitter
(Pdb) b splitter.split
Breakpoint 1 at /Users/beazley/Code/splitter.py:1
(Pdb) c
> /Users/beazley/Code/splitter.py(18)split()
-> fields = line.split(delimiter)
(Pdb)
```

Configuring the Debugger

If a `.pdbrc` file exists in the user's home directory or in the current directory, it's read in and executed as if it had been typed at the debugger prompt. This can be useful for specifying debugging commands that you want to execute each time the debugger is started (as opposed to having to interactively type the commands each time).

Program Profiling

The `profile` and `cProfile` modules are used to collect profiling information. Both modules work in the same way, but `cProfile` is implemented as a C extension, is significantly faster, and is more modern. Either module is used to collect both coverage information (that is, what functions get executed) as well as performance statistics. The easiest way to profile a program is to execute it from the command line as follows:

```
% python -m cProfile someprogram.py
```

Alternatively, the following function in the `profile` module can be used:

run(*command* [, *filename*])

Executes the contents of *command* using the `exec` statement under the profiler. *filename* is the name of a file in which raw profiling data is saved. If it's omitted, a report is printed to standard output.

The result of running the profiler is a report such as the following:

```
126 function calls (6 primitive calls) in 5.130 CPU seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.030    0.030    5.070    5.070 <string>:1(?)
 121/1    5.020    0.041    5.020    5.020 book.py:11(process)
     1    0.020    0.020    5.040    5.040 book.py:5(?)
     2    0.000    0.000    0.000    0.000 exceptions.py:101(__init__)
     1    0.060    0.060    5.130    5.130 profile:0(execfile('book.py'))
     0    0.000             0.000          profile:0(profiler)
```

Different parts of the report generated by `run()` are interpreted as follows:

Section	Description
`primitive calls`	Number of nonrecursive function calls
`ncalls`	Total number of calls (including self-recursion)
`tottime`	Time spent in this function (not counting subfunctions)
`percall`	`tottime/ncalls`
`cumtime`	Total time spent in the function
`percall`	`cumtime/(primitive calls)`
`filename:lineno(function)`	Location and name of each function

When there are two numbers in the first column (for example, `"121/1"`), the latter is the number of primitive calls and the former is the actual number of calls.

Simply inspecting the generated report of the profiler is often enough for most applications of this module—for example, if you simply want to see how your program is spending its time. However, if you want to save the data and analyze it further, the `pstats` module can be used. Consult http://docs.python.org/library/profile.html for more details about saving and analyzing the profile data.

Tuning and Optimization

This section covers some general rules of thumb that can be used to make Python programs run faster and use less memory. The techniques described here are by no means exhaustive but should give programmers some ideas when looking at their own code.

Making Timing Measurements

If you simply want to time a long-running Python program, the easiest way to do it is often just to run it until the control of something like the UNIX `time` command. Alternatively, if you have a block of long-running statements you want to time, you can insert calls to `time.clock()` to get a current reading of the elapsed CPU time or calls to `time.time()` to read the current wall-clock time. For example:

```
start_cpu = time.clock()
start_real= time.time()
statements
statements
end_cpu = time.clock()
end_real = time.time()
print("%f Real Seconds" % (end_real - start_real))
print("%f CPU seconds" % (end_cpu - start_cpu))
```

Keep in the mind that this technique really works only if the code to be timed runs for a reasonable period of time. If you have a fine-grained statement you want to benchmark, you can use the `timeit(`*`code [, setup`*`])` function in the `timeit` module. For example:

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2.0)','import math')
0.20388007164001465
>>> timeit('sqrt(2.0)','from math import sqrt')
0.14494490623474121
```

In this example, the first argument to `timeit()` is the code you want to benchmark. The second argument is a statement that gets executed once in order to set up the execution environment. The `timeit()` function runs the supplied statement one million times and reports the execution time. The number of repetitions can be changed by supplying a `number=`*`count`* keyword argument to `timeit()`.

The `timeit` module also has a function `repeat()` that can be used to make measurements. This function works the same way as `timeit()` except that it repeats the timing measurement three times and returns a list of the results. For example:

```
>>> from timeit import repeat
>>> repeat('math.sqrt(2.0)','import math')
[0.20306601524353027, 0.19715800285339355, 0.20907392501831055]
>>>
```

When making performance measurement, it is common to refer to the associated *speedup*, which usually refers to the original execution time divided by the new execution time. For example, in the previous timing measurements, using `sqrt(2.0)` instead of `math.sqrt(2.0)` represents a speedup of 0.20388/0.14494 or about 1.41. Sometimes this gets reported as a percentage by saying the speedup is about 41 percent.

Making Memory Measurements

The `sys` module has a function `getsizeof()` that can be used to investigate the memory footprint (in bytes) of individual Python objects. For example:

```
>>> import sys
>>> sys.getsizeof(1)
14
>>> sys.getsizeof("Hello World")
52
>>> sys.getsizeof([1,2,3,4])
52
>>> sum(sys.getsizeof(x) for x in [1,2,3,4])
56
```

For containers such as lists, tuples, and dictionaries, the size that gets reported is just for the container object itself, not the cumulative size of all objects contained inside of it. For instance, in the previous example, the reported size of the list `[1,2,3,4]` is actually smaller than the space required for four integers (which are 14 bytes each). This is because the contents of the list are not included in the total. You can use `sum()` as shown here to calculate the total size of the list contents.

Be aware that the `getsizeof()` function is only going to give you a rough idea of overall memory use for various objects. Internally, the interpreter aggressively shares objects via reference counting so the actual memory consumed by an object might be far less than you first imagine. Also, given that C extensions to Python can allocate memory outside of the interpreter, it may be difficult to precisely get a measurement of overall memory use. Thus, a secondary technique for measuring the actual memory footprint is to inspect your running program from an operating system process viewer or task manager.

Frankly, a better way to get a handle on memory use may be to sit down and be analytical about it. If you know your program is going to allocate various kinds of data structures and you know what kinds of data will be stored in those structures (that is, ints, floats, strings, and so on), you can use the results of the `getsizeof()` function to obtain figures for calculating an upper bound on your program's memory footprint—or at the very least, you can get enough information to carry out a "back of the envelope" estimate.

Disassembly

The `dis` module can be used to disassemble Python functions, methods, and classes into low-level interpreter instructions. The module defines a function `dis()` that can be used like this:

```
>>> from dis import dis
>>> dis(split)
  2           0 LOAD_FAST                0 (line)
              3 LOAD_ATTR                0 (split)
              6 LOAD_FAST                1 (delimiter)
              9 CALL_FUNCTION            1
             12 STORE_FAST               2 (fields)

  3          15 LOAD_GLOBAL              1 (types)
             18 JUMP_IF_FALSE           58 (to 79)
             21 POP_TOP

  4          22 BUILD_LIST               0
             25 DUP_TOP
             26 STORE_FAST               3 (_[1])
             29 LOAD_GLOBAL              2 (zip)
             32 LOAD_GLOBAL              1 (types)
             35 LOAD_FAST                2 (fields)
             38 CALL_FUNCTION            2
             41 GET_ITER
        >>   42 FOR_ITER                25 (to 70)
             45 UNPACK_SEQUENCE          2
             48 STORE_FAST               4 (ty)
             51 STORE_FAST               5 (val)
             54 LOAD_FAST                3 (_[1])
             57 LOAD_FAST                4 (ty)
             60 LOAD_FAST                5 (val)
             63 CALL_FUNCTION            1
             66 LIST_APPEND
             67 JUMP_ABSOLUTE           42
        >>   70 DELETE_FAST              3 (_[1])
             73 STORE_FAST               2 (fields)
             76 JUMP_FORWARD             1 (to 80)
        >>   79 POP_TOP

  5     >>   80 LOAD_FAST                2 (fields)
             83 RETURN_VALUE
>>>
```

Expert programmers can use this information in two ways. First, a disassembly will show you exactly what operations are involved in executing a function. With careful study, you might spot opportunities for making speedups. Second, if you are programming with threads, each line printed in the disassembly represents a single interpreter operation—each of which has atomic execution. Thus, if you are trying to track down a tricky race condition, this information might be useful.

Tuning Strategies

The following sections outline a few optimization strategies that, in the opinion of the author, have proven to be useful with Python code.

Understand Your Program

Before you optimize anything, know that speedup obtained by optimizing part of a program is directly related to that part's total contribution to the execution time. For example, if you optimize a function by making it run 10 times as fast but that function only contributes to 10 percent of the program's total execution time, you're only going to get an overall speedup of about 9%–10%. Depending on the effort involved in making the optimization, this may or may not be worth it.

It is always a good idea to first use the profiling module on code you intend to optimize. You really only want to focus on functions and methods where your program spends most of its time, not obscure operations that are called only occasionally.

Understand Algorithms

A poorly implemented O(n log n) algorithm will outperform the most finely tuned $O(n^3)$ algorithm. Don't optimize inefficient algorithms—look for a better algorithm first.

Use the Built-In Types

Python's built-in tuple, list, set, and dictionary types are implemented entirely in C and are the most finely tuned data structures in the interpreter. You should actively use these types to store and manipulate data in your program and resist the urge to build your own custom data structures that mimic their functionality (that is, binary search trees, linked lists, and so on).

Having said that, you should still look more closely at types in the standard library. Some library modules provide new types that outperform the built-ins at certain tasks. For instance, the `collection.deque` type provides similar functionality to a list but has been highly optimized for the insertion of new items at both ends. A list, in contrast, is only efficient when appending items at the end. If you insert items at the front, all of the other elements need to be shifted in order to make room. The time required to do this grows as the list gets larger and larger. Just to give you an idea of the difference, here is a timing measurement of inserting one million items at the front of a list and a deque:

```
>>> from timeit import timeit
>>> timeit('s.appendleft(37)',
...        'import collections; s = collections.deque()',
...        number=1000000)
0.24434304237365723
>>> timeit('s.insert(0,37)', 's = []', number=1000000)
612.95199513435364
```

Don't Add Layers

Any time you add an extra layer of abstraction or convenience to an object or a function, you will slow down your program. However, there is also a trade-off between usability and performance. For instance, the whole point of adding an extra layer is often to simplify coding, which is also a good thing.

To illustrate with a simple example, consider a program that makes use of the `dict()` function to create dictionaries with string keys like this:

```
s = dict(name='GOOG',shares=100,price=490.10)
# s = {'name':'GOOG', 'shares':100, 'price':490.10 }
```

A programmer might create dictionaries in this way to save typing (you don't have to put quotes around the key names). However, this alternative way of creating a dictionary also runs much more slowly because it adds an extra function call.

```
>>> timeit("s = {'name':'GOOG','shares':100,'price':490.10}")
0.38917303085327148
>>> timeit("s = dict(name='GOOG',shares=100,price=490.10)")
0.94420003890991211
```

If your program creates millions of dictionaries as it runs, then you should know that the first approach is faster. With few exceptions, any feature that adds an enhancement or changes the way in which an existing Python object works will run more slowly.

Know How Classes and Instances Build Upon Dictionaries

User-defined classes and instances are built using dictionaries. Because of this, operations that look up, set, or delete instance data are almost always going to run more slowly than directly performing these operations on a dictionary. If all you are doing is building a simple data structure for storing data, a dictionary may be a more efficient choice than defining a class.

Just to illustrate the difference, here is a simple class that represents a holding of stock:

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

If you compare the performance of using this class against a dictionary, the results are interesting. First, let's compare the performance of simply creating instances:

```
>>> from timeit import timeit
>>> timeit("s = Stock('GOOG',100,490.10)","from stock import Stock")
1.3166780471801758
>>> timeit("s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.37812089920043945
>>>
```

Here, the speedup of creating new objects is about 3.5. Next, let's look at the performance of performing a simple calculation:

```
>>> timeit("s.shares*s.price",
...        "from stock import Stock; s = Stock('GOOG',100,490.10)")
0.29100513458251953
>>> timeit("s['shares']*s['price']",
...        "s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.23622798919677734
>>>
```

Here, the speedup is about 1.2. The lesson here is that just because you can define a new object using a `class`, it's not the only way to work with data. Tuples and dictionaries are often good enough. Using them will make your program run more quickly and use less memory.

Use `__slots__`

If your program creates a large number of instances of user-defined classes, you might consider using the `__slots__` attribute in a class definition. For example:

```
class Stock(object):
    __slots__ = ['name','shares','price']
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

`__slots__` is sometimes viewed as a safety feature because it restricts the set of attribute names. However, it is really more of a performance optimization. Classes that use `__slots__` don't use a dictionary for storing instance data (instead, a more efficient internal data structure is used). So, not only will instances use far less memory, but access to instance data is also more efficient. In some cases, simply adding `__slots__` will make a program run noticeably faster without making any other changes.

There is one caution with using `__slots__`, however. Adding this feature to a class may cause other code to break mysteriously. For example, it is generally well-known that instances store their data in a dictionary that can be accessed as the `__dict__` attribute. When slots are defined, this attribute doesn't exist so any code that relies on `__dict__` will fail.

Avoid the (.) Operator

Whenever you use the (.) to look up an attribute on an object, it always involves a name lookup. For example, when you say `x.name`, there is a lookup for the variable name `"x"` in the environment and then a lookup for the attribute `"name"` on `x`. For user-defined objects, attribute lookup may involve looking in the instance dictionary, the class dictionary, and the dictionaries of base-classes.

For calculations involving heavy use of methods or module lookups, it is almost always better to eliminate the attribute lookup by putting the operation you want to perform into a local variable first. For example, if you were performing a lot of square root operations, it is faster to use `'from math import sqrt'` and `'sqrt(x)'` rather than typing `'math.sqrt(x)'`. In the first part of this section, we saw that this approach resulted in speedup of about 1.4.

Obviously you should not try to eliminate attribute lookups everywhere in your program because it will make your code very difficult to read. However, for performance-critical sections, this is a useful technique.

Use Exceptions to Handle Uncommon Cases

To avoid errors, you might be inclined to add extra checks to a program. For example:

```
def parse_header(line):
    fields = line.split(":")
    if len(fields) != 2:
        raise RuntimeError("Malformed header")
    header, value = fields
    return header.lower(), value.strip()
```

However, an alternative way to handle errors is to simply let the program generate an exception and to catch it. For example:

```
def parse_header(line):
    fields = line.split(":")
    try:
        header, value = fields
        return header.lower(), value.strip()
    except ValueError:
        raise RuntimeError("Malformed header")
```

If you benchmark both versions on a properly formatted line, the second version of code runs about 10 percent faster. Setting up a `try` block for code that normally doesn't raise an exceptions runs more quickly than executing an `if` statement.

Avoid Exceptions for Common Cases

Don't write code that uses exception handling for the common case. For example, suppose you had a program that performed a lot of dictionary lookups, but most of these lookups were for keys that didn't exist. Now, consider two approaches to performing a lookup:

```
 # Approach 1 : Perform a lookup and catch an exception
try:
    value = items[key]
except KeyError:
    value = None

# Approach 2: Check if the key exists and perform a lookup
if key in items:
   value = items[key]
else:
   value = None
```

In a simple performance measurement where the key is not found, the second approach runs more than 17 times faster! In case you were wondering, this latter approach also runs almost twice as fast as using `items.get(key)` because the `in` operator is faster to execute than a method call.

Embrace Functional Programming and Iteration

List comprehensions, generator expressions, generators, coroutines, and closures are much more efficient than most Python programmers realize. For data processing especially, list comprehensions and generator expressions run significantly more quickly than code that manually iterates over data and carries out similar operations. These operations also run much more quickly than legacy Python code that uses functions such as `map()` and `filter()`. Generators can be used to write code that not only runs fast, but which makes efficient use of memory.

Use Decorators and Metaclasses

Decorators and metaclasses are features that are used to modify functions and classes. However, because they operate at the time of function or class definition, they can be used in ways that lead to improved performance—especially if a program has many optional features that might be turned on or off. Chapter 6, "Functions and Functional Programming," has an example of using a decorator to enable logging of functions, but in a way that does not impact performance when logging is disabled.

Part 2: MongoDB

Table of Contents

Foreword

In the last 10 years, the Internet has challenged relational databases in ways nobody could have foreseen. Having used MySQL at large and growing Internet companies during this time, I've seen this happen firsthand. First you have a single server with a small data set. Then you find yourself setting up replication so you can scale out reads and deal with potential failures. And, before too long, you've added a caching layer, tuned all the queries, and thrown even more hardware at the problem.

Eventually you arrive at the point when you need to shard the data across multiple clusters and rebuild a ton of application logic to deal with it. And soon after that you realize that you're locked into the schema you modeled so many months before.

Why? Because there's so much data in your clusters now that altering the schema will take a long time and involve a lot of precious DBA time. It's easier just to work around it in code. This can keep a small team of developers busy for many months. In the end, you'll always find yourself wondering if there's a better way—or why more of these features are not built into the core database server.

Keeping with tradition, the Open Source community has created a plethora of "better ways" in response to the ballooning data needs of modern web applications. They span the spectrum from simple in-memory key/value stores to complicated SQL-speaking MySQL/InnoDB derivatives. But the sheer number of choices has made finding the right solution more difficult. I've looked at many of them.

I was drawn to MongoDB by its pragmatic approach. MongoDB doesn't try to be everything to everyone. Instead it strikes the right balance between features and complexity, with a clear bias toward making previously difficult tasks far easier. In other words, it has the features that really matter to the vast majority of today's web applications: indexes, replication, sharding, a rich query syntax, and a very flexible data model. All of this comes without sacrificing speed.

Like MongoDB itself, this book is very straightforward and approachable. New MongoDB users can start with Chapter 1 and be up and running in no time. Experienced users will appreciate this book's breadth and authority. It's a solid reference for advanced administrative topics such as replication, backups, and sharding, as well as popular client APIs.

Having recently started to use MongoDB in my day job, I have no doubt that this book will be at my side for the entire journey—from the first install to production deployment of a sharded and replicated cluster. It's an essential reference to anyone seriously looking at using MongoDB.

—Jeremy Zawodny
Craigslist Software Engineer
August 2010

How This Book Is Organized

Getting Up to Speed with MongoDB

In Chapter 1, *Introduction*, we provide some background about MongoDB: why it was created, the goals it is trying to accomplish, and why you might choose to use it for a project. We go into more detail in Chapter 2, *Getting Started*, which provides an introduction to the core concepts and vocabulary of MongoDB. Chapter 2 also provides a first look at working with MongoDB, getting you started with the database and the shell.

Developing with MongoDB

The next two chapters cover the basic material that developers need to know to work with MongoDB. In Chapter 3, *Creating, Updating, and Deleting Documents*, we describe how to perform those basic write operations, including how to do them with different levels of safety and speed. Chapter 4, *Querying*, explains how to find documents and create complex queries. This chapter also covers how to iterate through results and options for limiting, skipping, and sorting results.

Advanced Usage

The next three chapters go into more complex usage than simply storing and retrieving data. Chapter 5, *Indexing*, explains what indexes are and how to use them with MongoDB. It also covers tools you can use to examine or modify the indexes used to perform a query, and it covers index administration. Chapter 6, *Aggregation*, covers a number of techniques for aggregating data with MongoDB, including counting, finding distinct values, grouping documents, and using MapReduce. Chapter 7, *Advanced Topics*, is a mishmash of important tidbits that didn't fit into any of the previous categories: file storage, server-side JavaScript, database commands, and database references.

Administration

The next three chapters are less about programming and more about the operational aspects of MongoDB. Chapter 8, *Administration*, discusses options for starting the database in different ways, monitoring a MongoDB server, and keeping deployments secure. Chapter 8 also covers how to keep proper backups of the data you've stored in MongoDB. In Chapter 9, *Replication*, we explain how to set up replication with MongoDB, including standard master-slave configuration and setups with automatic failover. This chapter also covers how MongoDB replication works and options for tweaking it. Chapter 10, *Sharding*, describes how to scale MongoDB horizontally: it covers what autosharding is, how to set it up, and the ways in which it impacts applications.

Developing Applications with MongoDB

In Chapter 11, *Example Applications*, we provide example applications using MongoDB, written in Java, PHP, Python, and Ruby. These examples illustrate how to map the concepts described earlier in the book to specific languages and problem domains.

Appendixes

Appendix A, *Installing MongoDB*, explains MongoDB's versioning scheme and how to install it on Windows, OS X, and Linux. Appendix B, *mongo: The Shell*, includes some useful shell tips and tools. Finally, Appendix C, *MongoDB Internals*, details a little about how MongoDB works internally: its storage engine, data format, and wire protocol.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, collection names, database names, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, command-line utilities, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

Introduction

MongoDB is a powerful, flexible, and scalable data store. It combines the ability to scale out with many of the most useful features of relational databases, such as secondary indexes, range queries, and sorting. MongoDB is also incredibly featureful: it has tons of useful features such as built-in support for MapReduce-style aggregation and geospatial indexes.

There is no point in creating a great technology if it's impossible to work with, so a lot of effort has been put into making MongoDB easy to get started with and a pleasure to use. MongoDB has a developer-friendly data model, administrator-friendly configuration options, and natural-feeling language APIs presented by drivers and the database shell. MongoDB tries to get out of your way, letting you program instead of worrying about storing data.

A Rich Data Model

MongoDB is a *document-oriented* database, not a relational one. The primary reason for moving away from the relational model is to make scaling out easier, but there are some other advantages as well.

The basic idea is to replace the concept of a "row" with a more flexible model, the "document." By allowing embedded documents and arrays, the document-oriented approach makes it possible to represent complex hierarchical relationships with a single record. This fits very naturally into the way developers in modern object-oriented languages think about their data.

MongoDB is also schema-free: a document's keys are not predefined or fixed in any way. Without a schema to change, massive data migrations are usually unnecessary. New or missing keys can be dealt with at the application level, instead of forcing all data to have the same shape. This gives developers a lot of flexibility in how they work with evolving data models.

Easy Scaling

Data set sizes for applications are growing at an incredible pace. Advances in sensor technology, increases in available bandwidth, and the popularity of handheld devices that can be connected to the Internet have created an environment where even small-scale applications need to store more data than many databases were meant to handle. A terabyte of data, once an unheard-of amount of information, is now commonplace.

As the amount of data that developers need to store grows, developers face a difficult decision: how should they scale their databases? Scaling a database comes down to the choice between scaling up (getting a bigger machine) or scaling out (partitioning data across more machines). Scaling up is often the path of least resistance, but it has drawbacks: large machines are often very expensive, and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. For the type of large web application that most people aspire to build, it is either impossible or not cost-effective to run off of one machine. Alternatively, it is both extensible and economical to scale *out*: to add storage space or increase performance, you can buy another commodity server and add it to your cluster.

MongoDB was designed from the beginning to scale out. Its document-oriented data model allows it to automatically split up data across multiple servers. It can balance data and load across a cluster, redistributing documents automatically. This allows developers to focus on programming the application, not scaling it. When they need more capacity, they can just add new machines to the cluster and let the database figure out how to organize everything.

Tons of Features…

It's difficult to quantify what a feature is: anything above and beyond what a relational database provides? Memcached? Other document-oriented databases? However, no matter what the baseline is, MongoDB has some really nice, unique tools that are not (all) present in any other solution.

Indexing
: MongoDB supports generic secondary indexes, allowing a variety of fast queries, and provides unique, compound, and geospatial indexing capabilities as well.

Stored JavaScript
: Instead of stored procedures, developers can store and use JavaScript functions and values on the server side.

Aggregation
: MongoDB supports MapReduce and other aggregation tools.

Fixed-size collections
: Capped collections are fixed in size and are useful for certain types of data, such as logs.

File storage
: MongoDB supports an easy-to-use protocol for storing large files and file metadata.

Some features common to relational databases are not present in MongoDB, notably joins and complex multirow transactions. These are architectural decisions to allow for scalability, because both of those features are difficult to provide efficiently in a distributed system.

. . . Without Sacrificing Speed

Incredible performance is a major goal for MongoDB and has shaped many design decisions. MongoDB uses a binary wire protocol as the primary mode of interaction with the server (as opposed to a protocol with more overhead, like HTTP/REST). It adds dynamic padding to documents and preallocates data files to trade extra space usage for consistent performance. It uses memory-mapped files in the default storage engine, which pushes the responsibility for memory management to the operating system. It also features a dynamic query optimizer that "remembers" the fastest way to perform a query. In short, almost every aspect of MongoDB was designed to maintain high performance.

Although MongoDB is powerful and attempts to keep many features from relational systems, it is not intended to do everything that a relational database does. Whenever possible, the database server offloads processing and logic to the client side (handled either by the drivers or by a user's application code). Maintaining this streamlined design is one of the reasons MongoDB can achieve such high performance.

Simple Administration

MongoDB tries to simplify database administration by making servers administrate themselves as much as possible. Aside from starting the database server, very little administration is necessary. If a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to a master. In a distributed environment, the cluster needs to be told only that a new node exists to automatically integrate and configure it.

MongoDB's administration philosophy is that the server should handle as much of the configuration as possible automatically, allowing (but not requiring) users to tweak their setups if needed.

But Wait, That's Not All . . .

Throughout the course of the book, we will take the time to note the reasoning or motivation behind particular decisions made in the development of MongoDB. Through those notes we hope to share the philosophy behind MongoDB. The best way to summarize the MongoDB project, however, is through its main focus—to create a full-featured data store that is scalable, flexible, and fast.

Getting Started

MongoDB is very powerful, but it is still easy to get started with. In this chapter we'll introduce some of the basic concepts of MongoDB:

- A *document* is the basic unit of data for MongoDB, roughly equivalent to a row in a relational database management system (but much more expressive).
- Similarly, a *collection* can be thought of as the schema-free equivalent of a table.
- A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections and permissions.
- MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation.
- Every document has a special key, `"_id"`, that is unique across the document's collection.

Documents

At the heart of MongoDB is the concept of a *document*: an ordered set of keys with associated values. The representation of a document differs by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary. In JavaScript, for example, documents are represented as objects:

```
{"greeting" : "Hello, world!"}
```

This simple document contains a single key, `"greeting"`, with a value of `"Hello, world!"`. Most documents will be more complex than this simple one and often will contain multiple key/value pairs:

```
{"greeting" : "Hello, world!", "foo" : 3}
```

This example is a good illustration of several important concepts:

- Key/value pairs in documents are ordered—the earlier document is distinct from the following document:

  ```
  {"foo" : 3, "greeting" : "Hello, world!"}
  ```

  > In most cases the ordering of keys in documents is not important. In fact, in some programming languages the default representation of a document does not even maintain ordering (e.g., dictionaries in Python and hashes in Perl or Ruby 1.8). Drivers for those languages usually have some mechanism for specifying documents with ordering for the rare cases when it is necessary. (Those cases will be noted throughout the text.)

- Values in documents are not just "blobs." They can be one of several different data types (or even an entire embedded document—see "Embedded Documents" on page 20). In this example the value for `"greeting"` is a string, whereas the value for `"foo"` is an integer.

The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:

- Keys must not contain the character \0 (the `null` character). This character is used to signify the end of a key.
- The . and $ characters have some special properties and should be used only in certain circumstances, as described in later chapters. In general, they should be considered reserved, and drivers will complain if they are used inappropriately.
- Keys starting with _ should be considered reserved; although this is not strictly enforced.

MongoDB is type-sensitive and case-sensitive. For example, these documents are distinct:

```
{"foo" : 3}
{"foo" : "3"}
```

As are as these:

```
{"foo" : 3}
{"Foo" : 3}
```

A final important thing to note is that documents in MongoDB cannot contain duplicate keys. For example, the following is not a legal document:

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

Collections

A *collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Schema-Free

Collections are *schema-free*. This means that the documents within a single collection can have any number of different "shapes." For example, both of the following documents could be stored in a single collection:

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

Note that the previous documents not only have different types for their values (string versus integer) but also have entirely different keys. Because any document can be put into any collection, the question often arises: "Why do we need separate collections at all?" It's a good question—with no need for separate schemas for different kinds of documents, why *should* we use more than one collection? There are several good reasons:

- Keeping different kinds of documents in the same collection can be a nightmare for developers and admins. Developers need to make sure that each query is only returning documents of a certain kind or that the application code performing a query can handle documents of different shapes. If we're querying for blog posts, it's a hassle to weed out documents containing author data.
- It is much faster to get a list of collections than to extract a list of the types in a collection. For example, if we had a `type` key in the collection that said whether each document was a "skim," "whole," or "chunky monkey" document, it would be much slower to find those three values in a single collection than to have three separate collections and query for their names (see "Subcollections" on page 8).
- Grouping documents of the same kind together in the same collection allows for data locality. Getting several blog posts from a collection containing only posts will likely require fewer disk seeks than getting the same posts from a collection containing posts and author data.
- We begin to impose some structure on our documents when we create indexes. (This is especially true in the case of unique indexes.) These indexes are defined per collection. By putting only documents of a single type into the same collection, we can index our collections more efficiently.

As you can see, there are sound reasons for creating a schema and for grouping related types of documents together. MongoDB just relaxes this requirement and allows developers more flexibility.

Naming

A collection is identified by its name. Collection names can be any UTF-8 string, with a few restrictions:

- The empty string ("") is not a valid collection name.
- Collection names may not contain the character \0 (the `null` character) because this delineates the end of a collection name.
- You should not create any collections that start with *system.*, a prefix reserved for system collections. For example, the *system.users* collection contains the database's users, and the *system.namespaces* collection contains information about all of the database's collections.
- User-created collections should not contain the reserved character $ in the name. The various drivers available for the database do support using $ in collection names because some system-generated collections contain it. You should not use $ in a name unless you are accessing one of these collections.

Subcollections

One convention for organizing collections is to use namespaced subcollections separated by the . character. For example, an application containing a blog might have a collection named *blog.posts* and a separate collection named *blog.authors*. This is for organizational purposes only—there is no relationship between the *blog* collection (it doesn't even have to exist) and its "children."

Although subcollections do not have any special properties, they are useful and incorporated into many MongoDB tools:

- GridFS, a protocol for storing large files, uses subcollections to store file metadata separately from content chunks (see Chapter 7 for more information about GridFS).
- The MongoDB web console organizes the data in its DBTOP section by subcollection (see Chapter 8 for more information on administration).
- Most drivers provide some syntactic sugar for accessing a subcollection of a given collection. For example, in the database shell, `db.blog` will give you the *blog* collection, and `db.blog.posts` will give you the *blog.posts* collection.

Subcollections are a great way to organize data in MongoDB, and their use is highly recommended.

Databases

In addition to grouping documents by collection, MongoDB groups collections into *databases*. A single instance of MongoDB can host several databases, each of which can be thought of as completely independent. A database has its own permissions, and each database is stored in separate files on disk. A good rule of thumb is to store all data for a single application in the same database. Separate databases are useful when storing data for several application or users on the same MongoDB server.

Like collections, databases are identified by name. Database names can be any UTF-8 string, with the following restrictions:

- The empty string ("") is not a valid database name.
- A database name cannot contain any of these characters: ' ' (a single space), ., $, /, \, or \0 (the null character).
- Database names should be all lowercase.
- Database names are limited to a maximum of 64 bytes.

One thing to remember about database names is that they will actually end up as files on your filesystem. This explains why many of the previous restrictions exist in the first place.

There are also several reserved database names, which you can access directly but have special semantics. These are as follows:

admin
: This is the "root" database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases. There are also certain server-wide commands that can be run only from the *admin* database, such as listing all of the databases or shutting down the server.

local
: This database will never be replicated and can be used to store any collections that should be local to a single server (see Chapter 9 for more information about replication and the local database).

config
: When Mongo is being used in a sharded setup (see Chapter 10), the *config* database is used internally to store information about the shards.

By prepending a collection's name with its containing database, you can get a fully qualified collection name called a *namespace*. For instance, if you are using the *blog.posts* collection in the *cms* database, the namespace of that collection would be `cms.blog.posts`. Namespaces are limited to 121 bytes in length and, in practice, should be less than 100 bytes long. For more on namespaces and the internal representation of collections in MongoDB, see Appendix C.

Getting and Starting MongoDB

MongoDB is almost always run as a network server that clients can connect to and perform operations on. To start the server, run the `mongod` executable:

```
$ ./mongod
./mongod --help for help and startup options
Sun Mar 28 12:31:20 Mongo DB : starting : pid = 44978 port = 27017
dbpath = /data/db/ master = 0 slave = 0  64-bit
Sun Mar 28 12:31:20 db version v1.5.0-pre-, pdfile version 4.5
Sun Mar 28 12:31:20 git version: ...
Sun Mar 28 12:31:20 sys info: ...
Sun Mar 28 12:31:20 waiting for connections on port 27017
Sun Mar 28 12:31:20 web admin interface listening on port 28017
```

Or if you're on Windows, run this:

```
$ mongod.exe
```

> For detailed information on installing MongoDB on your system, see Appendix A.

When run with no arguments, `mongod` will use the default data directory, */data/db/* (or *C:\data\db* on Windows), and port 27017. If the data directory does not already exist or is not writable, the server will fail to start. It is important to create the data directory (e.g., *mkdir -p /data/db/*), and to make sure your user has permission to write to the directory, before starting MongoDB. The server will also fail to start if the port is not available—this is often caused by another instance of MongoDB that is already running.

The server will print some version and system information and then begin waiting for connections. By default, MongoDB listens for socket connections on port 27017.

`mongod` also sets up a very basic HTTP server that listens on a port 1,000 higher than the main port, in this case 28017. This means that you can get some administrative information about your database by opening a web browser and going to *http://localhost:28017*.

You can safely stop `mongod` by typing Ctrl-c in the shell that is running the server.

> For more information on starting or stopping MongoDB, see "Starting and Stopping MongoDB" on page 111, and for more on the administrative interface, see "Using the Admin Interface" on page 115.

MongoDB Shell

MongoDB comes with a JavaScript shell that allows interaction with a MongoDB instance from the command line. The shell is very useful for performing administrative functions, inspecting a running instance, or just playing around. The `mongo` shell is a crucial tool for using MongoDB and is used extensively throughout the rest of the text.

Running the Shell

To start the shell, run the `mongo` executable:

```
$ ./mongo
MongoDB shell version: 1.6.0
url: test
connecting to: test
type "help" for help
>
```

The shell automatically attempts to connect to a MongoDB server on startup, so make sure you start `mongod` before starting the shell.

The shell is a full-featured JavaScript interpreter, capable of running arbitrary JavaScript programs. To illustrate this, let's perform some basic math:

```
> x = 200
200
> x / 5;
40
```

We can also leverage all of the standard JavaScript libraries:

```
> Math.sin(Math.PI / 2);
1
> new Date("2010/1/1");
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

We can even define and call JavaScript functions:

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

Note that you can create multiline commands. The shell will detect whether the JavaScript statement is complete when you press Enter and, if it is not, will allow you to continue writing it on the next line.

A MongoDB Client

Although the ability to execute arbitrary JavaScript is cool, the real power of the shell lies in the fact that it is also a stand-alone MongoDB client. On startup, the shell connects to the *test* database on a MongoDB server and assigns this database connection to the global variable `db`. This variable is the primary access point to MongoDB through the shell.

The shell contains some add-ons that are not valid JavaScript syntax but were implemented because of their familiarity to users of SQL shells. The add-ons do not provide any extra functionality, but they are nice syntactic sugar. For instance, one of the most important operations is selecting which database to use:

```
> use foobar
switched to db foobar
```

Now if you look at the `db` variable, you can see that it refers to the *foobar* database:

```
> db
foobar
```

Because this is a JavaScript shell, typing a variable will convert the variable to a string (in this case, the database name) and print it.

Collections can be accessed from the `db` variable. For example, `db.baz` returns the *baz* collection in the current database. Now that we can access a collection in the shell, we can perform almost any database operation.

Basic Operations with the Shell

We can use the four basic operations, create, read, update, and delete (CRUD), to manipulate and view data in the shell.

Create

The `insert` function adds a document to a collection. For example, suppose we want to store a blog post. First, we'll create a local variable called `post` that is a JavaScript object representing our document. It will have the keys `"title"`, `"content"`, and `"date"` (the date that it was published):

```
> post = {"title" : "My Blog Post",
... "content" : "Here's my blog post.",
... "date" : new Date()}
{
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

This object is a valid MongoDB document, so we can save it to the *blog* collection using the `insert` method:

```
> db.blog.insert(post)
```

The blog post has been saved to the database. We can see it by calling `find` on the collection:

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

You can see that an `"_id"` key was added and that the other key/value pairs were saved as we entered them. The reason for `"_id"`'s sudden appearance is explained at the end of this chapter.

Read

`find` returns all of the documents in a collection. If we just want to see one document from a collection, we can use `findOne`:

```
> db.blog.findOne()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

`find` and `findOne` can also be passed criteria in the form of a query document. This will restrict the documents matched by the query. The shell will automatically display up to 20 documents matching a `find`, but more can be fetched. See Chapter 4 for more information on querying.

Update

If we would like to modify our post, we can use `update`. `update` takes (at least) two parameters: the first is the criteria to find which document to update, and the second is the new document. Suppose we decide to enable comments on the blog post we created earlier. We'll need to add an array of comments as the value for a new key in our document.

The first step is to modify the variable `post` and add a `"comments"` key:

```
> post.comments = []
[ ]
```

Then we perform the update, replacing the post titled "My Blog Post" with our new version of the document:

```
> db.blog.update({title : "My Blog Post"}, post)
```

Now the document has a `"comments"` key. If we call `find` again, we can see the new key:

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
    "comments" : [ ]
}
```

Delete

`remove` deletes documents permanently from the database. Called with no parameters, it removes all documents from a collection. It can also take a document specifying criteria for removal. For example, this would remove the post we just created:

```
> db.blog.remove({title : "My Blog Post"})
```

Now the collection will be empty again.

Tips for Using the Shell

Because `mongo` is simply a JavaScript shell, you can get a great deal of help for it by simply looking up JavaScript documentation online. The shell also includes built-in help that can be accessed by typing `help`:

```
> help
HELP
    show dbs                    show database names
    show collections            show collections in current database
    show users                  show users in current database
    show profile                show recent system.profile entries w. time >= 1ms
    use <db name>               set current database to <db name>
    db.help()                   help on DB methods
    db.foo.help()               help on collection methods
    db.foo.find()               list objects in collection foo
    db.foo.find( { a : 1 } )    list objects in foo where a == 1
    it                          result of the last line evaluated
```

Help for database-level commands is provided by `db.help();`, and help at the collections can be accessed with `db.foo.help();`.

A good way of figuring out what a function is doing is to type it without the parentheses. This will print the JavaScript source code for the function. For example, if we are curious about how the `update` function works or cannot remember the order of parameters, we can do the following:

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
```

```
                    upsert ? true : false, multi ? true : false);
}
```

There is also an autogenerated API of all the JavaScript functions provided by the shell at *http://api.mongodb.org/js*.

Inconvenient collection names

Fetching a collection with `db.collectionName` almost always works, unless the collection name actually is a property of the database class. For instance, if we are trying to access the *version* collection, we cannot say `db.version` because `db.version` is a database function. (It returns the version of the running MongoDB server.)

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
```

`db`'s collection-returning behavior is only a fallback for when JavaScript cannot find a matching property. When there is a property with the same name as the desired collection, we can use the `getCollection` function:

```
> db.getCollection("version");
test.version
```

This can also be handy for collections with invalid JavaScript in their names. For example, *foo-bar* is a valid collection name, but it's variable subtraction in JavaScript. You can get the *foo-bar* collection with `db.getCollection("foo-bar")`.

In JavaScript, `x.y` is identical to `x['y']`. This means that subcollections can be accessed using variables, not just literal names. That is, if you needed to perform some operation on every *blog* subcollection, you could iterate through them with something like this:

```
var collections = ["posts", "comments", "authors"];

for (i in collections) {
    doStuff(db.blog[collections[i]]);
}
```

Instead of this:

```
doStuff(db.blog.posts);
doStuff(db.blog.comments);
doStuff(db.blog.authors);
```

Data Types

The beginning of this chapter covered the basics of what a document is. Now that you are up and running with MongoDB and can try things on the shell, this section will dive a little deeper. MongoDB supports a wide range of data types as values in documents. In this section, we'll outline all of the supported types.

Basic Data Types

Documents in MongoDB can be thought of as "JSON-like" in that they are conceptually similar to objects in JavaScript. JSON is a simple representation of data: the specification can be described in about one paragraph (*http://www.json.org* proves it) and lists only six data types. This is a good thing in many ways: it's easy to understand, parse, and remember. On the other hand, JSON's expressive capabilities are limited, because the only types are null, boolean, numeric, string, array, and object.

Although these types allow for an impressive amount of expressivity, there are a couple of additional types that are crucial for most applications, especially when working with a database. For example, JSON has no date type, which makes working with dates even more annoying than it usually is. There is a number type, but only one—there is no way to differentiate floats and integers, never mind any distinction between 32-bit and 64-bit numbers. There is no way to represent other commonly used types, either, such as regular expressions or functions.

MongoDB adds support for a number of additional data types while keeping JSON's essential key/value pair nature. Exactly how values of each type are represented varies by language, but this is a list of the commonly supported types and how they are represented as part of a document in the shell:

null
: Null can be used to represent both a null value and a nonexistent field:

```
{"x" : null}
```

boolean
: There is a boolean type, which will be used for the values `'true'` and `'false'`:

```
{"x" : true}
```

32-bit integer
: This cannot be represented on the shell. As mentioned earlier, JavaScript supports only 64-bit floating point numbers, so 32-bit integers will be converted into those.

64-bit integer
: Again, the shell cannot represent these. The shell will display them using a special embedded document; see the section "Numbers" on page 18 for details.

64-bit floating point number
: All numbers in the shell will be of this type. Thus, this will be a floating-point number:

```
{"x" : 3.14}
```

As will this:

```
{"x" : 3}
```

string
: Any string of UTF-8 characters can be represented using the string type:

```
{"x" : "foobar"}
```

symbol
: This type is not supported by the shell. If the shell gets a symbol from the database, it will convert it into a string.

object id
: An object id is a unique 12-byte ID for documents. See the section "_id and ObjectIds" on page 20 for details:

```
{"x" : ObjectId()}
```

date
: Dates are stored as milliseconds since the epoch. The time zone is not stored:

```
{"x" : new Date()}
```

regular expression
: Documents can contain regular expressions, using JavaScript's regular expression syntax:

```
{"x" : /foobar/i}
```

code
: Documents can also contain JavaScript code:

```
{"x" : function() { /* ... */ }}
```

binary data
: Binary data is a string of arbitrary bytes. It cannot be manipulated from the shell.

maximum value
: BSON contains a special type representing the largest possible value. The shell does not have a type for this.

minimum value
: BSON contains a special type representing the smallest possible value. The shell does not have a type for this.

undefined
: Undefined can be used in documents as well (JavaScript has distinct types for null and undefined):

```
{"x" : undefined}
```

array
: Sets or lists of values can be represented as arrays:

```
{"x" : ["a", "b", "c"]}
```

embedded document
: Documents can contain entire documents, embedded as values in a parent document:

```
{"x" : {"foo" : "bar"}}
```

Numbers

JavaScript has one "number" type. Because MongoDB has three number types (4-byte integer, 8-byte integer, and 8-byte float), the shell has to hack around JavaScript's limitations a bit. By default, any number in the shell is treated as a double by MongoDB. This means that if you retrieve a 4-byte integer from the database, manipulate its document, and save it back to the database *even without changing the integer*, the integer will be resaved as a floating-point number. Thus, it is generally a good idea not to overwrite entire documents from the shell (see Chapter 3 for information on making changes to the values of individual keys).

Another problem with every number being represented by a double is that there are some 8-byte integers that cannot be accurately represented by 8-byte floats. Therefore, if you save an 8-byte integer and look at it in the shell, the shell will display it as an embedded document indicating that it might not be exact. For example, if we save a document with a `"myInteger"` key whose value is the 64-bit integer, `3`, and then look at it in the shell, it will look like this:

```
> doc = db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa08"),
    "myInteger" : {
        "floatApprox" : 3
    }
}
```

The number is not changed in the database (unless you modify and resave the object from the shell, in which case it will turn into a float); the embedded document just indicates that the shell is displaying a floating-point approximation of an 8-byte integer. If this embedded document has only one key, it is, in fact, exact.

If you insert an 8-byte integer that cannot be accurately displayed as a double, the shell will add two keys, `"top"` and `"bottom"`, containing the 32-bit integers representing the 4 high-order bytes and 4 low-order bytes of the integer, respectively. For instance, if we insert `9223372036854775807`, the shell will show us the following:

```
> db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa09"),
    "myInteger" : {
        "floatApprox" : 9223372036854776000,
        "top" : 2147483647,
        "bottom" : 4294967295
    }
}
```

The `"floatApprox"` embedded documents are special and can be manipulated as numbers as well as documents:

```
> doc.myInteger + 1
4
```

```
> doc.myInteger.floatApprox
3
```

All 4-byte integers can be represented exactly by an 8-byte floating-point number, so they are displayed normally.

Dates

In JavaScript, the `Date` object is used for MongoDB's date type. When creating a new `Date` object, always call `new Date(...)`, not just `Date(...)`. Calling the constructor as a function (that is, not including `new`) returns a string representation of the date, not an actual `Date` object. This is not MongoDB's choice; it is how JavaScript works. If you are not careful to always use the `Date` constructor, you can end up with a mishmash of strings and dates. Strings do not match dates, and vice versa, so this can cause problems with removing, updating, querying...pretty much everything.

For a full explanation of JavaScript's `Date` class and acceptable formats for the constructor, see ECMAScript specification section 15.9 (available for download at *http://www.ecmascript.org*).

Dates in the shell are displayed using local time zone settings. However, dates in the database are just stored as milliseconds since the epoch, so they have no time zone information associated with them. (Time zone information could, of course, be stored as the value for another key.)

Arrays

Arrays are values that can be interchangeably used for both ordered operations (as though they were lists, stacks, or queues) and unordered operations (as though they were sets).

In the following document, the key `"things"` has an array value:

```
{"things" : ["pie", 3.14]}
```

As we can see from the example, arrays can contain different data types as values (in this case, a string and a floating-point number). In fact, array values can be any of the supported values for normal key/value pairs, even nested arrays.

One of the great things about arrays in documents is that MongoDB "understands" their structure and knows how to "reach inside" of arrays to perform operations on their contents. This allows us to query on arrays and build indexes using their contents. For instance, in the previous example, MongoDB can query for all documents where 3.14 is an element of the `"things"` array. If this is a common query, you can even create an index on the `"things"` key to improve the query's speed.

MongoDB also allows atomic updates that modify the contents of arrays, such as reaching into the array and changing the value *pie* to *pi*. We'll see more examples of these types of operations throughout the text.

Embedded Documents

Embedded documents are entire MongoDB documents that are used as the *value* for a key in another document. They can be used to organize data in a more natural way than just a flat structure.

For example, if we have a document representing a person and want to store his address, we can nest this information in an embedded `"address"` document:

```
{
    "name" : "John Doe",
    "address" : {
        "street" : "123 Park Street",
        "city" : "Anytown",
        "state" : "NY"
    }
}
```

The value for the `"address"` key in the previous example is another document with its own values for `"street"`, `"city"`, and `"state"`.

As with arrays, MongoDB "understands" the structure of embedded documents and is able to "reach inside" of them to build indexes, perform queries, or make updates.

We'll discuss schema design in depth later, but even from this basic example, we can begin to see how embedded documents can change the way we work with data. In a relational database, the previous document would probably be modeled as two separate rows in two different tables (one for "people" and one for "addresses"). With MongoDB we can embed the address document directly within the person document. When used properly, embedded documents can provide a more natural (and often more efficient) representation of information.

The flip side of this is that we are basically denormalizing, so there can be more data repetition with MongoDB. Suppose "addresses" were a separate table in a relational database and we needed to fix a typo in an address. When we did a join with "people" and "addresses," we'd get the updated address for everyone who shares it. With MongoDB, we'd need to fix the typo in each person's document.

_id and ObjectIds

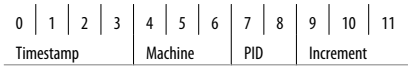
Every document stored in MongoDB must have an `"_id"` key. The `"_id"` key's value can be any type, but it defaults to an `ObjectId`. In a single collection, every document must have a unique value for `"_id"`, which ensures that every document in a collection can be uniquely identified. That is, if you had two collections, each one could have a document where the value for `"_id"` was 123. However, neither collection could contain more than one document where `"_id"` was 123.

ObjectIds

`ObjectId` is the default type for `"_id"`. It is designed to be lightweight, while still being easy to generate in a globally unique way across disparate machines. This is the main reason why MongoDB uses `ObjectId`s as opposed to something more traditional, like an autoincrementing primary key: it is difficult and time-consuming to synchronize autoincrementing primary keys across multiple servers. Because MongoDB was designed from the beginning to be a distributed database, dealing with many nodes is an important consideration. The `ObjectId` type, as we'll see, is easy to generate in a sharded environment.

`ObjectId`s use 12 bytes of storage, which gives them a string representation that is 24 hexadecimal digits: 2 digits for each byte. This causes them to appear larger than they are, which makes some people nervous. It's important to note that even though an `ObjectId` is often represented as a giant hexadecimal string, the string is actually twice as long as the data being stored.

If you create multiple new `ObjectId`s in rapid succession, you can see that only the last few digits change each time. In addition, a couple of digits in the middle of the `ObjectId` will change (if you space the creations out by a couple of seconds). This is because of the manner in which `ObjectId`s are created. The 12 bytes of an `ObjectId` are generated as follows:



The first four bytes of an `ObjectId` are a timestamp in seconds since the epoch. This provides a couple of useful properties:

- The timestamp, when combined with the next five bytes (which will be described in a moment), provides uniqueness at the granularity of a second.
- Because the timestamp comes first, it means that `ObjectId`s will sort in *roughly* insertion order. This is not a strong guarantee but does have some nice properties, such as making `ObjectId`s efficient to index.
- In these four bytes exists an implicit timestamp of when each document was created. Most drivers expose a method for extracting this information from an `ObjectId`.

Because the current time is used in `ObjectId`s, some users worry that their servers will need to have synchronized clocks. This is not necessary because the actual value of the timestamp doesn't matter, only that it is often new (once per second) and increasing.

The next three bytes of an `ObjectId` are a unique identifier of the machine on which it was generated. This is usually a hash of the machine's hostname. By including these bytes, we guarantee that different machines will not generate colliding `ObjectId`s.

To provide uniqueness among different processes generating `ObjectId`s concurrently on a single machine, the next two bytes are taken from the process identifier (PID) of the `ObjectId`-generating process.

These first nine bytes of an `ObjectId` guarantee its uniqueness across machines and processes for a single second. The last three bytes are simply an incrementing counter that is responsible for uniqueness within a second in a single process. This allows for up to 256^3 (16,777,216) unique `ObjectId`s to be generated *per process* in a single second.

Autogeneration of _id

As stated previously, if there is no `"_id"` key present when a document is inserted, one will be automatically added to the inserted document. This can be handled by the MongoDB server but will generally be done by the driver on the client side. There are a couple of reasons for that:

- Although `ObjectId`s are designed to be lightweight and easy to generate, there is still some overhead involved in their generation. The decision to generate them on the client side reflects an overall philosophy of MongoDB: work should be pushed out of the server and to the drivers whenever possible. This philosophy reflects the fact that, even with scalable databases like MongoDB, it is easier to scale out at the application layer than at the database layer. Moving work to the client side reduces the burden requiring the database to scale.
- By generating `ObjectId`s on the client side, drivers are capable of providing richer APIs than would be otherwise possible. For example, a driver might have its `insert` method either return the generated `ObjectId` or inject it directly into the document that was inserted. If the driver allowed the server to generate `ObjectId`s, then a separate query would be required to determine the value of `"_id"` for an inserted document.

CHAPTER 3

Creating, Updating, and Deleting Documents

This chapter covers the basics of moving data into and out of the database, including the following:

- Adding new documents to a collection
- Removing documents from a collection
- Updating existing documents
- Choosing the correct level of safety versus speed for all of these operations

Inserting and Saving Documents

Inserts are the basic method for adding data to MongoDB. To insert a document into a collection, use the collection's `insert` method:

```
> db.foo.insert({"bar" : "baz"})
```

This will add an `"_id"` key to the document (if one does not already exist) and save it to MongoDB.

Batch Insert

If you have a situation where you are inserting multiple documents into a collection, you can make the insert faster by using batch inserts. Batch inserts allow you to pass an array of documents to the database.

Sending dozens, hundreds, or even thousands of documents at a time can make inserts significantly faster. A batch insert is a single TCP request, meaning that you do not incur the overhead of doing hundreds of individual requests. It can also cut insert time by eliminating a lot of the header processing that gets done for each message. When an individual document is sent to the database, it is prefixed by a header that tells the database to do an insert operation on a certain collection. By using batch insert, the database doesn't need to reprocess this information for each document.

Batch inserts are intended to be used in applications, such as for inserting a couple hundred sensor data points into an analytics collection at once. They are useful only if you are inserting multiple documents into a single collection: you cannot use batch inserts to insert into multiple collections with a single request. If you are just importing raw data (for example, from a data feed or MySQL), there are command-line tools like `mongoimport` that can be used instead of batch insert. On the other hand, it is often handy to munge data before saving it to MongoDB (converting dates to the date type or adding a custom `"_id"`) so batch inserts can be used for importing data, as well.

Current versions of MongoDB do not accept messages longer than 16MB, so there is a limit to how much can be inserted in a single batch insert.

Inserts: Internals and Implications

When you perform an insert, the driver you are using converts the data structure into BSON, which it then sends to the database (see Appendix C for more on BSON). The database understands BSON and checks for an `"_id"` key and that the document's size does not exceed 4MB, but other than that, it doesn't do data validation; it just saves the document to the database as is. This has a couple of side effects, most notably that you can insert invalid data and that your database is fairly secure from injection attacks.

All of the drivers for major languages (and most of the minor ones, too) check for a variety of invalid data (documents that are too large, contain non-UTF-8 strings, or use unrecognized types) before sending anything to the database. If you are running a driver that you are not sure about, you can start the database server with the `--objcheck` option, and it will examine each document's structural validity before inserting it (at the cost of slower performance).

> Documents larger than 4MB (when converted to BSON) cannot be saved to the database. This is a somewhat arbitrary limit (and may be raised in the future); it is mostly to prevent bad schema design and ensure consistent performance. To see the BSON size (in bytes) of the document *doc*, run `Object.bsonsize(doc)` from the shell.
>
> To give you an idea of how much 4MB is, the entire text of *War and Peace* is just 3.14MB.

MongoDB does not do any sort of code execution on inserts, so they are not vulnerable to injection attacks. Traditional injection attacks are impossible with MongoDB, and alternative injection-type attacks are easy to guard against in general, but inserts are particularly invulnerable.

Removing Documents

Now that there's data in our database, let's delete it.

```
> db.users.remove()
```

This will remove all of the documents in the *users* collection. This doesn't actually remove the collection, and any indexes created on it will still exist.

The `remove` function optionally takes a query document as a parameter. When it's given, only documents that match the criteria will be removed. Suppose, for instance, that we want to remove everyone from the *mailing.list* collection where the value for `"opt-out"` is `true`:

```
> db.mailing.list.remove({"opt-out" : true})
```

Once data has been removed, it is gone forever. There is no way to undo the remove or recover deleted documents.

Remove Speed

Removing documents is usually a fairly quick operation, but if you want to clear an entire collection, it is faster to *drop* it (and then re-create any indexes).

For example, in Python, suppose we insert a million dummy elements with the following:

```
for i in range(1000000):
    collection.insert({"foo": "bar", "baz": i, "z": 10 - i})
```

Now we'll try to remove all of the documents we just inserted, measuring the time it takes. First, here's a simple `remove`:

```
import time

from pymongo import Connection

db = Connection().foo
collection = db.bar

start = time.time()

collection.remove()
collection.find_one()

total = time.time() - start
print "%d seconds" % total
```

On a MacBook Air, this script prints "46.08 seconds."

If the `remove` and `find_one` are replaced by `db.drop_collection("bar")`, the time drops to .01 seconds! This is obviously a vast improvement, but it comes at the expense of granularity: we cannot specify any criteria. The whole collection is dropped, and all of its indexes are deleted.

Updating Documents

Once a document is stored in the database, it can be changed using the `update` method. `update` takes two parameters: a query document, which locates documents to update, and a modifier document, which describes the changes to make to the documents found.

Updates are atomic: if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied. Thus, conflicting updates can safely be sent in rapid-fire succession without any documents being corrupted: the last update will "win."

Document Replacement

The simplest type of update fully replaces a matching document with a new one. This can be useful to do a dramatic schema migration. For example, suppose we are making major changes to a user document, which looks like the following:

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "name" : "joe",
    "friends" : 32,
    "enemies" : 2
}
```

We want to change that document into the following:

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "username" : "joe",
    "relationships" :
        {
            "friends" : 32,
            "enemies" : 2
        }
}
```

We can make this change by replacing the document using an `update`:

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
{
    "friends" : 32,
    "enemies" : 2
}
```

```
> joe.username = joe.name;
"joe"
> delete joe.friends;
true
> delete joe.enemies;
true
> delete joe.name;
true
> db.users.update({"name" : "joe"}, joe);
```

Now, doing a `findOne` shows that the structure of the document has been updated.

A common mistake is matching more than one document with the criteria and then create a duplicate `"_id"` value with the second parameter. The database will throw an error for this, and nothing will be changed.

For example, suppose we create several documents with the same `"name"`, but we don't realize it:

```
> db.people.find()
{"_id" : ObjectId("4b2b9f67a1f631733d917a7b"), "name" : "joe", "age" : 65},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7c"), "name" : "joe", "age" : 20},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7d"), "name" : "joe", "age" : 49},
```

Now, if it's Joe #2's birthday, we want to increment the value of his `"age"` key, so we might say this:

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),
    "name" : "joe",
    "age" : 20
}
> joe.age++;
> db.people.update({"name" : "joe"}, joe);
E11001 duplicate key on update
```

What happened? When you call update, the database will look for a document matching `{"name" : "joe"}`. The first one it finds will be the 65-year-old Joe. It will attempt to replace that document with the one in the `joe` variable, but there's already a document in this collection with the same `"_id"`. Thus, the update will fail, because `"_id"` values must be unique. The best way to avoid this situation is to make sure that your update always specifies a unique document, perhaps by matching on a key like `"_id"`.

Using Modifiers

Usually only certain portions of a document need to be updated. Partial updates can be done extremely efficiently by using atomic *update modifiers*. Update modifiers are special keys that can be used to specify complex update operations, such as altering, adding, or removing keys, and even manipulating arrays and embedded documents.

Suppose we were keeping website analytics in a collection and wanted to increment a counter each time someone visited a page. We can use update modifiers to do this increment atomically. Each URL and its number of page views is stored in a document that looks like this:

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

Every time someone visits a page, we can find the page by its URL and use the `"$inc"` modifier to increment the value of the `"pageviews"` key.

```
> db.analytics.update({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
```

Now, if we do a `find`, we see that `"pageviews"` has increased by one.

```
> db.analytics.find()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```

> Perl and PHP programmers are probably thinking that any character would have been a better choice than $. Both of these languages use $ as a variable prefix and will replace $-prefixed strings with their variable value in double-quoted strings. However, MongoDB started out as a JavaScript database, and $ is a special character that isn't interpreted differently in JavaScript, so it was used. It is an annoying historical relic from MongoDB's primordial soup.
>
> There are several options for Perl and PHP programmers. First, you could just escape the $: `"\$foo"`. You can use single quotes, which don't do variable interpolation: `'$foo'`. Finally, both drivers allow you to define your own character that will be used instead of $. In Perl, set `$MongoDB::BSON::char`, and in PHP set `mongo.cmd_char` in *php.ini* to =, :, ?, or any other character that you would like to use instead of $. Then, if you choose, say, ~, you would use ~inc instead of \$inc and ~gt instead of \$gt.
>
> Good choices for the special character are characters that will not naturally appear in key names (don't use _ or x) and are not characters that have to be escaped themselves, which will gain you nothing and be confusing (such as \ or, in Perl, @).

When using modifiers, the value of `"_id"` cannot be changed. (Note that `"_id"` can be changed by using whole-document replacement.) Values for any other key, including other uniquely indexed keys, can be modified.

Getting started with the "$set" modifier

`"$set"` sets the value of a key. If the key does not yet exist, it will be created. This can be handy for updating schema or adding user-defined keys. For example, suppose you have a simple user profile stored as a document that looks something like the following:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

This is a pretty bare-bones user profile. If the user wanted to store his favorite book in his profile, he could add it using `"$set"`:

```
> db.users.update({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "war and peace"}})
```

Now the document will have a "favorite book" key:

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin",
    "favorite book" : "war and peace"
}
```

If the user decides that he actually enjoys a different book, `"$set"` can be used again to change the value:

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" : "green eggs and ham"}})
```

`"$set"` can even change the type of the key it modifies. For instance, if our fickle user decides that he actually likes quite a few books, he can change the value of the "favorite book" key into an array:

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" :
...     ["cat's cradle", "foundation trilogy", "ender's game"]}})
```

If the user realizes that he actually doesn't like reading, he can remove the key altogether with `"$unset"`:

```
> db.users.update({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

Now the document will be the same as it was at the beginning of this example.

You can also use `"$set"` to reach in and change embedded documents:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.update({"author.name" : "joe"}, {"$set" : {"author.name" : "joe schmoe"
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe schmoe",
        "email" : "joe@example.com"
    }
}
```

You must always use a $ modifier for adding, changing, or removing keys. A common error people often make when starting out is to try to set the value of `"foo"` to `"bar"` by doing an update that looks like this:

```
> db.coll.update(criteria, {"foo" : "bar"})
```

This will not function as intended. It actually does a full-document replacement, replacing the matched document with `{"foo" : "bar"}`. Always use $ operators for modifying individual key/value pairs.

Incrementing and decrementing

The `"$inc"` modifier can be used to change the value for an existing key or to create a new key if it does not already exist. It is very useful for updating analytics, karma, votes, or anything else that has a changeable, numeric value.

Suppose we are creating a game collection where we want to save games and update scores as they change. When a user starts playing, say, a game of pinball, we can insert a document that identifies the game by name and user playing it:

```
> db.games.insert({"game" : "pinball", "user" : "joe"})
```

When the ball hits a bumper, the game should increment the player's score. As points in pinball are given out pretty freely, let's say that the base unit of points a player can earn is 50. We can use the `"$inc"` modifier to add 50 to the player's score:

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 50}})
```

If we look at the document after this update, we'll see the following:

```
> db.games.findOne()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 50
}
```

The score key did not already exist, so it was created by "$inc" and set to the increment amount: 50.

If the ball lands in a "bonus" slot, we want to add 10,000 to the score. This can be accomplished by passing a different value to "$inc":

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 10000}})
```

Now if we look at the game, we'll see the following:

```
> db.games.find()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 10050
}
```

The "score" key existed and had a numeric value, so the server added 10,000 to it.

"$inc" is similar to "$set", but it is designed for incrementing (and decrementing) numbers. "$inc" can be used only on values of type integer, long, or double. If it is used on any other type of value, it will fail. This includes types that many languages will automatically cast into numbers, like nulls, booleans, or strings of numeric characters:

```
> db.foo.insert({"count" : "1"})
> db.foo.update({}, {$inc : {count : 1}})
Cannot apply $inc modifier to non-number
```

Also, the value of the "$inc" key must be a number. You cannot increment by a string, array, or other non-numeric value. Doing so will give a "Modifier "$inc" allowed for numbers only" error message. To modify other types, use "$set" or one of the array operations described in a moment.

Array modifiers

An extensive class of modifiers exists for manipulating arrays. Arrays are common and powerful data structures: not only are they lists that can be referenced by index, but they can also double as sets.

Array operators can be used only on keys with array values. For example, you cannot push on to an integer or pop off of a string, for example. Use "$set" or "$inc" to modify scalar values.

"$push" adds an element to the end of an array if the specified key already exists and creates a new array if it does not. For example, suppose that we are storing blog posts and want to add a "comments" key containing an array. We can push a comment onto the nonexistent "comments" array, which will create the array and add the comment:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
> db.blog.posts.update({"title" : "A blog post"}, {$push : {"comments" :
... {"name" : "joe", "email" : "joe@example.com", "content" : "nice post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

Now, if we want to add another comment, we can simple use "$push" again:

```
> db.blog.posts.update({"title" : "A blog post"}, {$push : {"comments" :
... {"name" : "bob", "email" : "bob@example.com", "content" : "good post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

A common use is wanting to add a value to an array only if the value is not already present. This can be done using a "$ne" in the query document. For example, to push an author onto a list of citations, but only if he isn't already there, use the following:

```
> db.papers.update({"authors cited" : {"$ne" : "Richie"}},
... {$push : {"authors cited" : "Richie"}})
```

This can also be done with "$addToSet", which is useful for cases where "$ne" won't work or where "$addToSet" describes what is happening better.

For instance, suppose you have a document that represents a user. You might have a set of email addresses that they have added:

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com"
    ]
}
```

When adding another address, you can use "$addToSet" to prevent duplicates:

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@gmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
    ]
}
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@hotmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
    ]
}
```

You can also use "$addToSet" in conjunction with "$each" to add multiple unique values, which cannot be done with the "$ne"/"$push" combination. For instance, we could use these modifiers if the user wanted to add more than one email address:

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")}, {"$addToSet" :
... {"emails" : {"$each" : ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{

    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
        "joe@python.org"
    ]
}
```

There are a few ways to remove elements from an array. If you want to treat the array like a queue or a stack, you can use "$pop", which can remove elements from either end. {$pop : {*key* : 1}} removes an element from the end of the array. {$pop : {*key* : -1}} removes it from the beginning.

Sometimes an element should be removed based on specific criteria, rather than its position in the array. "$pull" is used to remove elements of an array that match the given criteria. For example, suppose we have a list of things that need to be done but not in any specific order:

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

If we do the laundry first, we can remove it from the list with the following:

```
> db.lists.update({}, {"$pull" : {"todo" : "laundry"}})
```

Now if we do a find, we'll see that there are only two elements remaining in the array:

```
> db.lists.find()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "todo" : [
        "dishes",
        "dry cleaning"
    ]
}
```

Pulling removes all matching documents, not just a single match. If you have an array that looks like [1, 1, 2, 1] and pull 1, you'll end up with a single-element array, [2].

Positional array modifications

Array manipulation becomes a little trickier when we have multiple values in an array and want to modify some of them. There are two ways to manipulate values in arrays: by position or by using the position operator (the "$" character).

Arrays use 0-based indexing, and elements can be selected as though their index were a document key. For example, suppose we have a document containing an array with a few embedded documents, such as a blog post with comments:

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b329a216cc613d5ee930192"),
```

```
    "content" : "...",
    "comments" : [
        {
            "comment" : "good post",
            "author" : "John",
            "votes" : 0
        },
        {
            "comment" : "i thought it was too short",
            "author" : "Claire",
            "votes" : 3
        },
        {
            "comment" : "free watches",
            "author" : "Alice",
            "votes" : -1
        }
    ]
}
```

If we want to increment the number of votes for the first comment, we can say the following:

```
> db.blog.update({"post" : post_id},
... {"$inc" : {"comments.0.votes" : 1}})
```

In many cases, though, we don't know what index of the array to modify without querying for the document first and examining it. To get around this, MongoDB has a positional operator, `"$"`, that figures out which element of the array the query document matched and updates that element. For example, if we have a user named John who updates his name to Jim, we can replace it in the comments by using the positional operator:

```
db.blog.update({"comments.author" : "John"},
... {"$set" : {"comments.$.author" : "Jim"}})
```

The positional operator updates only the first match. Thus, if John had left more than one comment, his name would be changed only for the first comment he left.

Modifier speed

Some modifiers are faster than others. `$inc` modifies a document in place: it does not have to change the size of a document, only the value of a key, so it is very efficient. On the other hand, array modifiers might change the size of a document and can be slow. (`"$set"` can modify documents in place if the size isn't changing but otherwise is subject to the same performance limitations as array operators.)

MongoDB leaves some padding around a document to allow for changes in size (and, in fact, figures out how much documents usually change in size and adjusts the amount of padding it leaves accordingly), but it will eventually have to allocate new space for a document if you make it much larger than it was originally. Compounding this slowdown, as arrays get longer, it takes MongoDB a longer amount of time to traverse the whole array, slowing down each array modification.

A simple program in Python can demonstrate the speed difference. This program inserts a single key and increments its value 100,000 times.

```
from pymongo import Connection

import time

db = Connection().performance_test
db.drop_collection("updates")
collection = db.updates

collection.insert({"x": 1})

# make sure the insert is complete before we start timing
collection.find_one()

start = time.time()

for i in range(100000):
    collection.update({}, {"$inc" : {"x" : 1}})

# make sure the updates are complete before we stop timing
collection.find_one()

print time.time() - start
```

On a MacBook Air this took 7.33 seconds. That's more than 13,000 updates per second (which is pretty good for a fairly anemic machine). Now, let's try it with a document with a single array key, pushing new values onto that array 100,000 times:

```
for i in range(100000):
    collection.update({}, {'$push' : {'x' : 1}})
```

This program took 67.58 seconds to run, which is less than 1,500 updates per second.

Using `"$push"` and other array modifiers is encouraged and often necessary, but it is good to keep in mind the trade-offs of such updates. If `"$push"` becomes a bottleneck, it may be worth pulling an embedded array out into a separate collection.

Upserts

An *upsert* is a special type of update. If no document is found that matches the update criteria, a new document will be created by combining the criteria and update documents. If a matching document is found, it will be updated normally. Upserts can be very handy because they eliminate the need to "seed" your collection: you can have the same code create and update documents.

Let's go back to our example recording the number of views for each page of a website. Without an upsert, we might try to find the URL and increment the number of views or create a new document if the URL doesn't exist. If we were to write this out as a JavaScript program (instead of a series of shell commands—scripts can be run with `mongo *scriptname.js*`), it might look something like the following:

```
// check if we have an entry for this page
blog = db.analytics.findOne({url : "/blog"})

// if we do, add one to the number of views and save
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// otherwise, create a new document for this page
else {
  db.analytics.save({url : "/blog", pageviews : 1})
}
```

This means we are making a round-trip to the database, plus sending an update or insert, every time someone visits a page. If we are running this code in multiple processes, we are also subject to a race condition where more than one document can be inserted for a given URL.

We can eliminate the race condition and cut down on the amount of code by just sending an upsert (the third parameter to `update` specifies that this should be an upsert):

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"visits" : 1}}, true)
```

This line does exactly what the previous code block does, except it's faster and atomic! The new document is created using the criteria document as a base and applying any modifier documents to it. For example, if you do an upsert that matches a key and has an increment to the value of that key, the increment will be applied to the match:

```
> db.math.remove()
> db.math.update({"count" : 25}, {"$inc" : {"count" : 3}}, true)
> db.math.findOne()
{
    "_id" : ObjectId("4b3295f26cc613d5ee93018f"),
    "count" : 28
}
```

The `remove` empties the collection, so there are no documents. The upsert creates a new document with a `"count"` of 25 and then increments that by 3, giving us a document where `"count"` is 28. If the upsert option were not specified, `{"count" : 25}` would not match any documents, so nothing would happen.

If we run the upsert again (with the criteria `{count : 25}`), it will create another new document. This is because the criteria does not match the only document in the collection. (Its `"count"` is 28.)

The save Shell Helper

`save` is a shell function that lets you insert a document if it doesn't exist and update it if it does. It takes one argument: a document. If the document contains an `"_id"` key, `save` will do an upsert. Otherwise, it will do an insert. This is just a convenience function so that programmers can quickly modify documents in the shell:

```
> var x = db.foo.findOne()
> x.num = 42
42
> db.foo.save(x)
```

Without `save`, the last line would have been a more cumbersome `db.foo.update({"_id" : x._id}, x)`.

Updating Multiple Documents

Updates, by default, update only the first document found that matches the criteria. If there are more matching documents, they will remain unchanged. To modify all of the documents matching the criteria, you can pass `true` as the fourth parameter to `update`.

> `update`'s behavior may be changed in the future (the server may update all matching documents by default and update one only if `false` is passed as the fourth parameter), so it is recommended that you always specify whether you want a multiple update.
>
> Not only is it more obvious what the update should be doing, but your program won't break if the default is ever changed.

Multiupdates are a great way of performing schema migrations or rolling out new features to certain users. Suppose, for example, we want to give a gift to every user who has a birthday on a certain day. We can use multiupdate to add a `"gift"` to their account:

```
> db.users.update({birthday : "10/13/1978"},
... {$set : {gift : "Happy Birthday!"}}, false, true)
```

This would add the `"gift"` key to all user documents with birthdays on October 13, 1978.

To see the number of documents updated by a multiple update, you can run the `getLastError` database command (which might be better named `"getLastOpStatus"`). The `"n"` key will contain the number of documents affected by the update:

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

`"n"` is `5`, meaning that five documents were affected by the update. `"updatedExisting"` is `true`, meaning that the update modified existing document(s). For more on database commands and their responses, see Chapter 7.

Returning Updated Documents

You can get some limited information about what was updated by calling `getLastError`, but it does not actually return the updated document. For that, you'll need the `findAndModify` command.

`findAndModify` is called differently than a normal update and is a bit slower, because it must wait for a database response. It is handy for manipulating queues and performing other operations that need get-and-set style atomicity.

Suppose we have a collection of processes run in a certain order. Each is represented with a document that has the following form:

```
{
    "_id" : ObjectId(),
    "status" : state,
    "priority" : N
}
```

`"status"` is a string that can be "READY," "RUNNING," or "DONE." We need to find the job with the highest priority in the "READY" state, run the process function, and then update the status to "DONE." We might try querying for the ready processes, sorting by priority, and updating the status of the highest-priority process to mark it is "RUNNING." Once we have processed it, we update the status to "DONE." This looks something like the following:

```
ps = db.processes.find({"status" : "READY").sort({"priority" : -1}).limit(1).next()
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "RUNNING"}})
do_something(ps);
db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

This algorithm isn't great, because it is subject to a race condition. Suppose we have two threads running. If one thread (call it A) retrieved the document and another thread (call it B) retrieved the same document before A had updated its status to "RUNNING," then both threads would be running the same process. We can avoid this by checking the status as part of the update query, but this becomes complex:

```
var cursor = db.processes.find({"status" : "READY"}).sort({"priority" : -1}).limit(1);
while ((ps = cursor.next()) != null) {
    ps.update({"_id" : ps._id, "status" : "READY"},
              {"$set" : {"status" : "RUNNING"}});

    var lastOp = db.runCommand({getlasterror : 1});
    if (lastOp.n == 1) {
        do_something(ps);
        db.processes.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
        break;
    }
    cursor =  db.processes.find({"status" : "READY"}).sort({"priority" : -1}).limit(1);
}
```

Also, depending on timing, one thread may end up doing all the work while another thread is uselessly trailing it. Thread A could always grab the process, and then B would try to get the same process, fail, and leave A to do all the work. Situations like this are perfect for `findAndModify`. `findAndModify` can return the item and update it in a single operation. In this case, it looks like the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}})
{
    "ok" : 1,
    "value" : {
        "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
        "priority" : 1,
        "status" : "READY"
    }
}
```

Notice that the status is still "READY" in the returned document. The document is returned before the modifier document is applied. If you do a find on the collection, though, you will see that the document's `"status"` has been updated to "RUNNING":

```
> db.processes.findOne({"_id" : ps.value._id})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "RUNNING"
}
```

Thus, the program becomes the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}}).value
> do_something(ps)
> db.process.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

`findAndModify` can have either an `"update"` key or a `"remove"` key. A `"remove"` key indicates that the matching document should be removed from the collection. For instance, if we wanted to simply remove the job instead of updating its status, we could run the following:

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "remove" : true).value
> do_something(ps)
```

The values for each key in the `findAndModify` command are as follows:

`findAndModify`
: A string, the collection name.

`query`
: A query document, the criteria with which to search for documents.

`sort`
: Criteria by which to sort results.

`update`
: A modifier document, the update to perform on the document found.

`remove`
: Boolean specifying whether the document should be removed.

`new`
: Boolean specifying whether the document returned should be the updated document or the preupdate document. Defaults to the preupdate document.

Either `"update"` or `"remove"` must be included, but not both. If no matching document is found, the command will return an error.

`findAndModify` has a few limitations. First, it can update or remove only one document at a time. There is also no way to use it for an upsert; it can update only existing documents.

The price of using `findAndModify` over a traditional update is speed: it is a bit slower. That said, it is no slower than one might expect: it takes roughly the same amount of time as a `find`, `update`, and `getLastError` command performed in serial.

The Fastest Write This Side of Mississippi

The three operations that this chapter focused on (inserts, removes, and updates) seem instantaneous because none of them waits for a database response. They are not asynchronous; they can be thought of as "fire-and-forget" functions: the client sends the documents to the server and immediately continues. The client never receives an "OK, got that" or a "not OK, could you send that again?" response.

The benefit to this is that the speed at which you can perform these operations is terrific. You are often only limited by the speed at which your client can send them and the speed of your network. This works well most of the time; however, sometimes something goes wrong: a server crashes, a rat chews through a network cable, or a data center is in a flood zone. If the server disappears, the client will happily send some writes to a server that isn't there, entirely unaware of its absence. For some applications, this is acceptable. Losing a couple of seconds of log messages, user clicks, or analytics in a hardware failure is not the end of the world. For others, this is not the behavior the programmer wants (payment-processing systems spring to mind).

Safe Operations

Suppose you are writing an ecommerce application. If someone orders something, the application should probably take a little extra time to make sure the order goes through. That is why you can do a "safe" version of these operations, where you check whether there was an error in execution and attempt to redo them.

> MongoDB developers made unchecked operations the default because of their experience with relational databases. Many applications written on top of relational databases do not care about or check the return codes, yet they incur the performance penalty of their application waiting for them to arrive. MongoDB pushes this option to the user. This way, programs that collect log messages or real-time analytics don't have to wait for return codes that they don't care about.

The safe version of these operations runs a `getLastError` command immediately following the operation to check whether it succeeded (see "Database Commands" on page 93 for more on commands). The driver waits for the database response and then handles errors appropriately, throwing a catchable exception in most cases. This way, developers can catch and handle database errors in whatever way feels "natural" for their language. When an operation is successful, the `getLastError` response also contains some additional information (e.g., for an update or remove, it includes the number of documents affected).

> The same `getLastError` command that powers safe mode also contains functionality for checking that operations have been successfully replicated. For more on this feature, see "Blocking for Replication" on page 140.

The price of performing "safe" operations is performance: waiting for a database response takes an order of magnitude longer than sending the message, ignoring the client-side cost of handling exceptions. (This cost varies by language but is usually fairly heavyweight.) Thus, applications should weigh the importance of their data (and the consequences if some of it is lost) versus the speed needed.

> When in doubt, use safe operations. If they aren't fast enough, start making less important operations fire-and-forget.

More specifically:

- If you live dangerously, use fire-and-forget operations exclusively.

- If you want to live longer, save valuable user input (account sign-ups, credit card numbers, emails) with safe operations and do everything else with fire-and-forget operations.
- If you are cautious, use safe operations exclusively. If your application is automatically generating hundreds of little pieces of information to save (e.g., page, user, or advertising statistics), these can still use the fire-and-forget operation.

Catching "Normal" Errors

Safe operations are also a good way to debug "strange" database behavior, not just for preventing the apocalyptic scenarios described earlier. Safe operations should be used extensively while developing, even if they are later removed before going into production. They can protect against many common database usage errors, most commonly duplicate key errors.

Duplicate key errors often occur when users try to insert a document with a duplicate value for the `"_id"` key. MongoDB does not allow multiple documents with the same `"_id"` in the same collection. If you do a safe insert and a duplicate key error occurs, the server error will be picked up by the safety check, and an exception will be thrown. In unsafe mode, there is no database response, and you might not be aware that the insert failed.

For example, using the shell, you can see that inserting two documents with the same `"_id"` will not work:

```
> db.foo.insert({"_id" : 123, "x" : 1})
> db.foo.insert({"_id" : 123, "x" : 2})
E11000 duplicate key error index: test.foo.$_id_  dup key: { : 123.0 }
```

If we examine the collection, we can see that only the first document was successfully inserted. Note that this error can occur with any unique index, not just the one on `"_id"`. The shell always checks for errors; in the drivers it is optional.

Requests and Connections

For each connection to a MongoDB server, the database creates a queue for that connection's requests. When the client sends a request, it will be placed at the end of its connection's queue. Any subsequent requests on the connection will occur after the enqueued operation is processed. Thus, a single connection has a consistent view of the database and can always read its own writes.

Note that this is a per-connection queue: if we open two shells, we will have two connections to the database. If we perform an insert in one shell, a subsequent query in the other shell might not return the inserted document. However, within a single shell, if we query for the document after inserting, the document will be returned. This behavior can be difficult to duplicate by hand, but on a busy server, interleaved inserts/queries are very likely to occur. Often developers run into this when they insert data in one thread and then check that it was successfully inserted in another. For a second or two, it looks like the data was not inserted, and then it suddenly appears.

This behavior is especially worth keeping in mind when using the Ruby, Python, and Java drivers, because all three drivers use connection pooling. For efficiency, these drivers open multiple connections (a *pool*) to the server and distribute requests across them. They all, however, have mechanisms to guarantee that a series of requests is processed by a single connection. There is detailed documentation on connection pooling in various languages on the MongoDB wiki.

CHAPTER 4
Querying

This chapter looks at querying in detail. The main areas covered are as follows:

- You can perform ad hoc queries on the database using the `find` or `findOne` functions and a query document.
- You can query for ranges, set inclusion, inequalities, and more by using `$` conditionals.
- Some queries cannot be expressed as query documents, even using $ conditionals. For these types of complex queries, you can use a `$where` clause to harness the full expressive power of JavaScript.
- Queries return a database cursor, which lazily returns batches of documents as you need them.
- There are a lot of metaoperations you can perform on a cursor, including skipping a certain number of results, limiting the number of results returned, and sorting results.

Introduction to find

The `find` method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection, from no documents at all to the entire collection. Which documents get returned is determined by the first argument to `find`, which is a document specifying the query to be performed.

An empty query document (i.e., `{}`) matches everything in the collection. If `find` isn't given a query document, it defaults to `{}`. For example, the following:

```
> db.c.find()
```

returns everything in the collection *c*.

When we start adding key/value pairs to the query document, we begin restricting our search. This works in a straightforward way for most types. Integers match integers, booleans match booleans, and strings match strings. Querying for a simple type is as easy as specifying the value that you are looking for. For example, to find all documents where the value for `"age"` is 27, we can add that key/value pair to the query document:

```
> db.users.find({"age" : 27})
```

If we have a string we want to match, such as a `"username"` key with the value `"joe"`, we use that key/value pair instead:

```
> db.users.find({"username" : "joe"})
```

Multiple conditions can be strung together by adding more key/value pairs to the query document, which gets interpreted as "condition1 AND condition2 AND ... AND conditionN." For instance, to get all users who are 27-year-olds with the username "joe," we can query for the following:

```
> db.users.find({"username" : "joe", "age" : 27})
```

Specifying Which Keys to Return

Sometimes, you do not need all of the key/value pairs in a document returned. If this is the case, you can pass a second argument to `find` (or `findOne`) specifying the keys you want. This reduces both the amount of data sent over the wire and the time and memory used to decode documents on the client side.

For example, if you have a user collection and you are interested only in the `"username"` and `"email"` keys, you could return just those keys with the following query:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

As you can see from the previous output, the `"_id"` key is always returned, even if it isn't specifically listed.

You can also use this second parameter to exclude specific key/value pairs from the results of a query. For instance, you may have documents with a variety of keys, and the only thing you know is that you never want to return the `"fatal_weakness"` key:

```
> db.users.find({}, {"fatal_weakness" : 0})
```

This can even prevent `"_id"` from being returned:

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

Limitations

There are some restrictions on queries. The value of a query document must be a constant as far as the database is concerned. (It can be a normal variable in your own code.) That is, it cannot refer to the value of another key in the document. For example, if we were keeping inventory and we had both `"in_stock"` and `"num_sold"` keys, we could compare their values by querying the following:

```
> db.stock.find({"in_stock" : "this.num_sold"}) // doesn't work
```

There are ways to do this (see "$where Queries" on page 55), but you will usually get better performance by restructuring your document slightly, such that a normal query will suffice. In this example, we could instead use the keys `"initial_stock"` and `"in_stock"`. Then, every time someone buys an item, we decrement the value of the `"in_stock"` key by one. Finally, we can do a simple query to check which items are out of stock:

```
> db.stock.find({"in_stock" : 0})
```

Query Criteria

Queries can go beyond the exact matching described in the previous section; they can match more complex criteria, such as ranges, OR-clauses, and negation.

Query Conditionals

`"$lt"`, `"$lte"`, `"$gt"`, and `"$gte"` are all comparison operators, corresponding to <, <=, >, and >=, respectively. They can be combined to look for a range of values. For example, to look for users who are between the ages of 18 and 30 inclusive, we can do this:

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

These types of range queries are often useful for dates. For example, to find people who registered before January 1, 2007, we can do this:

```
> start = new Date("01/01/2007")
> db.users.find({"registered" : {"$lt" : start}})
```

An exact match on a date is less useful, because dates are only stored with millisecond precision. Often you want a whole day, week, or month, making a range query necessary.

To query for documents where a key's value is not equal to a certain value, you must use another conditional operator, `"$ne"`, which stands for "not equal." If you want to find all users who do not have the username "joe," you can query for them using this:

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

`"$ne"` can be used with any type.

OR Queries

There are two ways to do an OR query in MongoDB. `"$in"` can be used to query for a variety of values for a single key. `"$or"` is more general; it can be used to query for any of the given values across multiple keys.

If you have more than one possible value to match for a single key, use an array of criteria with `"$in"`. For instance, suppose we were running a raffle and the winning ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

`"$in"` is very flexible and allows you to specify criteria of different types as well as values. For example, if we are gradually migrating our schema to use usernames instead of user ID numbers, we can query for either by using this:

```
> db.users.find({"user_id" : {"$in" : [12345, "joe"]})
```

This matches documents with a `"user_id"` equal to 12345, and documents with a `"user_id"` equal to `"joe"`.

If `"$in"` is given an array with a single value, it behaves the same as directly matching the value. For instance, `{ticket_no : {$in : [725]}}` matches the same documents as `{ticket_no : 725}`.

The opposite of `"$in"` is `"$nin"`, which returns documents that don't match any of the criteria in the array. If we want to return all of the people who didn't win anything in the raffle, we can query for them with this:

```
> db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
```

This query returns everyone who did not have tickets with those numbers.

`"$in"` gives you an OR query for a single key, but what if we need to find documents where `"ticket_no"` is 725 or `"winner"` is `true`? For this type of query, we'll need to use the `"$or"` conditional. `"$or"` takes an array of possible criteria. In the raffle case, using `"$or"` would look like this:

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

`"$or"` can contain other conditionals. If, for example, we want to match any of the three `"ticket_no"` values or the `"winner"` key, we can use this:

```
> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}},
                           {"winner" : true}]})
```

With a normal AND-type query, you want to narrow your results down as far as possible in as few arguments as possible. OR-type queries are the opposite: they are most efficient if the first arguments match as many documents as possible.

$not

`"$not"` is a metaconditional: it can be applied on top of any other criteria. As an example, let's consider the modulus operator, `"$mod"`. `"$mod"` queries for keys whose values, when divided by the first value given, have a remainder of the second value:

```
> db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

The previous query returns users with `"id_num"`s of 1, 6, 11, 16, and so on. If we want, instead, to return users with `"id_num"`s of 2, 3, 4, 5, 7, 8, 9, 10, 12, and so on, we can use `"$not"`:

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

`"$not"` can be particularly useful in conjunction with regular expressions to find all documents that don't match a given pattern (regular expression usage is described in the section "Regular Expressions" on page 50).

Rules for Conditionals

If you look at the update modifiers in the previous chapter and previous query documents, you'll notice that the $-prefixed keys are in different positions. In the query, `"$lt"` is in the inner document; in the update, `"$inc"` is the key for the outer document. This generally holds true: conditionals are an inner document key, and modifiers are always a key in the outer document.

Multiple conditions can be put on a single key. For example, to find all users between the ages of 20 and 30, we can query for both `"$gt"` and `"$lt"` on the `"age"` key:

```
> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

Any number of conditionals can be used with a single key. Multiple update modifiers *cannot* be used on a single key, however. For example, you cannot have a modifier document such as `{"$inc" : {"age" : 1}, "$set" : {age : 40}}` because it modifies `"age"` twice. With query conditionals, no such rule applies.

Type-Specific Queries

As covered in Chapter 2, MongoDB has a wide variety of types that can be used in a document. Some of these behave specially in queries.

null

`null` behaves a bit strangely. It does match itself, so if we have a collection with the following documents:

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

we can query for documents whose `"y"` key is `null` in the expected way:

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

However, `null` not only matches itself but also matches "does not exist." Thus, querying for a key with the value `null` will return all documents lacking that key:

```
> db.c.find({"z" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

If we only want to find keys whose value is `null`, we can check that the key is `null` and exists using the `"$exists"` conditional:

```
> db.c.find({"z" : {"$in" : [null], "$exists" : true}})
```

Unfortunately, there is no `"$eq"` operator, which makes this a little awkward, but `"$in"` with one element is equivalent.

Regular Expressions

Regular expressions are useful for flexible string matching. For example, if we want to find all users with the name Joe or joe, we can use a regular expression to do case-insensitive matching:

```
> db.users.find({"name" : /joe/i})
```

Regular expression flags (`i`) are allowed but not required. If we want to match not only various capitalizations of joe, but also joey, we can continue to improve our regular expression:

```
> db.users.find({"name" : /joey?/i})
```

MongoDB uses the Perl Compatible Regular Expression (PCRE) library to match regular expressions; any regular expression syntax allowed by PCRE is allowed in MongoDB. It is a good idea to check your syntax with the JavaScript shell before using it in a query to make sure it matches what you think it matches.

> MongoDB can leverage an index for queries on prefix regular expressions (e.g., `/^joey/`), so queries of that kind can be fast.

Regular expressions can also match themselves. Very few people insert regular expressions into the database, but if you insert one, you can match it with itself:

```
> db.foo.insert({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
```

```
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "bar" : /baz/
}
```

Querying Arrays

Querying for elements of an array is simple. An array can mostly be treated as though each element is the value of the overall key. For example, if the array is a list of fruits, like this:

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

the following query:

```
> db.food.find({"fruit" : "banana"})
```

will successfully match the document. We can query for it in much the same way as though we had a document that looked like the (illegal) document: `{"fruit" : "apple", "fruit" : "banana", "fruit" : "peach"}`.

$all

If you need to match arrays by more than one element, you can use `"$all"`. This allows you to match a list of elements. For example, suppose we created a collection with three elements:

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

Then we can find all documents with both `"apple"` and `"banana"` elements by querying with `"$all"`:

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})
      {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
      {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

Order does not matter. Notice `"banana"` comes before `"apple"` in the second result. Using a one-element array with `"$all"` is equivalent to not using `"$all"`. For instance, `{fruit : {$all : ['apple']}` will match the same documents as `{fruit : 'apple'}`.

You can also query by exact match using the entire array. However, exact match will not match a document if any elements are missing or superfluous. For example, this will match the first document shown previously:

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

But this will not:

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

and neither will this:

```
> db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

If you want to query for a specific element of an array, you can specify an index using the syntax *`key.index`*:

```
> db.food.find({"fruit.2" : "peach"})
```

Arrays are always 0-indexed, so this would match the third array element against the string `"peach"`.

$size

A useful conditional for querying arrays is `"$size"`, which allows you to query for arrays of a given size. Here's an example:

```
> db.food.find({"fruit" : {"$size" : 3}})
```

One common query is to get a range of sizes. `"$size"` cannot be combined with another $ conditional (in this example, `"$gt"`), but this query can be accomplished by adding a `"size"` key to the document. Then, every time you add an element to the array, increment the value of `"size"`. If the original update looked like this:

```
> db.food.update({"$push" : {"fruit" : "strawberry"}})
```

it can simply be changed to this:

```
> db.food.update({"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

Incrementing is extremely fast, so any performance penalty is negligible. Storing documents like this allows you to do queries such as this:

```
> db.food.find({"size" : {"$gt" : 3}})
```

Unfortunately, this technique doesn't work as well with the `"$addToSet"` operator.

The $slice operator

As mentioned earlier in this chapter, the optional second argument to `find` specifies the keys to be returned. The special `"$slice"` operator can be used to return a subset of elements for an array key.

For example, suppose we had a blog post document and we wanted to return the first 10 comments:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

Alternatively, if we wanted the last 10 comments, we could use -10:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

`"$slice"` can also return pages in the middle of the results by taking an offset and the number of elements to return:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

This would skip the first 23 elements and return the 24th through 34th. If there are fewer than 34 elements in the array, it will return as many as possible.

Unless otherwise specified, all keys in a document are returned when `"$slice"` is used. This is unlike the other key specifiers, which suppress unmentioned keys from being returned. For instance, if we had a blog post document that looked like this:

```
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

and we did a `"$slice"` to get the last comment, we'd get this:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -1}})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

Both `"title"` and `"content"` are still returned, even though they weren't explicitly included in the key specifier.

Querying on Embedded Documents

There are two ways of querying for an embedded document: querying for the whole document or querying for its individual key/value pairs.

Querying for an entire embedded document works identically to a normal query. For example, if we have a document that looks like this:

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },


    "age" : 45
}
```

we can query for someone named Joe Schmoe with the following:

```
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

However, if Joe decides to add a middle name key, suddenly this query won't work anymore; it doesn't match the entire embedded document! This type of query is also order-sensitive; `{"last" : "Schmoe", "first" : "Joe"}` would not be a match.

If possible, it's usually a good idea to query for just a specific key or keys of an embedded document. Then, if your schema changes, all of your queries won't suddenly break because they're no longer exact matches. You can query for embedded keys using dot-notation:

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

Now, if Joe adds more keys, this query will still match his first and last names.

This dot-notation is the main difference between query documents and other document types. Query documents can contain dots, which mean "reach into an embedded document." Dot-notation is also the reason that documents to be inserted cannot contain the . character. Oftentimes people run into this limitation when trying to save URLs as keys. One way to get around it is to always perform a global replace before inserting or after retrieving, substituting a character that isn't legal in URLs for the dot (.) character.

Embedded document matches can get a little tricky as the document structure gets more complicated. For example, suppose we are storing blog posts and we want to find comments by Joe that were scored at least a five. We could model the post as follows:

```
> db.blog.find()
{
    "content" : "...",
    "comments" : [
        {
            "author" : "joe",
            "score" : 3,
            "comment" : "nice post"
        },
        {
            "author" : "mary",
            "score" : 6,
            "comment" : "terrible post"
        }
    ]
}
```

Now, we can't query using `db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}})`. Embedded document matches have to match the whole document, and this doesn't match the `"comment"` key. It also wouldn't work to do `db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}})`,

because the author criteria could match a different comment than the score criteria. That is, it would return the document shown earlier; it would match `"author" : "joe"` in the first comment and `"score" : 6` in the second comment.

To correctly group criteria without needing to specify every key, use `"$elemMatch"`. This vaguely named conditional allows you to partially specify criteria to match a single embedded document in an array. The correct query looks like this:

```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe",
                                              "score" : {"$gte" : 5}}}})
```

`"$elemMatch"` allows us to "group" our criteria. As such, it's only needed when you have more than one key you want to match on in an embedded document.

$where Queries

Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are `"$where"` clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query.

The most common case for this is wanting to compare the values for two keys in a document, for instance, if we had a list of items and wanted to return documents where any two of the values are equal. Here's an example:

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

In the second document, `"spinach"` and `"watermelon"` have the same value, so we'd like that document returned. It's unlikely MongoDB will ever have a $ conditional for this, so we can use a `"$where"` clause to do it with JavaScript:

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...             return true;
...         }
...     }
... }
... return false;
... }});
```

If the function returns `true`, the document will be part of the result set; if it returns `false`, it won't be.

We used a function earlier, but you can also use strings to specify a `"$where"` query; the following two `"$where"` queries are equivalent:

```
> db.foo.find({"$where" : "this.x + this.y == 10"})
> db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})
```

`"$where"` queries should not be used unless strictly necessary: they are much slower than regular queries. Each document has to be converted from BSON to a JavaScript object and then run through the `"$where"` expression. Indexes cannot be used to satisfy a `"$where"`, either. Hence, you should use `"$where"` only when there is no other way of doing the query. You can cut down on the penalty by using other query filters in combination with `"$where"`. If possible, an index will be used to filter based on the non-`$where` clauses; the `"$where"` expression will be used only to fine-tune the results.

Another way of doing complex queries is to use MapReduce, which is covered in the next chapter.

Cursors

The database returns results from `find` using a *cursor*. The client-side implementations of cursors generally allow you to control a great deal about the eventual output of a query. You can limit the number of results, skip over some number of results, sort results by any combination of keys in any direction, and perform a number of other powerful operations.

To create a cursor with the shell, put some documents into a collection, do a query on them, and assign the results to a local variable (variables defined with `"var"` are local). Here, we create a very simple collection and query it, storing the results in the `cursor` variable:

```
> for(i=0; i<100; i++) {
... db.c.insert({x : i});
... }
> var cursor = db.collection.find();
```

The advantage of doing this is that you can look at one result at a time. If you store the results in a global variable or no variable at all, the MongoDB shell will automatically iterate through and display the first couple of documents. This is what we've been seeing up until this point, and it is often the behavior you want for seeing what's in a collection but not for doing actual programming with the shell.

To iterate through the results, you can use the `next` method on the cursor. You can use `hasNext` to check whether there is another result. A typical loop through results looks like the following:

```
> while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
```

`cursor.hasNext()` checks that the next result exists, and `cursor.next()` fetches it.

The `cursor` class also implements the iterator interface, so you can use it in a `forEach` loop:

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
... print(x.name);
... });
adam
matt
zak
```

When you call `find`, the shell does not query the database immediately. It waits until you actually start requesting results to send the query, which allows you to chain additional options onto a query before it is performed. Almost every method on a cursor object returns the cursor itself so that you can chain them in any order. For instance, all of the following are equivalent:

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

At this point, the query has not been executed yet. All of these functions merely build the query. Now, suppose we call the following:

```
> cursor.hasNext()
```

At this point, the query will be sent to the server. The shell fetches the first 100 results or first 4MB of results (whichever is smaller) at once so that the next calls to `next` or `hasNext` will not have to make trips to the server. After the client has run through the first set of results, the shell will again contact the database and ask for more results. This process continues until the cursor is exhausted and all results have been returned.

Limits, Skips, and Sorts

The most common query options are limiting the number of results returned, skipping a number of results, and sorting. All of these options must be added before a query is sent to the database.

To set a limit, chain the `limit` function onto your call to `find`. For example, to only return three results, use this:

```
> db.c.find().limit(3)
```

If there are fewer than three documents matching your query in the collection, only the number of matching documents will be returned; `limit` sets an upper limit, not a lower limit.

`skip` works similarly to `limit`:

```
> db.c.find().skip(3)
```

This will skip the first three matching documents and return the rest of the matches. If there are less than three documents in your collection, it will not return any documents.

`sort` takes an object: a set of key/value pairs where the keys are key names and the values are the sort directions. Sort direction can be 1 (ascending) or -1 (descending). If multiple keys are given, the results will be sorted in that order. For instance, to sort the results by `"username"` ascending and `"age"` descending, we do the following:

```
> db.c.find().sort({username : 1, age : -1})
```

These three methods can be combined. This is often handy for pagination. For example, suppose that you are running an online store and someone searches for *mp3*. If you want 50 results per page sorted by price from high to low, you can do the following:

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

If they click Next Page to see more results, you can simply add a skip to the query, which will skip over the first 50 matches (which the user already saw on page 1):

```
> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

However, large skips are not very performant, so there are suggestions on avoiding them in a moment.

Comparison order

MongoDB has a hierarchy as to how types compare. Sometimes you will have a single key with multiple types, for instance, integers and booleans, or strings and nulls. If you do a sort on a key with a mix of types, there is a predefined order that they will be sorted in. From least to greatest value, this ordering is as follows:

1. Minimum value
2. `null`
3. Numbers (integers, longs, doubles)
4. Strings
5. Object/document
6. Array
7. Binary data
8. Object ID
9. Boolean
10. Date
11. Timestamp
12. Regular expression
13. Maximum value

Avoiding Large Skips

Using `skip` for a small number of documents is fine. For a large number of results, `skip` can be slow (this is true in nearly every database, not just MongoDB) and should be avoided. Usually you can build criteria into the documents themselves to avoid

having to do large skips, or you can calculate the next query based on the result from the previous one.

Paginating results without skip

The easiest way to do pagination is to return the first page of results using limit and then return each subsequent page as an offset from the beginning.

```
> // do not use: slow for large skips
> var page1 = db.foo.find(criteria).limit(100)
> var page2 = db.foo.find(criteria).skip(100).limit(100)
> var page3 = db.foo.find(criteria).skip(200).limit(100)
...
```

However, depending on your query, you can usually find a way to paginate without `skip`s. For example, suppose we want to display documents in descending order based on `"date"`. We can get the first page of results with the following:

```
> var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

Then, we can use the `"date"` value of the last document as the criteria for fetching the next page:

```
var latest = null;

// display first page
while (page1.hasNext()) {
   latest = page1.next();
   display(latest);
}

// get next page
var page2 = db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

Now the query does not need to include a `skip`.

Finding a random document

One fairly common problem is how to get a random document from a collection. The naive (and slow) solution is to count the number of documents and then do a `find`, skipping a random number of documents between 0 and the size of the collection.

```
> // do not use
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

It is actually highly inefficient to get a random element this way: you have to do a count (which can be expensive if you are using criteria), and skipping large numbers of elements can be time-consuming.

It takes a little forethought, but if you know you'll be looking up a random element on a collection, there's a much more efficient way to do so. The trick is to add an extra random key to each document when it is inserted. For instance, if we're using the shell, we could use the `Math.random()` function (which creates a random number between 0 and 1):

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
```

Now, when we want to find a random document from the collection, we can calculate a random number and use that as query criteria, instead of doing a `skip`:

```
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
```

There is a slight chance that `random` will be greater than any of the `"random"` values in the collection, and no results will be returned. We can guard against this by simply returning a document in the other direction:

```
> if (result == null) {
... result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

If there aren't any documents in the collection, this technique will end up returning `null`, which makes sense.

This technique can be used with arbitrarily complex queries; just make sure to have an index that includes the random key. For example, if we want to find a random plumber in California, we can create an index on `"profession"`, `"state"`, and `"random"`:

```
> db.people.ensureIndex({"profession" : 1, "state" : 1, "random" : 1})
```

This allows us to quickly find a random result (see Chapter 5 for more information on indexing).

Advanced Query Options

There are two types of queries: *wrapped* and *plain*. A plain query is something like this:

```
> var cursor = db.foo.find({"foo" : "bar"})
```

There are a couple options that "wrap" the query. For example, suppose we perform a sort:

```
> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})
```

Instead of sending `{"foo" : "bar"}` to the database as the query, the query gets wrapped in a larger document. The shell converts the query from `{"foo" : "bar"}` to `{"$query" : {"foo" : "bar"}, "$orderby" : {"x" : 1}}`.

Most drivers provide helpers for adding arbitrary options to queries. Other helpful options include the following:

`$maxscan : integer`
: Specify the maximum number of documents that should be scanned for the query.

`$min : document`
: Start criteria for querying.

`$max : document`
: End criteria for querying.

`$hint : document`
: Tell the server which index to use for the query.

`$explain : boolean`
: Get an explanation of how the query will be executed (indexes used, number of results, how long it takes, etc.), instead of actually doing the query.

`$snapshot : boolean`
: Ensure that the query's results will be a consistent snapshot from the point in time when the query was executed. See the next section for details.

Getting Consistent Results

A fairly common way of processing data is to pull it out of MongoDB, change it in some way, and then save it again:

```
cursor = db.foo.find();

while (cursor.hasNext()) {
    var doc = cursor.next();
    doc = process(doc);
    db.foo.save(doc);
}
```

This is fine for a small number of results, but it breaks down for large numbers of documents. To see why, imagine how the documents are actually being stored. You can picture a collection as a list of documents that looks something like Figure 4-1. Snowflakes represent documents, because every document is beautiful and unique.

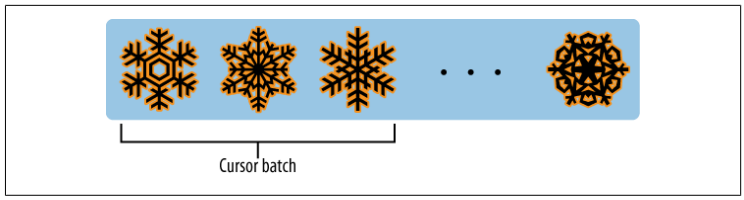


Figure 4-1. A collection being queried

Now, when we do a `find`, it starts returning results from the beginning of the collection and moves right. Your program grabs the first 100 documents and processes them. When you save them back to the database, if a document does not have the padding available to grow to its new size, like in Figure 4-2, it needs to be relocated. Usually, a document will be relocated to the end of a collection (Figure 4-3).

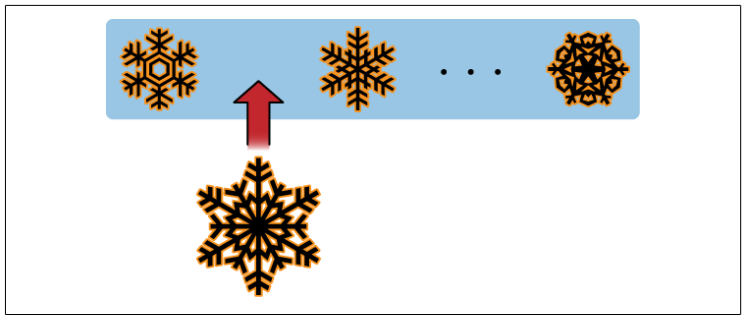


Figure 4-2. An enlarged document may not fit where it did before

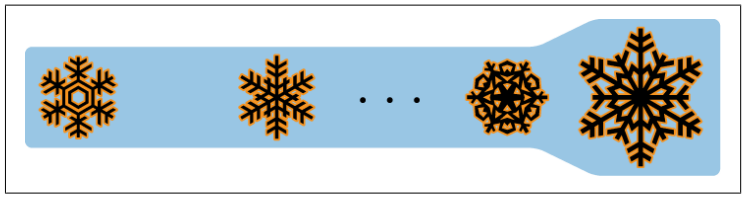


Figure 4-3. MongoDB relocates updated documents that don't fit in their original position

Now our program continues to fetch batches of documents. When it gets toward the end, it will return the relocated documents again (Figure 4-4)!

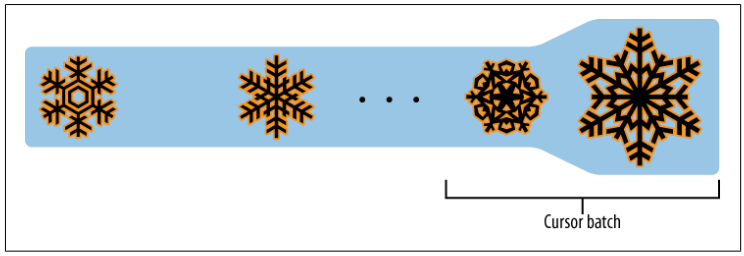


Figure 4-4. A cursor may return these relocated documents again in a later batch

The solution to this problem is to *snapshot* your query. If you add the `"$snapshot"` option, the query will be run against an unchanging view of the collection. All queries that return a single batch of results are effectively snapshotted. Inconsistencies arise

only when the collection changes under a cursor while it is waiting to get another batch of results.

Cursor Internals

There are two sides to a cursor: the client-facing cursor and the database cursor that the client-side one represents. We have been talking about the client-side one up until now, but we are going to take a brief look at what's happening on the server side.

On the server side, a cursor takes up memory and resources. Once a cursor runs out of results or the client sends a message telling it to die, the database can free the resources it was using. Freeing these resources lets the database use them for other things, which is good, so we want to make sure that cursors can be freed quickly (within reason).

There are a couple of conditions that can cause the death (and subsequent cleanup) of a cursor. First, when a cursor finishes iterating through the matching results, it will clean itself up. Another way is that, when a cursor goes out of scope on the client side, the drivers send the database a special message to let it know that it can kill that cursor. Finally, even if the user hasn't iterated through all the results and the cursor is still in scope, after 10 minutes of inactivity, a database cursor will automatically "die."

This "death by timeout" is usually the desired behavior: very few applications expect their users to sit around for minutes at a time waiting for results. However, sometimes you might know that you need a cursor to last for a long time. In that case, many drivers have implemented a function called `immortal`, or a similar mechanism, which tells the database not to time out the cursor. If you turn off a cursor's timeout, you must iterate through all of its results or make sure it gets closed. Otherwise, it will sit around in the database hogging resources forever.

CHAPTER 5
Indexing

Indexes are the way to make queries go *vroom*. Database indexes are similar to a book's index: instead of looking through the whole book, the database takes a shortcut and just looks in the index, allowing it to do queries orders of magnitude faster. Once it finds the entry in the index, it can jump right to the location of the desired document.

Extending this metaphor to the breaking point, creating database indexes is like deciding how a book's index will be organized. You have the advantage of knowing what kinds of queries you'll be doing and thus what types of information the database will need to find quickly. If all of your queries involve the `"date"` key, for example, you probably need an index on `"date"` (at least). If you will be querying for usernames, you don't need to index the `"user_num"` key, because you aren't querying on it.

Introduction to Indexing

It can be tricky to figure out what the optimal index for your queries is, but it is well worth the effort. Queries that otherwise take minutes can be instantaneous with the proper indexes.

> MongoDB's indexes work almost identically to typical relational database indexes, so if you are familiar with those, you can skim this section for syntax specifics. We'll go over some indexing basics, but keep in mind that it's an extensive topic and most of the material out there on index optimization for MySQL/Oracle/SQLite will apply equally well to MongoDB.

Suppose that you are querying for a single key, such as the following:

```
> db.people.find({"username" : "mark"})
```

When only a single key is used in the query, that key can be indexed to improve the query's speed. In this case, you would create an index on `"username"`. To create the index, use the `ensureIndex` method:

```
> db.people.ensureIndex({"username" : 1})
```

An index needs to be created only once for a collection. If you try to create the same index again, nothing will happen.

An index on a key will make queries on that key fast. However, it may not make other queries fast, even if they contain the indexed key. For example, this wouldn't be very performant with the previous index:

```
> db.people.find({"date" : date1}).sort({"date" : 1, "username" : 1})
```

The server has to "look through the whole book" to find the desired dates. This process is called a *table scan*, which is basically what you'd do if you were looking for information in a book without an index: you start at page 1 and read through the whole thing. In general, you want to avoid making the server do table scans, because it is very slow for large collections.

As a rule of thumb, you should create an index that contains all of the keys in your query. For instance, to optimize the previous query, you should have an index on `date` and `username`:

```
> db.ensureIndex({"date" : 1, "username" : 1})
```

The document you pass to `ensureIndex` is of the same form as the document passed to the `sort` function: a set of keys with value 1 or -1, depending on the direction you want the index to go. If you have only a single key in the index, direction is irrelevant. A single key index is analogous to a book's index that is organized in alphabetical order: whether it goes from A–Z or Z–A, it's going to be fairly obvious where to find entries starting with *M*.

If you have more than one key, you need to start thinking about index direction. For example, suppose we have a collection of users:

```
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
```

Let's say we index them with `{"username" : 1, "age" : -1}`. MongoDB will organize the users as follows:

```
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }

{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
```

The usernames are in strictly increasing alphabetical order, and within each name group the ages are in decreasing order. This optimizes sorting by `{"username" : 1, "age" : -1}` but is less efficient at sorting by `{"username" : 1, "age" : 1}`. If we wanted to optimize `{"username" : 1, "age" : 1}`, we would create an index on `{"username" : 1, "age" : 1}` to organize ages in ascending order.

The index on `username` and `age` also makes queries on `username` fast. In general, if an index has N keys, it will make queries on any prefix of those keys fast. For instance, if we have an index that looks like `{"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}`, we effectively have an index on `{"a" : 1}`, `{"a" : 1, "b" : 1}`, `{"a" : 1, "b" : 1, "c" : 1}`, and so on. Queries that would use the index `{"b" : 1}`, `{"a" : 1, "c" :1}`, and so on will not be optimized: only queries that can use a prefix of the index can take advantage of it.

The MongoDB query optimizer will reorder query terms to take advantage of indexes: if you query for `{"x" : "foo", "y" : "bar"}` and you have an index on `{"y" : 1, "x" : 1}`, MongoDB will figure it out.

The disadvantage to creating an index is that it puts a little bit of overhead on every insert, update, and remove. This is because the database not only needs to do the operation but also needs to make a note of it in any indexes on the collection. Thus, the absolute minimum number of indexes should be created. There is a built-in maximum of 64 indexes per collection, which is more than almost any application should need.

> Do not index every key. This will make inserts slow, take up lots of space, and probably not speed up your queries very much. Figure out what queries you are running, what the best indexes are for these queries, and make sure that the server is using the indexes you've created using the `explain` and `hint` tools described in the next section.

Sometimes the most efficient solution is actually not to use an index. In general, if a query is returning a half or more of the collection, it will be more efficient for the database to just do a table scan instead of having to look up the index and then the value for almost every single document. Thus, for queries such as checking whether a key exists or determining whether a boolean value is true or false, it may actually be better to not use an index at all.

Scaling Indexes

Suppose we have a collection of status messages from users. We want to query by user and date to pull up all of a user's recent statuses. Using what we've learned so far, we might create an index that looks like the following:

```
> db.status.ensureIndex({user : 1, date : -1})
```

This will make the query for user and date efficient, but it is not actually the best index choice.

Imagine this as a book index again. We would have a list of documents sorted by user and then subsorted by date, so it would look something like the following:

```
User 123 on March 13, 2010
User 123 on March 12, 2010
User 123 on March 11, 2010
User 123 on March 5, 2010
User 123 on March 4, 2010
User 124 on March 12, 2010
User 124 on March 11, 2010
...
```

This looks fine at this scale, but imagine if the application has millions of users who have dozens of status updates per day. If the index entries for each user's status messages take up a page's worth of space on disk, then for every "latest statuses" query, the database will have to load a different page into memory. This will be very slow if the site becomes popular enough that not all of the index fits into memory.

If we flip the index order to `{date : -1, user : 1}`, the database can keep the last couple days of the index in memory, swap less, and thus query for the latest statuses for any user much more quickly.

Thus, there are several questions to keep in mind when deciding what indexes to create:

1. What are the queries you are doing? Some of these keys will need to be indexed.
2. What is the correct direction for each key?
3. How is this going to scale? Is there a different ordering of keys that would keep more of the frequently used portions of the index in memory?

If you can answer these questions, you are ready to index your data.

Indexing Keys in Embedded Documents

Indexes can be created on keys in embedded documents in the same way that they are created on normal keys. For example, if we want to be able to search blog post comments by date, we can create an index on the `"date"` key in the array of embedded `"comments"` documents:

```
> db.blog.ensureIndex({"comments.date" : 1})
```

Indexes on keys in embedded documents are identical to those on top-level keys, and the two can be combined in compound indexes.

Indexing for Sorts

As your collection grows, you'll need to create indexes for any large sorts your queries are doing. If you call `sort` on a key that is not indexed, MongoDB needs to pull all of that data into memory to sort it. Thus, there's a limit on the amount you can sort without an index: you can't sort a terabyte of data in memory. Once your collection is too big to sort in memory, MongoDB will just return an error for queries that try.

Indexing the sort allows MongoDB to pull the sorted data in order, allowing you to sort any amount of data without running out of memory.

Uniquely Identifying Indexes

Each index in a collection has a string name that uniquely identifies the index and is used by the server to delete or manipulate it. Index names are, by default, *`keyname1_dir1_keyname2_dir2_..._keynameN_dirN`*, where `keynameX` is the index's key and `dirX` is the index's direction (1 or -1). This can get unwieldy if indexes contain more than a couple keys, so you can specify your own name as one of the options to `ensureIndex`:

```
> db.foo.ensureIndex({"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}, {"name" : "alphabet"})
```

There is a limit to the number of characters in an index name, so complex indexes may need custom names to be created. A call to `getLastError` will show whether the index creation succeeded or why it didn't.

Unique Indexes

Unique indexes guarantee that, for a given key, every document in the collection will have a unique value. For example, if you want to make sure no two documents can have the same value in the `"username"` key, you can create a unique index:

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true})
```

Keep in mind that `insert`, by default, does not check whether the document was actually inserted. Therefore, you may want to use safe inserts if you are inserting documents that might have a duplicate value for a unique key. This way, you will get a duplicate key error when you attempt to insert such a document.

A unique index that you are probably already familiar with is the index on `"_id"`, which is created whenever you create a normal collection. This is a normal unique index, aside from the fact that it cannot be deleted.

> If a key does not exist, the index stores its value as `null`. Therefore, if you create a unique index on a key and try to insert more than one document that is missing the indexed key, the inserts will fail because you already have a document with a value of `null`.

Dropping Duplicates

When building unique indexes for an existing collection, some values may be duplicates. If there are any duplicates, this will cause the index building to fail. In some cases, you may just want to drop all of the documents with duplicate values. The `dropDups` option will save the first document found and remove any subsequent documents with duplicate values:

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
```

This is a bit of a drastic option; it might be better to create a script to preprocess the data if it is important.

Compound Unique Indexes

You can also create a compound unique index. If you do this, individual keys can have the same values, but the combination of values for all keys must be unique.

GridFS, the standard method for storing large files in MongoDB (see Chapter 7), uses a compound unique index. The collection that holds the file content has a unique index on `{files_id : 1, n : 1}`, which allows documents that look like (in part) the following:

```
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 2}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 3}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 4}
```

Note that all of the values for `"files_id"` are the same, but `"n"` is different. If you attempted to insert `{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}` again, the database would complain about the duplicate key.

Using explain and hint

`explain` is an incredibly handy tool that will give you lots of information about your queries. You can run it on any query by tacking it on to a cursor. `explain` returns a document, not the cursor itself, unlike most cursor methods:

```
> db.foo.find().explain()
```

`explain` will return information about the indexes used for the query (if any) and stats about timing and the number of documents scanned.

Let's take an example. The index `{"username" : 1}` works well to speed up a simple key/value pair lookup, but many queries are more complicated. For example, suppose you are querying and sorting the following:

```
> db.people.find({"age" : 18}).sort({"username" : 1})
```

You may be unsure if the database is using the index you created or how effective it is. If you run `explain` on the query, it will return the index currently being used for the query, how long it took, and how many documents the database needed to scan to find the results.

For a very simple query ({}) on a database with no indexes (other than the index on `"_id"`) and 64 documents, the output for `explain` looks like this:

```
> db.people.find().explain()
{
    "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 64,
    "nscannedObjects" : 64,
    "n" : 64,
    "millis" : 0,
    "allPlans" : [
        {
            "cursor" : "BasicCursor",
            "indexBounds" : [ ]
        }
    ]
}
```

The important parts of this result are as follows:

`"cursor" : "BasicCursor"`
: This means that the query did not use an index (unsurprisingly, because there was no query criteria). We'll see what this value looks like for an indexed query in a moment.

`"nscanned" : 64`
: This is the number of documents that the database looked through. You want to make sure this is as close to the number returned as possible.

`"n" : 64`
: This is the number of documents returned. We're doing pretty well here, because the number of documents scanned exactly matches the number returned. Of course, given that we're returning the entire collection, it would be difficult to do otherwise.

`"millis" : 0`
: The number of milliseconds it took the database to execute the query. 0 is a good time to shoot for.

Now, let's suppose we have an index on the "age" key and we're querying for users in their 20s. If we run explain on this query, we'll see the following:

```
> db.c.find({age : {$gt : 20, $lt : 30}}).explain()
{
    "cursor" : "BtreeCursor age_1",
    "indexBounds" : [
        [
            {
                "age" : 20
            },
            {
                "age" : 30
            }
        ]
    ],
    "nscanned" : 14,
    "nscannedObjects" : 12,
    "n" : 12,
    "millis" : 1,
    "allPlans" : [
        {
            "cursor" : "BtreeCursor age_1",
            "indexBounds" : [
                [
                    {
                        "age" : 20
                    },
                    {
                        "age" : 30
                    }
                ]
            ]
        }
    ]
}
```

There are some differences here, because this query uses an index. Thus, some of explain's outputted keys have changed:

"cursor" : "BtreeCursor age_1"
: This query is not using a BasicCursor, like the first query was. Indexes are stored in data structures called B-trees, so, when a query uses an index, it uses a special type of cursor called a BtreeCursor.

  This value also identifies the name of the index being used: age_1. Using this name, we could query the *system.indexes* collection to find out more about this index (e.g., whether it is unique or what keys it includes):

```
> db.system.indexes.find({"ns" : "test.c", "name" : "age_1"})
{
    "_id" : ObjectId("4c0d211478b4eaaf7fb28565"),
    "ns" : "test.c",
    "key" : {
        "age" : 1

    },
    "name" : "age_1"
}
```

"allPlans" : [...]
: This key lists all of the plans MongoDB considered for the query. The choice here was fairly obvious, because we had an index on "age" and we were querying for "age". If we had multiple overlapping indexes and a more complex query, "allPlans" would contain information about all of the possible plans that could have been used.

Let's take a slightly more complicated example: suppose you have an index on {"username" : 1, "age" : 1} and an index on {"age" : 1, "username" : 1}. What happens if you query for username and age? Well, it depends on the query:

```
> db.c.find({age : {$gt : 10}, username : "sally"}).explain()
{
        "cursor" : "BtreeCursor username_1_age_1",
        "indexBounds" : [
                [
                        {
                                "username" : "sally",
                                "age" : 10
                        },
                        {
                                "username" : "sally",
                                "age" : 1.7976931348623157e+308
                        }
                ]
        ],
        "nscanned" : 13,
        "nscannedObjects" : 13,
        "n" : 13,
        "millis" : 5,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor username_1_age_1",
                        "indexBounds" : [
                                [
                                        {
                                                "username" : "sally",
                                                "age" : 10
                                        },
                                        {
                                                "username" : "sally",
                                                "age" : 1.7976931348623157e+308
                                        }
                                ]
                        ]
                }
        ],
        "oldPlan" : {
                "cursor" : "BtreeCursor username_1_age_1",
                "indexBounds" : [
                        [
                                {
                                        "username" : "sally",
                                        "age" : 10
                                },
                                {
                                        "username" : "sally",
                                        "age" : 1.7976931348623157e+308
                                }
                        ]
                ]
        }
}
```

We are querying for an exact match on "username" and a range of values for "age", so the database chooses to use the {"username" : 1, "age" : 1} index, reversing the terms of the query. If, on the other hand, we query for an exact age and a range of names, MongoDB will use the other index:

```
> db.c.find({"age" : 14, "username" : /.*/}).explain()
{
        "cursor" : "BtreeCursor age_1_username_1 multi",
        "indexBounds" : [
                [
                        {
                                "age" : 14,
                                "username" : ""
                        },
                        {
                                "age" : 14,
                                "username" : {

                                }
                        }
                ],
                [
                        {
                                "age" : 14,
                                "username" : /.*/
                        },
                        {
                                "age" : 14,
                                "username" : /.*/
                        }
                ]
        ],
        "nscanned" : 2,
        "nscannedObjects" : 2,
        "n" : 2,
        "millis" : 2,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor age_1_username_1 multi",
                        "indexBounds" : [
                                [
                                        {
                                                "age" : 14,
                                                "username" : ""
                                        },
                                        {
                                                "age" : 14,
                                                "username" : {

                                                }
                                        }
                                ],
                                [
                                        {
                                                "age" : 14,
                                                "username" : /.*/
                                        },
                                        {
                                                "age" : 14,
                                                "username" : /.*/
                                        }
                                ]
                        ]
                }
        ]
}
```

If you find that Mongo is using different indexes than you want it to for a query, you can force it to use a certain index by using hint. For instance, if you want to make sure MongoDB uses the {"username" : 1, "age" : 1} index on the previous query, you could say the following:

```
> db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})
```

Hinting is usually unnecessary. MongoDB has a query optimizer and is very clever about choosing which index to use. When you first do a query, the query optimizer tries out a number of query plans concurrently. The first one to finish will be used, and the rest of the query executions are terminated. That query plan will be remembered for future queries on the same keys. The query optimizer periodically retries other plans, in case you've added new data and the previously chosen plan is no longer best. The only part you should need to worry about is giving the query optimizer useful indexes to choose from.

Index Administration

Metainformation about indexes is stored in the *system.indexes* collection of each database. This is a reserved collection, so you cannot insert documents into it or remove documents from it. You can manipulate its documents only through ensureIndex and the dropIndexes database command.

The *system.indexes* collection has detailed information about each index, but the *system.namespaces* collection also lists their names. If you look at this collection, you

can see that there are at least two documents for each collection: one for the collection itself and one for each index it contains. For a collection with just the standard `"_id"` index, its *system.namespaces* entries would look like this:

```
{ "name" : "test.foo" }
{ "name" : "test.foo.$_id_"}
```

If we add a compound index on the name and age keys, a new document is added to *system.namespaces* with the following form:

```
{ "name" : "test.foo.$name_1_age_1" }
```

In Chapter 2, we mentioned that collection names are limited to 121 bytes in length. This somewhat strange limit is because the `"_id"` index's namespace needs 6 extra bytes (`".$_id_"`), bringing the namespace length for that index up to a more sensical 127 bytes.

It is important to remember that the size of the collection name plus the index name cannot exceed 127 bytes. If you approach the maximum namespace size or have large index names, you may have to use custom index names to avoid creating namespaces that are too long. It's generally easier just to keep database, collection, and key names to reasonable lengths.

Changing Indexes

As you and your application grow old together, you may find that your data or queries have changed and that the indexes that used to work well no longer do. You can add new indexes to existing collections at any time with `ensureIndex`:

```
> db.people.ensureIndex({"username" : 1}, {"background" : true})
```

Building indexes is time-consuming and resource-intensive. Using the `{"background" : true}` option builds the index in the background, while handling incoming requests. If you do not include the `background` option, the database will block all other requests while the index is being built.

Blocking lets index creation go faster but means your application will be unresponsive during the build. Even building an index in the background can take a toll on normal operations, so it is best to do it during an "off" time; building indexes in the background will still add extra load to your database, but it will not bring it grinding to a halt.

Creating indexes on existing documents is slightly faster than creating the index first and then inserting all of the documents. Of course, creating an index beforehand is an option only if you're populating a new collection from scratch.

If an index is no longer necessary, you can remove it with the `dropIndexes` command and the index name. Often, you may have to look at the *system.indexes* collection to figure out what the index name is, because even the autogenerated names vary somewhat from driver to driver:

```
> db.runCommand({"dropIndexes" : "foo", "index" : "alphabet"})
```

You can drop all indexes on a collection by passing in `*` as the value for the `index` key:

```
> db.runCommand({"dropIndexes" : "foo", "index" : "*"})
```

Another way to delete all indexes is to drop the collection. This will also delete the index on `_id` (and all of the documents in the collection). Removing all of the documents in a collection (with `remove`) does not affect the indexes; they will be repopulated when new documents are inserted.

Geospatial Indexing

There is another type of query that is becoming increasingly common (especially with the emergence of mobile devices): finding the nearest N things to a current location. MongoDB provides a special type of index for coordinate plane queries, called a *geospatial index*.

Suppose we wanted to find the nearest coffee shop to where we are now, given a latitude and longitude. We need to create a special index to do this type of query efficiently, because it needs to search in two dimensions. A geospatial index can be created using the `ensureIndex` function, but by passing `"2d"` as a value instead of 1 or -1:

```
> db.map.ensureIndex({"gps" : "2d"})
```

The `"gps"` key must be some type of pair value, that is, a two-element array or embedded document with two keys. These would all be valid:

```
{ "gps" : [ 0, 100 ] }
{ "gps" : { "x" : -30, "y" : 30 } }
{ "gps" : { "latitude" : -180, "longitude" : 180 } }
```

The keys can be anything; for example, `{"gps" : {"foo" : 0, "bar" : 1}}` is valid.

By default, geospatial indexing assumes that your values are going to range from -180 to 180 (which is convenient for latitude and longitude). If you are using it for other values, you can specify what the minimum and maximum values will be as options to `ensureIndex`:

```
> db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
```

This will create a spatial index calibrated for a 2,000 × 2,000 light-year square.

Geospatial queries can be performed in two ways: as a normal query (using `find`) or as a database command. The `find` approach is similar to performing a nongeospatial query, except the conditional `"$near"` is used. It takes an array of the two target values:

```
> db.map.find({"gps" : {"$near" : [40, -73]}})
```

This finds all of the documents in the *map* collection, in order by distance from the point (40, -73). A default limit of 100 documents is applied if no limit is specified. If you don't need that many results, you should set a limit to conserve server resources. For example, the following code returns the 10 nearest documents to (40, -73):

```
> db.map.find({"gps" : {"$near" : [40, -73]}}).limit(10)
```

This same type of query can be performed using the `geoNear` command:

```
> db.runCommand({geoNear : "map", near : [40, -73], num : 10});
```

`geoNear` also returns the distance of each result document from the query point. The distance is in whatever units you inserted the data in; if you inserted degrees of longitude and latitude, the distance is in degrees. `find` with `"$near"` doesn't give you the distance for each point but must be used if you have more than 4MB of results.

MongoDB also allows you to find all of the documents within a shape, as well as near a point. To find points in a shape, we use the `"$within"` conditional instead of the `"$near"` conditional. `"$within"` can take an increasing number of shapes; check the online documentation for geospatial indexing to see the most up-to-date list of what's supported (*http://www.mongodb.org/display/DOCS/Geospatial+Indexing*). As of this writing, there are two options: you can query for all points within a rectangle or a circle. To use a rectangle, use the `"$box"` option:

```
> db.map.find({"gps" : {"$within" : {"$box" : [[10, 20], [15, 30]]}}})
```

`"$box"` takes a two-element array: the first element specifies the coordinates of the lower-left corner, the second element the upper right.

Also, you can find all points within a circle with `"$center"`, which takes an array with the center point and then a radius:

```
> db.map.find({"gps" : {"$within" : {"$center" : [[12, 25], 5]}}})
```

Compound Geospatial Indexes

Applications often need to search for more complex criteria than just a location. For example, a user might want to find all coffee shops or pizza parlors near where they are. To facilitate this type of query, you can combine geospatial indexes with normal indexes. In this situation, for instance, we might want to query on both the `"location"` key and a `"desc"` key, so we'd create a compound index:

```
> db.ensureIndex({"location" : "2d", "desc" : 1})
```

Then we can quickly find the nearest coffee shop:

```
> db.map.find({"location" : {"$near" : [-70, 30]}, "desc" : "coffeeshop"}).limit(1)
{
    "_id" : ObjectId("4c0d1348928a815a720a0000"),
    "name" : "Mud",
    "location" : [x, y],
    "desc" : ["coffee", "coffeeshop", "muffins", "espresso"]
}
```

Note that creating an array of keywords is a good way to perform user-defined searches.

The Earth Is Not a 2D Plane

MongoDB's geospatial indexes assumes that whatever you're indexing is a flat plane. This means that results aren't perfect for spherical shapes, like the earth, especially near the poles. The problem is that the distance between lines of longitude in the Yukon is much shorter than the distance between them at the equator. There are various projections that can be used to convert the earth to a 2D plane with varying levels of accuracy and corresponding levels of complexity.

CHAPTER 6

Aggregation

MongoDB provides a number of aggregation tools that go beyond basic query functionality. These range from simply counting the number of documents in a collection to using MapReduce to do complex data analysis.

count

The simplest aggregation tool is `count`, which returns the number of documents in the collection:

```
> db.foo.count()
0
> db.foo.insert({"x" : 1})
> db.foo.count()
1
```

Counting the total number of documents in a collection is fast regardless of collection size.

You can also pass in a query, and Mongo will count the number of results for that query:

```
> db.foo.insert({"x" : 2})
> db.foo.count()
2
> db.foo.count({"x" : 1})
1
```

This can be useful for getting a total for pagination: "displaying results 0–10 of 439." Adding criteria does make the count slower, however.

distinct

The `distinct` command finds all of the distinct values for a given key. You must specify a collection and key:

```
> db.runCommand({"distinct" : "people", "key" : "age"})
```

For example, suppose we had the following documents in our collection:

```
{"name" : "Ada", "age" : 20}
{"name" : "Fred", "age" : 35}
{"name" : "Susan", "age" : 60}
{"name" : "Andy", "age" : 35}
```

If you call distinct on the `"age"` key, you will get back all of the distinct ages:

```
> db.runCommand({"distinct" : "people", "key" : "age"})
{"values" : [20, 35, 60], "ok" : 1}
```

A common question at this point is if there's a way to get all of the distinct *keys* in a collection. There is no built-in way of doing this, although you can write something to do it yourself using MapReduce (described in a moment).

group

`group` allows you to perform more complex aggregation. You choose a key to group by, and MongoDB divides the collection into separate groups for each value of the chosen key. For each group, you can create a result document by aggregating the documents that are members of that group.

> If you are familiar with SQL, `group` is similar to SQL's `GROUP BY`.

Suppose we have a site that keeps track of stock prices. Every few minutes from 10 a.m. to 4 p.m., it gets the latest price for a stock, which it stores in MongoDB. Now, as part of a reporting application, we want to find the closing price for the past 30 days. This can be easily accomplished using `group`.

The collection of stock prices contains thousands of documents with the following form:

```
{"day" : "2010/10/03", "time" : "10/3/2010 03:57:01 GMT-400", "price" : 4.23}
{"day" : "2010/10/04", "time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27}
{"day" : "2010/10/03", "time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10}
{"day" : "2010/10/06", "time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
{"day" : "2010/10/04", "time" : "10/4/2010 08:34:50 GMT-400", "price" : 4.01}
```

> You should never store money amounts as floating-point numbers because of inexactness concerns, but we'll do it anyway in this example for simplicity.

We want our results to be a list of the latest time and price for each day, something like this:

```
[
    {"time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10},
    {"time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27},
    {"time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
]
```

We can accomplish this by splitting the collection into sets of documents grouped by day then finding the document with the latest timestamp for each day and adding it to the result set. The whole function might look something like this:

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }
... }}})
```

Let's break this command down into its component keys:

`"ns" : "stocks"`
: This determines which collection we'll be running the group on.

`"key" : "day"`
: This specifies the key on which to group the documents in the collection. In this case, that would be the `"day"` key. All of the documents with a `"day"` key of a given value will be grouped together.

`"initial" : {"time" : 0}`
: The first time the `reduce` function is called for a given group, it will be passed the initialization document. This same accumulator will be used for each member of a given group, so any changes made to it can be persisted.

`"$reduce" : function(doc, prev) { ... }`
: This will be called once for each document in the collection. It is passed the current document and an accumulator document: the result so far for that group. In this example, we want the `reduce` function to compare the current document's time with the accumulator's time. If the current document has a later time, we'll set the accumulator's day and price to be the current document's values. Remember that there is a separate accumulator for each group, so there is no need to worry about different days using the same accumulator.

In the initial statement of the problem, we said that we wanted only the last 30 days worth of prices. Our current solution is iterating over the entire collection, however. This is why you can include a `"condition"` that documents must satisfy in order to be processed by the group command at all:

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",

... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }},
... "condition" : {"day" : {"$gt" : "2010/09/30"}}
... }})
```

> Some documentation refers to a `"cond"` or `"q"` key, both of which are identical to the `"condition"` key (just less descriptive).

Now the command will return an array of 30 documents, each of which is a group. Each group has the key on which the group was based (in this case, `"day" : `*`string`*) and the final value of `prev` for that group. If some of the documents do not contain the key, these will be grouped into a single group with a `day : null` element. You can eliminate this group by adding `"day" : {"$exists" : true}` to the `"condition"`. The `group` command also returns the total number of documents used and the number of distinct values for `"key"`:

```
> db.runCommand({"group" : {...}})
{
    "retval" :
        [
            {
                "day" : "2010/10/04",
                "time" : "Mon Oct 04 2010 11:28:39 GMT-0400 (EST)"
                "price" : 4.27
            },
            ...
        ],
    "count" : 734,
    "keys" : 30,
    "ok" : 1
}
```

We explicitly set the `"price"` for each group, and the `"time"` was set by the initializer and then updated. The `"day"` is included because the key being grouped by is included by default in each `"retval"` embedded document. If you don't want to return this key, you can use a finalizer to change the final accumulator document into anything, even a nondocument (e.g., a number or string).

Using a Finalizer

Finalizers can be used to minimize the amount of data that needs to be transferred from the database to the user, which is important, because the `group` command's output needs to fit in a single database response. To demonstrate this, we'll take the example

of a blog where each post has tags. We want to find the most popular tag for each day. We can group by day (again) and keep a count for each tag. This might look something like this:

```
> db.posts.group({
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     }
... }})
```

This will return something like this:

```
[
    {"day" : "2010/01/12", "tags" : {"nosql" : 4, "winter" : 10, "sledding" : 2}},
    {"day" : "2010/01/13", "tags" : {"soda" : 5, "php" : 2}},
    {"day" : "2010/01/14", "tags" : {"python" : 6, "winter" : 4, "nosql": 15}}
]
```

Then we could find the largest value in the `"tags"` document on the client side. However, sending the entire tags document for every day is a lot of extra overhead to send to the client: an entire set of key/value pairs for each day, when all we want is a single string. This is why `group` takes an optional `"finalize"` key. `"finalize"` can contain a function that is run on each group once, right before the result is sent back to the client. We can use a `"finalize"` function to trim out all of the cruft from our results:

```
> db.runCommand({"group" : {
... "ns" : "posts",
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     },
... "finalize" : function(prev) {
...     var mostPopular = 0;
...     for (i in prev.tags) {
...         if (prev.tags[i] > mostPopular) {
...             prev.tag = i;
...             mostPopular = prev.tags[i];
...         }
...     }
...     delete prev.tags
... }}})
```

Now, we're only getting the information we want; the server will send back something like this:

```
[
    {"day" : "2010/01/12", "tag" : "winter"},
    {"day" : "2010/01/13", "tag" : "soda"},
    {"day" : "2010/01/14", "tag" : "nosql"}
]
```

`finalize` can either modify the argument passed in or return a new value.

Using a Function as a Key

Sometimes you may have more complicated criteria that you want to group by, not just a single key. Suppose you are using `group` to count how many blog posts are in each category. (Each blog post is in a single category.) Post authors were inconsistent, though, and categorized posts with haphazard capitalization. So, if you group by category name, you'll end up with separate groups for "MongoDB" and "mongodb." To make sure any variation of capitalization is treated as the same key, you can define a function to determine documents' grouping key.

To define a grouping function, you must use a `$keyf` key (instead of `"key"`). Using `"$keyf"` makes the `group` command look something like this:

```
> db.posts.group({"ns" : "posts",
... "$keyf" : function(x) { return x.category.toLowerCase(); },
... "initializer" : ... })
```

`"$keyf"` allows you can group by arbitrarily complex criteria.

MapReduce

MapReduce is the Uzi of aggregation tools. Everything described with `count`, `distinct`, and `group` can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem, sends chunks of it to different machines, and lets each machine solve its part of the problem. When all of the machines are finished, they merge all of the pieces of the solution back into a full solution.

MapReduce has a couple of steps. It starts with the map step, which *maps* an operation onto every document in a collection. That operation could be either "do nothing" or "emit these keys with *X* values." There is then an intermediary stage called the shuffle step: keys are grouped and lists of emitted values are created for each key. The reduce takes this list of values and *reduces* it to a single element. This element is returned to the shuffle step until each key has a list containing a single value: the result.

The price of using MapReduce is speed: `group` is not particularly speedy, but MapReduce is slower and is not supposed to be used in "real time." You run MapReduce as a background job, it creates a collection of results, and then you can query that collection in real time.

We'll go through a couple of MapReduce examples because it is incredibly useful and powerful but also a somewhat complex tool.

Example 1: Finding All Keys in a Collection

Using MapReduce for this problem might be overkill, but it is a good way to get familiar with how MapReduce works. If you already understand MapReduce, feel free to skip ahead to the last part of this section, where we cover MongoDB-specific MapReduce considerations.

MongoDB is schemaless, so it does not keep track of the keys in each document. The best way, in general, to find all the keys across all the documents in a collection is to use MapReduce. In this example, we'll also get a count of how many times each key appears in the collection. This example doesn't include keys for embedded documents, but it would be a simple addition to the `map` function to do so.

For the mapping step, we want to get every key of every document in the collection. The `map` function uses a special function to "return" values that we want to process later: `emit`. `emit` gives MapReduce a key (like the one used by `group` earlier) and a value. In this case, we emit a count of how many times a given key appeared in the document (once: `{count : 1}`). We want a separate count for each key, so we'll call emit for every key in the document. `this` is a reference to the current document we are mapping:

```
> map = function() {
... for (var key in this) {
...     emit(key, {count : 1});
... }};
```

Now we have a ton of little `{count : 1}` documents floating around, each associated with a key from the collection. An array of one or more of these `{count : 1}` documents will be passed to the `reduce` function. The `reduce` function is passed two arguments: `key`, which is the first argument from `emit`, and an array of one or more `{count : 1}` documents that were emitted for that key:

```
> reduce = function(key, emits) {
... total = 0;
... for (var i in emits) {
...     total += emits[i].count;
... }
... return {"count" : total};
... }
```

`reduce` must be able to be called repeatedly on results from either the map phase or previous reduce phases. Therefore, `reduce` must return a document that can be re-sent to `reduce` as an element of its second argument. For example, say we have the key `x` mapped to three documents: `{count : 1, id : 1}`, `{count : 1, id : 2}`, and `{count : 1, id : 3}`. (The ID keys are just for identification purposes.) MongoDB might call `reduce` in the following pattern:

```
> r1 = reduce("x", [{count : 1, id : 1}, {count : 1, id : 2}])
{count : 2}
> r2 = reduce("x", [{count : 1, id : 3}])
{count : 1}
> reduce("x", [r1, r2])
{count : 3}
```

You cannot depend on the second argument always holding one of the initial documents (`{count : 1}` in this case) or being a certain length. `reduce` should be able to be run on any combination of `emit` documents and `reduce` return values.

Altogether, this MapReduce function would look like this:

```
> mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" : reduce})
{
    "result" : "tmp.mr.mapreduce_1266787811_1",
    "timeMillis" : 12,
    "counts" : {
        "input" : 6
        "emit" : 14
        "output" : 5
    },
    "ok" : true
}
```

The document MapReduce returns gives you a bunch of metainformation about the operation:

`"result" : "tmp.mr.mapreduce_1266787811_1"`
: This is the name of the collection the MapReduce results were stored in. This is a temporary collection that will be deleted when the connection that did the MapReduce is closed. We will go over how to specify a nicer name and make the collection permanent in a later part of this chapter.

`"timeMillis" : 12`
: How long the operation took, in milliseconds.

`"counts" : { ... }`
: This embedded document contains three keys:

  `"input" : 6`
  : The number of documents sent to the `map` function.

  `"emit" : 14`
  : The number of times `emit` was called in the `map` function.

  `"output" : 5`
  : The number of documents created in the result collection.

  `"counts"` is mostly useful for debugging.

If we do a find on the resulting collection, we can see all of the keys and their counts from our original collection:

```
> db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

Each of the key values becomes an `"_id"`, and the final result of the reduce step(s) becomes the `"value"`.

Example 2: Categorizing Web Pages

Suppose we have a site where people can submit links to other pages, such as reddit.com. Submitters can tag a link as related to certain popular topics, e.g., "politics," "geek," or "icanhascheezburger." We can use MapReduce to figure out which topics are the most popular, as a combination of recent and most-voted-for.

First, we need a `map` function that emits tags with a value based on the popularity and recency of a document:

```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;

        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

Now we need to reduce all of the emitted values for a tag into a single score for that tag:

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

The final collection will end up with a full list of URLs for each tag and a score showing how popular that particular tag is.

MongoDB and MapReduce

Both of the previous examples used only the `mapreduce`, `map`, and `reduce` keys. These three keys are required, but there are many optional keys that can be passed to the MapReduce command.

`"finalize" : function`
: A final step to send `reduce`'s output to.

`"keeptemp" : boolean`
: If the temporary result collection should be saved when the connection is closed.

`"output" : string`
: Name for the output collection. Setting this option implies `keeptemp : true`.

`"query" : document`
: Query to filter documents by before sending to the `map` function.

`"sort" : document`
: Sort to use on documents before sending to the map (useful in conjunction with the `limit` option).

`"limit" : integer`
: Maximum number of documents to send to the `map` function.

`"scope" : document`
: Variables that can be used in any of the JavaScript code.

`"verbose" : boolean`
: Whether to use more verbose output in the server logs.

The finalize function

As with the previous `group` command, MapReduce can be passed a `finalize` function that will be run on the last `reduce`'s output before it is saved to a temporary collection.

Returning large result sets is less critical with MapReduce than `group`, because the whole result doesn't have to fit in 4MB. However, the information will be passed over the wire eventually, so `finalize` is a good chance to take averages, chomp arrays, and remove extra information in general.

Keeping output collections

By default, Mongo creates a temporary collection while it is processing the MapReduce with a name that you are unlikely to choose for a collection: a dot-separated string containing *mr*, the name of the collection you're MapReducing, a timestamp, and the job's ID with the database. It ends up looking something like *mr.stuff.18234210220.2*. MongoDB will automatically destroy this collection when the connection that did the MapReduce is closed. (You can also drop it manually when you're done with it.) If you want to persist this collection even after disconnecting, you can specify `keeptemp : true` as an option.

If you'll be using the temporary collection regularly, you may want to give it a better name. You can specify a more human-readable name with the `out` option, which takes a string. If you specify `out`, you need not specify `keeptemp : true`, because it is implied. Even if you specify a "pretty" name for the collection, MongoDB will use the autogenerated collection name for intermediate steps of the MapReduce. When it has finished, it will automatically and atomically rename the collection from the autogenerated name to your chosen name. This means that if you run MapReduce multiple times with the same target collection, you will never be using an incomplete collection for operations.

The output collection created by MapReduce is a normal collection, which means that there is no problem with doing a MapReduce on it, or a MapReduce on the results from that MapReduce, ad infinitum!

MapReduce on a subset of documents

Sometimes you need to run MapReduce on only part of a collection. You can add a query to filter the documents before they are passed to the `map` function.

Every document passed to the `map` function needs to be deserialized from BSON into a JavaScript object, which is a fairly expensive operation. If you know that you will need to run MapReduce only on a subset of the documents in the collection, adding a filter can greatly speed up the command. The filter is specified by the `"query"`, `"limit"`, and `"sort"` keys.

The `"query"` key takes a query document as a value. Any documents that would ordinarily be returned by that query will be passed to the `map` function. For example, if we have an application tracking analytics and want a summary for the last week, we can use MapReduce on only the most recent week's documents with the following command:

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "query" : {"date" : {"$gt" : week_ago}}})
```

The `sort` option is mostly useful in conjunction with `limit`. `limit` can be used on its own, as well, to simply provide a cutoff on the number of documents sent to the `map` function.

If, in the previous example, we wanted an analysis of the last 10,000 page views (instead of the last week), we could use `limit` and `sort`:

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "limit" : 10000, "sort" : {"date" : -1}})
```

`query`, `limit`, and `sort` can be used in any combination, but `sort` isn't useful if `limit` isn't present.

Using a scope

MapReduce can take a code type for the `map`, `reduce`, and `finalize` functions, and, in most languages, you can specify a scope to be passed with code. However, MapReduce ignores this scope. It has its own scope key, `"scope"`, and you must use that if there are client-side values you want to use in your MapReduce. You can set them using a plain document of the form `variable_name : value`, and they will be available in your `map`, `reduce`, and `finalize` functions. The scope is immutable from within these functions.

For instance, in the example in the previous section, we calculated the recency of a page using `1/(new Date() - this.date)`. We could, instead, pass in the current date as part of the scope with the following code:

```
> db.runCommand({"mapreduce" : "webpages", "map" : map, "reduce" : reduce,
                 "scope" : {now : new Date()}})
```

Then, in the `map` function, we could say `1/(now - this.date)`.

Getting more output

There is also a verbose option for debugging. If you would like to see the progress of your MapReduce as it runs, you can specify `"verbose" : true`.

You can also use `print` to see what's happening in the `map`, `reduce`, and `finalize` functions. `print` will print to the server log.

CHAPTER 7

Advanced Topics

MongoDB supports some advanced functionality that goes well beyond the capabilities discussed so far. When you want to become a power user, this chapter has you covered; in it we'll discuss the following:

- Using database commands to take advantage of advanced features
- Working with capped collections, a special type of collection
- Leveraging GridFS for storing large files
- Taking advantage of MongoDB's support for server-side JavaScript
- Understanding what database references are and when you should consider using them

Database Commands

In the previous chapters we've seen how to create, read, update, and delete documents in MongoDB. In addition to these basic operations, MongoDB supports a wide range of advanced operations that are implemented as *commands*. Commands implement all of the functionality that doesn't fit neatly into "create, read, update, delete."

We've already seen a couple of commands in the previous chapters; for instance, we used the `getLastError` command in Chapter 3 to check the number of documents affected by an update:

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

In this section, we'll take a closer look at commands to see exactly what they are and how they're implemented. We'll also describe some of the most useful commands that are supported by MongoDB.

How Commands Work

One example of a database command that you are probably familiar with is `drop`: to drop a collection from the shell, we run `db.test.drop()`. Under the hood, this function is actually running the `drop` command—we can perform the exact same operation using `runCommand`:

```
> db.runCommand({"drop" : "test"});
{
    "nIndexesWas" : 1,
    "msg" : "indexes dropped for collection",
    "ns" : "test.test",
    "ok" : true
}
```

The document we get as a result is the *command response*, which contains information about whether the command was successful, as well as any other information that the command might provide. The command response will always contain the key `"ok"`. If `"ok"` is `true`, the command was successful, and if it is `false`, the command failed for some reason.

> In version 1.5 and earlier, the value of `"ok"` was `1.0` or `0.0` instead of `true` or `false`, respectively.

If `"ok"` is `false`, then an additional key will be present, `"errmsg"`. The value of `"errmsg"` is a string explaining why the command failed. As an example, let's try running the `drop` command again, on the collection that we just dropped:

```
> db.runCommand({"drop" : "test"});
{ "errmsg" : "ns not found", "ok" : false }
```

Commands in MongoDB are actually implemented as a special type of query that gets performed on the *$cmd* collection. `runCommand` just takes a command document and performs the equivalent query, so our drop call becomes the following:

```
db.$cmd.findOne({"drop" : "test"});
```

When the MongoDB server gets a query on the *$cmd* collection, it handles it using special logic, rather than the normal code for handling queries. Almost all MongoDB drivers provide a helper method like `runCommand` for running commands, but commands can always be run using a simple query if necessary.

Some commands require administrator access and must be run on the *admin* database. If such a command is run on any other database, it will return an "access denied" error.

Command Reference

At the time of this writing, MongoDB supports more than 75 different commands, and more commands are being added all the time. There are two ways to get an up-to-date list of all of the commands supported by a MongoDB server:

- Run `db.listCommands()` from the shell, or run the equivalent `listCommands` command from any other driver.
- Browse to the *http://localhost:28017/_commands* URL on the MongoDB admin interface (for more on the admin interface see Chapter 8).

The following list contains some of the most frequently used MongoDB commands along with example documents showing how each command should be represented:

buildInfo

```
{"buildInfo" : 1}
```

Admin-only command that returns information about the MongoDB server's version number and host operating system.

collStats

```
{"collStats" : collection}
```

Gives some stats about a given collection, including its data size, the amount of storage space allocated for it, and the size of its indexes.

distinct

```
{"distinct" : collection, "key": key, "query": query}
```

Gets a list of distinct values for *`key`* in documents matching *`query`*, across a given collection.

drop

```
{"drop" : collection}
```

Removes all data for *`collection`*.

dropDatabase

```
{"dropDatabase" : 1}
```

Removes all data for the current database.

dropIndexes

```
{"dropIndexes" : collection, "index" : name}
```

Deletes the index named *`name`* from *`collection`*, or all indexes if *`name`* is `"*"`.

findAndModify

See Chapter 3 for a full reference on using the `findAndModify` command.

getLastError

```
{"getLastError" : 1[, "w" : w[, "wtimeout" : timeout]]}
```

Checks for errors or other status information about the last operation performed on this connection. The command will optionally block until *`w`* slaves have replicated the last operation on this connection (or until *`timeout`* milliseconds have gone by).

isMaster

```
{"isMaster" : 1}
```

Checks if this server is a master or slave.

listCommands

```
{"listCommands" : 1}
```

Returns a list of all database commands available on this server, as well as some information about each command.

listDatabases

```
{"listDatabases" : 1}
```

Admin-only command listing all databases on this server.

ping

```
{"ping" : 1}
```

Checks if a server is alive. This command will return immediately even if the server is in a lock.

renameCollection

```
{"renameCollection" : a, "to" : b}
```

Renames collection *`a`* to *`b`*, where both *`a`* and *`b`* are *full collection namespaces* (e.g., `"foo.bar"` for the collection *bar* in the *foo* database).

repairDatabase

```
{"repairDatabase" : 1}
```

Repairs and compacts the current database, which can be a long-running operation. See "Repair" on page 124 for more information.

serverStatus

```
{"serverStatus" : 1}
```

Gets administrative statistics for this server. See "Monitoring" on page 114 for more information.

Remember, there are far more supported commands than just those listed earlier. Others are documented as appropriate throughout the rest of the book, and for the full list, just run `listCommands`.

Capped Collections

We've already seen how normal collections in MongoDB are created dynamically and automatically grow in size to fit additional data. MongoDB also supports a different type of collection, called a *capped collection*, which is created in advance and is fixed in size (see Figure 7-1). Having fixed-size collections brings up an interesting question: what happens when we try to insert into a capped collection that is already full? The answer is that capped collections behave like circular queues: if we're out of space, the oldest document(s) will be deleted, and the new one will take its place (see Figure 7-2). This means that capped collections automatically age-out the oldest documents as new documents are inserted.

Certain operations are not allowed on capped collections. Documents cannot be removed or deleted (aside from the automatic age-out described earlier), and updates that would cause documents to move (in general updates that cause documents to grow in size) are disallowed. By preventing these two operations, we guarantee that documents in a capped collection are stored in insertion order and that there is no need to maintain a free list for space from removed documents.

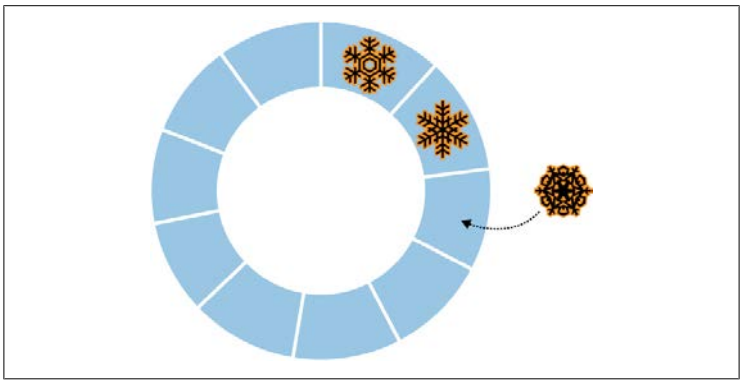


Figure 7-1. New documents are inserted at the end of the queue

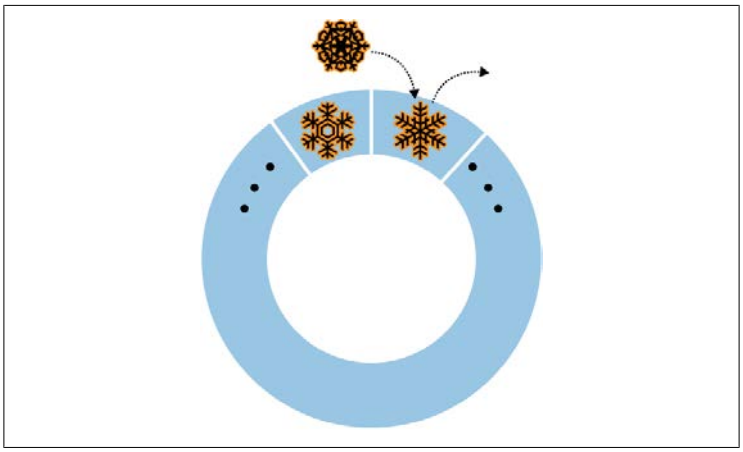


Figure 7-2. When the queue is full, the oldest element will be replaced by the newest

A final difference between capped and normal collections is that in a capped collection, there are no indexes by default, not even an index on `"_id"`.

Properties and Use Cases

The set of features and limitations possessed by capped collections combine to give them some interesting properties. First, inserts into a capped collection are extremely fast. When doing an insert, there is never a need to allocate additional space, and the server never needs to search through a free list to find the right place to put a document. The inserted document can always be placed directly at the "tail" of the collection, overwriting old documents if needed. By default, there are also no indexes to update on an insert, so an insert is essentially a single `memcpy`.

Another interesting property of capped collections is that queries retrieving documents in insertion order are very fast. Because documents are always stored in insertion order, queries for documents in that order just walk over the collection, returning documents in the exact order that they appear on disk. By default, any `find` performed on a capped collection will always return results in insertion order.

Finally, capped collections have the useful property of automatically aging-out old data as new data is inserted. The combination of fast inserts, fast queries for documents sorted by insertion order, and automatic age-out makes capped collections ideal for use cases like logging. In fact, the primary motivation for including capped collections in MongoDB is so that they can be used to store an internal replication log, the *oplog* (for more on replication and the oplog, see Chapter 9). Another good use case to consider for capped collections is caching of small numbers of documents. In general, capped collections are good for any case where the auto age-out property is helpful as opposed to undesirable and the limitations on available operations are not prohibitive.

Creating Capped Collections

Unlike normal collections, capped collections must be explicitly created before they are used. To create a capped collection, use the `create` command. From the shell, this can be done using `createCollection`:

```
> db.createCollection("my_collection", {capped: true, size: 100000});
{ "ok" : true }
```

The previous command creates a capped collection, *my_collection*, that is a fixed size of 100,000 bytes. `createCollection` has a couple of other options as well. We can specify a limit on the number of documents in a capped collection in addition to the limit on total collection size:

```
> db.createCollection("my_collection", {capped: true, size: 100000, max: 100});
{ "ok" : true }
```

> When limiting the number of documents in a capped collection, you must specify a size limit as well. Age-out will be based on the number of documents in the collection, *unless* the collection runs out of space before the limit is reached. In that case, age-out will be based on collection size, as in any other capped collection.

Another option for creating a capped collection is to convert an existing, regular collection into a capped collection. This can be done using the `convertToCapped` command—in the following example, we convert the *test* collection to a capped collection of 10,000 bytes:

```
> db.runCommand({convertToCapped: "test", size: 10000});
{ "ok" : true }
```

Sorting Au Naturel

There is a special type of sort that you can do with capped collections, called a *natural sort*. Natural order is just the order that documents appear on disk (see Figure 7-3).

Because documents in a capped collection are always kept in insertion order, natural order is the same as insertion order. As mentioned earlier, queries on a capped collection return documents in insertion order by default. You can also sort in reverse insertion order with a natural sort (see Figure 7-4):

```
> db.my_collection.find().sort({"$natural" : -1})
```

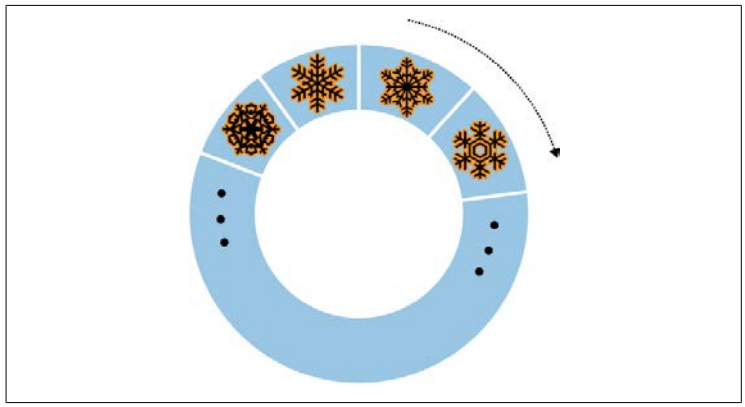


Figure 7-3. Sort by {"$natural" : 1}

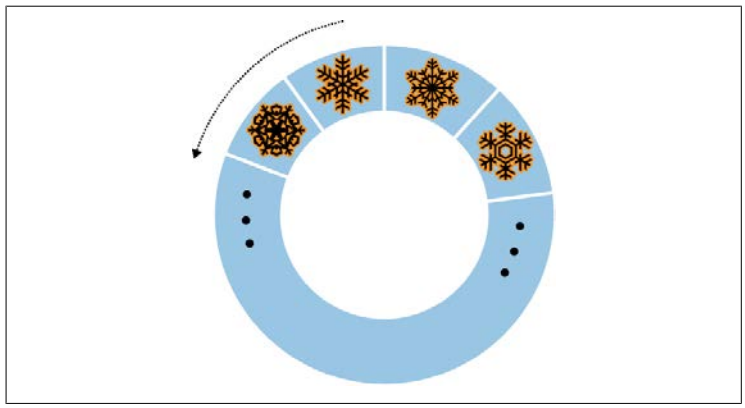


Figure 7-4. Sort by {"$natural" : -1}

Sorting by `{"$natural" : 1}` is identical to the default sort. Noncapped collections do not guarantee that documents are stored in any particular order, so their natural ordering is not as significant.

Tailable Cursors

Tailable cursors are a very special type of persistent cursor that are not closed when their results are exhausted. They were inspired by the *tail -f* command and, similar to the command, will continue fetching output for as long as possible. Because the cursors do not die when they runs out of results, they can continue to fetch new results as they are added to the collection. Tailable cursors can be used only on capped collections.

Again, the Mongo shell does not allow you to use tailable cursors, but using one in PHP looks something like the following:

```
$cursor = $collection->find()->tailable();

while (true) {
    if (!$cursor->hasNext()) {
        if ($cursor->dead()) {
            break;
        }
        sleep(1);
    }
    else {
        while (cursor->hasNext()) {
            do_stuff(cursor->getNext());
        }
    }
}
```

Although the cursor has not died, it will be either processing results or waiting for more results to arrive.

GridFS: Storing Files

GridFS is a mechanism for storing large binary files in MongoDB. There are several reasons why you might consider using GridFS for file storage:

- Using GridFS can simplify your stack. If you're already using MongoDB, GridFS obviates the need for a separate file storage architecture.
- GridFS will leverage any existing replication or autosharding that you've set up for MongoDB, so getting failover and scale-out for file storage is easy.
- GridFS can alleviate some of the issues that certain filesystems can exhibit when being used to store user uploads. For example, GridFS does not have issues with storing large numbers of files in the same directory.
- You can get great disk locality with GridFS, because MongoDB allocates data files in 2GB chunks.

Getting Started with GridFS: mongofiles

The easiest way to get up and running with GridFS is by using the `mongofiles` utility. `mongofiles` is included with all MongoDB distributions and can be used to upload, download, list, search for, or delete files in GridFS. As with any of the other command-line tools, run `mongofiles --help` to see the options available for `mongofiles`. The following session shows how to use `mongofiles` to upload a file from the filesystem to GridFS, list all of the files in GridFS, and download a file that we've previously uploaded:

```
$ echo "Hello, world" > foo.txt
$ ./mongofiles put foo.txt
connected to: 127.0.0.1
added file: { _id: ObjectId('4c0d2a6c3052c25545139b88'),
              filename: "foo.txt", length: 13, chunkSize: 262144,
              uploadDate: new Date(1275931244818),
              md5: "a7966bf58e23583c9a5a4059383ff850" }
done!
$ ./mongofiles list
connected to: 127.0.0.1
foo.txt 13
$ rm foo.txt
$ ./mongofiles get foo.txt
connected to: 127.0.0.1
done write to: foo.txt
$ cat foo.txt
Hello, world
```

In the previous example, we perform three basic operations using `mongofiles`: `put`, `list`, and `get`. The `put` operation takes a file in the filesystem and adds it to GridFS, `list` will list any files that have been added to GridFS, and `get` does the inverse of `put`: it takes a file from GridFS and writes it to the filesystem. `mongofiles` also supports two other operations: `search` for finding files in GridFS by filename and `delete` for removing a file from GridFS.

Working with GridFS from the MongoDB Drivers

We've seen how easy it is to work with GridFS from the command line, and it's equally easy to work with from the MongoDB drivers. For example, we can use PyMongo, the Python driver for MongoDB, to perform the same series of operations as we did with `mongofiles`:

```
>>> from pymongo import Connection
>>> import gridfs
>>> db = Connection().test
>>> fs = gridfs.GridFS(db)
>>> file_id = fs.put("Hello, world", filename="foo.txt")
>>> fs.list()
[u'foo.txt']
>>> fs.get(file_id).read()
'Hello, world'
```

The API for working with GridFS from PyMongo is very similar to that of `mongofiles`: we can easily perform the basic `put`, `get`, and `list` operations. Almost all of the MongoDB drivers follow this basic pattern for working with GridFS, while often exposing more advanced functionality as well. For driver-specific information on GridFS, please check out the documentation for the specific driver you're using.

Under the Hood

GridFS is a lightweight specification for storing files that is built on top of normal MongoDB documents. The MongoDB server actually does almost nothing to "special-case" the handling of GridFS requests; all of the work is handled by the client-side drivers and tools.

The basic idea behind GridFS is that we can store large files by splitting them up into *chunks* and storing each chunk as a separate document. Because MongoDB supports storing binary data in documents, we can keep storage overhead for chunks to a minimum. In addition to storing each chunk of a file, we store a single document that groups the chunks together and contains metadata about the file.

The chunks for GridFS are stored in their own collection. By default chunks will use the collection *fs.chunks*, but this can be overridden if needed. Within the chunks collection the structure of the individual documents is pretty simple:

```
{
    "_id" : ObjectId("..."),
    "n" : 0,
    "data" : BinData("..."),
    "files_id" : ObjectId("...")
}
```

Like any other MongoDB document, the chunk has its own unique `"_id"`. In addition, it has a couple of other keys. `"files_id"` is the `"_id"` of the file document that contains the metadata for this chunk. `"n"` is the chunk number; this attribute tracks the order that chunks were present in the original file. Finally, `"data"` contains the binary data that makes up this chunk of the file.

The metadata for each file is stored in a separate collection, which defaults to *fs.files*. Each document in the files collection represents a single file in GridFS and can contain any custom metadata that should be associated with that file. In addition to any user-defined keys, there are a couple of keys that are mandated by the GridFS specification:

_id
: A unique id for the file—this is what will be stored in each chunk as the value for the `"files_id"` key.

length
: The total number of bytes making up the content of the file.

chunkSize
: The size of each chunk comprising the file, in bytes. The default is 256K, but this can be adjusted if needed.

uploadDate
: A timestamp representing when this file was stored in GridFS.

md5
: An md5 checksum of this file's contents, generated on the server side.

Of all of the required keys, perhaps the most interesting (or least self-explanatory) is `"md5"`. The value for `"md5"` is generated by the MongoDB server using the `filemd5` command, which computes the md5 checksum of the uploaded chunks. This means that users can check the value of the `"md5"` key to ensure that a file was uploaded correctly.

When we understand the underlying GridFS specification, it becomes trivial to implement features that the driver we're using might not implement for us. For example, we can use the `distinct` command to get a list of unique filenames stored in GridFS:

```
> db.fs.files.distinct("filename")
[ "foo.txt" ]
```

Server-Side Scripting

JavaScript can be executed on the server using the `db.eval` function. It can also be stored in the database and is used in some database commands.

db.eval

`db.eval` is a function that allows you to execute arbitrary JavaScript on the MongoDB server. It takes a string of JavaScript, sends it to MongoDB (which executes it), and returns the result.

`db.eval` can be used to imitate multidocument transactions: `db.eval` locks the database, executes the JavaScript, and unlocks the database. There's no built-in rollback, but this does give you a guarantee of a series of operations occurring in a certain order (unless an error occurs).

There are two options for sending code: enclosing it in a function or not. The following two lines are equivalent:

```
> db.eval("return 1;")
1
> db.eval("function() { return 1; }")
1
```

Defining an enclosing function is necessary only if you are passing in arguments. These can be passed using `db.eval`'s second argument, which is an array of values. For example, if we wanted to pass the username as an argument to a function, we could say the following:

```
> db.eval("function(u) { print('Hello, '+u+'!'); }", [username])
```

You can pass in as many arguments as necessary. For instance, if we want a sum of three numbers, we can do the following:

```
> db.eval("function(x,y,z) { return x + y + z; }", [num1, num2, num3])
```

`num1` becomes `x`, `num2` becomes `y`, and `num3` becomes `z`. If you would like to use a variable number of parameters, arguments in JavaScript are stored in an `arguments` array when a function is called.

As a `db.eval` expression becomes more complex, debugging can be tricky. The JavaScript code is run by the database and often doesn't have useful line numbers in error messages. A good way of debugging is printing to the database log, which you can do with the print function:

```
> db.eval("print('Hello, world');");
```

Stored JavaScript

MongoDB has a special collection for each database called *system.js*, which can store JavaScript variables. These variables can then be used in any of MongoDB's JavaScript contexts, including `"$where"` clauses, `db.eval` calls, and MapReduce jobs. You can add variables to *system.js* with a simple insert:

```
> db.system.js.insert({"_id" : "x", "value" : 1})
> db.system.js.insert({"_id" : "y", "value" : 2})
> db.system.js.insert({"_id" : "z", "value" : 3})
```

This defines variables `x`, `y`, and `z` in the global scope. Now, if we want to find their sum, we can execute the following:

```
> db.eval("return x+y+z;")
6
```

system.js can be used to store JavaScript code as well as simple values. This can be handy for defining your own utilities. For example, if you want to create a logging function to use in JavaScript code, you can store it in *system.js*:

```
> db.system.js.insert({"_id" : "log", "value" :
... function(msg, level) {
...     var levels = ["DEBUG", "WARN", "ERROR", "FATAL"];
...     level = level ? level : 0; // check if level is defined
...     var now = new Date();
...     print(now + " " + levels[level] + msg);
... }})
```

Now, in any JavaScript context, you can call this log function:

```
> db.eval("x = 1; log('x is '+x); x = 2; log('x is greater than 1', 1);");
```

The database log will then contain something like this:

```
Fri Jun 11 2010 11:12:39 GMT-0400 (EST) DEBUG x is 1
Fri Jun 11 2010 11:12:40 GMT-0400 (EST) WARN x is greater than 1
```

There are downsides to using stored JavaScript: it keeps portions of your code out of source control, and it can obfuscate JavaScript sent from the client.

The best reason for storing JavaScript is if you have multiple parts of your code (or code in different programs or languages) using a single JavaScript function. Keeping such functions in a central location means they do not need to be updated in multiple places if changes are required. Stored JavaScript can also be useful if your JavaScript code is long and executed frequently, because storing it once can cut down on network transfer time.

Security

Executing JavaScript is one of the few times you must be careful about security with MongoDB. If done incorrectly, server-side JavaScript is susceptible to injection attacks similar to those that occur in a relational database. Luckily, it is very easy to prevent these attacks and use JavaScript safely.

Suppose you want to print "Hello, *username*!" to the user. If the username is in a variable called `username`, you could write a JavaScript function such as the following:

```
> func = "function() { print('Hello, "+username+"!'); }"
```

If `username` is a user-defined variable, it could contain the string `"'); db.dropDatabase(); print('"`, which would turn the code into this:

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!'); }"
```

Now your entire database has been dropped!

To prevent this, you should use a *scope* to pass in the username. In PHP, for example, this looks like this:

```
$func = new MongoCode("function() { print('Hello, "+username+"!'); }",
... array("username" => $username));
```

Now the database will harmlessly print this:

```
Hello, '); db.dropDatabase(); print('!
```

Most drivers have a special type for sending code to the database, since code can actually be a composite of a string and a scope. A scope is just a document mapping variable names to values. This mapping becomes a local scope for the JavaScript function being executed.

> The shell does not have a code type that includes scope; you can only use strings or JavaScript functions with it.

Database References

Perhaps one of the least understood features of MongoDB is its support for *database references*, or *DBRefs*. DBRefs are like URLs: they are simply a specification for uniquely identifying a reference to document. They do not automatically load the document any more than a URL automatically loads a web page into a site with a link.

What Is a DBRef?

A DBRef is an embedded document, just like any other embedded document in MongoDB. A DBRef, however, has specific keys that must be present. A simple example looks like the following:

```
{"$ref" : collection, "$id" : id_value}
```

The DBRef references a specific *`collection`* and an *`id_value`* that we can use to find a single document by its `"_id"` within that collection. These two pieces of information allow us to use a DBRef to uniquely identify and reference any document within a MongoDB database. If we want to reference a document in a different database, DBRefs support an optional third key that we can use, `"$db"`:

```
{"$ref" : collection, "$id" : id_value, "$db" : database}
```

> DBRefs are one place in MongoDB where the order of keys in a document matters. The first key in a DBRef must be `"$ref"`, followed by `"$id"`, and then (optionally) `"$db"`.

Example Schema

Let's look at an example schema that uses DBRefs to reference documents across collections. The schema consists of two collections, *users* and *notes*. Users can create notes, which can reference users or other notes. Here are a couple of user documents, each with a unique username as its `"_id"` and a separate free-form `"display_name"`:

```
{"_id" : "mike", "display_name" : "Mike D"}
{"_id" : "kristina", "display_name" : "Kristina C"}
```

Notes are a little more complex. Each has a unique `"_id"`. Normally this `"_id"` would probably be an `ObjectId`, but we use an integer here to keep the example concise. Notes also have an `"author"`, some `"text"`, and an optional set of `"references"` to other notes or users:

```
{"_id" : 5, "author" : "mike", "text" : "MongoDB is fun!"}
{"_id" : 20, "author" : "kristina", "text" : "... and DBRefs are easy, too",
 "references": [{"$ref" : "users", "$id" : "mike"}, {"$ref" : "notes", "$id" : 5}]}
```

The second note contains some references to other documents, each stored as a DBRef. Our application code can use those DBRefs to get the documents for the "mike" user and the "MongoDB is fun!" note, both of which are associated with Kristina's note. This dereferencing is easy to implement; we use the value of the `"$ref"` key to get the collection to query on, and we use the value of the `"$id"` key to get the `"_id"` to query for:

```
> var note = db.notes.findOne({"_id" : 20});
> note.references.forEach(function(ref) {
... printjson(db[ref.$ref].findOne({"_id" : ref.$id}));
... });
{ "_id" : "mike", "display_name" : "Mike D" }
{ "_id" : 5, "author" : "mike", "text" : "MongoDB is fun!" }
```

Driver Support for DBRefs

One thing that can be confusing about DBRefs is that not all drivers treat them as normal embedded documents. Some provide a special type for DBRefs that will be automatically translated to and from the normal document representation. This is mainly provided as a convenience for developers, because it can make working with DBRefs a little less verbose. As an example, here we represent the same note as earlier using PyMongo and its DBRef type:

```
>>> note = {"_id": 20, "author": "kristina",
...         "text": "... and DBRefs are easy, too",
...         "references": [DBRef("users", "mike"), DBRef("notes", 5)]}
```

When the note is saved, the DBRef instances will automatically be translated to the equivalent embedded documents. When the note is returned from a query, the opposite will happen, and we'll get DBRef instances back.

Some drivers also add other helpers for working with DBRefs, like methods to handle dereferencing or even mechanisms for automatically dereferencing DBRefs as they are returned in query results. This helper functionality tends to vary from driver to driver, so for up-to-date information on what's supported, you should reference driver-specific documentation.

When Should DBRefs Be Used?

DBRefs are not essential for representing references to other documents in MongoDB. In fact, even the previous example does some referencing using a different mechanism: the `"author"` key in each note just stores the value of the author document's `"_id"` key. We don't need to use a DBRef because we know that each author is a document in the *users* collection. We've seen another example of this type of referencing as well: the `"files_id"` key in GridFS chunk documents is just an `"_id"` reference to a file document. With this option in mind, we have a decision to make each time we need to store a reference: should we use a DBRef or just store an `"_id"`?

Storing `"_id"`s is nice because they are more compact than DBRefs and also can be a little more lightweight for developers to work with. DBRefs, on the other hand, are

capable of referencing documents in any collection (or even database) without the developer having to know or remember what collection the referenced document might reside in. DBRefs also enable drivers and tools to provide some more enhanced functionality (e.g., automatic dereferencing) and could allow for more advanced support on the server side in the future.

In short, the best times to use DBRefs are when you're storing heterogeneous references to documents in different collections, like in the previous example or when you want to take advantage of some additional DBRef-specific functionality in a driver or tool. Otherwise, it's generally best to just store an `"_id"` and use that as a reference, because that representation tends to be more compact and easier to work with.

CHAPTER 8
Administration

Administering MongoDB is usually a simple task. From taking backups to setting up multinode systems with replication, most administrative tasks are quick and painless. This reflects a general philosophy of MongoDB, which is to minimize the number of dials in the system. Whenever possible, configuration is done automatically by the system rather than forcing users and administrators to tweak configuration settings. That said, there are still some administrative tasks that require manual intervention.

In this chapter we'll be switching gears from the developer perspective and discussing what you need to know to work with MongoDB from the operations or administration side. Whether you're working for a startup where you are both the engineering *and* ops teams or you're a DBA looking to work with MongoDB, this is the chapter for you. Here's the big picture:

- MongoDB is run as a normal command-line program using the `mongod` executable.
- MongoDB features a built-in admin interface and monitoring functionality that is easy to integrate with third-party monitoring packages.
- MongoDB supports basic, database-level authentication including read-only users and a separate level of authentication for admin access.
- There are several different ways of backing up a MongoDB system, the choice of which depends on a couple of key considerations.

Starting and Stopping MongoDB

In Chapter 2, we covered the basics of starting MongoDB. This chapter will go into more detail about what administrators need to know to deploy Mongo robustly in production.

Starting from the Command Line

The MongoDB server is started with the `mongod` executable. `mongod` has many configurable startup options; to view all of them, run `mongod --help` from the command line. A couple of the options are widely used and important to be aware of:

--dbpath
: Specify an alternate directory to use as the data directory; the default is */data/db/* (or *C:\data\db* on Windows). Each `mongod` process on a machine needs its own data directory, so if you are running three instances of `mongod`, you'll need three separate data directories. When `mongod` starts up, it creates a *mongod.lock* file in its data directory, which prevents any other `mongod` process from using that directory. If you attempt to start another MongoDB server using the same data directory, it will give an error:

```
"Unable to acquire lock for lockfilepath: /data/db/mongod.lock."
```

--port
: Specify the port number for the server to listen on. By default, `mongod` uses port 27017, which is unlikely to be used by another process (besides other `mongod` processes). If you would like to run more than one `mongod` process, you'll need to specify different ports for each one. If you try to start `mongod` on a port that is already being used, it will give an error:

```
"Address already in use for socket: 0.0.0.0:27017"
```

--fork
: Fork the server process, running MongoDB as a daemon.

--logpath
: Send all output to the specified file rather than outputting on the command line. This will create the file if it does not exist, assuming you have write permissions to the directory. It will also overwrite the log file if it already exists, erasing any older log entries. If you'd like to keep old logs around, use the `--logappend` option in addition to `--logpath`.

--config
: Use a configuration file for additional options not specified on the command line. See "File-Based Configuration" on page 113 for details.

So, to start the server as a daemon listening on port 5586 and sending all output to *mongodb.log*, we could run this:

```
$ ./mongod --port 5586 --fork --logpath mongodb.log
forked process: 45082
all output going to: mongodb.log
```

When you first install and start MongoDB, it is a good idea to look at the log. This might be an easy thing to miss, especially if MongoDB is being started from an init script, but the log often contains important warnings that prevent later errors from occurring. If you don't see any warnings in the MongoDB log on startup, then you are all set. However, you might see something like this:

```
$ ./mongod
Sat Apr 24 11:53:49 Mongo DB : starting : pid = 18417 port = 27017
dbpath = /data/db/ master = 0 slave = 0  32-bit
****
WARNING: This is development version of MongoDB.
         Not recommended for production.
****

** NOTE: when using MongoDB 32 bit, you are limited to about
**       2 gigabytes of data see
**       http://blog.mongodb.org/post/137788967/32-bit-limitations
**       for more

Sat Apr 24 11:53:49 db version v1.5.1-pre-, pdfile version 4.5
Sat Apr 24 11:53:49 git version: f86d93fd949777d5fbe00bf9784ec0947d6e75b9
Sat Apr 24 11:53:49 sys info: Linux ubuntu 2.6.31-15-generic ...
Sat Apr 24 11:53:49 waiting for connections on port 27017
Sat Apr 24 11:53:49 web admin interface listening on port 28017
```

The MongoDB being run here is a development version—if you download a stable release, it will not have the first warning. The second warning occurs because we are running a 32-bit build of MongoDB. We are limited to about 2GB of data when running 32 bit, because MongoDB uses a memory-mapped file-based storage engine (see Appendix C for details on MongoDB's storage engine). If you are using a stable release on a 64-bit machine, you won't get either of these messages, but it's a good idea to understand how MongoDB logs work and get used to how they look.

The log preamble won't change when you restart the database, so feel free to run MongoDB from an init script and ignore the logs, once you know what they say. However, it's a good idea to check again each time you do an install, upgrade, or recover from a crash, just to make sure MongoDB and your system are on the same page.

File-Based Configuration

MongoDB supports reading configuration information from a file. This can be useful if you have a large set of options you want to use or are automating the task of starting up MongoDB. To tell the server to get options from a configuration file, use the `-f` or `--config` flags. For example, run `mongod --config ~/.mongodb.conf` to use *~/.mongodb.conf* as a configuration file.

The options supported in a configuration file are exactly the same as those accepted at the command line. Here's an example configuration file:

```
# Start MongoDB as a daemon on port 5586

port = 5586
fork = true # daemonize it!
```

```
logpath = mongodb.log
```

This configuration file specifies the same options we used earlier when starting with regular command-line arguments. It also highlights most of the interesting aspects of MongoDB configuration files:

- Any text on a line that follows the # character is ignored as a comment.
- The syntax for specifying options is *`option = value`*, where *`option`* is case-sensitive.
- For command-line switches like `--fork`, the value `true` should be used.

Stopping MongoDB

Being able to safely stop a running MongoDB server is at least as important as being able to start one. There are a couple of different options for doing this effectively.

The most basic way to stop a running MongoDB server is to send it a SIGINT or SIGTERM signal. If the server is running as the foreground process in a terminal, this can be done by pressing Ctrl-C. Otherwise, a command like `kill` can be used to send the signal. If `mongod` has 10014 as its PID, the command would be `kill -2 10014` (SIGINT) or `kill 10014` (SIGTERM).

When `mongod` receives a SIGINT or SIGTERM, it will do a clean shutdown. This means it will wait for any currently running operations or file preallocations to finish (this could take a moment), close all open connections, flush all data to disk, and halt.

> It is important not to send a SIGKILL message (`kill -9`) to a running MongoDB server. Doing so will cause the database to shut down without going through the steps outlined earlier and could lead to corrupt data files. If this happens, the database should be repaired (see "Repair" on page 124) before being started back up.

Another way to cleanly shut down a running server is to use the `shutdown` command, `{"shutdown" : 1}`. This is an admin command and must be run on the *admin* database. The shell features a helper function to make this easier:

```
> use admin
switched to db admin
> db.shutdownServer();
server should be down...
```

Monitoring

As the administrator of a MongoDB server, it's important to monitor the health and performance of your system. Fortunately, MongoDB has functionality that makes monitoring easy.

Using the Admin Interface

By default, starting `mongod` also starts up a (very) basic HTTP server that listens on a port 1,000 higher than the native driver port. This server provides an HTTP interface that can be used to see basic information about the MongoDB server. All of the information presented can also be seen through the shell, but the HTTP interface gives a nice, easy-to-read overview.

To see the admin interface, start the database and go to *http://localhost:28017* in a web browser. (Use 1,000 higher than the port you specified, if you used the `--port` option when starting MongoDB.) You'll see a page that looks like Figure 8-1.

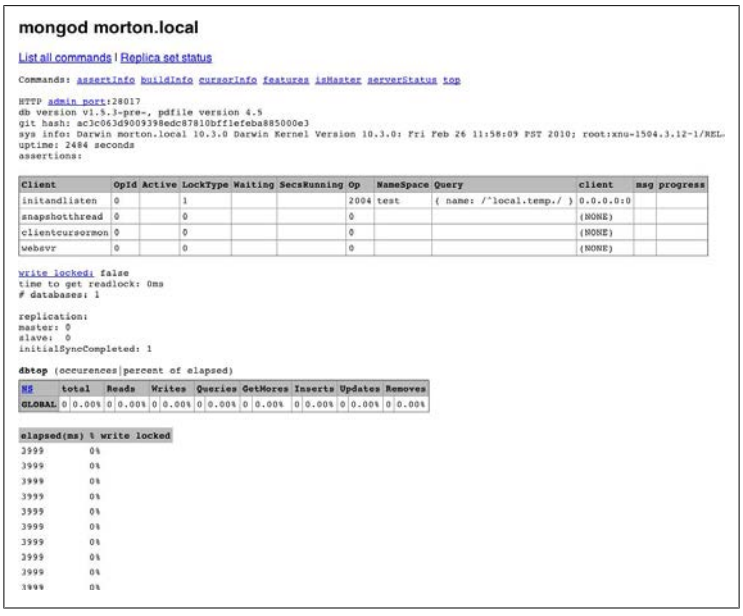


Figure 8-1. The admin interface

As you can see, this interface gives access to assertion, locking, indexing, and replication information about the MongoDB server. It also gives some more general information, like the log preamble and access to a list of available database commands.

To make full use of the admin interface (e.g., to access the command list), you'll need to turn on REST support with `--rest`. You can also turn off the admin interface altogether by starting `mongod` with the `--nohttpinterface` option.

> Do not attempt to connect a driver to the HTTP interface port, and do not try to connect to the native driver port via HTTP. The driver port handles only the native MongoDB wire protocol; it will not handle HTTP requests. For example, if you go to *http://localhost:27017* in a web browser, you will see:
>
> ```
> You are trying to access MongoDB on the native driver port.
> For http diagnostic access, add 1000 to the port number
> ```
>
> Similarly, you cannot use the native MongoDB wire protocol when connecting on the admin interface's port.

serverStatus

The most basic tool for getting statistics about a running MongoDB server is the `serverStatus` command, which has the following output (exact keys present may vary by platform/server version):

```
> db.runCommand({"serverStatus" : 1})
{
    "version" : "1.5.3",
    "uptime" : 166,
    "localTime" : "Thu Jun 10 2010 15:47:40 GMT-0400 (EDT)",
    "globalLock" : {
        "totalTime" : 165984675,
        "lockTime" : 91471425,
        "ratio" : 0.551083556358441
    },
    "mem" : {
        "bits" : 64,
        "resident" : 101,
        "virtual" : 2824,
        "supported" : true,
        "mapped" : 336
    },
    "connections" : {
        "current" : 141,
        "available" : 19859
    },
    "extra_info" : {
        "note" : "fields vary by platform"
    },
    "indexCounters" : {
        "btree" : {
            "accesses" : 1563,
            "hits" : 1563,
            "misses" : 0,
            "resets" : 0,
            "missRatio" : 0
        }
    },
    "backgroundFlushing" : {
        "flushes" : 2,
        "total_ms" : 44,

        "average_ms" : 22,
        "last_ms" : 36,
        "last_finished" : "Thu Jun 10 2010 15:46:54 GMT-0400 (EDT)"
    },
    "opcounters" : {
        "insert" : 38195,
        "query" : 8874,
        "update" : 4058,
        "delete" : 389,
        "getmore" : 888,
        "command" : 17731
    },
    "asserts" : {
        "regular" : 0,
        "warning" : 0,
        "msg" : 0,
        "user" : 5054,
        "rollovers" : 0
    },
    "ok" : true
}
```

> Raw status information can also be retrieved as JSON using the MongoDB HTTP interface, at the */_status* (*http://localhost:28017/_status*) URL: this includes the output of `serverStatus`, as well as the output of some other useful commands. See "Using the Admin Interface" on page 115 for more on the admin interface.

`serverStatus` provides a detailed look at what is going on inside a MongoDB server. Information such as the current server version, uptime (in seconds), and current number of connections is easily available. Some of the other information in the `serverStatus` response might need some explaining, however.

The value for `"globalLock"` gives a quick look at how much time a global write lock has been held on the server (the times are given in microseconds). `"mem"` contains information on how much data the server has memory mapped and what the virtual and resident memory sizes are for the server process (all in megabytes). `"indexCounters"` gives information on the number of B-Tree lookups that have had to go to disk (`"misses"`) versus successful lookups from memory (`"hits"`)—if this ratio starts to increase you should consider adding more RAM, or system performance might suffer. `"backgroundFlushing"` tells us how many background `fsync`s have been performed and how long they've taken. One of the most important pieces of the response is the `"opcounters"` document, which contains counters for each of the major operation types. Finally, `"asserts"` counts any assertions that have occurred on the server.

All of the counters in the `serverStatus` output are tracked from the time the server was started and will eventually roll over if the counts get high enough. When a rollover occurs for any counter, all counters will roll over, and the value of `"rollovers"` in the `"asserts"` document will increment.

mongostat

Although powerful, `serverStatus` is not exactly a user-friendly mechanism for monitoring server health and performance. Fortunately, MongoDB distributions also ship with `mongostat`, which puts a friendly face on the output of `serverStatus`.

`mongostat` prints some of the most important information available from `serverStatus`. It prints a new line every second, which gives a more real-time view to the static counters we saw previously. The columns printed by `mongostat` have names like *inserts/s*, *commands/s*, *vsize*, and *% locked*, each of which corresponds exactly to data available in `serverStatus`.

Third-Party Plug-Ins

Most administrators are probably already using monitoring packages to keep track of their servers, and the presence of `serverStatus` and the */_status* URL make it pretty easy to write a MongoDB plug-in for any such tool. At the time of this writing, MongoDB plug-ins exist for at least Nagios, Munin, Ganglia, and Cacti. For an up-to-date list of third-party plug-ins, check out the MongoDB documentation on monitoring tools.

Security and Authentication

One of the first priorities for any systems administrator is to ensure their systems are secure. The best way to handle security with MongoDB is to run it in a trusted environment, ensuring that only trusted machines are able to connect to the server. That said, MongoDB supports per connection authentication, albeit with a pretty coarse-grained permissions scheme.

Authentication Basics

Each database in a MongoDB instance can have any number of users. When security is enabled, only authenticated users of a database are able to perform read or write operations on it. In the context of authentication, MongoDB treats one database as special: *admin*. A user in the *admin* database can be thought of as a superuser. After authenticating, admin users are able to read or write from *any* database and are able to perform certain admin-only commands, like `listDatabases` or `shutdown`.

Before starting the database with security turned on, it's important that at least one admin user has been added. Let's run through a quick example, starting from a shell connected to a server without security turned on:

```
> use admin
switched to db admin
> db.addUser("root", "abcd");
{
    "user" : "root",
    "readOnly" : false,
    "pwd" : "1a0f1c3c3aa1d592f490a2addc559383"
}
> use test
switched to db test
> db.addUser("test_user", "efgh");
{
    "user" : "test_user",
    "readOnly" : false,
    "pwd" : "6076b96fc3fe6002c810268702646eec"
}
> db.addUser("read_only", "ijkl", true);
{
    "user" : "read_only",
    "readOnly" : true,
    "pwd" : "f497e180c9dc0655292fee5893c162f1"
}
```

Here we've added an admin user, root, and two users on the *test* database. One of those users, read_only, has read permissions only and cannot write to the database. From the shell, a read-only user is created by passing `true` as the third argument to `addUser`. To call `addUser`, you must have write permissions for the database in question; in this case we can call `addUser` on any database because we have not enabled security yet.

> The `addUser` method is useful for more than just adding new users: it can be used to change a user's password or read-only status. Just call `addUser` with the username and a new password or read-only setting for the user.

Now let's restart the server, this time adding the `--auth` command-line option to enable security. After enabling security, we can reconnect from the shell and try it:

```
> use test
switched to db test
> db.test.find();
error: { "$err" : "unauthorized for db [test] lock type: -1 " }
> db.auth("read_only", "ijkl");
1
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
> db.test.insert({"x" : 2});
unauthorized
> db.auth("test_user", "efgh");
1
> db.test.insert({"x": 2});
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
{ "_id" : ObjectId("4bb0088cbe17157d7b9cac07"), "x" : 2 }
> show dbs
assert: assert failed : listDatabases failed:{
    "assertion" : "unauthorized for db [admin] lock type: 1
",
    "errmsg" : "db assertion failure",
    "ok" : 0
}
> use admin
switched to db admin
> db.auth("root", "abcd");
1
> show dbs
admin
local
test
```

When we first connect, we are unable to perform any operations (read or write) on the *test* database. After authenticating as the read_only user, however, we are able to perform a simple `find`. When we try to insert data, we are again met with a failure because of the lack of authorization. test_user, which was not created as read-only, is able to insert data normally. As a nonadmin user, though, test_user is not able to list all of the available databases using the `show dbs` helper. The final step is to authenticate as an admin user, root, who is able to perform operations of any kind on any particular database.

How Authentication Works

Users of a given database are stored as documents in its *system.users* collection. The structure of a user document is {`"user" : `*`username`*`, "readOnly" : `*`true`*`, "pwd" : `*`password hash`*}. The *`password hash`* is a hash based on the *`username`* and password chosen.

Knowing where and how user information is stored makes performing some common administration tasks trivial. For example, to remove a user, simply remove the user document from the *system.users* collection:

```
> db.auth("test_user", "efgh");
1
> db.system.users.remove({"user" : "test_user"});
> db.auth("test_user", "efgh");
0
```

When a user authenticates, the server keeps track of that authentication by tying it to the connection used for the `authenticate` command. This means that if a driver or tool is employing connection pooling or fails over to another node, any authenticated users will need to reauthenticate on any new connections. Some drivers may be capable of handling this transparently, but if not, it will need to be done manually. If that is the case, then it might be best to avoid using `--auth` altogether (again, by deploying MongoDB in a trusted environment and handling authentication on the client side).

Other Security Considerations

There are a couple of options besides authentication that should be considered when locking down a MongoDB instance. First, even when using authentication, the MongoDB wire protocol is not encrypted. If that is a requirement, consider using SSH tunneling or another similar mechanism to encrypt traffic between clients and the MongoDB server.

We suggest always running your MongoDB servers behind a firewall or on a network accessible only through your application servers. If you do have MongoDB on a machine accessible to the outside world, however, it is recommended that you start it with the `--bindip` option, which allows you to specify a local IP address that `mongod` will be bound to. For instance, to only allow connections from an application server running on the same machine, you could run `mongod --bindip localhost`.

As documented in the section "Using the Admin Interface" on page 115, by default MongoDB starts up a very simple HTTP server that allows you to see information about current operations, locking, and replication from your browser. If you don't want this information exposed, you can turn off the admin interface by using the `--nohttpinterface` option.

Finally, you can entirely disallow server-side JavaScript execution by starting the database with `--noscripting`.

Backup and Repair

Taking backups is an important administrative task with any data storage system. Often, doing backups properly can be tricky, and the only thing worse than not taking backups at all is taking them incorrectly. Luckily, MongoDB has several different options that make taking backups a painless process.

Data File Backup

MongoDB stores all of its data in a *data directory*. By default, this directory is */data/db/* (or *C:\data\db* on Windows). The directory to use as the data directory is configurable through the `--dbpath` option when starting MongoDB. Regardless of where the data directory is, its contents form a complete representation of the data stored in MongoDB. This suggests that making a backup of MongoDB is as simple as creating a copy of all of the files in the data directory.

> It is not safe to create a copy of the data directory while MongoDB is running unless the server has done a full `fsync` and is not allowing writes. Such a backup will likely turn out to be corrupt and need repairing (see the section "Repair" on page 124).

Because it is not safe in general to copy the data directory while MongoDB is running, one option for taking a backup is to shut down the MongoDB server and then copy the data directory. Assuming the server is shut down safely (see the section "Starting and Stopping MongoDB" on page 111), the data directory will represent a safe snapshot of the data stored when it was shut down. That directory can be copied as a backup before restarting the server.

Although shutting down the server and copying the data directory is an effective and safe method of taking backups, it is not ideal. In the remainder of this chapter, we'll look at techniques for backing up MongoDB without requiring any downtime.

mongodump and mongorestore

One method for backing up a running instance of MongoDB is to use the `mongodump` utility that is included with all MongoDB distributions. `mongodump` works by querying against a running MongoDB server and writing all of the documents it contains to disk. Because `mongodump` is just a regular client, it can be run against a live instance of MongoDB, even one handling other requests and performing writes.

> Because `mongodump` operates using the normal MongoDB query mechanism, the backups it produces are not necessarily point-in-time snapshots of the server's data. This is especially evident if the server is actively handling writes during the course of the backup.
>
> Another consequence of the fact that `mongodump` acts through the normal query mechanism is that it can cause some performance degradation for other clients throughout the duration of the backup.

Like most of the command-line tools included with MongoDB, we can see the options available for `mongodump` by running with the `--help` option:

```
$ ./mongodump --help
options:
  --help                    produce help message
  -v [ --verbose ]          be more verbose (include multiple times for more
                            verbosity e.g. -vvvvv)
  -h [ --host ] arg         mongo host to connect to ("left,right" for pairs)
  -d [ --db ] arg           database to use
  -c [ --collection ] arg   collection to use (some commands)
  -u [ --username ] arg     username
  -p [ --password ] arg     password
  --dbpath arg              directly access mongod data files in the given path,
                            instead of connecting to a mongod instance - needs
                            to lock the data directory, so cannot be used if a
                            mongod is currently accessing the same path
  --directoryperdb          if dbpath specified, each db is in a separate
                            directory
  -o [ --out ] arg (=dump) output directory
```

Along with `mongodump`, MongoDB distributions include a corresponding tool for restoring data from a backup, `mongorestore`. `mongorestore` takes the output from running `mongodump` and inserts the backed-up data into a running instance of MongoDB. The following example session shows a hot backup of the database *test* to the *backup* directory, followed by a separate call to `mongorestore`:

```
$ ./mongodump -d test -o backup
connected to: 127.0.0.1
DATABASE: test   to     backup/test
    test.x to backup/test/x.bson
         1 objects
$ ./mongorestore -d foo --drop backup/test/
connected to: 127.0.0.1
backup/test/x.bson
     going into namespace [foo.x]
     dropping
     1 objects
```

In the previous example, we use `-d` to specify a database to restore to, in this case *foo*. This option allows us to restore a backup to a database with a different name than the original. We also use the `--drop` option, which will drop the collection (if it exists) before restoring data to it. Otherwise, the data will be merged into any existing collection, possibly overwriting some documents. Again, for a complete list of options, run `mongorestore --help`.

fsync and Lock

Although `mongodump` and `mongorestore` allow us to take backups without shutting down the MongoDB server, we lose the ability to get a point-in-time view of the data. MongoDB's `fsync` command allows us to copy the data directory of a running MongoDB server without risking any corruption.

The `fsync` command will force the MongoDB server to flush all pending writes to disk. It will also, optionally, hold a lock preventing any further writes to the database until the server is unlocked. This write lock is what allows the `fsync` command to be useful for backups. Here is an example of how to run the command from the shell, forcing an `fsync` and acquiring a write lock:

```
> use admin
switched to db admin
> db.runCommand({"fsync" : 1, "lock" : 1});
{
    "info" : "now locked against writes, use db.$cmd.sys.unlock.findOne() to unlock",
    "ok" : 1
}
```

At this point, the data directory represents a consistent, point-in-time snapshot of our data. Because the server is locked for writes, we can safely make a copy of the data directory to use as a backup. This is especially useful when running on a snapshotting filesystem, like LVM* or EBS†, where taking a snapshot of the data directory is a fast operation.

After performing the backup, we need to unlock the database again:

```
> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
> db.currentOp();
{ "inprog" : [ ] }
```

Here we run the `currentOp` command to ensure that the lock has been released. (It may take a moment after the unlock is first requested.)

The `fsync` command allows us to take very flexible backups, without shutting down the server or sacrificing the point-in-time nature of the backup. The price we've paid, however, is a momentary block against write operations. The only way to have a point-in-time snapshot without any downtime for reads *or* writes is to backup from a slave.

Slave Backups

Although the options discussed earlier provide a wide range of flexibility in terms of backups, none is as flexible as backing up from a slave server. When running MongoDB with replication (see Chapter 9), any of the previously mentioned backup techniques can be applied to a slave server rather than the master. The slave will always have a copy of the data that is nearly in sync with the master. Because we're not depending on the performance of the slave or its availability for reads or writes, we are free to use any of the three options above: shutting down, the dump and restore tools, or the `fsync` command. Backing up from a slave is the recommended way to handle data backups with MongoDB.

Repair

We take backups so that when a disaster occurs, which could be anything from a power failure to an elephant on the loose in the data center, our data is safe. There will unfortunately always be cases when a server with no backups (or slaves to failover to) fails. In the case of a power failure or a software crash, the disk will be fine when the machine comes back up. Because of the way MongoDB stores data, however, we are not guaranteed that the data on the disk is OK to use: corruption might have occurred (see Appendix C for more on MongoDB's storage engine). Luckily, MongoDB has built-in repairing functionality to attempt to recover corrupt data files.

* A logical volume manager for Linux

† Amazon's Elastic Block Store

A repair should be run after any unclean shutdown of MongoDB. If an unclean shutdown has occurred, you'll be greeted with the following warning when trying to start the server back up:

```
**************
old lock file: /data/db/mongod.lock. probably means unclean shutdown
recommend removing file and running --repair
see: http://dochub.mongodb.org/core/repair for more information
*************
```

The easiest way to repair all of the databases for a given server is to start up a server with `--repair`: `mongod --repair`. The underlying process of repairing a database is actually pretty easy to understand: all of the documents in the database are exported and then immediately imported, ignoring any that are invalid. After that is complete, all indexes are rebuilt. Understanding this mechanism explains some of the properties of repair. It can take a long time for large data sets, because all of the data is validated and all indexes are rebuilt. Repairing can also leave a database with fewer documents than it had before the corruption originally occurred, because any corrupt documents are simply ignored.

> Repairing a database will also perform a compaction. Any extra free space (which might exist after dropping large collections or removing large number of documents, for example) will be reclaimed after a repair.

To repair a single database on a running server, you can use the `repairDatabase` method from the shell. If we wanted to repair the database *test*, we would do the following:

```
> use test
switched to db test
> db.repairDatabase()
{ "ok" : 1 }
```

To do the same from a driver rather than the shell, issue the `repairDatabase` command, `{"repairDatabase" : 1}`.

Repairing to eliminate corruption should be treated as a last resort. The most effective way to manage data is to always stop the MongoDB server cleanly, use replication for failover, and take regular backups.

CHAPTER 9

Replication

Perhaps the most important job of any MongoDB administrator is making sure that replication is set up and functioning correctly. Use of MongoDB's replication functionality is always recommended in production settings, especially since the current storage engine does not provide single-server durability (see Appendix C for details). Replicas can be used purely for failover and data integrity, or they can be used in more advanced ways, such as for scaling out reads, taking hot backups, or as a data source for offline batch processing. In this chapter, we'll cover everything you need to know about replication.

Master-Slave Replication

Master-slave replication is the most general replication mode supported by MongoDB. This mode is very flexible and can be used for backup, failover, read scaling, and more (see Figures 9-1 and 9-2).

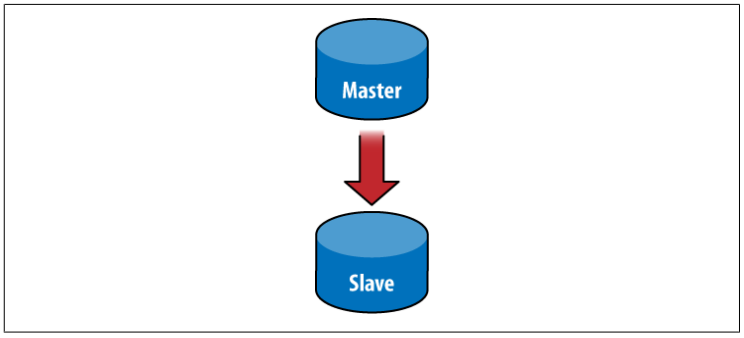


Figure 9-1. A master with one slave

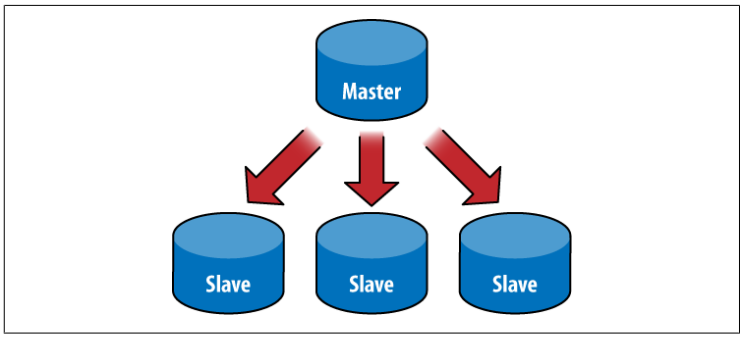


Figure 9-2. A master with three slaves

The basic setup is to start a master node and one or more slave nodes, each of which knows the address of the master. To start the master, run `mongod --master`. To start a slave, run `mongod --slave --source` *`master_address`*, where *`master_address`* is the address of the master node that was just started.

It is simple to try this on a single machine, although in production you would use multiple servers. First, create a directory for the master to store data in and choose a port (10000):

```
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000 --master
```

Now set up the slave, choosing a different data directory and port. For a slave, you also need to tell it who its master is with the `--source` option:

```
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001 --slave --source localhost:10000
```

All slaves must be replicated from a master node. There is currently no mechanism for replicating from a slave (*daisy chaining*), because slaves do not keep their own oplog (see "How It Works" on page 138 for more on the oplog).

There is no explicit limit on the number of slaves in a cluster, but having a thousand slaves querying a single master will likely overwhelm the master node. In practice, clusters with less than a dozen slaves tend to work well.

Options

There are a few other useful options in conjunction with master-slave replication:

--only
: Use on a slave node to specify only a single database to replicate. (The default is to replicate all databases.)

--slavedelay
: Use on a slave node to add a delay (in seconds) to be used when applying operations from the master. This makes it easy to set up delayed slaves, which can be useful in case a user accidentally deletes important documents or inserts bad data. Either of those operations will be replicated to all slaves. By delaying the application of operations, you have a window in which recovery from the bad operation is possible.

--fastsync
: Start a slave from a snapshot of the master node. This option allows a slave to bootstrap much faster than doing a full sync, if its data directory is initialized with a snapshot of the master's data.

--autoresync
: Automatically perform a full resync if this slave gets out of sync with the master (see "How It Works" on page 138).

--oplogSize
: Size (in megabytes) for the master's oplog (see "How It Works" on page 138 for more on the oplog).

Adding and Removing Sources

You can specify a master by starting your slave with the `--source` option, but you can also configure its source(s) from the shell.

Suppose we have a master at `localhost:27017`. We could start a slave without any source and then add the master to the *sources* collection:

```
$ ./mongod --slave --dbpath ~/dbs/slave --port 27018
```

Now we can add `localhost:27017` as a source for our slave by starting the shell and running the following:

```
> use local
> db.sources.insert({"host" : "localhost:27017"})
```

If you watch the slave's log, you can see it sync to `localhost:27017`.

If we do a `find` on the *sources* collection immediately after inserting the source, it will show us the document we inserted:

```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017"
}
```

Once the slave's log shows that it has finished syncing, the document will be updated to reflect this:

```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017",
    "source" : "main",
    "syncedTo" : {
        "t" : 1276530906000,
        "i" : 1
    },
    "localLogTs" : {
        "t" : 0,
        "i" : 0
    },
    "dbsNextPass" : {
        "test_db" : true
    }
}
```

Now, suppose we are going into production and we want to change the slave's configuration such that it slaves off of `prod.example.com`. We can change the source for the slave using `insert` and `remove`:

```
> db.sources.insert({"host" : "prod.example.com:27017"})
> db.sources.remove({"host" : "localhost:27017"})
```

As you can see, *sources* can be manipulated like a normal collection and provides a great deal of flexibility for managing slaves.

> If you slave off of two different masters with the same collections, MongoDB will attempt to merge them, but correctly doing so is not guaranteed. If you are using a single slave with multiple different masters, it is best to make sure the masters use different namespaces.

Replica Sets

A *replica set* is basically a master-slave cluster with automatic failover. The biggest difference between a master-slave cluster and a replica set is that a replica set does not have a single master: one is elected by the cluster and may change to another node if the current master goes down. However, they look very similar: a replica set always has a single master node (called a *primary*) and one or more slaves (called *secondaries*). See Figures 9-3, 9-4, and 9-5.

The nice thing about replica sets is how automatic everything is. First, the set itself does a lot of the administration for you, promoting slaves automatically and making sure you won't run into inconsistencies. For a developer, they are easy to use: you specify a few servers in a set, and the driver will automatically figure out all of the servers in the set and handle failover if the current master dies.

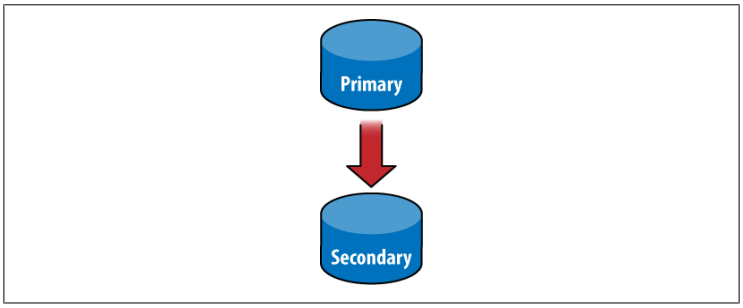


Figure 9-3. A replica set with two members

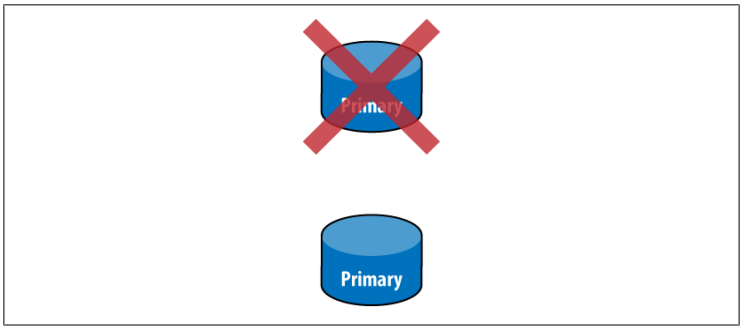


Figure 9-4. When the primary server goes down, the secondary server will become master

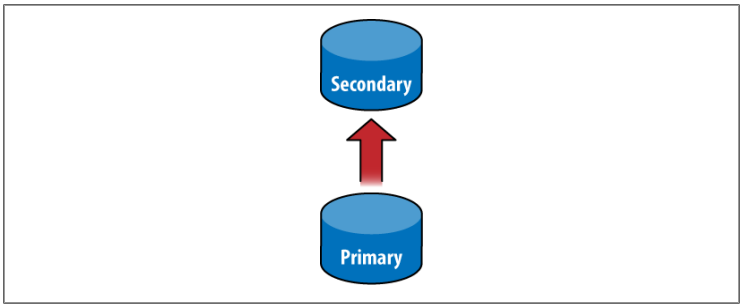


Figure 9-5. If the original primary comes back up, it will begin slaving off of the new primary

Initializing a Set

Setting up a replica set is a little more involved than setting up a master-slave cluster. We'll just start out by setting up the smallest set possible: two servers.

> You cannot specify `localhost` addresses as members, so you need to figure out what the hostname is of your machine. On *NIX, this can be done with the following:
>
> ```
> $ cat /etc/hostname
> morton
> ```

First, we create our data directories and choose ports for each server:

```
$ mkdir -p ~/dbs/node1 ~/dbs/node2
```

We have one more decision to make before we start up the servers: we must choose a *name* for this replica set. This name makes it easy to refer to the set as a whole and distinguish between sets. We'll call our replica set `"blort"`.

Now we can actually start up the servers. The only new option is `--replSet`, which lets the server know that it's a member of the `replSet` `"blort"` that contains another member at `morton:10002` (which hasn't been started yet):

```
$ ./mongod --dbpath ~/dbs/node1 --port 10001 --replSet blort/morton:10002
```

We start up the other server in the same way:

```
$ ./mongod --dbpath ~/dbs/node2 --port 10002 --replSet blort/morton:10001
```

If we wanted to add a third server, we could do so with either of these commands:

```
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001,morton:10002
```

One of the nice things about replica sets is that they are self-detecting: you can specify a single server in the set, and MongoDB will figure out and connect to the rest of the nodes automatically.

Once you have a few servers up, you'll notice that the server logs are complaining about the replica set not being initialized. This is because there's one more step: initializing the set in the shell.

Connect to one of the servers (we use `morton:10001` in the following example) with the shell. Initializing the set is a database command that has to be run only once:

```
$ ./mongo morton:10001/admin
MongoDB shell version: 1.5.3
connecting to localhost:10001/admin
type "help" for help
> db.runCommand({"replSetInitiate" : {
... "_id" : "blort",
... "members" : [
...     {
...         "_id" : 1,
...         "host" : "morton:10001"
...     },
...     {
...         "_id" : 2,
...         "host" : "morton:10002"
...     }
... ]}})
{
        "info" : "Config now saved locally.  Should come online in about a minute.",
        "ok" : true
}
```

The initialization document is a bit complicated, but going through it key by key should make sense:

`"_id" : "blort"`
: The name of this set.

`"members" : [...]`
: A list of servers in the set. You can add more later. Each server document has (at least) two keys:

  `"_id" : N`
  : Each server needs a unique ID.

  `"host" : hostname`
  : This is the key that actually specifies the host.

Now you should see some log messages about which server is being elected primary.

If we connect to the other server and do a `find` on the *local.system.replset* namespace, you can see that the configuration has been propagated to the other server in the set.

> At the time of this writing, replica sets are still under development and have not yet been released in a production version of MongoDB. As such, the information here is subject to change. For the most up-to-date documentation on replica sets, see the MongoDB wiki.

Nodes in a Replica Set

At any point in time, one node in the cluster is *primary*, and the rest are *secondary*. The primary node is essentially the master, the difference being that which node is designated as primary can vary over time.

There are several different types of nodes that can coexist in a replica set:

standard
: This is a regular replica set node. It stores a full copy of the data being replicated, takes part in voting when a new primary is being elected, and is capable of becoming the primary node in the set.

passive
: Passive nodes store a full copy of the data and participate in voting but will never become the primary node for the set.

arbiter
: An arbiter node participates only in voting; it does not receive any of the data being replicated and cannot become the primary node.

The difference between a standard node and a passive node is actually more of a sliding scale; each *participating node* (nonarbiter) has a *priority* setting. A node with priority `0` is passive and will never be selected as primary. Nodes with nonzero priority will be selected in order of decreasing priority, using freshness of data to break ties between nodes with the same priority. So, in a set with two priority `1` nodes and a priority `0.5` node, the third node will be elected primary only if neither of the priority `1` nodes are available.

Standard and passive nodes can be configured as part of a node's description, using the `priority` key:

```
> members.push({
... "_id" : 3,
... "host" : "morton:10003",
... "priority" : 40
... });
```

The default priority is 1, and priorities must be between 0 and 1000 (inclusive).

Arbiters are specified using the `"arbiterOnly"` key.

```
> members.push({
... "_id" : 4,
... "host" : "morton:10004",
... "arbiterOnly" : true
... });
```

There is more information about arbiters in the next section.

Secondary nodes will pull from the primary node's oplog and apply operations, just like a slave in a master-slave system. A secondary node will also write the operation to its own local oplog, however, so that it is capable of becoming the primary. Operations in the oplog also include a monotonically increasing ordinal. This ordinal is used to determine how up-to-date the data is on any node in the cluster.

Failover and Primary Election

If the current primary fails, the rest of the nodes in the set will attempt to elect a new primary node. This election process will be initiated by any node that cannot reach the primary. The new primary must be elected by a *majority* of the nodes in the set. Arbiter nodes participate in voting as well and are useful for breaking ties (e.g., when the participating nodes are split into two halves separated by a network partition). The new primary will be the node with the highest priority, using freshness of data to break ties between nodes with the same priority (see Figures 9-6, 9-7, and 9-8).

The primary node uses a heartbeat to track how many nodes in the cluster are visible to it. If this falls below a majority, the primary will automatically fall back to secondary status. This prevents the primary from continuing to function as such when it is separated from the cluster by a network partition.

Whenever the primary changes, the data on the new primary is assumed to be the most up-to-date data in the system. Any operations that have been applied on any other nodes (i.e., the former primary node) will be rolled back, even if the former primary comes back online. To accomplish this rollback, all nodes go through a resync process when connecting to a new primary. They look through their oplog for operations that have not been applied on the primary and query the new primary to get an up-to-date copy of any documents affected by such operations. Nodes that are currently in the process of resyncing are said to be *recovering* and will not be eligible for primary election until the process is complete.

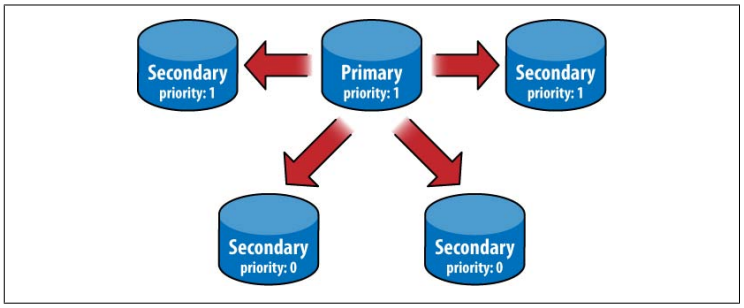


Figure 9-6. A replica set can have several servers of different priority levels

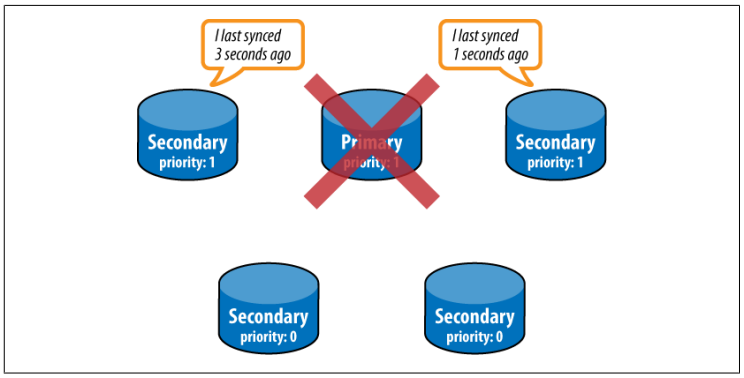


Figure 9-7. If the primary goes down, the highest-priority servers will compare how up-to-date they are

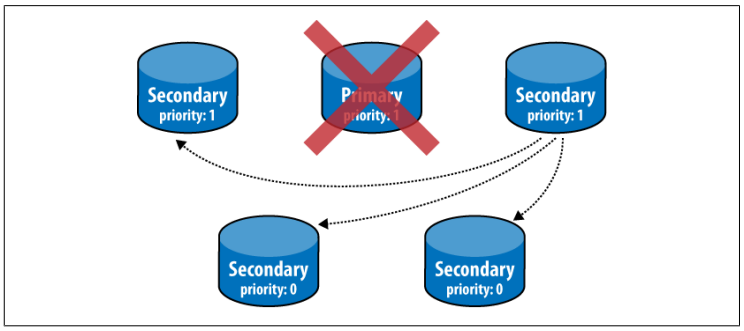


Figure 9-8. The highest-priority most-up-to-date server will become the new primary

Performing Operations on a Slave

The primary purpose and most common use case of a MongoDB slave is to function as failover mechanism in the case of data loss or downtime on the master node. There are other valid use cases for a MongoDB slave, however. A slave can be used as a source for taking backups (see Chapter 8). It can also be used for scaling out reads or for performing data processing jobs on.

Read Scaling

One way to scale reads with MongoDB is to issue queries against slave nodes. By issuing queries on slaves, the workload for the master is reduced. In general, this is a good approach to scaling when your workload is read heavy—if you have a more write-intensive workload, see Chapter 10 to learn how to scale with autosharding.

> One important note about using slaves to scale reads in MongoDB is that replication is asynchronous. This means that when data is inserted or updated on the master, the data on the slave will be out-of-date momentarily. This is important to consider if you are serving some requests using queries to slaves.

Scaling out reads with slaves is easy: just set up master-slave replication like usual, and make connections directly to the slave servers to handle queries. The only trick is that there is a special query option to tell a slave server that it is allowed to handle a query. (By default, queries will not be executed on a slave.) This option is called `slaveOkay`, and all MongoDB drivers provide a mechanism for setting it. Some drivers also provide facilities to automate the process of distributing queries to slaves—this varies on a per-driver basis, however.

Using Slaves for Data Processing

Another interesting technique is to use slaves as a mechanism for offloading intensive processing or aggregation to avoid degrading performance on the master. To do this, start a normal slave, but with the addition of the `--master` command-line argument. Starting with both `--slave` and `--master` may seem like a bit of a paradox. What it means, however, is that you'll be able to write to the slave, query on it like usual, and basically treat it like you would a normal MongoDB master node. In addition, the slave will continue to replicate data from the actual master. This way, you can perform blocking operations on the slave without ever affecting the performance of the master node.

> When using this technique, you should be sure never to write to any database on the slave that is being replicated from the master. The slave will not revert any such writes in order to properly mirror the master.
>
> The slave should also not have any of the databases that are being replicated when it first starts up. If it does, those databases will not ever be fully synced but will just update with new operations.

How It Works

At a very high level, a replicated MongoDB setup always consists of at least two servers, or nodes. One node is the master and is responsible for handling normal client requests. The other node(s) is a slave and is responsible for mirroring the data stored on the master. The master keeps a record of all operations that have been performed on it. The slave periodically polls the master for any new operations and then performs them on its copy of the data. By performing all of the same operations that have been performed on the master node, the slave keeps its copy of the data up-to-date with the master's.

The Oplog

The record of operations kept by the master is called the *oplog*, short for operation log. The oplog is stored in a special database called *local*, in the *oplog.$main* collection. Each document in the oplog represents a single operation performed on the master server. The documents contain several keys, including the following:

ts
: *Timestamp* for the operation. The *timestamp* type is an internal type used to track when operations are performed. It is composed of a 4-byte timestamp and a 4-byte incrementing counter.

op
: Type of operation performed as a 1-byte code (e.g., "i" for an insert).

ns
: Namespace (collection name) where the operation was performed.

o
: Document further specifying the operation to perform. For an insert, this would be the document to insert.

One important note about the oplog is that it stores only operations that change the state of the database. A query, for example, would not be stored in the oplog. This makes sense because the oplog is intended only as a mechanism for keeping the data on slaves in sync with the master.

The operations stored in the oplog are also not exactly those that were performed on the master server itself. The operations are transformed before being stored such that they are idempotent. This means that operations can be applied multiple times on a slave with no ill effects, so long as the operations are applied in the correct order (e.g., an incrementing update, using `"$inc"`, will be transformed to a `"$set"` operation).

A final important note about the oplog is that it is stored in a capped collection (see "Capped Collections" on page 97). As new operations are stored in the oplog, they will automatically replace the oldest operations. This guarantees that the oplog does not grow beyond a preset bound. That bound is configurable using the `--oplogSize` option

when starting the server, which allows you to specify the size of the oplog in megabytes. By default, 64-bit instances will use 5 percent of available free space for the oplog. This space will be allocated in the *local* database and will be preallocated when the server starts.

Syncing

When a slave first starts up, it will do a full sync of the data on the master node. The slave will copy every document from the master node, which is obviously an expensive operation. After the initial sync is complete, the slave will begin querying the master's oplog and applying operations in order to stay up-to-date.

If the application of operations on the slave gets too far behind the actual operations being performed on the master, the slave will fall *out of sync*. An out-of-sync slave is unable to continue to apply operations to catch up to the master, because every operation in the master's oplog is too "new." This could happen if the slave has had downtime or is busy handling reads. It can also happen following a full sync, if the sync takes long enough that the oplog has rolled over by the time it is finished.

When a slave gets out of sync, replication will halt, and the slave will need to be fully resynced from the master. This resync can be performed manually by running the command `{"resync" : 1}` on the slave's *admin* database or automatically by starting the slave with the `--autoresync` option. Either way, doing a resync is a very expensive operation, and it's a situation that is best avoided by choosing a large enough oplog size.

To avoid out of sync slaves, it's important to have a large oplog so that the master can store a long history of operations. A larger oplog will obviously use up more disk space, but in general this is a good trade-off to make (hence the default oplog size of 5 percent of free space). For more information on sizing the oplog, see "Administration" on page 141.

Replication State and the Local Database

The *local database* is used for all internal replication state, on both the master and the slave. The local database's name is `local`, and its contents will never be replicated. Thus, the local database is guaranteed to be local to a single MongoDB server.

> Use of the local database isn't limited to MongoDB internals. If you have documents that you don't want to replicate, just store them in a collection in the local database.

Other replication state stored on the master includes a list of its slaves. (Slaves perform a handshake using the `handshake` command when they connect to the master.) This list is stored in the *slaves* collection:

```
> db.slaves.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567c"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
{ "_id" : ObjectId("4c12873Oe6e5c3096f40e0de"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" : 1 } }
```

Slaves also store state in the local database. They store a unique slave identifier in the `me` collection, and a list of *sources*, or nodes, that they are slaving from, in the `sources` collection:

```
> db.sources.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567b"), "host" : "localhost:27017",
  "source" : "main", "syncedTo" : { "t" : 1276283096000, "i" : 1 },
  "localLogTs" : { "t" : 0, "i" : 0 } }
```

Both the master and slave keep track of how up-to-date a slave is, using the timestamp stored in `"syncedTo"`. Each time the slave queries the oplog for new operations, it uses `"syncedTo"` to specify which new operations it needs to apply or to find out if it is out of sync.

Blocking for Replication

MongoDB's `getLastError` command allows developers to enforce guarantees about how up-to-date replication is by using the optional `"w"` parameter. Here we run a `getLastError` that will block until at least *N* servers have replicated the last write operation:

```
> db.runCommand({getLastError: 1, w: N});
```

If *N* is not present or is less than two, the command will return immediately. If *N* is two, the master won't respond to the command until at least one slave has replicated the last operation. (The master itself is included toward *N*.) The master uses the `"syncedTo"` information stored in *local.slaves* to track how up-to-date each slave is.

When specifying `"w"`, `getLastError` takes an additional parameter, `"wtimeout"`, which is a timeout in milliseconds. This allows the `getLastError` to time out and return an error before the last operation has replicated to *N* servers. (By default the command has no timeout.)

Blocking for replication will cause write operations to slow down significantly, particularly for large values of `"w"`. In practice, setting `"w"` to two or three for important operations will yield a good combination of efficiency and safety.

Administration

In this section, we'll introduce some administration concepts that are specific to replication.

Diagnostics

MongoDB includes a couple of useful administrative helpers for inspecting the status of replication. When connected to the master, use the `db.printReplicationInfo` function:

```
> db.printReplicationInfo();
    configured oplog size:   10.48576MB
    log length start to end: 34secs (0.01hrs)
    oplog first event time:  Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
    oplog last event time:   Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
    now:                     Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

This gives information about the size of the oplog and the date ranges of operations contained in the oplog. In this example, the oplog is about 10MB and is only able to fit about 30 seconds of operations. This is almost certainly a case where we should increase the size of the oplog (see the next section). We want the log length to be *at least* as long as the time it takes to do a full resync—that way, we don't run into a case where a slave is already out of sync by the time its initial sync (or resync) is finished.

> The log length is computed by taking the time difference between the first and last operation in the oplog. If the server has just started, then the first operation will be relatively recent. In that case, the log length will be small, even though the oplog probably still has free space available. The log length is a more useful metric for servers that have been operational long enough for the oplog to "roll over."

We can also get some information when connected to the slave, using the `db.printSlaveReplicationInfo` function:

```
> db.printSlaveReplicationInfo();
    source:   localhost:27017
    syncedTo: Tue Mar 30 2010 16:44:01 GMT-0400 (EDT)
    = 12secs ago (0hrs)
```

This will show a list of sources for the slave, each with information about how far behind the master it is. In this case, we are only 12 seconds behind the master.

Changing the Oplog Size

If we find that the oplog size needs to be changed, the simplest way to do so is to stop the master, delete the files for the *local* database, and restart with a new setting for `--oplogSize`. To change the oplog size to *size*, we shut down the master and run the following:

```
$ rm /data/db/local.*
$ ./mongod --master --oplogSize size
```

size is specified in megabytes.

> Preallocating space for a large oplog can be time-consuming and might cause too much downtime for the master node. It is possible to manually preallocate data files for MongoDB if that is the case; see the MongoDB documentation on halted replication for more detailed information.

After restarting the master, any slaves should either be restarted with the `--autoresync` or have a manual resync performed.

Replication with Authentication

If you are using replication in tandem with MongoDB's support for authentication (see "Authentication Basics" on page 118), there is some additional configuration that needs to be performed to allow the slave to access the data on the master. On both the master and the slave, a user needs to be added to the *local* database, with the same username and password on each node. Users on the *local* database are similar to users on *admin*; they have full read and write permissions on the server.

When the slave attempts to connect to the master, it will authenticate using a user stored in `local.system.users`. The first username it will try is "repl," but if no such user is found, it will just use the first available user in `local.system.users`. So, to set up replication with authentication, run the following code on both the master *and* any slaves, replacing *password* with a secure password:

```
> use local
switched to db local
> db.add User("repl", password);
{
    "user" : "repl",
    "readOnly" : false,
    "pwd" : "..."
}
```

The slave will then be able to replicate from the master.

CHAPTER 10
Sharding

Sharding is MongoDB's approach to scaling out. Sharding allows you to add more machines to handle increasing load and data size without affecting your application.

Introduction to Sharding

Sharding refers to the process of splitting data up and storing different portions of the data on different machines; the term *partitioning* is also sometimes used to describe this concept. By splitting data up across machines, it becomes possible to store more data and handle more load without requiring large or powerful machines.

Manual sharding can be done with almost any database software. It is when an application maintains connections to several different database servers, each of which are completely independent. The application code manages storing different data on different servers and querying against the appropriate server to get data back. This approach can work well but becomes difficult to maintain when adding or removing nodes from the cluster or in the face of changing data distributions or load patterns.

MongoDB supports *autosharding*, which eliminates some of the administrative headaches of manual sharding. The cluster handles splitting up data and rebalancing automatically. Throughout the rest of this book (and most MongoDB documentation in general), the terms *sharding* and *autosharding* are used interchangeably, but it's important to note the difference between that and manual sharding in an application.

Autosharding in MongoDB

The basic concept behind MongoDB's sharding is to break up collections into smaller *chunks*. These chunks can be distributed across *shards* so that each shard is responsible for a subset of the total data set. We don't want our application to have to know what shard has what data, or even that our data is broken up across multiple shards, so we run a routing process called `mongos` in front of the shards. This router knows where all of the data is located, so applications can connect to it and issue requests normally. As far as the application knows, it's connected to a normal `mongod`. The router, knowing what data is on which shard, is able to forward the requests to the appropriate shard(s). If there are responses to the request, the router collects them and sends them back to the application.

In a nonsharded MongoDB setup, you would have a client connecting to a `mongod` process, like in Figure 10-1. In a sharded setup, like Figure 10-2, the client connects to a `mongos` process, which abstracts the sharding away from the application. From the application's point of view, a sharded setup looks just like a nonsharded setup. There is no need to change application code when you need to scale.

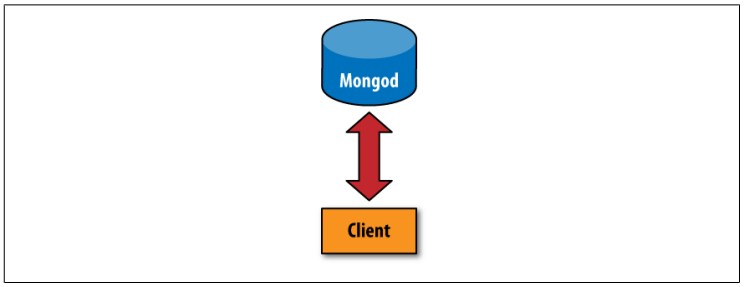


Figure 10-1. Nonsharded client connection

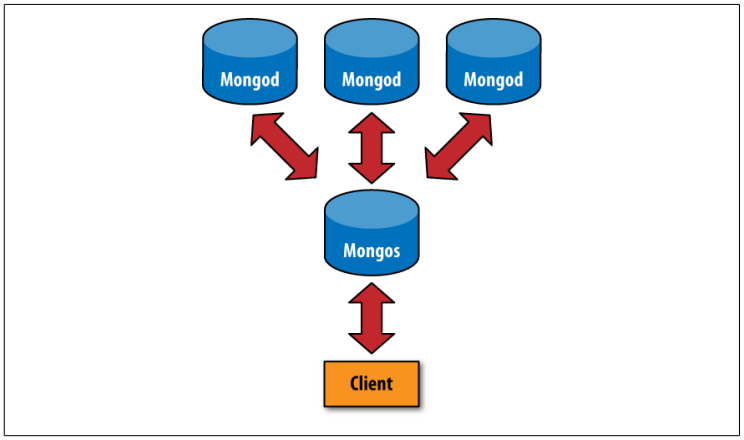


Figure 10-2. Sharded client connection

If you are using an old version of MongoDB, you should upgrade to at least 1.6.0 before using sharding. Sharding has been around for a while, but the first production-ready release is 1.6.0.

When to Shard

One question people often have is when to start sharding. There are a couple of signs that sharding might be a good idea:

- You've run out of disk space on your current machine.
- You want to write data faster than a single `mongod` can handle.
- You want to keep a larger proportion of data in memory to improve performance.

In general, you should start with a nonsharded setup and convert it to a sharded one, if and when you need.

The Key to Sharding: Shard Keys

When you set up sharding, you choose a key from a collection and use that key's values to split up the data. This key is called a *shard key*.

Let's look at an example to see how this works: suppose we had a collection of documents representing people. If we chose `"name"` as our shard key, one shard could hold documents where the `"name"` started with A–F, the next shard could hold names from G–P, and the final shard would hold names from Q–Z. As you added (or removed) shards, MongoDB would rebalance this data so that each shard was getting a balanced amount of traffic and a sensible amount of data (e.g., if a shard is getting a lot of traffic, it might have less data than a shard with data that is less "hot").

Sharding an Existing Collection

Suppose we have an existing collection of logs and we want to shard it. If we enable sharding and tell MongoDB to use `"timestamp"` as the shard key, we'll have a single shard with all of our data. We can insert any data we'd like, and it will all go to that one shard.

Now, suppose we add a new shard. Once this shard is up and running, MongoDB will break up the collection into two pieces, called *chunks*. A chunk contains all of the documents for a range of values for the shard key, so one chunk would have documents with a timestamp value between -∞ and, say, June 26, 2003, and the other chunk would have timestamps between June 27, 2003 and ∞. One of these chunks would then be moved to the new shard.

If we get a new document with a timestamp value before June 27, 2003, we'll add that to the first chunk; otherwise, we'll add the document to the second chunk.

Incrementing Shard Keys Versus Random Shard Keys

The distribution of inserts across shards is very dependent on which key we're sharding on.

If we choose to shard on something like `"timestamp"`, where the value is probably going to increase and not jump around a lot, we'll be sending all of the inserts to one shard (the one with the [June 27, 2003, ∞] chunk). Notice that, if we add a new shard and it splits the data again, we'll still be inserting on just one server. If we add a new shard, MongoDB might split [June 27, 2003, ∞] into [June 27, 2003, December 12, 2010) and [December 12, 2010, ∞]. We'll always have a chunk that will be "some date through infinity," which is where our inserts will be going. This isn't good for a very high write load, but it will make queries on the shard key very efficient.

If we have a high write load and want to evenly distribute writes across multiple shards, we should pick a shard key that will jump around more. This could be a hash of the timestamp in the log example or a key like `"logMessage"`, which won't have any particular pattern to it.

Whether your shard key jumps around or increases steadily, it is important to choose a key that will vary somewhat. If, for example, we had a `"logLevel"` key that had only values `"DEBUG"`, `"WARN"`, or `"ERROR"`, MongoDB won't be able to break up your data into more than three chunks (because there are only three different values). If you have a key with very little variation and want to use it as a shard key anyway, you can do so by creating a compound shard key on that key and a key that varies more, like `"logLevel"` and `"timestamp"`.

Determining which key to shard on and creating shard keys should be reminiscent of indexing, because the two concepts are similar. In fact, often your shard key will just be the index you use most often.

How Shard Keys Affect Operations

To the end user, a sharded setup should be indistinguishable from a nonsharded one. However, it can be useful, especially when setting up sharding, to understand how different queries will be done depending on the shard key chosen.

Suppose we have the collection described in the previous section, which is sharded on the `"name"` key and has three shards with names ranging from A to Z. Different queries will be executed in different ways:

`db.people.find({"name" : "Susan"})`
: `mongos` will send this query directly to the Q–Z shard, receive a response from that shard, and send it to the client.

`db.people.find({"name" : {"$lt" : "L"}})`
: `mongos` will send the query to the A–F and G–P shards in serial. It will forward their responses to the client.

`db.people.find().sort({"email" : 1})`
: `mongos` will query all of the shards and do a merge sort when it gets the results to make sure it is returning them in the correct order. `mongos` uses cursors to retrieve data from each server, so it does not need to get the entire data set in order to start sending batches of results to the client.

`db.people.find({"email" : "joe@example.com"})`
: `mongos` does not keep track of the `"email"` key, so it doesn't know which shard to send this to. Thus, it sends the query to all of the shards in serial.

If we insert a new document, `mongos` will send that document to the appropriate shard, based on the value of its `"name"` key.

Setting Up Sharding

There are two steps to setting up sharding: starting the actual servers and then deciding how to shard your data.

Sharding basically involves three different components working together:

shard
: A shard is a container that holds a subset of a collection's data. A shard is either a single `mongod` server (for development/testing) or a replica set (for production). Thus, even if there are many servers in a shard, there is only one master, and all of the servers contain the same data.

`mongos`
: This is the router process and comes with all MongoDB distributions. It basically just routes requests and aggregates responses. It doesn't store any data or configuration information. (Although it does cache information from the config servers.)

config server
: Config servers store the configuration of the cluster: which data is on which shard. Because `mongos` doesn't store anything permanently, it needs somewhere to get the shard configuration. It syncs this data from the config servers.

If you are working with MongoDB already, you probably have a shard ready to go. (Your current `mongod` can become your first shard.) The following section shows how to create a new shard from scratch, but feel free to use your existing database instead.

Starting the Servers

First we need to start up our config server and `mongos`. The config server needs to be started first, because `mongos` uses it to get its configuration. The config server can be started like any other `mongod` process:

```
$ mkdir -p ~/dbs/config
$ ./mongod --dbpath ~/dbs/config --port 20000
```

A config server does not need much space or resources. (A generous estimate is 1KB of config server space per 200MB of actual data.)

Now you need a `mongos` process for your application to connect to. Routing servers don't even need a data directory, but they need to know where the config server is:

```
$ ./mongos --port 30000 --configdb localhost:20000
```

Shard administration is always done through a `mongos`.

Adding a shard

A shard is just a normal `mongod` instance (or replica set):

```
$ mkdir -p ~/dbs/shard1
$ ./mongod --dbpath ~/dbs/shard1 --port 10000
```

Now we'll connect to the `mongos` process we started and add the shard to the cluster. Start up a shell connected to your `mongos`:

```
$ ./mongo localhost:30000/admin
MongoDB shell version: 1.6.0
url: localhost:30000/admin
connecting to localhost:30000/admin
type "help" for help
>
```

Make sure you're connected to `mongos`, not a `mongod`. Now you can add this shard with the `addshard` database command:

```
> db.runCommand({addshard : "localhost:10000", allowLocal : true})
{
    "added" : "localhost:10000",
    "ok" : true
}
```

The `"allowLocal"` key is necessary only if you are running the shard on `localhost`. MongoDB doesn't want to let you accidentally set up a cluster locally, so this lets it know that you're just in development and know what you're doing. If you're in production, you should have shards on different machines (although there can be some overlap; see the next section for details).

Whenever we want to add a new shard, we can run the `addshard` database command. MongoDB will take care of integrating it into the cluster.

Sharding Data

MongoDB won't just distribute every piece of data you've ever stored: you have to explicitly turn sharding on at both the database and collection levels. Let's look at an example: we'll shard the *bar* collection in the *foo* database on the `"_id"` key. First, we enable sharding for *foo*:

```
> db.runCommand({"enablesharding" : "foo"})
```

Sharding a database results in its collections being stored on different shards and is a prerequisite to sharding one of its collections.

Once you've enabled sharding on the database level, you can shard a collection by running the `shardcollection` command:

```
> db.runCommand({"shardcollection" : "foo.bar", "key" : {"_id" : 1}})
```

Now the collection will be sharded by the `"_id"` key. When we start adding data, it will automatically distribute itself across our shards based on the values of `"_id"`.

Production Configuration

The example in the previous section is fine for trying sharding or for development. However, when you move an application into production, you'll want a more robust setup. To set up sharding with no points of failure, you'll need the following:

- Multiple config servers
- Multiple `mongos` servers
- Replica sets for each shard
- `w` set correctly (see the previous chapter for information on `w` and replication)

A Robust Config

Setting up multiple config servers is simple. As of this writing, you can have one config server (for development) or three config servers (for production).

Setting up multiple config servers is the same as setting up one; you just do it three times:

```
$ mkdir -p ~/dbs/config1 ~/dbs/config2 ~/dbs/config3
$ ./mongod --dbpath ~/dbs/config1 --port 20001
$ ./mongod --dbpath ~/dbs/config2 --port 20002
$ ./mongod --dbpath ~/dbs/config3 --port 20003
```

Then, when you start a `mongos`, you should connect it to all three config servers:

```
$ ./mongos --configdb localhost:20001,localhost:20002,localhost:20003
```

Config servers use two-phase commit, not the normal MongoDB asynchronous replication, to maintain separate copies of the cluster's configuration. This ensures that they always have a consistent view of the cluster's state. It also means that if a single config server is down, the cluster's configuration information will go read-only. Clients are still able to do both reads and writes, but no rebalancing will happen until all of the config servers are back up.

Many mongos

You can also run as many `mongos` processes as you want. One recommended setup is to run a `mongos` process for every application server. That way, each application server can talk to `mongos` locally, and if the server goes down, no one will be trying to talk to a `mongos` that isn't there.

A Sturdy Shard

In production, each shard should be a replica set. That way, an individual server can fail without bringing down the whole shard. To add a replica set as a shard, pass its name and a seed to the `addshard` command.

For example, say we have a replica set named `"foo"` containing a server at *prod.example.com:27017* (among other servers). We could add this set to the cluster with the following:

```
> db.runCommand({"addshard" : "foo/prod.example.com:27017"})
```

If *prod.example.com* goes down, `mongos` will know that it is connected to a replica set and use the new primary for that set.

Physical Servers

This may seem like an overwhelming number of machines: three config servers, at least two `mongod`s per shard, and as many `mongos` processes as you want. However, not everything has to have its own machine. The main thing to avoid is putting an entire component on one machine. For example, avoid putting all three config servers, all of your `mongos` processes, or an entire replica set on one machine. However, a config server and `mongos` processes can happily share a box with a member of a replica set.

Sharding Administration

Sharding information is mostly stored in the *config* database, which can be accessed from any connection to a `mongos` process.

config Collections

All of the code in the following sections assume that you are running a shell connected to a `mongos` process and have already run `use config`.

Shards

You can find a list of shards in the *shards* collection:

```
> db.shards.find()
{ "_id" : "shard0", "host" : "localhost:10000" }
{ "_id" : "shard1", "host" : "localhost:10001" }
```

Each shard is assigned a unique, human-readable `_id`.

Databases

The *databases* collection contains a list of databases that exist on the shards and information about them:

```
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
{ "_id" : "x", "partitioned" : false, "primary" : "shard0" }
{
    "_id" : "test",
    "partitioned" : true,
    "primary" : "shard0",
    "sharded" : {
        "test.foo" : {
            "key" : {"x" : 1},
            "unique" : false
        }
    }
}
```

Every database available is listed here and has some basic information available.

`"_id" : string`
: The `_id` is the database's name.

`"partitioned" : boolean`
: If `"partitioned"` is true, then the `enablesharding` command has been run on this database.

`"primary" : string`
: The value corresponds to a shard `"_id"` and indicates where this database's "home" is. A database always has a home, whether it is sharded. In a sharded setup, a new database will be created on a random shard. This home is where it will start creating data files. If it is sharded, it will use other servers as well, but it will start out on this shard.

Chunks

Chunk information is stored in the *chunks* collection. This is where things get more interesting; you can actually see how your data has been divided up across the cluster:

```
> db.chunks.find()
{
    "_id" : "test.foo-x_MinKey",
    "lastmod" : { "t" : 1276636243000, "i" : 1 },
    "ns" : "test.foo",
    "min" : {
        "x" : { $minKey : 1 }
    },
    "max" : {
        "x" : { $maxKey : 1 }
    },


    "shard" : "shard0"
}
```

This is what a collection with a single chunk will look like: the chunk range goes from -∞ (`MinKey`) to ∞ (`MaxKey`).

Sharding Commands

We've already covered some of the basic commands, such as adding chunks and enabling sharding on a collection. There are a couple more commands that are useful for administering a cluster.

Getting a summary

The `printShardingStatus` function will give you a quick summary of the previous collections:

```
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0", "host" : "localhost:10000" }
      { "_id" : "shard1", "host" : "localhost:10001" }
  databases:
      { "_id" : "admin", "partitioned" : false, "primary" : "config" }
      { "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
      { "_id" : "x", "partitioned" : false, "primary" : "shard0" }
      { "_id" : "test", "partitioned" : true, "primary" : "shard0",
          "sharded" : { "test.foo" : { "key" : { "x" : 1 }, "unique" : false } } }
              test.foo chunks:
                  { "x" : { $minKey : 1 } } -->> { "x" : { $maxKey : 1 } } on : shard0
                  { "t" : 1276636243000, "i" : 1 }
```

Removing a shard

Shards can be removed from a cluster with the `removeshard` command. `removeshard` drains all of the chunks on a given shard to the other shards.

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "started draining" : "localhost:10000",
    "ok" : 1
}
```

As the shard is drained, `removeshard` will give you the status of how much remains on the shard.

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "msg" : "already draining...",
    "remaining" : {
        "chunks" : 39,
        "dbs" : 2
    },
    "ok" : 1
}
```

Finally, when the shard has finished draining, `removeshard` shows that the shard has been successfully removed.

> As of version 1.6.0, if a removed shard was the primary shard for a database, the database has to be manually moved (using the `moveprimary` command):
>
> ```
> > db.runCommand({"moveprimary" : "test", "to" : "localhost:10001"})
> {
>     "primary" : "localhost:10001",
>     "ok" : 1
> }
> ```
>
> This will likely be automated in future releases.

CHAPTER 11
Example Applications

Throughout this text, almost all of the examples have been in JavaScript. This chapter explores using MongoDB with languages that are more likely to be used in a real application.

Chemical Search Engine: Java

The Java driver is the oldest MongoDB driver. It has been used in production for years and is stable and a popular choice for enterprise developers.

We'll be using the Java driver to build a search engine for chemical compounds, heavily inspired by *http://www.chemeo.com*. This search engine has the chemical and physical properties of thousands of compounds on file, and its goal is to make this information fully searchable.

Installing the Java Driver

The Java driver comes as a JAR file that can be downloaded from Github. To install, add the JAR to your classpath.

All of the Java classes you will probably need to use in a normal application are in the `com.mongodb` and `com.mongodb.gridfs` packages. There are a number of other packages included in the .JAR that are useful if you are planning on manipulating the driver's internals or expanding its functionality, but most applications can ignore them.

Using the Java Driver

Like most things in Java, the API is a bit verbose (especially compared to the other languages' APIs). However, all of the concepts are similar to using the shell, and almost all of the method names are identical.

The `com.mongodb.Mongo` class creates a connection to a MongoDB server. You can access a database from the connection and then get a collection from the database:

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;

class ChemSearch {

    public static void main(String[] args) {
        Mongo connection = new Mongo();
        DB db = connection.getDB("search");
        DBCollection chemicals = db.getCollection("chemicals");

        /* ... */
    }
}
```

This will connect to `localhost:27017` and get the `search.chemicals` namespace.

Documents in Java must be instances of `org.bson.DBObject`, an interface that is basically an ordered `java.util.Map`. While there are a few ways to create a document in Java, the simplest one is to use the `com.mongodb.BasicDBObject` class. Thus, creating the document that could be represented by the shell as `{"x" : 1, "y" : "foo"}` would look like this:

```
BasicDBObject doc = new BasicDBObject();
doc.put("x", 1);
doc.put("y", "foo");
```

If we wanted to add an embedded document, such as `"z" : {"hello" : "world"}`, we would create another BasicDBObject and then `put` it in the top-level one:

```
BasicDBObject z = new BasicDBObject();
z.put("hello", "world");

doc.put("z", z);
```

Then we would have the document `{"x" : 1, "y" : "foo", "z" : {"hello" : "world"}}`.

From there, all of the other methods implemented by the Java driver are similar to the shell. For instance, we could say `chemicals.insert(doc)` or `chemicals.find(doc)`. There is full API documentation for the Java driver at *http://api.mongodb.org/java* and some articles on specific areas of interest (concurrency, data types, etc.) at the MongoDB Java Language Center.

Schema Design

The interesting thing about this problem is that there are thousands of possible properties for each chemical, and we want to be able to search for any of them efficiently. Take two simple examples: silicon and silicon nitride. A document representing silicon might look something like this:

```
{
    "name" : "silicon",

    "mw" : 32.1173
}
```

`mw` stands for "molecular weight."

Silicon nitride might have a couple other properties, so its document would look like this:

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH°gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S°gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    }
}
```

MongoDB lets us store chemicals with any number or structure of properties, which makes this application nicely extensible, but there's no efficient way to index it in its current form. To be able to quickly search for any property, we would need to index almost every key! As we learned in Chapter 5, this is a bad idea.

There is a solution. We can take advantage of the fact that MongoDB indexes every element of an array, so we can store all of the properties we want to search for in an array with common key names. For example, with silicon nitride we can add an array just for indexing containing each property of the given chemical:

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH°gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S°gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    },
    "index" : [
        {"name" : "mw", "value" : 42.0922},
        {"name" : "ΔfH°gas", "value" : 372.38},
        {"name" : "S°gas", "value" : 216.81}
    ]
}
```

Silicon, on the other hand, would have a single-element array with just the molecular weight:

```
{
    "name" : "silicon",
    "mw" : 32.1173,
    "index" : [
        {"name" : "mw", "value" : 32.1173}
    ]
}
```

Now, all we need to do is create a compound index on the `"index.name"` and `"index.value"` keys. Then we'll be able to do a fairly quick search through the chemical compounds for any attribute.

Writing This in Java

Going back to our original Java code snippet, we'll create a compound index with the `ensureIndex` function:

```
BasicDBObject index = new BasicDBObject();
index.put("index.name", 1);
index.put("index.value", 1);

chemicals.ensureIndex(index);
```

Creating a document for, say, silicon nitride is not difficult, but it is verbose:

```
public static DBObject createSiliconNitride() {
    BasicDBObject sn = new BasicDBObject();
    sn.put("name", "silicon nitride");
    sn.put("mw", 42.0922);

    BasicDBObject deltafHgas = new BasicDBObject();
    deltafHgas.put("value", 372.38);
    deltafHgas.put("units", "kJ/mol");

    sn.put("ΔfH°gas", deltafHgas);

    BasicDBObject sgas = new BasicDBObject();
    sgas.put("value", 216.81);
    sgas.put("units", "J/mol×K");

    sn.put("S°gas", sgas);

    ArrayList<BasicDBObject> index = new ArrayList<BasicDBObject>();
    index.add(BasicDBObjectBuilder.start()
              .add("name", "mw").add("value", 42.0922).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "ΔfH°gas").add("value", 372.38).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "S°gas").add("value", 216.81).get());

    sn.put("index", index);

    return sn;
}
```

Arrays can be represented by anything that implements `java.util.List`, so we create a `java.util.ArrayList` of embedded documents for the chemical's properties.

Issues

One issue with this structure is that, if we are querying for multiple criteria, search order matters. For example, suppose we are looking for all documents with a molecular weight of less than 1000, a boiling point greater than 0°, and a freezing point of -20°. Naively, we could do this query by concatenating the criteria in an `$all` conditional:

```
BasicDBObject criteria = new BasicDBObject();

BasicDBObject all = new BasicDBObject();

BasicDBObject mw = new BasicDBObject("name", "mw");
mw.put("value", new BasicDBObject("$lt", 1000));

BasicDBObject bp = new BasicDBObject("name", "bp");
bp.put("value", new BasicDBObject("$gt", 0));

BasicDBObject fp = new BasicDBObject("name", "fp");
fp.put("value", -20);

all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
all.put("$elemMatch", fp);
criteria.put("index", new BasicDBObject("$all", all));

chemicals.find(criteria);
```

The problem with this approach is that MongoDB can use an index only for the first item in an `$all` conditional. Suppose there are 1 million documents with a `"mw"` key whose value is less than 1,000. MongoDB can use the index for that part of the query, but then it will have to scan for the boiling and freezing points, which will take a long time.

If we know some of the characteristics of our data, for instance, that there are only 43 chemicals with a freezing point of -20°, we can rearrange the `$all` to do that query first:

```
all.put("$elemMatch", fp);
all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
criteria.put("index", new BasicDBObject("$all", all));
```

Now the database can quickly find those 43 elements and, for the subsequent clauses, has to scan only 43 elements (instead of 1 million). Figuring out a good ordering for arbitrary searches is the real trick of course, of course. This could be done with pattern recognition and data aggregation algorithms that are beyond the scope of this book.

News Aggregator: PHP

We will be creating a basic news aggregation application: users submit links to interesting sites, and other users can comment and vote on the quality of the links (and other

comments). This will involve creating a tree of comments and implementing a voting system.

Installing the PHP Driver

The MongoDB PHP driver is a PHP extension. It is easy to install on almost any system. It should work on any system with PHP 5.1 or newer installed.

Windows install

Look at the output of `phpinfo()` and determine the version of PHP you are running (PHP 5.2 and 5.3 are supported on Windows; 5.1 is not), including VC version, if shown. If you are using Apache, you should use VC6; otherwise, you're probably running a VC9 build. Some obscure Zend installs use VC8. Also notice whether it is thread-safe (usually abbreviated "ts").

While you're looking at `phpinfo()`, make a note of the `extension_dir` value, which is where we'll need to put the extension.

Now that you know what you're looking for, go to Github. Download the package that matches your PHP version, VC version, and thread safety. Unzip the package, and move *php_mongo.dll* to the `extension_dir` directory.

Finally, add the following line to your *php.ini* file:

```
extension=php_mongo.dll
```

If you are running an application server (Apache, WAMPP, and so on), restart it. The next time you start PHP, it will automatically load the Mongo extension.

Mac OS X Install

It is easiest to install the extension through PECL, if you have it available. Try running the following:

```
$ pecl install mongo
```

Some Macs do not, however, come with PECL or the correct PHP libraries to install extensions.

If PECL does not work, you can download binary builds for OS X, available at Github (*http://www.github.com/mongodb/mongo-php-driver/downloads*). Run `php -i` to see what version of PHP you are running and what the value of `extension_dir` is, and then download the correct version. (It will have "osx" in the filename.) Unarchive the extension, and move *mongo.so* to the directory specified by `extension_dir`.

After the extension is installed via either method, add the following line to your *php.ini* file:

```
extension=mongo.so
```

Restart any application server you might have running, and the Mongo extension will be loaded the next time PHP starts.

Linux and Unix install

Run the following:

```
$ pecl install mongo
```

Then add the following line to your *php.ini* file:

```
extension=mongo.so
```

Restart any application server you might have running, and the Mongo extension will be loaded the next time PHP is started.

Using the PHP Driver

The `Mongo` class is a connection to the database. By default, the constructor attempts to connect to a database server running locally on the default port.

You can use the `__get` function to get a database from the connection and a collection from the database (even a subcollection from a collection). For example, this connects to MongoDB and gets the `bar` collection in the `foo` database:

```
<?php

$connection = new Mongo();

$collection = $connection->foo->bar;

?>
```

You can continue chaining getters to access subcollections. For example, to get the `bar.baz` collection, you can say the following:

```
$collection = $connection->foo->bar->baz;
```

Documents are represented by associative arrays in PHP. Thus, something like `{"foo" : "bar"}` in JavaScript could be represented as `array("foo" => "bar")` in PHP. Arrays are also represented as arrays in PHP, which sometimes leads to confusion: `["foo", "bar", "baz"]` in JavaScript is equivalent to `array("foo", "bar", "baz")`.

The PHP driver uses PHP's native types for null, booleans, numbers, strings, and arrays. For all other types, there is a Mongo-prefixed type: `MongoCollection` is a collection, `MongoDB` is a database, and `MongoRegex` is a regular expression. There is extensive documentation in the PHP manual for all of these classes.

Designing the News Aggregator

We'll be creating a simple news aggregator, where users can submit links to interesting stories and other users can vote and comment on them. We will just be covering two aspects of it: creating a tree of comments and handling votes.

To store the submissions and comments, we need only a single collection, *`posts`*. The initial posts linking to some article will look something like the following:

```
{
    "_id" : ObjectId(),
    "title" : "A Witty Title",
    "url" : "http://www.example.com",
    "date" : new Date(),
    "votes" : 0,
    "author" : {
        "name" : "joe",
        "_id" : ObjectId(),
    }
}
```

The comments will be almost identical, but they need a `"content"` key instead of a `"url"` key.

Trees of Comments

There are several different ways to represent a tree in MongoDB; the choice of which representation to use depends on the types of query being performed.

We'll be storing an *array of ancestors* tree: each node will contain an array of its parent, grandparent, and so on. So, if we had the following comment structure:

```
original link
|- comment 1
|   |- comment 3 (reply to comment 1)
|   |- comment 4 (reply to comment 1)
|       |- comment 5 (reply to comment 4)
|- comment 2
|   |- comment 6 (reply to comment 2)
```

then comment 5's array of ancestors would contain the original link's `_id`, comment 1's `_id`, and comment 4's `_id`. Comment 2's ancestors would be the original link's `_id` and comment 2's `_id`. This allows us to easily search for "all comments for link *X*" or "the subtree of comment 2's replies."

This method of storing comments assumes that we are going to have a lot of them and that we might be interested in seeing just parts of a comment thread. If we knew that we always wanted to display all of the comments and there weren't going to be thousands, we could store the entire tree of comments as an embedded document in the submitted link's document.

Using the array of ancestors approach, when someone wants to create a new comment, we need to add a new document to the collection. To create this document, we create a leaf document by linking it to the parent's `"_id"` value and its array of ancestors.

```
function createLeaf($parent, $replyInfo) {
    $child = array(
        "_id" => new MongoId(),
        "content" => $replyInfo['content'],
        "date" => new MongoDate(),
        "votes" => 0,
        "author" => array(
            "name" => $replyInfo['name'],
            "name" => $replyInfo['name'],
        ),
        "ancestors" => $parent['ancestors'],
        "parent" => $parent['_id']
    );

    // add the parent's _id to the ancestors array
    $child['ancestors'][] = $parent['_id'];

    return $child;
}
```

Then we can add the new comment to the *posts* collection:

```
$comment = createLeaf($parent, $replyInfo);

$posts = $connection->news->posts;
$posts->insert($comment);
```

We can get a list of the latest submissions (sans comments) with the following:

```
$cursor = $posts->find(array("ancestors" => array('$size' => 0)));
$cursor = $cursor->sort(array("date" => -1));
```

If someone wants to see the comments for a given post, we can find them all with the following:

```
$cursor = $posts->find(array("ancestors" => $postId));
```

In fact, we can use this query to access any subtree of comments. If the root of the subtree is passed in as `$postId`, every child will contain `$postId` in its ancestor's array and be returned.

To make these queries fast, we should index the `"date"` and `"ancestors"` keys:

```
$pageOfComments = $posts->ensureIndex(array("date" => -1, "ancestors" => 1));
```

Now we can quickly query for the main page, a tree of comments, or a subtree of comments.

Voting

There are many ways of implementing voting, depending on the functionality and information you want: do you allow up and down votes? Will you prevent users from voting more than once? Will you allow them to switch their vote? Do you care *when* people voted, to see if a link is trending? Each of these requires a different solution with far more complex coding than the simplest way of doing it: using `"$inc"`:

```
$posts->update(array("_id" => $postId), array('$inc' => array("votes", 1)));
```

For a controversial or popular link, we wouldn't want people to be able to vote hundreds of times, so we want to limit users to one vote each. A simple way to do this is to add a `"voters"` array to keep track of who has voted on this post, keeping an array of user `"_id"` values. When someone tries to vote, we do an update that checks the user `"_id"` against the array of `"_id"` values:

```
$posts->update(array("_id" => $postId, "voters" => array('$ne' => $userId)),
    array('$inc' => array("votes", 1), '$push' => array("voters" => $userId)));
```

This will work for up to a couple million users. For larger voting pools, we would hit the 4MB limit, and we would have to special-case the most popular links by putting spillover votes into a new document.

Custom Submission Forms: Ruby

MongoDB is a popular choice for Ruby developers, likely because the document-oriented approach meshes well with Ruby's dynamism and flexibility. In this example we'll use the MongoDB Ruby driver to build a framework for custom form submissions, inspired by a *New York Times* blog post about how it uses MongoDB to handle submission forms (*http://open.blogs.nytimes.com/2010/05/25/building-a-better-submission-form/*). For even more documentation on using MongoDB from Ruby, check out the Ruby Language Center.

Installing the Ruby Driver

The Ruby driver is available as a RubyGem, hosted at *http://rubygems.org*. Installation using the gem is the easiest way to get up and running. Make sure you're using an up-to-date version of RubyGems (with `gem update --system`) and then install the *mongo* gem:

```
$ gem install mongo
Successfully installed bson-1.0.2
Successfully installed mongo-1.0.2
2 gems installed
Installing ri documentation for bson-1.0.2...
Building YARD (yri) index for bson-1.0.2...
Installing ri documentation for mongo-1.0.2...
Building YARD (yri) index for mongo-1.0.2...


Installing RDoc documentation for bson-1.0.2...
Installing RDoc documentation for mongo-1.0.2...
```

Installing the *mongo* gem will also install the *bson* gem on which it depends. The *bson* gem handles all of the BSON encoding and decoding for the driver (for more on BSON, see "BSON" on page 179). The *bson* gem will also make use of C extensions available in the *bson_ext* gem to improve performance, if that gem has been installed. For maximum performance, be sure to install *bson_ext*:

```
$ gem install bson_ext
Building native extensions.  This could take a while...
Successfully installed bson_ext-1.0.1
1 gem installed
```

If *bson_ext* is on the load path, it will be used automatically.

Using the Ruby Driver

To connect to an instance of MongoDB, use the `Mongo::Connection` class. Once we have an instance of `Mongo::Connection`, we can get an individual database (here we use the *stuffy* database) using bracket notation:

```
> require 'rubygems'
 => true
> require 'mongo'
 => true
> db = Mongo::Connection.new["stuffy"]
```

The Ruby driver uses hashes to represent documents. Aside from that, the API is similar to that of the shell with most method names being the same. (Although the Ruby driver uses underscore_naming, whereas the shell often uses camelCase.) To insert the document `{"x" : 1}` into the *bar* collection and query for the result, we would do the following:

```
> db["bar"].insert :x => 1
 => BSON::ObjectID('4c168343e6fb1b106f000001')
> db["bar"].find_one
 => {"_id"=>BSON::ObjectID('4c168343e6fb1b106f000001'), "x"=>1}
```

There are some important gotchas about documents in Ruby that you need to be aware of:

- Hashes are ordered in Ruby 1.9, which matches how documents work in MongoDB. In Ruby 1.8, however, hashes are unordered. The driver provides a special type, `BSON::OrderedHash`, which must be used instead of a regular hash whenever key order is important.
- Hashes being saved to MongoDB can have symbols as either keys or values. Hashes returned from MongoDB will have symbol values wherever they were present in the input, but any symbol keys will be returned as strings. So, `{:x => :y}` will become `{"x" => :y}`. This is a side effect of the way documents are represented in BSON (see Appendix C for more on BSON).

Custom Form Submission

The problem at hand is to generate custom forms for user-submitted data and to handle user submissions for those forms. Forms are created by editors and can contain arbitrary fields, each with different types and rules for validation. Here we'll leverage the ability to embed documents and store each field as a separate document within a form. A form document for a comment submission form might look like this:

```
comment_form = {
    :_id => "comments",
    :fields => [
        {
            :name => "name",
            :label => "Your Name",
            :help_text => "Required",
            :required => true,
            :type => "string",
            :max_length => 200
        },
        {
            :name => "email",
            :label => "Your E-mail Address",
            :help_text => "Required, but will not be displayed",
            :required => true,
            :type => "email"
        },
        {
            :name => "comment",
            :label => "Your Comment",
            :help_text => "Comments will be moderated",
            :required => true,
            :type => "string",
            :word_limit => 200
        }
    ]
}
```

This form shows some of the benefits of working with a document-oriented database like MongoDB. First, we're able to embed the form's fields directly within the form document. We don't need to store them separately and do a join—we can get the entire representation for a form by querying for a single document. We're also able to specify different keys for different types of fields. In the previous example, the name field has a `:max_length`, key and the comment field has a `:word_limit` key, while the email field has neither.

In this example we use `"_id"` to store a human-readable name for our form. This works well because we need to index on the form name anyway to make queries efficient. Because the `"_id"` index is a unique index, we're also guaranteed that form names will be unique across the system.

When an editor adds a new form, we simply save the resultant document. To save the `comment_form` document that we created, we'd do the following:

```
db["forms"].save comment_form
```

Each time we want to render a page with the comment form, we can query for the form document by its name:

```
db["forms"].find_one :_id => "comments"
```

The single document returned contains all the information we need in order to render the form, including the name, label, and type for each input field that needs to be rendered. When a form needs to be changed, editors can easily add a field or specify additional constraints for an existing field.

When we get a user submission for a form, we can run the same query as earlier to get the relevant form document. We'll need this in order to validate that the user's submission includes values for all required fields and meets any other requirements specified in our form. After validation, we can save the submission as a separate document in a *submissions* collection. A submission for our comment form might look like this:

```
comment_submission = {
    :form_id => "comments",
    :name => "Mike D.",
    :email => "mike@example.com",
    :comment => "MongoDB is flexible!"
}
```

We're again leveraging the document model by including custom keys for each submission (here we use `:name`, `:email`, and `:comment`). The only key that we require in each submission is `:form_id`. This allows us to efficiently retrieve all submissions for a certain form:

```
db["submissions"].find :form_id => "comments"
```

To perform this query, we should have an index on `:form_id`:

```
db["submissions"].create_index :form_id
```

We can also use `:form_id` to retrieve the form document for a given submission.

Ruby Object Mappers and Using MongoDB with Rails

There are several libraries written on top of the basic Ruby driver to provide things like models, validations, and associations for MongoDB documents. The most popular of these tools seem to be MongoMapper and Mongoid. If you're used to working with tools like ActiveRecord or DataMapper, you might consider using one of these object mappers in addition to the basic Ruby driver.

MongoDB also works nicely with Ruby on Rails, especially when working with one of the previously mentioned mappers. There are up-to-date instructions on integrating MongoDB with Rails on the MongoDB site.

Real-Time Analytics: Python

The Python driver for MongoDB is called *PyMongo*. In this section, we'll use PyMongo to implement some real-time tracking of metrics for a web application. The most up-to-date documentation on PyMongo is available at *http://api.mongodb.org/python*.

Installing PyMongo

PyMongo is available in the Python Package Index and can be installed using `easy_install` (*http://pypi.python.org/pypi/setuptools*):

```
$ easy_install pymongo
Searching for pymongo
Reading http://pypi.python.org/simple/pymongo/
Reading http://github.com/mongodb/mongo-python-driver
Best match: pymongo 1.6
Downloading ...
Processing pymongo-1.6-py2.6-macosx-10.6-x86_64.egg
Moving ...
Adding pymongo 1.6 to easy-install.pth file

Installed ...
Processing dependencies for pymongo
Finished processing dependencies for pymongo
```

This will install PyMongo and will attempt to install an optional C extension as well. If the C extension fails to build or install, everything will continue to work, but performance will suffer. An error message will be printed during install in that case.

As an alternative to `easy_install`, PyMongo can also be installed by running `python setup.py install` from a source checkout.

Using PyMongo

We use the `pymongo.connection.Connection` class to connect to a MongoDB server. Here we create a new `Connection` and use attribute-style access to get the *analytics* database:

```
from pymongo import Connection
db = Connection().analytics
```

The rest of the API for PyMongo is similar to the API of the MongoDB shell; like the Ruby driver, PyMongo uses underscore_naming instead of camelCase, however. Documents are represented using dictionaries in PyMongo, so to insert and retrieve the document `{"a" : [1, 2, 3]}`, we do the following:

```
db.test.insert({"a": [1, 2, 3]})
db.test.find_one()
```

Dictionaries in Python are unordered, so PyMongo provides an ordered subclass of `dict`, `pymongo.son.SON`. In most places where ordering is required, PyMongo provides APIs that hide it from the user. If not, applications can use `SON` instances instead of dictionaries to ensure their documents maintain key order.

MongoDB for Real-Time Analytics

MongoDB is a great tool for tracking metrics in real time for a couple of reasons:

- Upsert operations (see Chapter 3) allow us to send a single message to either create a new tracking document or increment the counters on an existing document.
- The upsert we send does not wait for a response; it's fire-and-forget. This allows our application code to avoid blocking on each analytics update. We don't need to wait and see whether the operation is successful, because an error in analytics code wouldn't get reported to a user anyway.
- We can use an `$inc` update to increment a counter without having to do a separate query and update operation. We also eliminate any contention issues if multiple updates are happening simultaneously.
- MongoDB's update performance is very good, so doing one or more updates per request for analytics is reasonable.

Schema

In our example we will be tracking page views for our site, with hourly roll-ups. We'll track both total page views as well as page views for each individual URL. The goal is to end up with a collection, *hourly*, containing documents like this:

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 5 }
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "url" : "/bar", "views" : 5 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/", "views" : 12 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/bar", "views" : 3 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 10 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "url" : "/foo", "views" : 21 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "url" : "/", "views" : 3 }
...
```

Each document represents all of the page views for a single URL in a given hour. If a URL gets no page views in an hour, there is no document for it. To track total page views for the entire site, we'll use a separate collection, *hourly_totals*, which has the following documents:

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "views" : 10 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "views" : 25 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "views" : 24 }
...
```

The difference here is just that we don't need a `"url"` key, because we're doing site-wide tracking. If our entire site doesn't get any page views during an hour, there will be no document for that hour.

Handling a Request

Each time our application receives a request, we need to update our analytics collections appropriately. We need to add a page view both to the *hourly* collection for the specific URL requested and to the *hourly_totals* collection. Let's define a function that takes a URL and updates our analytics appropriately:

```
from datetime import datetime

def track(url):
    hour = datetime.utcnow().replace(minute=0, second=0, microsecond=0)
    db.hourly.update({"hour": hour, "url": url},
                     {"$inc": {"views": 1}}, upsert=True)
    db.hourly_totals.update({"hour": hour},
                            {"$inc": {"views": 1}}, upsert=True)
```

We'll also want to make sure that we have indexes in place to be able to perform these updates efficiently:

```
from pymongo import ASCENDING

db.hourly.create_index([("url", ASCENDING), ("hour", ASCENDING)], unique=True)
db.hourly_totals.create_index("hour", unique=True)
```

For the *hourly* collection, we create a compound index on `"url"` and `"hour"`, while for *hourly_totals* we just index on `"hour"`. Both of the indexes are created as unique, because we want only one document for each of our roll-ups.

Now, each time we get a request, we just call `track` a single time with the requested URL. It will perform two upserts; each will create a new roll-up document if necessary or increment the `"views"` for an existing roll-up.

Using Analytics Data

Now that we're tracking page views, we need a way to query that data and put it to use. Here we print the hourly page view totals for the last 10 hours:

```
from pymongo import DESCENDING

for rollup in db.hourly_totals.find().sort("hour", DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

This query will be able to leverage the index we've already created on `"hour"`. We can perform a similar operation for an individual *`url`*:

```
for rollup in db.hourly.find({"url": url}).sort("hour", DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

The only difference is that here we add a query document for selecting an individual `"url"`. Again, this will leverage the compound index we've already created on `"url"`, and `"hour"`.

Other Considerations

One thing we might want to consider is running a periodic cleaning task to remove old analytics documents. If we're displaying only the last 10 hours of data, then we can conserve space by not keeping around a month's worth of documents. To remove all documents older than 24 hours, we can do the following, which could be run using *cron* or a similar mechanism:

```
from datetime import timedelta

remove_before = datetime.utcnow() - timedelta(hours=24)

db.hourly.remove({"hour": {"$lt": remove_before}})
db.hourly_totals.remove({"hour": {"$lt": remove_before}})
```

In this example, the first `remove` will actually need to do a table scan because we haven't defined an index on `"hour"`. If we need to perform this operation efficiently (or any other operation querying by `"hour"` for all URLs), we should consider adding a second index on `"hour"` to the *hourly* collection.

Another important note about this example is that it would be easy to add tracking for other metrics besides page views or to do roll-ups on a window other than hourly (or even to do roll-ups on multiple windows at once). All we need to do is to tweak the `track` function to perform upserts tracking whatever metric we're interested in, at whatever granularity we want.

APPENDIX A

Installing MongoDB

Installing MongoDB is a simple process on most platforms. Precompiled binaries are available for Linux, Mac OS X, Windows, and Solaris. This means that, on most platforms, you can download the archive from *http://www.mongodb.org*, inflate it, and run the binary. The MongoDB server requires a directory it can write database files to and a port it can listen for connections on. This section covers the entire install on the two variants of system: Windows and everything else (Linux, Max, Solaris).

When we speak of "installing MongoDB," generally what we are talking about is setting up `mongod`, the core database server. `mongod` is used in a single-server setup as either master or slave, as a member of a replica sets, and as a shard. Most of the time, this will be the MongoDB process you are using. Other binaries that come with the download are covered in Chapter 8.

Choosing a Version

MongoDB uses a fairly simple versioning scheme: even-point releases are stable, and odd-point releases are development versions. For example, anything starting with 1.6 is a stable release, such as 1.6.0, 1.6.1, and 1.6.15. Anything starting with 1.7 is a development release, such as 1.7.0, 1.7.2, or 1.7.10. Let's take the 1.6/1.7 release as a sample case to demonstrate how the versioning timeline works:

1. Developers release 1.6.0. This is a major release and will have an extensive changelog. Everyone in production is advised to upgrade as soon as possible.
2. After the developers start working on the milestones for 1.8, they release 1.7.0. This is the new development branch that is fairly similar to 1.6.0, but probably with an extra feature or two and maybe some bugs.
3. As the developers continue to add features, they will release 1.7.1, 1.7.2, and so on.
4. Any bug fixes or "nonrisky" features will be backported to the 1.6 branch, and 1.6.1, 1.6.2, and so on, will be released. Developers are conservative about what is backported; few new features are ever added to a stable release. Generally, only bug fixes are ported.
5. After all of the major milestones have been reached for 1.8.0, developers will release something like, say, 1.7.5.
6. After extensive testing of 1.7.5, usually there are a couple minor bugs that need to be fixed. Developers fix these bugs and release 1.7.6.
7. Developers repeat step 6 until no new bugs are apparent, and then 1.7.6 (or whatever the latest release ended up being) is renamed 1.8.0. That is, the last development release in a series becomes the new stable release.
8. Start over from step 1, incrementing all versions by .2.

Thus, the initial releases in a development branch may be highly unstable (x.y.0, x.y.1, x.y.2), but usually, by the time a branch gets to x.y.5, it's fairly close to production-ready. You can see how close a production release is by browsing the core server roadmap on the MongoDB bug tracker.

If you are running in production, you should use a stable release unless there are features in the development release that you need. Even if you need certain features from a development release, it is worth first getting in touch with the developers through the mailing list and IRC to let them know you are planning on going into production with a development release and get advice about keeping your data safe. (Of course, this is always a good idea.)

If you are just starting development on a project, using a development release may be a better choice. By the time you deploy to production, there will probably be a stable release (MongoDB keeps a regular cycle of stable releases every couple of months), and you'll get to use the latest features. However, you must balance this against the possibility that you would run into server bugs, which could be confusing or discouraging to most new users.

Windows Install

To install MongoDB on Windows, download the Windows zip from the MongoDB downloads page. Use the advice in the previous section to choose the correct version of MongoDB. There are 32-bit and 64-bit releases for Windows, so select whichever version you're running. When you click the link, it will download the *.zip*. Use your favorite extraction tool to unzip the archive.

Now you need to make a directory in which MongoDB can write database files. By default, MongoDB tries to use *C:\data\db* as its data directory. You can create this directory or any other empty directory anywhere on the filesystem. If you chose to use a directory other than *C:\data\db*, you'll need to specify the path when you start MongoDB, which is covered in a moment.

Now that you have a data directory, open the command prompt (*cmd.exe*). Navigate to the directory where you unzipped the MongoDB binaries and run the following:

```
$ bin\mongod.exe
```

If you chose a directory other than *C:\data\db*, you'll have to specify it here, with the `--dbpath` argument:

```
$ bin\mongod.exe --dbpath C:\Documents and Settings\Username\My Documents\db
```

See the Chapter 8 section for more common options, or run `mongod.exe --help` to see all options.

Installing as a Service

MongoDB can also be installed as a service on Windows. To install, simply run with the full path, escape any spaces, and use the `--install` option. For example:

```
$ C:\mongodb-windows-32bit-1.6.0\bin\mongod.exe
    --dbpath "\"C:\Documents and Settings\Username\My Documents\db\"" --install
```

It can then be started and stopped from the Control Panel.

POSIX (Linux, Mac OS X, and Solaris) Install

Choose a version of MongoDB, based on the advice in the section "Choosing a Version" on page 173. Go to the MongoDB downloads page, and select the correct version for your OS.

> If you are using a Mac, check whether you're running 32-bit or 64-bit. Macs are especially picky that you choose the correct build and will refuse to start MongoDB and give confusing error messages if you choose the wrong build. You can check what you're running by clicking the apple in the upper-left corner and selecting the About This Mac option.

You must create a directory for the database to put its files. By default, the database will use */data/db*, although you can specify any other directory. If you create the default directory, make sure it has the correct write permissions. You can create the directory and set the permissions by running the following:

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

`mkdir -p` creates the directory and all its parents, if necessary (i.e., if the */data* directory didn't exist, it will create the `/data` directory and then the */data/db* directory). `chown` changes the ownership of */data/db* so that your user can write to it. Of course, you can also just create a directory in your home folder and specify that MongoDB should use that when you start the database, to avoid any permissions issues.

Decompress the *.tar.gz* file you downloaded from *http://www.mongodb.org*.

```
$ tar zxf mongodb-linux-i686-1.6.0.tar.gz
$ cd mongodb-linux-i686-1.6.0
```

Now you can start the database:

```
$ bin/mongod
```

Or if you'd like to use an alternate database path, specify it with the `--dbpath` option:

```
$ bin/mongod --dbpath ~/db
```

See Chapter 8 for a summary of the most common options, or run `mongod` with `--help` to see all the possible options.

Installing from a Package Manager

On these systems, there are many package managers that can also be used to install MongoDB. If you prefer using one of these, there are official packages for Debian and Ubuntu and unofficial ones for Red Hat, Gentoo, and FreeBSD. If you use an unofficial version, make sure to check the logs when you start the database; sometimes these packages are not built with UTF-8 support.

On Mac, there are also unofficial packages in Homebrew and MacPorts. If you go for the MacPorts version, be forewarned: it takes hours to compile all the Boost libraries, which are MongoDB prerequisites. Start the download, and leave it overnight.

Regardless of the package manager you use, it is a good idea to figure out where it is putting the MongoDB log files before you have a problem and need to find them. It's important to make sure they're being saved properly in advance of any possible issues.

APPENDIX B
mongo: The Shell

Throughout this text, we use the `mongo` binary, which is the database shell. We generally assume that you are running it on the same machine as `mongod` and that you are running `mongod` on the default port, but if you are not, you can specify this on startup and have the shell connect to another server:

```
$ bin/mongo staging.example.com:20000
```

This would connect to a `mongod` running at *staging.example.com* on port 20000.

The shell also, by default, starts out connected to the `test` database. If you'd like `db` to refer to a different database, you can use */dbname* after the server address:

```
$ bin/mongo localhost:27017/admin
```

This connects to *mongod* running locally on the default port, but `db` will immediately refer to the `admin` database.

You can also start the shell without connecting to any database by using the `--nodb` option. This is useful if you'd like to just play around with JavaScript or connect later:

```
$ bin/mongo --nodb
MongoDB shell version: 1.5.3
type "help" for help
>
```

Keep in mind that *db* isn't the only database connection you can have. You can connect to as many databases as you would like from the shell, which can be handy in multi-server environments. Simply use the `connect()` method, and assign the resulting connection to any variable you'd like. For instance, with sharding, we might want `mongos` to refer to the `mongos` server and also have a connection to each shard:

```
> mongos = connect("localhost:27017")
connecting to: localhost:27017
localhost:27017
> shard0 = connect("localhost:30000")
connecting to: localhost:30000
localhost:30000
> shard1 = connect("localhost:30001")


connecting to: localhost:30001
localhost:30001
```

Then, we can use `mongos`, `shard0`, or `shard1` as the `db` variable is usually used. (Although special helpers, such as `use foo` or `show collections`, will not work.)

Shell Utilities

There are a number of useful shell functions that were not covered earlier.

For administrating multiple databases, it can be useful to have multiple database variables, not `db`. For example, with sharding, you may want to maintain a separate variable pointing to your `config` database:

```
> config = db.getSisterDB("config")
config
> config.shards.find()
...
```

You can even connect to multiple servers within a single shell using the `connect` function:

```
> shard_db = connect("shard.example.com:27017/mydb")
connecting to shard.example.com:27017/mydb
mydb
>
```

The shell can also run shell commands:

```
> runProgram("echo", "Hello", "world")
shell: started mongo program echo Hello world
0
> sh6487| Hello world
```

(The output looks strange because the shell is running.)

APPENDIX C
MongoDB Internals

For the most part, users of MongoDB can treat it as a black box. When trying to understand performance characteristics or looking to get a deeper understanding of the system, it helps to know a little bit about the internals of MongoDB, though.

BSON

Documents in MongoDB are an abstract concept—the concrete representation of a document varies depending on the driver/language being used. Because documents are used extensively for communication in MongoDB, there also needs to be a representation of documents that is shared by all drivers, tools, and processes in the MongoDB ecosystem. That representation is called Binary JSON (BSON).

BSON is a lightweight binary format capable of representing any MongoDB document as a string of bytes. The database understands BSON, and BSON is the format in which documents are saved to disk.

When a driver is given a document to insert, use as a query, and so on, it will encode that document to BSON before sending it to the server. Likewise, documents being returned to the client from the server are sent as BSON strings. This BSON data is decoded by the driver to its native document representation before being returned to the client.

The BSON format has three primary goals:

Efficiency
: BSON is designed to represent data efficiently, without using much extra space. In the worst case BSON is slightly less efficient than JSON and in the best case (e.g., when storing binary data or large numerics), it is much more efficient.

Traversability
: In some cases, BSON does sacrifice space efficiency to make the format easier to traverse. For example, string values are prefixed with a length rather than relying on a terminator to signify the end of a string. This traversability is useful when the MongoDB server needs to introspect documents.

Performance
: Finally, BSON is designed to be fast to encode to and decode from. It uses C-style representations for types, which are fast to work with in most programming languages.

For the exact BSON specification, see *http://www.bsonspec.org*.

Wire Protocol

Drivers access the MongoDB server using a lightweight TCP/IP wire protocol. The protocol is documented on the MongoDB wiki but basically consists of a thin wrapper around BSON data. For example, an insert message consists of 20 bytes of header data (which includes a code telling the server to perform an insert and the message length), the collection name to insert into, and a list of BSON documents to insert.

Data Files

Inside of the MongoDB data directory, which is */data/db/* by default, there are separate files for each database. Each database has a single *.ns* file and several data files, which have monotonically increasing numeric extensions. So, the database *foo* would be stored in the files *foo.ns*, *foo.0*, *foo.1*, *foo.2*, and so on.

The numeric data files for a database will double in size for each new file, up to a maximum file size of 2GB. This behavior allows small databases to not waste too much space on disk, while keeping large databases in mostly contiguous regions on disk.

MongoDB also preallocates data files to ensure consistent performance. (This behavior can be disabled using the `--noprealloc` option.) Preallocation happens in the background and is initiated every time that a data file is filled. This means that the MongoDB server will always attempt to keep an extra, empty data file for each database to avoid blocking on file allocation.

Namespaces and Extents

Within its data files, each database is organized into *namespaces*, each storing a specific type of data. The documents for each collection have their own namespace, as does each index. Metadata for namespaces is stored in the database's *.ns* file.

The data for each namespace is grouped on disk into sections of the data files, called *extents*. In Figure C-1 the *foo* database has three data files, the third of which has been preallocated and is empty. The first two data files have been divided up into extents belonging to several different namespaces.

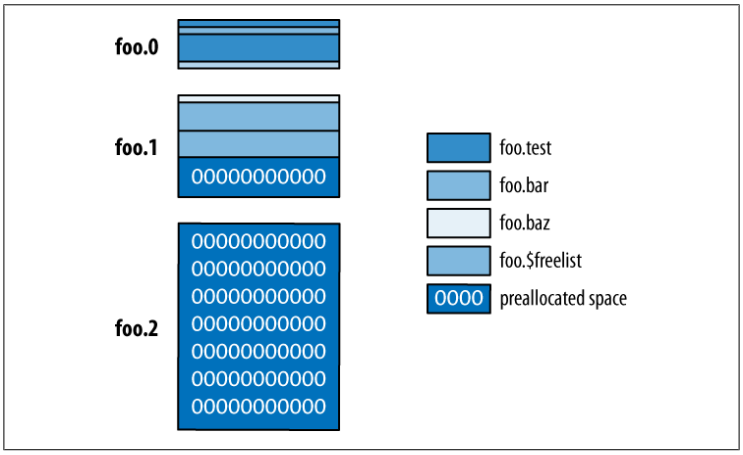


Figure C-1. Namespaces and extents

Figure C-1 shows us several interesting things about namespaces and extents. Each namespace can have several different extents, which are not (necessarily) contiguous on disk. Like data files for a database, extents for a namespace grow in size with each new allocation. This is done to balance wasted space used by a namespace versus the desire to keep data for a namespace mostly contiguous on disk. The figure also shows a special namespace, *$freelist*, which keeps track of extents that are no longer in use (e.g., extents from a dropped collection or index). When a namespace allocates a new extent, it will first search the freelist to see whether an appropriately sized extent is available.

Memory-Mapped Storage Engine

The default storage engine (and only supported storage engine at the time of this writing) for MongoDB is a memory-mapped engine. When the server starts up, it memory maps all its data files. It is then the responsibility of the operating system to manage flushing data to disk and paging data in and out. This storage engine has several important properties:

- MongoDB's code for managing memory is small and clean, because most of that work is pushed to the operating system.
- The virtual size of a MongoDB server process is often very large, exceeding the size of the entire data set. This is OK, because the operating system will handle keeping the amount of data resident in memory contained.

- MongoDB cannot control the order that data is written to disk, which makes it impossible to use a writeahead log to provide single-server durability. Work is ongoing on an alternative storage engine for MongoDB to provide single-server durability.
- 32-bit MongoDB servers are limited to a total of about 2GB of data per *`mongod`*. This is because all of the data must be addressable using only 32 bits.

Part 3: Python and MongoDB

Table of Contents

Preface

I've been building production database-driven applications for about 10 years. I've worked with most of the usual relational databases (MSSQL Server, MySQL, PostgreSQL) and with some very interesting nonrelational databases (Freebase.com's Graphd/MQL, Berkeley DB, MongoDB). MongoDB is at this point the system I enjoy working with the most, and choose for most projects. It sits somewhere at a crossroads between the performance and pragmatism of a relational system and the flexibility and expressiveness of a semantic web database. It has been central to my success in building some quite complicated systems in a short period of time.

I hope that after reading this book you will find MongoDB to be a pleasant database to work with, and one which doesn't get in the way between you and the application you wish to build.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/0636920021513

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank Ariel Backenroth, Aseem Mohanty and Eugene Ciurana for giving detailed feedback on the first draft of this book. I would also like to thank the O'Reilly team for making it a great pleasure to write the book. Of course, thanks to all the people at 10gen without whom MongoDB would not exist and this book would not have been possible.

CHAPTER 1

Getting Started

Introduction

First released in 2009, MongoDB is relatively new on the database scene compared to contemporary giants like Oracle which trace their first releases to the 1970's. As a document-oriented database generally grouped into the NoSQL category, it stands out among distributed key value stores, Amazon Dynamo clones and Google BigTable re-implementations. With a focus on rich operator support and high performance Online Transaction Processing (OLTP), MongoDB is in many ways closer to MySQL than to batch-oriented databases like HBase.

The key differences between MongoDB's document-oriented approach and a traditional relational database are:

1. MongoDB does not support joins.
2. MongoDB does not support transactions. It does have some support for atomic operations, however.
3. MongoDB schemas are flexible. Not all documents in a collection must adhere to the same schema.

1 and 2 are a direct result of the huge difficulties in making these features scale across a large distributed system while maintaining acceptable performance. They are tradeoffs made in order to allow for horizontal scalability. Although MongoDB lacks joins, it does introduce some alternative capabilites, e.g. embedding, which can be used to solve many of the same data modeling problems as joins. Of course, even if embedding doesn't quite work, you can always perform your join in application code, by making multiple queries.

The lack of transactions can be painful at times, but fortunately MongoDB supports a fairly decent set of atomic operations. From the basic atomic increment and decrement operators to the richer "findAndModify", which is essentially an atomic read-modify-write operator.

It turns out that a flexible schema can be very beneficial, especially when you expect to be iterating quickly. While up front schema design—as used in the relational model—has its place, there is often a heavy cost in terms of maintenance. Handling schema updates in the relational world is of course doable, but comes with a price.

In MongoDB, you can add new properties at any time, dynamically, without having to worry about ALTER TABLE statements that can take hours to run and complicated data migration scripts. However, this approach does come with its own tradeoffs. For example, type enforcement must be carefully handled by the application code. Custom document versioning might be desirable to avoid large conditional blocks to handle heterogeneous documents in the same collection.

The dynamic nature of MongoDB lends itself quite naturally to working with a dynamic language such as Python. The tradeoffs between a dynamically typed language such as Python and a statically typed language such as Java in many respects mirror the tradeoffs between the flexible, document-oriented model of MongoDB and the up-front and statically typed schema definition of SQL databases.

Python allows you to express MongoDB documents and queries natively, through the use of existing language features like nested dictionaries and lists. If you have worked with JSON in Python, you will immediately be comfortable with MongoDB documents and queries.

For these reasons, MongoDB and Python make a powerful combination for rapid, iterative development of horizontally scalable backend applications. For the vast majority of modern Web and mobile applications, we believe MongoDB is likely a better fit than RDBMS technology.

Finding Reference Documentation

MongoDB, Python, 10gen's PyMongo driver and each of the Web frameworks mentioned in this book all have good reference documentation online.

For MongoDB, we would strongly suggest bookmarking and at least skimming over the official MongoDB manual which is available in a few different formats and constantly updated at *http://www.mongodb.org/display/DOCS/Manual*. While the manual describes the JavaScript interface via the `mongo` console utility as opposed to the Python interface, most of the code snippets should be easily understood by a Python programmer and more-or-less portable to PyMongo, albeit sometimes with a little bit of work. Furthermore, the MongoDB manual goes into greater depth on certain advanced and technical implementation and database administration topics than is possible in this book.

For the Python language and standard library, you can use the `help()` function in the interpreter or the `pydoc` tool on the command line to get API documentation for any methods or modules. For example:

```
pydoc string
```

The latest Python language and API documentation is also available for online browsing at *http://docs.python.org/*.

10gen's PyMongo driver has API documentation available online to go with each release. You can find this at *http://api.mongodb.org/python/*. Additionally, once you have the PyMongo driver package installed on your system, a summary version of the API documentation should be available to you in the Python interpreter via the `help()` function. Due to an issue with the `virtualenv` tool mentioned in the next section, "pydoc" does not work inside a virtual environment. You must instead run `python -m pydoc pymongo`.

Installing MongoDB

For the purposes of development, it is recommended to run a MongoDB server on your local machine. This will permit you to iterate quickly and try new things without fear of destroying a production database. Additionally, you will be able to develop with MongoDB even without an Internet connection.

Depending on your operating system, you may have multiple options for how to install MongoDB locally.

Most modern UNIX-like systems will have a version of MongoDB available in their package management system. This includes FreeBSD, Debian, Ubuntu, Fedora, CentOS and ArchLinux. Installing one of these packages is likely the most convenient approach, although the version of MongoDB provided by your packaging vendor may lag behind the latest release from 10gen. For local development, as long as you have the latest major release, you are probably fine.

10gen also provides their own MongoDB packages for many systems which they update very quickly on each release. These can be a little more work to get installed but ensure you are running the latest-and-greatest. After the initial setup, they are typically trivial to keep up-to-date. For a production deployment, where you likely want to be able to update to the most recent stable MongoDB version with a minimum of hassle, this option probably makes the most sense.

In addition to the system package versions of MongoDB, 10gen provide binary zip and tar archives. These are independent of your system package manager and are provided in both 32-bit and 64-bit flavours for OS X, Windows, Linux and Solaris. 10gen also provide statically-built binary distributions of this kind for Linux, which may be your best option if you are stuck on an older, legacy Linux system lacking the modern libc and other library versions. Also, if you are on OS X, Windows or Solaris, these are probably your best bet.

Finally, you can always build your own binaries from the source code. Unless you need to make modifications to MongoDB internals yourself, this method is best avoided due to the time and complexity involved.

In the interests of simplicity, we will provide the commands required to install a stable version of MongoDB using the system package manager of the most common UNIX-like operating systems. This is the easiest method, assuming you are on one of these platforms. For Mac OS X and Windows, we provide instructions to install the binary packages from 10gen.

Ubuntu / Debian:

```
sudo apt-get update; sudo apt-get install mongodb
```

Fedora:

```
sudo yum install mongo-stable-server
```

FreeBSD:

```
sudo pkg_add -r mongodb
```

Windows:

Go to *http://www.mongodb.org* and download the latest production release zip file for Windows—choosing 32-bit or 64-bit depending on your system. Extract the contents of the zipfile to a location like `C:\mongodb` and add the `bin` directory to your PATH.

Mac OS X:

Go to *http://www.mongodb.org* and download the latest production release compressed tar file for OS X—choosing 32-bit or 64-bit depending on your system. Extract the contents to a location like `/usr/local/` or `/opt` and add the `bin` directory to your $PATH. For exmaple:

```
cd /tmp
wget http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-1.8.3-rc1.tgz
tar xfz mongodb-osx-x86_64-1.8.3-rc1.tgz
sudo mkdir /usr/local/mongodb
sudo cp -r mongodb-osx-x86_64-1.8.3-rc1/bin /usr/local/mongodb/
export PATH=$PATH:/usr/local/mongodb/bin
```

Install MongoDB on OS X with Mac Ports

If you would like to try a third-party system package management system on Mac OS X, you may also install MongoDB (and Python, in fact) through Mac Ports. Mac Ports is similar to FreeBSD ports, but for OS X.

A word of warning though: Mac Ports compiles from source, and so can take considerably longer to install software compared with simply grabbing the binaries. Futhermore, you will need to have Apple's Xcode Developer Tools installed, along with the X11 windowing environment.

The first step is to install Mac Ports from *http://www.macports.org*. We recommend downloading and installing their DMG package.

Once you have Mac Ports installed, you can install MongoDB with the command:

```
sudo port selfupdate; sudo port install mongodb
```

To install Python 2.7 from Mac Ports use the command:

```
sudo port selfupdate; sudo port install python27
```

Running MongoDB

On some platforms—such as Ubuntu—the package manager will automatically start the mongod daemon for you, and ensure it starts on boot also. On others, such as Mac OS X, you must write your own script to start it, and manually integrate with launchd so that it starts on system boot.

Note that before you can start MongoDB, its data and log directories must exist.

If you wish to have MongoDB start automatically on boot on Windows, 10gen have a document describing how to set this up at *http://www.mongodb.org/display/DOCS/Windows+Service*

To have MongoDB start automatically on boot under Mac OS X, first you will need a plist file. Save the following (changing db and log paths appropriately) to `/Library/LaunchDaemons/org.mongodb.mongod.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>RunAtLoad</key>
        <true/>
        <key>Label</key>
        <string>org.mongo.mongod</string>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/local/mongodb/bin/mongod</string>
                <string>--dbpath</string>

                <string>/usr/local/mongodb/data/</string>
                <string>--logpath</string>
                <string>/usr/local/mongodb/log/mongodb.log</string>
        </array>
</dict>
</plist>
```

Next run the following commands to activate the startup script with launchd:

```
sudo launchctl load /Library/LaunchDaemons/org.mongodb.mongod.plist
sudo launchctl start org.mongodb.mongod
```

A quick way to test whether there is a MongoDB instance already running on your local machine is to type `mongo` at the command-line. This will start the MongoDB admin console, which attempts to connect to a database server running on the default port (27017).

In any case, you can always start MongoDB manually from the command-line. This is a useful thing to be familiar with in case you ever want to test features such as replica sets or sharding by running multiple mongod instances on your local machine.

Assuming the mongod binary is in your $PATH, run:

```
mongod --logpath <path/to/mongo.logfile> --port <port to listen on> --dbpath <path/to/
data directory>
```

Setting up a Python Environment with MongoDB

In order to be able to connect to MongoDB with Python, you need to install the PyMongo driver package. In Python, the best practice is to create what is known as a "virtual environment" in which to install your packages. This isolates them cleanly from any "system" packages you have installed and yields the added bonus of not requiring root privileges to install additional Python packages. The tool to create a "virtual environment" is called virtualenv.

There are two approaches to installing the virtualenv tool on your system—manually and via your system package management tool. Most modern UNIX-like systems will have the virtualenv tool in their package repositories. For example, on Mac OS X with Mac Ports, you can run `sudo port install py27-virtualenv` to install virtualenv for Python 2.7. On Ubuntu you can run `sudo apt-get install python-virtualenv`. Refer to the documentation for your OS to learn how to install it on your specific platform.

In case you are unable or simply don't want to use your system's package manager, you can always install it yourself, by hand. In order to manually install it, you must have the Python setuptools package. You may already have setuptools on your system. You can test this by running `python -c import setuptools` on the command line. If nothing is printed and you are simply returned to the prompt, you don't need to do anything. If an ImportError is raised, you need to install setuptools.

To manually install setuptools, first download the file *http://peak.telecommunity.com/dist/ez_setup.py*

Then run `python ez_setup.py` as root.

For Windows, first download and install the latest Python 2.7.x package from *http://www.python.org*. Once you have installed Python, download and install the Windows setuptools installer package from *http://pypi.python.org/pypi/setuptools/*. After installing Python 2.7 and setuptools, you will have the easy_install tool available on your machine in the Python scripts directory—default is C:\Python27\Scripts\.

Once you have setuptools installed on your system, run `easy_install virtualenv` as root.

Now that you have the "virtualenv" tool available on your machine, you can create your first virtual Python environment. You can do this by executing the command `virtualenv --no-site-packages myenv`. You do not need—and indeed should not want—to run this command with root privileges. This will create a virtual environment in the directory "myenv". The --no-site-packages option to the "virtualenv" utility instructs it to create a clean Python environment, isolated from any existing packages installed in the system.

You are now ready to install the PyMongo driver.

With the "myenv" directory as your working directory (i.e. after "cd myenv"), simply execute `bin/easy_install pymongo`. This will install the latest stable version of PyMongo into your virtual Python environment. To verify that this worked successfully, execute the command `bin/python -c import pymongo`, making sure that the "myenv" directory is still your working directory, as with the previous command.

Assuming Python did not raise an ImportError, you now have a Python virtualenv with the PyMongo driver correctly installed and are ready to connect to MongoDB and start issuing queries!

CHAPTER 2

Reading and Writing to MongoDB with Python

MongoDB is a document-oriented database. This is different from a relational database in two significant ways. Firstly, not all entries must adhere to the same schema. Secondly you can embed entries inside of one another. Despite these major differences, there are analogs to SQL concepts in MongoDB. A logical group of entries in a SQL database is termed a table. In MongoDB, the analogous term is a collection. A single entry in a SQL databse is termed a row. In MongoDB, the analog is a document.

Table 2-1. Comparison of SQL/RDBMS and MongoDB Concepts and Terms

Concept	SQL	MongoDB
One User	One Row	One Document
All Users	Users Table	Users Collection
One Username Per User (1-to-1)	Username Column	Username Property
Many Emails Per User (1-to-many)	SQL JOIN with Emails Table	Embed relevant email doc in User Document
Many Items Owned by Many Users (many-to-many)	SQL JOIN with Items Table	Programmatically Join with Items Collection

Hence, in MongoDB, you are mostly operating on documents and collections of documents. If you are familiar with JSON, a MongoDB document is essentially a JSON document with a few extra features. From a Python perspective, it is a Python dictionary.

Consider the following example of a user document with a username, first name, surname, date of birth, email address and score:

```
from datetime import datetime
user_doc = {
    "username" : "janedoe",
    "firstname" : "Jane",
```

```
    "surname" : "Doe",
    "dateofbirth" : datetime(1974, 4, 12),
    "email" : "janedoe74@example.com",
    "score" : 0
}
```

As you can see, this is a native Python object. Unlike SQL, there is no special syntax to deal with. The PyMongo driver transparently supports Python datetime objects. This is very convenient when working with datetime instances—the driver will transparently marshall the values for you in both reads and writes. You should never have to write datetime conversion code yourself.

Instead of grouping things inside of tables, as in SQL, MongoDB groups them in collections. Like SQL tables, MongoDB collections can have indexes on particular document properties for faster lookups and you can read and write to them using complex query predicates. Unlike SQL tables, documents in a MongoDB collection do not all have to conform to the same schema.

Returning to our user example above, such documents would be logically grouped in a "users" collection.

Connecting to MongoDB with Python

The PyMongo driver makes connecting to a MongoDB database quite straight forward. Furthermore, the driver supports some nice features right out of the box, such as connection pooling and automatic reconnect on failure (when working with a replicated setup). If you are familiar with more traditional RDBMS/SQL systems—for example MySQL—you are likely used to having to deploy additional software, or possibly even write your own, to handle connection pooling and automatic reconnect. 10gen very thoughtfully relieved us of the need to worry about these details when working with MongoDB and the PyMongo driver. This takes a lot of the headache out of running a production MongoDB-based system.

You instantiate a Connection object with the necessary parameters. By default, the Connection object will connect to a MongoDB server on localhost at port 27017. To be explicit, we'll pass those parameters along in our example:

```
""" An example of how to connect to MongoDB """
import sys

from pymongo import Connection
from pymongo.errors import ConnectionFailure

def main():
    """ Connect to MongoDB """
    try:
        c = Connection(host="localhost", port=27017)
        print "Connected successfully"
    except ConnectionFailure, e:
        sys.stderr.write("Could not connect to MongoDB: %s" % e)

        sys.exit(1)

if __name__ == "__main__":
    main()
```

As you can see, a ConnectionFailure exception can be thrown by Connection instantiation. It is usually a good idea to handle this exception and output something informative to your users.

Getting a Database Handle

Connection objects themselves are not all that frequently used when working with MongoDB in Python. Typically you create one once, and then forget about it. This is because most of the real interaction happens with Database and Collection objects. Connection objects are just a way to get a handle on your first Databse object. In fact, even if you lose reference to the Connection object, you can always get it back because Database objects have a reference to the Connection object.

Getting a Database object is easy once you have a Connection instance. You simply need to know the name of the database, and the username and password to access it if you are using authorization on it.

```
""" An example of how to get a Python handle to a MongoDB database """
import sys

from pymongo import Connection
from pymongo.errors import ConnectionFailure

def main():
    """ Connect to MongoDB """
    try:
        c = Connection(host="localhost", port=27017)
    except ConnectionFailure, e:
        sys.stderr.write("Could not connect to MongoDB: %s" % e)
        sys.exit(1)
    # Get a Database handle to a database named "mydb"
    dbh = c["mydb"]

    # Demonstrate the db.connection property to retrieve a reference to the
    # Connection object should it go out of scope. In most cases, keeping a
    # reference to the Database object for the lifetime of your program should
    # be sufficient.

    assert dbh.connection == c
    print "Successfully set up a database handle"

if __name__ == "__main__":
    main()
```

Inserting a Document into a Collection

Once you have a handle to your database, you can begin inserting data. Let us imagine we have a collection called "users", containing all the users of our game. Each user has a username, a first name, surname, date of birth, email address and a score. We want to add a new user:

```
""" An example of how to insert a document """
import sys

from datetime import datetime
from pymongo import Connection
from pymongo.errors import ConnectionFailure

def main():
    try:
        c = Connection(host="localhost", port=27017)
    except ConnectionFailure, e:
        sys.stderr.write("Could not connect to MongoDB: %s" % e)
        sys.exit(1)
    dbh = c["mydb"]
    assert dbh.connection == c
    user_doc = {
        "username" : "janedoe",
        "firstname" : "Jane",
        "surname" : "Doe",
        "dateofbirth" : datetime(1974, 4, 12),
        "email" : "janedoe74@example.com",
        "score" : 0
    }

    dbh.users.insert(user_doc, safe=True)
    print "Successfully inserted document: %s" % user_doc

if __name__ == "__main__":
    main()
```

Note that we don't have to tell MongoDB to create our collection "users" before we insert to it. Collections are created lazily in MongoDB, whenever you access them. This has the advantage of being very lightweight, but can occasionally cause problems due to typos. These can be hard to track down unless you have good test coverage. For example, imagine you accidentally typed:

```
# dbh.usrs is a typo, we mean dbh.users!  Unlike an RDBMS, MongoDB won't
# protect you from this class of mistake.
dbh.usrs.insert(user_doc)
```

The code would execute correctly and no errors would be thrown. You might be left scratching your head wondering why your user document isn't there. We recommend being extra vigilant to double check your spelling when addressing collections. Good test coverage can also help find bugs of this sort.

Another feature of MongoDB inserts to be aware of is primary key auto-generation. In MongoDB, the `_id` property on a document is treated specially. It is considered to be the primary key for that document, and is expected to be unique unless the collection has been explcitly created without an index on `_id`. By default, if no `_id` property is present in a document you insert, MongoDB will generate one itself. When MongoDB generates an `_id` property itself, it uses the type ObjectId. A MongoDB ObjectId is a 96-bit value which is expected to have a very high probability of being unique when created. It can be considered similar in purpose to a UUID object as defined by RFC 4122. MongoDB ObjectIds have the nice property of being almost-certainly-unique upon generation, hence no central coordination is required.

This contrasts sharply with the common RDBMS idiom of using auto-increment primary keys. Guaranteeing that an auto-increment key is not already in use usually requires consulting some centralized system. When the intention is to provide a horizontally scalable, de-centralized and fault-tolerant database—as is the case with MongoDB—auto-increment keys represent an ugly bottleneck.

By employing ObjectId as your `_id`, you leave the door open to horizontal scaling via MongoDB's sharding capabilities. While you can in fact supply your own value for the `_id` property if you wish—so long as it is globally unique—this is best avoided unless there is a strong reason to do otherwise. Examples of cases where you may be forced to provide your own `_id` property value include migration from RDBMS systems which utilized the previously-mentioned auto-increment primary key idiom.

Note that an ObjectId can be just as easily generated on the client-side, with PyMongo, as by the server. To generate an ObjectId with PyMongo, you simply instantiate `pymongo.objectid.ObjectId`.

Write to a Collection Safely and Synchronously

By default, the PyMongo driver performs asynchronous writes. Write operations include insert, update, remove and findAndModify.

Asynchronous writes are unsafe in the sense that they are not checked for errors and so execution of your program could continue without any guarantees of the write having completed successfully. While asynchronous writes improve performance by not blocking execution, they can easily lead to nasty race conditions and other nefarious data integrity bugs. For this reason, we recommend you almost always use safe, synchronous, blocking writes. It seems rare in practice to have truly "fire-and-forget" writes where there are aboslutely no consequences for failures. That being said, one common example where asynchronous writes may make sense is when you are writing non-critical logs or analytics data to MongoDB from your application.

Unless you are certain you don't need synchronous writes, we recommend that you pass the "safe=True" keyword argument to inserts, updates, removes and findAndModify operations:

```
# safe=True ensures that your write
# will succeed or an exception will be thrown
dbh.users.insert(user_doc, safe=True)
```

Guaranteeing Writes to Multiple Database Nodes

The term node refers to a single instance of the MongoDB daemon process. Typically there is a single MongoDB node per machine, but for testing or development cases you can run multiple nodes on one machine.

Replica Set is the MongoDB term for the database's enhanced master-slave replication configuration. This is similar to the traditional master-slave replication you find in RDBMS such as MySQL and PostgreSQL in that a single node handles writes at a given time. In MongoDB master selection is determined by an election protocol and during failover a slave is automatically promoted to master without requiring operator intervention. Furthermore, the PyMongo driver is Replica Set-aware and performs automatic reconnect on failure to the new master. MongoDB Replica Sets, therefore, represent a master-slave replication configuration with excellent failure handling out of the box. For anyone who has had to manually recover from a MySQL master failure in a production environment, this feature is a welcome relief.

By default, MongoDB will return success for your write operation once it has been written to a single node in a Replica Set.

However, for added safety in case of failure, you may wish your write to be committed to two or more replicas before returning success. This can help ensure that in case of catastrophic failure, at least one of the nodes in the Replica Set will have your write.

PyMongo makes it easy to specify how many nodes you would like your write to be replicated to before returning success. You simply set a parameter named "w" to the number of servers in each write method call.

For example:

```
# w=2 means the write will not succeed until it has
# been written to at least 2 servers in a replica set.
dbh.users.insert(user_doc, w=2)
```

Note that passing any value of "w" to a write method in PyMongo implies setting "safe=True" also.

Introduction to MongoDB Query Language

MongoDB queries are represented as a JSON-like structure, just like documents. To build a query, you specify a document with properties you wish the results to match. MongoDB treats each property as having an implicit boolean AND. It natively supports boolean OR queries, but you must use a special operator ($or) to achieve it. In addition to exact matches, MongoDB has operators for greater than, less than, etc.

Sample query document to match all documents in the users collection with firstname "jane":

```
q  = {
    "firstname" : "jane"
}
```

If we wanted to retrieve all documents with firstname "jane" AND surname "doe", we would write:

```
q = {
    "firstname" : "jane",
    "surname" : "doe"
}
```

If we wanted to retrieve all documents with a score value of greater than 0, we would write:

```
q = {
    "score" : { "$gt" : 0 }
}
```

Notice the use of the special "$gt" operator. The MongoDB query language provides a number of such operators, enabling you to build quite complex queries.

See the section on MongoDB Query Operators for details.

Reading, Counting, and Sorting Documents in a Collection

In many situations, you only want to retrieve a single document from a collection. This is especially true when documents in your collection are unique on some property. A good example of this is a users collection, where each username is guaranteed unique.

```
# Assuming we already have a database handle in scope named dbh
# find a single document with the username "janedoe".
user_doc = dbh.users.find_one({"username" : "janedoe"})
if not user_doc:
    print "no document found for username janedoe"
```

Notice that `find_one()` will return `None` if no document is found.

Now imagine you wish to find all documents in the users collection which have a firstname property set to "jane" and print out their email addresses. MongoDB will return a Cursor object for us, to stream the results. PyMongo handles result streaming as you iterate, so if you have a huge number of results they are not all stored in memory at once.

```
# Assuming we already have a database handle in scope named dbh
# find all documents with the firstname "jane".
# Then iterate through them and print out the email address.
users = dbh.users.find({"firstname":"jane"})
for user in users:
    print user.get("email")
```

Notice in the above example that we use the Python `dict` class' `get` method. If we were certain that every single result document contained the "email" property, we could have used dictionary access instead.

```
for user in users:
    print user["email"]
```

If you only wish to retrieve a subset of the properties from each document in a collection during a read, you can pass those as a dictionary via an additional parameter. For example, suppose that you only wish to retrieve the email address for each user with firstname "jane":

```
# Only retrieve the "email" field from each matching document.
users = dbh.users.find({"firstname":"jane"}, {"email":1})
for user in users:
    print user.get("email")
```

If you are retrieving a large result set, requesting only the properties you need can reduce network and decoding overhead, potentially increasing performance.

Sometimes you are not so interested in the query results themselves, but are looking to find the size of the result set for a given query. A common example is an analytics situation where you want a count of how many documents are in your users' collections. MonogDB supports efficient server-side counting of result sets with the `count()` method on `Cursor` objects:

```
# Find out how many documents are in users collection, efficiently
userscount = dbh.users.find().count()
print "There are %d documents in users collection" % userscount
```

MongoDB can also perform result sorting for you on the server-side. Especially if you are sorting results on a property which has an index, it can sort these far more efficiently than your client program can. PyMongo Cursor objects have a `sort()` method which takes a Python 2-tuple comprising the property to sort on, and the direction. The PyMongo `sort()` method is analogous to the SQL ORDER BY statement. Direction can either be `pymongo.ASCENDING` or `pymongo.DESCENDING`. For example:

```
# Return all user with firstname "jane" sorted
# in descending order by birthdate (ie youngest first)
users = dbh.users.find(
    {"firstname":"jane"}).sort(("dateofbirth", pymongo.DESCENDING))
for user in users:
    print user.get("email")
```

In addition to the `sort()` method on the PyMongo Cursor object, you may also pass sort instructions to the `find()` and `find_one()` methods on the PyMongo Collection object. Using this facility, the above example may be rewritten as:

```
# Return all user with firstname "jane" sorted
# in descending order by birthdate (ie youngest first)
users = dbh.users.find({"firstname":"jane"},
    sort=[("dateofbirth", pymongo.DESCENDING)])
for user in users:
    print user.get("email")
```

Another situation you may encounter—especially when you have large result sets—is that you wish to only fetch a limited number of results. This is frequently combined with server-side sorting of results. For example, imagine you are generating a high score table which displays only the top ten scores. PyMongo Cursor objects have a `limit()` method which enables this. The `limit()` method is analogous to the SQL LIMIT statement.

```
# Return at most 10 users sorted by score in descending order
# This may be used as a "top 10 users highscore table"
users = dbh.users.find().sort(("score", pymongo.DESCENDING)).limit(10)
for user in users:
    print user.get("username"), user.get("score", 0)
```

If you know in advance that you only need a limited number of results from a query, using `limit()` can yield a performance benefit. This is because it may greatly reduce the size of the results data which must be sent by MongoDB. Note that a limit of 0 is equivalent to no limit.

Additionally, MongoDB can support skipping to a specific offset in a result set through the `Cursor.skip()` method provided by PyMongo. When used with `limit()` this enables result pagination which is frequently used by clients when allowing end-users to browse very large result sets. `skip()` is analogous to the SQL OFFSET statement. For example, imagine a Web application which displays 20 users per page, sorted alphabetically by surname , and needs to fetch the data to build the second page of results for a user. The query used by the Web application might look like this:

```
# Return at most 20 users sorted by name,
# skipping the first 20 results in the set
users = dbh.users.find().sort(
    ("surname", pymongo.ASCENDING)).limit(20).skip(20)
```

Finally, when traversing very large result sets, where the underlying documents may be modified by other programs at the same time, you may wish to use MongoDB's Snapshot Mode. Imagine a busy site with hundreds of thousands of users. You are developing an analytics program to count users and build various statistics about usage patterns and so on. However, this analytics program is intended to run against the live, production database. Since this is such a busy site, real users are frequently performing actions on the site which may result in modifications to their corresponding user documents—while your analytics program is running. Due to quirks in MongoDB's cur-

soring mechanism, in this kind of situation your program could easily see duplicates in your query result set. Duplicate data could throw off the accuracy of your analysis program, and so it is best avoided. This is where Snapshot Mode comes in.

MongoDB's Snapshot Mode guarantees that documents which are modified during the lifetime of a query are returned only once in a result set. In other words, duplicates are eliminated, and you should not have to worry about them.

However, Snapshot Mode does have some limitations. Snapshot Mode cannot be used with sorting, nor can it be used with an index on any property other than `_id`.

To use Snapshot Mode with PyMongo, simply pass "snapshot=True" as a parameter to the `find()` method:

```
# Traverse the entire users collection, employing Snapshot Mode
# to eliminate potential duplicate results.
for user in dbh.users.find(snapshot=True):
    print user.get("username"), user.get("score", 0)
```

Updating Documents in a Collection

Update queries in MongoDB consist of two parts: a document spec which informs the database of the set of documents to be updated, and the update document itself.

The first part, the document spec, is the same as the query document which you use with `find()` or `find_one()`.

The second part, the update document, can be used in two ways. The simplest is to supply the full document which will replace the matched document in the collection. For example, suppose you had the following document in your users collection:

```
user_doc = {
    "username" : "janedoe",
    "firstname" : "Jane",
    "surname" : "Doe",
    "dateofbirth" : datetime(1974, 4, 12),
    "email" : "janedoe74@example.com",
    "score" : 0
}
```

Now let's say you wish to update the document with username "janedoe" to change the email address to "janedoe74@example2.com". We build a completely new document which is identical to the original, except for the new email address.

```
# first query to get a copy of the current document
import copy
old_user_doc = dbh.users.find_one({"username":"janedoe"})
new_user_doc = copy.deepcopy(old_user_doc)
# modify the copy to change the email address
new_user_doc["email"] = "janedoe74@example2.com"
# run the update query
# replace the matched document with the contents of new_user_doc
dbh.users.update({"username":"janedoe"}, new_user_doc, safe=True)
```

Building the whole replacement document can be cumbersome, and worse, can introduce race conditions. Imagine you want to increment the score property of the "janedoe" user. In order to achieve this with the replacement approach, you would have to first fetch the document, modify it with the incremented score, then write it back to the database. With that approach, you could easily lose other score changes if something else were to update the score in between you reading and writing it.

In order to solve this problem, the update document supports an additional set of MongoDB operators called "update modifiers". These update modifiers include operators such as atomic increment/decrement, atomic list push/pop and so on. It is very helpful to be aware of which update modifiers are available and what they can do when designing your application. Many of these will be described in their own recipes throughout this book.

To illustrate usage of "update modifiers", let's return to our original example of changing only the email address of the document with username "janedoe". We can use the $set update modifier in our update document to avoid having to query before updating. $set changes the value of an individual property or a group of properties to whatever you specify.

```
# run the update query, using the $set update modifier.
# we do not need to know the current contents of the document
# with this approach, and so avoid an initial query and
# potential race condition.
dbh.users.update({"username":"janedoe"},
    {"$set":{"email":"janedoe74@example2.com"}}, safe=True)
```

You can also set multiple properties at once using the $set update modifier:

```
# update the email address and the score at the same time
# using $set in a single write.
dbh.users.update({"username":"janedoe"},
    {"$set":{"email":"janedoe74@example2.com", "score":1}}, safe=True)
```

At the time of writing, the PyMongo driver, even if you specify a document spec to the update method which matches multiple documents in a collection, only applies the update to the first document matched.

In other words, even if you believe your update document spec matches every single document in the collection, your update will only write to one of those documents.

For example, let us imagine we wish to set a flag on every document in our users collection which has a score of 0:

```
# even if every document in your collection has a score of 0,
# only the first matched document will have its "flagged" property set to True.
dbh.users.update({"score":0},{"$set":{"flagged":True}}, safe=True)
```

In order to have your update query write multiple documents, you must pass the "multi=True" parameter to the update method.

```
# once we supply the "multi=True" parameter, all matched documents
# will be updated
dbh.users.update({"score":0},{"$set":{"flagged":True}}, multi=True, safe=True)
```

Although the default value for the multi parameter to the update method is currently False—meaning only the first matched document will be updated—this may change. The PyMongo documentation currently recommends that you explicitly set multi=False if you are relying on this default, to avoid breakage in future. Note that this should only impact you if you are working with a collection where your documents are not unique on the property you are querying on in your document spec.

Deleting Documents from a Collection

If you wish to permanently delete documents from a collection, it is quite easy to do so. The PyMongo Collection object has a `remove()` method. As with reads and updates, you specify which documents you want to remove by way of a document spec. For example, to delete all documents from the users collection with a score of 1, you would use the following code:

```
# Delete all documents in user collection with score 1
dbh.users.remove({"score":1}, safe=True)
```

Note that the `remove()` method takes a `safe` parameter. As mentioned in the earlier section "Write to a Collection Safely and Synchronously", it is recommended to set the `safe` parameter to `True` on write methods to ensure the operation has completed. It is also worth noting that `remove()` will not raise any exception or error if no documents are matched.

Finally, if you wish to delete all documents in a collection, you can pass None as a parameter to `remove()`:

```
# Delete all documents in user collection
dbh.users.remove(None, safe=True)
```

Clearing a collection with `remove()` differs from dropping the collection via `drop_collection()` in that the indexes will remain intact.

MongoDB Query Operators

As mentioned previously, MongoDB has quite a rich set of query operators and predicates. In Table 2-2 we provide a table with the meaning of each one, along with a sample usage and the SQL equivalent where applicable.

Table 2-2. MongoDB query operators

Operator	Meaning	Example	SQL Equivalent
$gt	Greater Than	"score":{"$gt":0}	>
$lt	Less Than	"score":{"$lt":0}	<
$gte	Greater Than or Equal	"score":{"$gte":0}	>=
$lte	Less Than or Equal	"score":{"$lte":0}	<=
$all	Array Must Contain All	"skills":{"$all":["mongodb","python"]}	N/A
$exists	Property Must Exist	"email":{"$exists":True}	N/A
$mod	Modulo X Equals Y	"seconds":{"$mod":[60,0]}	MOD()
$ne	Not Equals	"seconds":{"$ne":60}	!=
$in	In	"skills":{"$in":["c","c++"]}	IN
$nin	Not In	"skills":{"$nin":["php","ruby","perl"]}	NOT IN
$nor	Nor	"$nor":[{"language":"english"},{"country":"usa"}]	N/A
$or	Or	"$or":[{"language":"english"},{"country":"usa"}]	OR
$size	Array Must Be Of Size	"skills":{"$size":3}	N/A

If you do not fully understand the meaning or purpose of some of these operators immediately do not worry. We shall discuss the practical use of some of the more advanced operators in detail in Chapter 3.

MongoDB Update Modifiers

As mentioned in the section "Updating Documents in a Collection", MongoDB comes with a set of operators for performing atomic modifications on documents.

Table 2-3. MongoDB update modifiers

Modifier	Meaning	Example
$inc	Atomic Increment	"$inc":{"score":1}
$set	Set Property Value	"$set":{"username":"niall"}
$unset	Unset (delete) Property	"$unset":{"username":1}
$push	Atomic Array Append (atom)	"$push":{"emails":"foo@example.com"}
$pushAll	Atomic Array Append (list)	"$pushall":{"emails":["foo@example.com","foo2@example.com"]}
$addToSet	Atomic Append-If-Not-Present	"$addToSet":{"emails":"foo@example.com"}
$pop	Atomic Array Tail Remove	"$pop":{"emails":1}
$pull	Atomic Conditional Array Item Removal	"$pull":{"emails":"foo@example.com"}
$pullAll	Atomic Array Multi Item Removal	"$pullAll":{"emails":["foo@example.com", "foo2@example.com"]}
$rename	Atomic Property Rename	"$rename":{"emails":"old_emails"}

As with the MongoDB query operators listed earlier in this chapter, this table is mostly for your reference. These operators will be introduced in greater detail in Chapter 3.

CHAPTER 3
Common MongoDB and Python Patterns

After some time working with MongoDB and Python to solve different problems, various patterns and best practices begin to emerge. Just as with any programming language and database system, there are established approaches for modeling data along with known methods for answering queries as quickly and efficiently as possible.

While there are myriad sources of such knowledge for traditional RDBM systems like MySQL, there are far fewer resources available for MongoDB. This chapter is an attempt to address this.

A Uniquely Document-Oriented Pattern: Embedding

While the ability of MongoDB documents to contain sub-documents has been mentioned previously in this book, it has not been explored in detail. In fact, embedding is an extremely important modeling technique when working with MongoDB and can have important performance and scalability implications. In particular, embedding can be used to solve many data modeling problems usually solved by a join in traditional RDBMS. Furthermore, embedding is perhaps more intuitive and easier to understand than a join.

What exactly is meant by embedding? In Python terms, when the value of a key in a dictionary is yet another dictionary, we say that you are embedding the latter in the former. For example:

```
my_document = {
    "name":"foo document",
    "data":{"name":"bar document"}
}
```

Here, "data" is a sub-document embedded in the top-level document "my_document".

Embedding sub-documents can be a useful, natural technique to reduce clutter or namespace collisions. For example consider the case where a "user" document should reference Facebook, Twitter and IRC account usernames, passwords and associated details—in addition to storing a "username" property native to your applicaton:

```
user_doc = {
    "username":"foouser",
    "twitter":{
         "username":"footwitter",
         "password":"secret",
         "email":"twitter@example.com"
    },
    "facebook":{
        "username":"foofacebook",
        "password":"secret",
        "email":"facebook@example.com"
    },
    "irc":{
        "username":"fooirc",
        "password":"secret",
    }
}
```

Note that in MongoDB documents—just as in Python dictionaries—property names (a.k.a. keys) are unique. In other words, a single document can only ever have one "username" property. This rule also applies to properties in embedded sub-documents. This uniqueness constraint can actually be exploited and enable some useful patterns. Specifically, see the section titled "Fast Accounting Pattern".

Of course, embedded sub-documents can be queried against just like their top-level counterparts. For example, it would be completely legal to attempt to query for the above document in a collection called "users" with the following statement:

```
user_doc = dbh.users.find_one({"facebook.username":"foofacebook"})
```

As you can see, the dot (".") is used to denote keys in an embedded sub-document. This should be familiar to anybody who has worked with objects in JavaScript, where object-style access via the dot notation can be used in parallel with dictionary-style access via square brackets. As MongoDB uses JavaScript heavily internally, this choice of notation is unsurprising. JSON is JavaScript Object Notation, afterall. The dot notation can also be used in update statements with update modifiers such as `$set` to set the value of an individual sub-property:

```
# update modifiers such as $set also support the dot notation
dbh.users.update({"facebook.username":"foofacebook"},
    {"$set":{"facebook.username":"bar"}}, safe=True)
```

This use of embedded sub-documents is useful, but perhaps even more useful is to embed multiple sub-documents under a single key. In other words, a property whose value is a list or array of sub-documents. In MongoDB, this is a legal and very useful construct. This is a very natural way to model one-to-many relationships, or parent-child relationships. Consider the example of a "user" document which can reference multiple email addresses for that user. In the relational model, this would typically be achieved with two tables—one for users, and one for the email addresses associated with each user. A join query could then be used to retrieve a user along which each of its email addresses.

In MongoDB, a natural approach to model a one-to-many relationship would be to simply have a property "emails" on the user document, the value of which is an array containing sub-documents, each representing an associated email account. For example:

```
# A user document demonstrating one-to-many relationships using embedding
# Here we map multiple email addresses (along with whether or not the email
# is the user's primary email address) to a single user.
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
        {
         "email":"foouser2@example2.com",
         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}
```

Not only does this work, but MongoDB has some specific features to help working with this type of embedded structure. Just as you can query for documents by the value of sub-documents directly embedded in the top-level document, documents can also be located by the value of sub-documents embedded in arrays. To do this, simply use the same dot (".") notation, as described earlier in this section. MongoDB transparently searches through arrays for sub-documents.

Returning to our earlier example of a single user with multiple email addresses, consider the following code:

```
# A user document demonstrating one-to-many relationships using embedding
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
```

```
        {
         "email":"foouser2@example2.com",
         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}
# Insert the user document
dbh.users.insert(user_doc, safe=True)
# Retrieve the just-inserted document via one of its many email addresses
user_doc_result = dbh.users.find_one({"emails.email":"foouser1@example.com"})
# Assert that the original user document and the query result are the same
assert user_doc == user_doc_result
```

In addition to MongoDB understanding lists of sub-documents to enable querying for embedded values via the dot notation, there are also useful update modifiers. `$pull`, `$push` and their variants are the most helpful, enabling atomic append and removal of sub-documents to and from embedded lists. Consider the case where a user no longer wishes a particular email address to be linked to his or her account. The naive way to remove that email address from their user document would be to first query for their user document, modify it in your application code so it no longer contains the removed email address, and then send an update query to the database. Not only is this cumbersome, it also introduces a race condition, as the underlying user document may have been modified by another process in between your read and write:

```
# Naive method to remove an email address from a user document
# Cumbersome and has a race condition
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
        {
         "email":"foouser2@example2.com",
         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}
# Insert the user document
dbh.users.insert(user_doc, safe=True)
# Retrieve the just-inserted document via username
user_doc_result = dbh.users.find_one({"username":"foouser"})
# Remove the "foouser2@example2.com" email address sub-document from the embedded list
del user_doc_result["emails"][1]


# Now write the new emails property to the database
# May cause data to be lost due to the race between read and write
dbh.users.update({"username":"foouser"},{"$set":{"emails":user_doc_result}},
safe=True)
```

The three most common operations on sub-documents embedded in a list property are: Deletion, insertion and modification. Each of these can be performed atomically with the provided update modifiers. First let's demonstrate the use of `$pull` to remove the sub-document matching "foouser2@example2.com" in a simple and race-free way:

```
# Atomically remove an email address from a user document race-free using the
# $pull update modifier
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
        {
         "email":"foouser2@example2.com",
         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}
# Insert the user document
dbh.users.insert(user_doc, safe=True)
# Use $pull to atomically remove the "foouser2@example2.com" email sub-document
dbh.users.update({"username":"foouser"},
    {"$pull":{"emails":{"email":"foouser2@example2.com"}}}, safe=True)
```

In this example, `$pull` is used to match an embedded document with `"email":"foouser2@example2.com"` in the "emails" field. `$pull` will remove the entire document from the array in an atomic fashion, meaning there is no opportunity for a race condition. You can also use query modifiers with `$pull`, for example to remove all sub-documents with a "primary" value that is not equal to True, you could write the following:

```
# Use $pull to atomically remove all email sub-documents with primary not equal to True
dbh.users.update({"username":"foouser"},
    {"$pull":{"emails":{"primary":{"$ne":True}}}, safe=True)
```

The full range of query modifiers (see table in Chapter 2) are available for use, including $gt, $lt and so on. Additionally, `$pull` can be used with arrays containing atoms (numbers, strings, dates, ObjectIDs etc). In other words, `$pull` doesn't work only with embedded documents—if you store a list of primitive types in an array, you can remove elements atomically with `$pull` too.

The `$push` update modifier is used to atomically append an element to an array. At the time of writing, `$push` can only support adding items to the end of the array—there is no update modifier to add an element to the beginning of an array, or to insert it at an arbitrary index. `$push` is simple to use, because, unlike `$pull`, it does not take any field match or conditional arguments.

For example, to atomically add a new email address to our user document, we could use the following query:

```
# Use $push to atomically append a new email sub-document to the user document
new_email = {"email":"fooemail4@exmaple4.com", "primary":False}
dbh.users.update({"username":"foouser"},
    {"$push":{"emails":new_email}}, safe=True)
```

The final case is updating an existing sub-document in-place. This can be achieved using what is called the "positional" operator. The positional operator is represented by the dollar sign ("$"). Basically, it is replaced by the server with the index of the item matched by the document spec. For example, suppose we wish to make our user document's "foouser2@example2.com" email address primary. We could issue the following update query to modify it in-place:

```
# Demonstrate usage of the positional operator ($) to modify
# matched sub-documents in-place.
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
        {
         "email":"foouser2@example2.com",
         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}
# Insert the user document
dbh.users.insert(user_doc, safe=True)
# Now make the "foouser2@example2.com" email address primrary
dbh.users.update({"emails.email":"foouser2@example2.com"},
    {"$set":{"emails.$.primary":True}}, safe=True)
# Now make the "foouser1@example.com" email address not primary
dbh.users.update({"emails.email":"foouser1@example.com"},
    {"$set":{"emails.$.primary":False}}, safe=True)
```

Note that the `$` operator cannot be used with upserts (see section on upserts later in this chapter) additionally it only works with the first matched element.

When working with embedding, it is important to be aware of the performance characteristics of documents and sub-documents in MongoDB. First and foremost, when a document is fetched from the database to answer a query, the entire document—including any and all embedded sub-documents—is loaded into memory. This means that there is no extra cost (aside from the additional network and decode/encode CPU overhead incurred by a larger result set) to fetch embedded data. Once the top-level document has been retrieved, its sub-documents are immediately avaialble, too. Contrast this with a relational schema design utilizing joins, where the database may need to read from one or more additional tables to fetch associated data. Depending on the situation, these joins could impact query performance considerably.

Secondly, it is also very important to be aware that there is a size limit on documents in MongoDB. Additionally, the document size limit has been increased over successive major MongoDB releases. In MongoDB 1.4.x and 1.6.x, the maximum document size was 4MB but in 1.8.x it was increased to 16MB. One can expect that this limit may continue to increase—perhaps eventually to be arbitrarily large—but for now, keep in mind that documents have a finite size when modeling your data.

In practice, it is rare to reach even a 4MB document size, unless the design is such that documents continue to grow over time. For example, a scenario where new properties are created on an hourly or daily basis. In such cases, it is wise to ensure there is some application logic to handle purging old/expired embedded sub-documents to prevent the limit being hit.

Another example would be building a document publishing platform which embedded every single document posted by a user as a sub-document inside of the user document. While performance would be excellent since a single query for the user document could retieve all their published documents in a single shot, it is quite likely that some users would eventually publish more than 16MB of content.

Hence there is often a judgement call to be made when designing MongoDB schemas: To embed, or not to embed.

The alternative to embedding is storing the documents in a separate collection and performing a join in your own application code, by querying twice or more. Usually many-to-many relationships are modeled in this way, while one-to-many relationships are embedded.

Fast Lookups: Using Indexes with MongoDB

The role of indexes in MongoDB is very similar to their role in traditional RDBMS such as MySQL, PostgreSQL, etc. MongoDB offers two kinds of indexes out-of-the-box: Btree indexes and geospatial indexes. The btree indexes in MongoDB are much the same as the equivalents in MySQL or PostgreSQL. When in a relational system you would put an index on a column to get fast lookups on that field, you do an analogous thing in MongoDB by placing an index on a particular property in a collection. Just as

with an RDBMS, MongoDB indexes can span multiple fields (a.k.a. compound indexes)—useful if you know in advance that you will be querying based on the value of more than a single property. A compound index would be useful for example if you were querying documents by first name and last name. In MongoDB, btree indexes can have a "direction". This direction is only useful in the case of compound indexes, where the index direction should match the sort direction or range query direction for optimal performance. For example, if you are querying a range (say, A through C) on first name and last name and then sorting in ascending order on last name, your compound index direction should also be ascending.

Using a btree index will incur a performance hit on writes, as the database must now update the index in addition to the data. For this reason, it is wise to choose your indexes carefully. Avoid superfluous indexes if at all possible. Indexes also take up valuable storage—not so much of an issue with on-disk space today given the low price-per-terrabyte—but in memory, too. Your database will run fastest when it resides entirely in memory, and indexes can considerably add to its size. It is a classic Computer Science time vs. space tradeoff scenario.

MongoDB btree indexes can also be used to enforce a unique constraint on a particular property in a collection. By default, the `_id` primary key property has a unique index created in MongoDB. The unique constraint will prevent the protected property from ever having a duplicate value within the collection. This can be useful for values which are expected to be globally unique in the collection—a common example being usernames. Beware of over-reliance on this feature, however, as in the current implementation of sharding, unique indexes are supported only on the `_id` property—otherwise they are not enforced globally across the cluster.

Btree indexes also transparently support indexing multi-value properties, that is, properties where the value is an array. Each item in the array will be properly stored in the index to enable fast retrieval of the parent document. This can be useful for performant implementations of tagging, where each tag is stored as a string inside a "tags" list property on the document. Lookups for documents matching one or more of those tags (potentially using the `$in` query operator) will then be looked up in the "tags" index. Furthermore, btree indexes are equally well supported when placed on embedded sub-documents. If, for example, you store email addresses as embedded sub-documents, and you wish to be able to look up by the value of the email address using an index, MongoDB allows this. Hence the following document and query could take advantage of an index:

```
user_doc = {
    "username":"foouser",
    "emails":[
        {
         "email":"foouser1@example.com",
         "primary":True
        },
        {
         "email":"foouser2@example2.com",

         "primary":False
        },
        {
         "email":"foouser3@example3.com",
         "primary":False
        }
    ]
}

dbh.users.insert(user_doc)
# If we place an index on property "emails.email",
# e.g. dbh.users.create_index("emails.email")
# this find_one query can use a btree index
user = dbh.users.find_one({"emails.email":"foouser2@example2.com"})
```

Btree indexes in MongoDB are also important when performing server-side sorting of results. Without an index on the property you are sorting by, MongoDB will run out of memory when trying to sort anything greater than a relatively small results set (approx. 4Mb at time of writing). If you expect that you will be sorting result sets larger than 4Mb, you should specify an index on the sort key. It is easy to underestimate this and find exceptions are being raised on queries against larger, real-world data which were not anticipated during development.

To create an index with the PyMongo driver, use the `Collection.create_index()` method. This method can create single-key indexes or compound indexes. For a single-key index, only the key needs to be provided. A compound index is slightly more complicated—a list of 2-tuples (key, direction) must be supplied.

For example to create an index on the `username` property of a collection called `users`, you could write the following:

```
# Create index on username property
dbh.users.create_index("username")
```

To create a compound index, for example on the first_name and last_name, with an ascending direction, you could specify:

```
# Create a compound index on first_name and last_name properties
# with ascending index direction
dbh.users.create_index([("first_name", pymongo.ASCENDING), ("last_name",
pymongo.ASCENDING)])
```

Indexes in MongoDB each have names. By default, MongoDB will generate a name, but you may wish to give a custom name—particularly for compound indexes where the generated names are not especially readable by humans. To give a custom name during creation, supply the `name=<str>` parameter to the `create_index()` method:

```
# Create a compound index called "name_idx" on first_name and last_name properties
# with ascending index direction
dbh.users.create_index([
    ("first_name", pymongo.ASCENDING),
    ("last_name", pymongo.ASCENDING)
    ],
    name="name_idx")
```

It should be noted that index creation locks the database by default. For large collections, index creation can be time consuming. To help mitigate the impact of these operations on live, production databases, MongoDB is capable of building indexes in the background, without blocking database access. Building an index in the background may take slightly longer, and will still cause additional load on the system, but the database should otherwise remain available.

To specify that an index should be built in the background, pass the `background=True` parameter to the `create_index()` method:

```
# Create index in the background
# Database remains usable
dbh.users.create_index("username", background=True)
```

As mentioned earlier in this section, MongoDB btree indexes can be used to enforce a uniqueness constraint on a particular property. Unique constraints can be applied to both single-key indexes and compound indexes. To create an index with a unique constraint, simply pass the `unique=True` parameter to the `create_index()` method:

```
# Create index with unique constraint on username property
dbh.users.create_index("username", unique=True)
```

Be aware that unique indexes in MongoDB do not function exactly the same as indexes in RDBMS systems. In particular, a document with a missing property will be added to the index as if it the value of that property were null. This means that when a unique constraint is added to a btree index in MongoDB, the database will prevent you from having multiple documents in the collection which are missing the indexed property. For example, if you have created a unique index for the `username` property in a users collection, only one document in that collection may be permitted to lack a `username` property. Writes of additional documents without a `username` property will raise an exception. If you try to add a unique index to a collection which already contains duplicates on the specified property, MongoDB will (unsurprisingly) raise an exception. However, if you don't mind throwing away duplicate data, you can instruct MongoDB to drop all but the first document it finds using the `dropDups` or `drop_dups` parameter:

```
# Create index with unique constraint on username property
# instructing MongoDB to drop all duplicates after the first document it finds.
dbh.users.create_index("username", unique=True, drop_dups=True)
# Could equally be written:
# dbh.users.create_index("username", unique=True, dropDups=True)
```

Over time, your schema may evolve and you may find that a particular index is no longer needed. Fortunately, indexes are easy to remove in MongoDB. The `Collection.drop_index()` method deletes one index at a time. If you created your index with a custom name (as described above), you must supply this same name to the `drop_index()` method in order to delete it. For example:

```
# Create index on username property called "username_idx"
dbh.users.create_index("username", name="username_idx")
# Delete index called "username_idx"
dbh.users.drop_index("username_idx")
```

If, on the other hand, you did not give your index a custom name, you can delete by passing the original index specifier. For example:

```
# Create a compound index on first_name and last_name properties
# with ascending index direction
dbh.users.create_index([("first_name", pymongo.ASCENDING), ("last_name",
pymongo.ASCENDING)])
# Delete this index
dbh.users.drop_index([("first_name", pymongo.ASCENDING), ("last_name",
pymongo.ASCENDING)])
```

All indexes in a collection can be dropped in a single statement using the `Collection.drop_indexes()` method.

If you wish to programatically inspect the indexes on your collections from Python, you can use the `Collection.index_information()` method. This returns a dictionary in which each key is the name of an index. The value associated with each key is an additional dictionary. These second-level dictionaries always contain a special key called `key`, which is an entry containing the original index specifier—including index direction. This original index specifier was the data passed to the `create_index()` method when the index was first created. The second-level dictionaries may also contain additional options such as unique constraints and so on.

Location-based Apps with MongoDB: GeoSpatial Indexing

As mentioned in the previous section on indexes, MongoDB has support for two kinds of index: Btree and geospatial. Btree indexes have been covered quite thoroughly in the preceeding section, however we have not yet described GeoSpatial indexes.

First of all, let us discuss why geospatial indexing might be useful at all. Many apps today are being built with the requirement of location-awareness. Typically this translates into features where points of interest (POI) near a particular user location may be rapidly retrieved from a database. For example, a location-aware mobile app might wish to quickly fetch a list of nearby coffeeshops, based upon the current GPS coordinates. The complicating issue, fundamentally, is that the world is both quite large and quite full of interesting points—and so to try to answer such a query by iterating through the entire list of all POIs in the world to find ones which are nearby would take an unacceptably long time. Hence the need for some sort of GeoSpatial indexing, to speed up these searches.

Fortunately for anybody tasked with building location-aware applications, MongoDB is one of the rare few databases with out-of-the-box support for geospatial indexing. MongoDB uses geohashing, a public domain algorithm developed by Gustavo Niemeyer, which translates geographic proximity into lexical proximity. Hence, a database supporting range queries (such as MongoDB) can be efficiently used to query for points near and within bounds.

It should be noted that at present, MongoDB's geospatial indexing support is limited purely to point-based querying. The supplied operators can only be used for finding individual points—not routes or sub-shapes.

MongoDB provides the `$near` and `$within` operators which constitute the primary means for performing geospatial queries in the system. Using `$near`, you can efficiently sort documents in a collection by their proximity to a given point. The `$within` operator allows you to specify a bounds for the query. Supported boundary definitions include `$box` for a rectangular shape, `$circle` for a circle. In MongoDB 1.9 and up, the `$polygon` operator allows for convex and concave polygon boundaries.

Before you can use the geospatial queries, you must have a geospatial index. In MongoDB versions up to and including 1.8.x, geospatial indexes are limited to a single index per collection. This means that each document can have only one location property queried efficiently by MongoDB. This can have some important implications for schema design which is why it is good to know from the outset.

Geospatial indexes by default limit acceptable values for the location property on documents to those within GPS. That is, co-ordinates must be in the range -180 .. +180. If you have co-ordinates outside of this range, MongoDB will raise an exception when you attempt to create the geospatial index on the colleciton. If you wish to index values outside of the range of regular GPS, you can specify this at index creation time.

The location property on your documents must be either an array or sub-document where the first two items are the x and y co-ordinates to be indexed. The order of the co-ordinates (whether x,y or y,x) does not matter so long as it is consistent on all documents. For example, your document could look like any of the following:

```
# location property is an array with x,y ordering
user_doc = {
    "username":"foouser",
    "user_location":[x,y]
}
```

```
# location property is an array with y,x ordering
user_doc = {
    "username":"foouser",
    "user_location":[y,x]
}
```

```
import bson
# location property is a sub-document with y,x ordering
loc = bson.SON()
loc["y"] = y
loc["x"] = x
user_doc = {
    "username":"foouser",
    "user_location":loc
}
```

```
import bson
# location property is a sub-document with x,y ordering
loc = bson.SON()
loc["x"] = x
loc["y"] = y
user_doc = {
    "username":"foouser",
    "user_location":loc
}
```

Note that in Python the default dictionary type (`dict` class), order is not preserved. When using location in a sub-document from Python, use a `bson.SON` object instead. `bson.SON` comes with the PyMongo driver, and is used in exactly the same way as Python's `dict` class.

Once the documents in your collection have their location properties correctly formed, we can create the geospatial index. As with btree indexes, geospatial indexes in MongoDB are created with PyMongo's `Collection.create_index()` method. Due to the one-geospatial-index-per-collection limitation in MongoDB versions up to and including 1.8.x, if you are planning to query by other properties in addition to the location property, you can make your geospatial index a compound index. For exmaple, if you know that you will be searching your collection by both "username" and "user_location" properties, you could create a compound geo index across both fields. This can help to work around the single geospatial index limitation in many cases.

Returning to our example of documents in a collection called "users" with the location property being "user_location", we would create a geospatial index with the following statement:

```
# Create geospatial index on "user_location" property.
dbh.users.create_index([("user_location", pymongo.GEO2D)])
```

To create a compound geospatial index which would let us query efficiently on location and username, we could issue this statement:

```
# Create geospatial index on "user_location" property.
dbh.users.create_index([("user_location", pymongo.GEO2D), ("username",
pymongo.ASCENDING)])
```

Now that we have geospatial indexes available, we can try out some efficient location-based queries. The `$near` operator is pretty easy to understand, so we shall start there. As has already been explained, `$near` will sort query results by proximity to specified point. By default, `$near` will try to find the closest 100 results.

An important performance consideration which is not mentioned clearly in the official MongoDB documentation is that when using `$near`, you will almost always want to specify a maximum distance on the query. Without a clamp on the maximum distance, in order to return the specified number of results (default 100) MongoDB has to search through the entire database. This takes a lot of time. In most cases, a max distance of around 5 degrees should be sufficient. Since we are using decimal degrees (a.k.a GPS) co-ordinates, the units of max distance is degrees. 1 degree is roughly 69 miles. If you only care about a relatively small set of results (for example, the nearest 10 coffee shops), limiting the query to 10 results should also aid performance.

Let's start with an example of finding the nearest 10 users to the point 40, 40 limiting to a max distance of 5 degrees:

```
# Find the 10 users nearest to the point 40, 40 with max distance 5 degrees
nearest_users = dbh.users.find(
    {"user_location":
        {"$near" : [40, 40],
         "$maxDistance":5}}).limit(10)
# Print the users
for user in nearest_users:
    # assume user_location property is array x,y
    print "User %s is at location %s,%s" %(user["username"], user["user_location"][0],
        user["user_location"[1])
```

Next let us try using the `$within` geospatial operator to find points within a certain boundary. This can be useful when searching for POI's in a specific county/city or even well-defined neighbourhood within a city. In the real world, these boundaries are fuzzy and changing constantly, however there are good enough databases available for them to be useful.

To specify a rectangle to search within, you simply provide the lower-left and top-right co-ordinates as elements in an array. For example:

```
box = [[50.73083, -83.99756], [50.741404,  -83.988135]]
```

We could query for points within this bound by using the following geospatial query:

```
box = [[50.73083, -83.99756], [50.741404,  -83.988135]]
users_in_boundary = dbh.users.find({"user_location":{"$within": {"$box":box}}})
```

To specify a citcle to search within, you just supply the center point and the radius. As with `$maxDistance` mentioned previously, the units of the radius are degrees. Here is how we could make a geospatial lookup for 10 users within a radius of 5 degress centered at the point 40, 40:

```
users_in_circle = dbh.users.find({"user_location":{"$within":{"$center":[40, 40,
5]}}}).limit(10)
```

Notice that with the circle boundary using `$center`, we pass an array, the first two values of which are the co-ordinates of the center and the third parameter is the radius (in degrees).

All the queries we've mentioned so far which make use of a geospatial index actually are not entirely accurate. This is because they use a flat earth model where each arc degree of latitude and longitude translates to the same distance everywhere on the earth. In reality, the earth is a sphere and so these values differ depending upon where you are. Fortunately, MongoDB in 1.8.x and up implements a spherical model of the earth for geospatial queries.

The new spherical model can be used by employing the `$nearSphere` and `$circleSphere` variants on the `$near` and `$circle` operators. MongoDB's spherical model has a few extra caveats. First and foremost, you must use (longitude, latitude) ordering of your co-ordinates. While there are many other application and formats which use the (latitude, longitude) ordering, you should be careful to re-order to use with MongoDB's spherical model. Secondly, unlike the `$near` and `$center` operators we just described, the units for distances with `$nearSphere` and `$centerSphere` are always expressed in radians. This includes when using `$maxDistance` with `$nearSphere` or `$centerSphere`. Luckily, it is not difficult to convert from a more humanly-understandable unit such as kilometers to radians. To translate from kilometers to radians, simply divide the kilometer value by the radius of the earth which is approximately 6371 km (or 3959 miles). To demonstrate, let's try our earlier example of finding the 10 users nearest to the point 40,40 with a max distance of 5 km—but this time using the spherical model:

```
# Find the 10 users nearest to the point 40, 40 with max distance 5 degrees
# Uses the spherical model provided by MongoDB 1.8.x and up

earth_radius_km = 6371.0
max_distance_km = 5.0
max_distance_radians = max_distance_km / earth_radius_km
nearest_users = dbh.users.find(
    {"user_location":
        {"$nearSphere" : [40, 40],
         "$maxDistance":max_distance_radians}}).limit(10)
# Print the users
for user in nearest_users:
    # assume user_location property is array x,y
    print "User %s is at location %s,%s" %(user["username"], user["user_location"][0],
        user["user_location"[1])
```

Code Defensively to Avoid KeyErrors and Other Bugs

One of the tradeoffs of a document-oriented database versus a relational database is that the database does not enforce schema for you. For this reason, when working with MongoDB you must be vigilant in your handling of database results. Do not blindly assume that results will always have the properties you expect.

Check for their presence before accessing them. Although Python will generally raise a KeyError and stop execution, depending on your application this still may result in loss of data integrity. Consider the case of updating every document in a collection one-by-one—a single unforeseen KeyError could leave the database in an inconsistent state, with some documents having been updated and others not.

For example,

```
all_user_emails = []
for username in ("jill", "sam", "cathy"):
    user_doc = dbh.users.find_one({"username":username})
    # KeyError will be raised if any of these does not exist
    dbh.emails.insert({"email":user_doc["email"]})
```

Sometimes you will want to have a default fallback to avoid KeyErrors should a document be returned which is missing a property required by your program. The Python `dict` class' `get` method easily lets you specify a default value for a property should it be missing.

Often it makes sense to use this defensively. For example, imagine we have a collection of user documents. Some users have a "score" property set to a number, while others do not. It would be safe to have a default fallback of zero (0) in the case that no score property is present. We can take a missing score to mean a zero score. The `dict` class's `get` method lets us do this easily:

```
total_score = 0
for username in ("jill", "sam", "cathy"):
    user_doc = dbh.users.find_one({"username":username})
    total_score += user_doc.get("score", 0)
```

This approach can also work well when looping over embedded lists. For example, to defensively handle the case where a document representing a particular product does not yet have a list of suppliers embedded (perhaps because it is not yet on the market, or is no longer being produced) you might write code like this:

```
# Email each supplier of this product.
# Default value is the empty list so no special casing
# is needed if the suppliers property is not present.
for supplier in product_doc.get("suppliers", []):
    email_supplier(supplier)
```

MongoDB also makes no guarantees about the type of a property's value on a given document.

In most RDBMS implementations (the notable exception I'm aware of being SQLite) the database will quite rigorously enforce column types. If you try to insert a string into an integer column, the database will reject the write.

MongoDB, on the other hand, will only in exceptional circumstances reject such writes. If you set the value of a property on one document to be a string and on another document in the same collection set the value of that property to a number, it will very happily store that.

```
# Perfectly legal insert - MongoDB will not complain
dbh.users.insert({"username":"janedoe"})
# Also perfectly legal - MongoDB will not complain
dbh.users.insert({"username":1337})
```

When you couple this with Python's willingness to let you forgo explicitly typing your variables, you can soon run into trouble. Perhaps the most common scenario is when writing inputs from Web applications to the database. Most WSGI-based Python frameworks will send you all HTTP POST and GET parameter values as strings—regardless of whether or not they are in fact strings.

Thus it is easy to insert or update a numeric property with a value that is a string. The best way, of course, to avoid errors of this nature is to prevent the wrong type of data ever being written to the database in the first place. Thus, in the context of a Web application, validating and/or coercing the types of any inputs to write queries before issuing them is strongly advised. You may consider using the FormEncode or Colander Python libraries to help with such validation.

Update-or-Insert: Upserts in MongoDB

A relatively common task in a database-powered application is to update an existing entry, or if not found insert it as a new record. MongoDB conveniently supports this as a single operation, freeing you from having to implement your own "if-exists-update-else-insert" logic. 10gen refer to this type of write operation as an "upsert".

In PyMongo, there are three possible methods one can use to perform an upsert. These are `Collection.save()`, `Collection.update()` and `Collection.find_and_modify()`. We shall start by describing `Collection.save()` as it is the most straight forward method.

In the earlier section "Inserting a Document into a Collection" we used the `Collection.insert()` method to write a new document to the collection. However, we could have just as easily used the `save()` method. `save()` offers almost identical functionality to `insert()` with the following exceptions: `save()` can perform upserts and `save()` cannot insert multiple documents in a single call.

`save()` is quite easy to understand. If you pass it a document without an `_id` property it will perform an `insert()`, creating a brand new document. If, on the other hand, you pass it a document which contains an `_id` property it will update the existing document corresponding to that `_id` value, overwriting the already-present document with the document passed to `save()`.

This is the essence of an upsert: If a document already exists, update it. Otherwise create a new document.

`save()` can be useful because it supports both writing new documents and modifying existing documents, most likely ones retrieved from MongoDB via a read query. Having a single method which is capable of both modes of operation reduces the need for conditional handling in your client code, thus simplifying your program.

More useful, perhaps, is the "upsert=True" parameter which may be passed to `Collection.update()`. As has been discussed in the section "Updating Documents in a Collection" and is further described in the section "MongoDB Update Modifiers", the `update()` method supports the use of "update modifiers". These rich operators enable you to perform writes more complex than the basic "overwrite but keep existing _id" semantics of the `save()` method.

For example, imagine you are writing a method, `edit_or_add_session()`. This method either edits an existing document, or inserts a new one. Furthermore, semantics of the method dictate that the method will always be called with a session_id, but that the session_id may or may not already be present in the database. The naive implementation would first query to see whether a session document already existed, then conditionally either insert a new session document or update the existing document:

```
# Naive, bad implementation without upsert=True
def edit_or_add_session(description, session_id):
    # We must query first, becase we don't know whether this session_id already exists.
    # If we attempt to update a non-existent document, no write will occur.
    session_doc = dbh.sessions.find_one({"session_id":session_id})
    if session_doc:
        dbh.sessions.update({"session_id":session_id},
            {"$set":{"session_description":description}}, safe=True)
    else:
        dbh.sessions.insert({"session_description":description,
"session_id":session_id},
            safe=True)
```

However, by employing the upsert feature of `Collection.update()`, this can be implemented in a single method call—simplifying the code considerably and eliminating the need for an extra read query:

```
# Good implementation using upsert=True
def edit_or_add_session(description, session_id):
    dbh.sessions.update({"session_id":session_id},
        {"$set":{"session_description":description}}, safe=True, upsert=True)
```

Note that we could not have implemented the above semantics using `Cursor.save()` because we are testing for existence on the property "session_id" rather than "_id". Recall that the `save()` upsert method only works with "_id".

The trick to understanding upserts with the `update()` method is to consider the two execution cases seperately. In the case that the document already exists, then the update document will be processed normally—just as with a regular `update()` call without the "upsert=True" parameter. However, in the case that the document does not already exist, the document written (upserted) will match both the document spec supplied as the first argument to the `update()` call and the update document with any modifiers it contains. In other words, the observed behaviour is that the document is first created with the properties specified in the document spec—in this case, `"session_id":session_id`—and then the update document is executed against that. That may not accurately reflect what is happening internally in the daemon or driver, but that is equivalent to whatever does go on.

Atomic Read-Write-Modify: MongoDB's findAndModify

We've already introduced the atomic update modifiers supported by MongoDB. These are very powerful and enable race-free write operations of many kinds—including array manipulation and increment/decrement. However, it is often necessary to be able to modify the document atomically, and also return the result of the atomic operation—in a single step.

For example, imagine a billing system. Each user document has an "account_balance" property. There may be writes which alter the account balance—let's say an account top-up event which adds money to the account, and a purchase action which takes money from the account. Consider the following implementation:

```
# User X adds $20 to his/her account, so we atomically increment
# account_balance
dbh.users.update({"username":username}, {"$inc":{"account_balance":20}}, safe=True)
# Fetch the updated account balance to display to user
new_account_balance = dbh.users.find_one({"username":username},
    {"account_balance":1})["account_balance"]
```

This will work fine assuming no other writes to the account balance occur between the write and read operations. There is an obvious race condition. If a purchase action were to take place between the balance update and the balance read, the user may be displeased to be presented with a post-payment balance of less than what they expected!

```
# User X adds $20 to his/her account, so we atomically increment
# account_balance
dbh.users.update({"username":username}, {"$inc":{"account_balance":20}}, safe=True)

# In the meantime, in another thread or process, there is a payment operation,
# which decrements the account balance:
dbh.users.update({"username":username}, {"$dec":{"account_balance":5}}, safe=True)

# Fetch the updated account balance to display to user
new_account_balance = dbh.users.find_one({"username":username},
    {"account_balance":1})["account_balance"]
```

What you want in this kind of situation is a way to update the account balance and return the new value in a single, atomic operation. MongoDB's findAndModify command allows you to do just this. PyMongo provides a wrapper around findAndModify in the `Collection.find_and_modify()` method. Using this method, we can rewrite the code to a single, atomic operation:

```
# User X adds $20 to his/her account, so we atomically increment
# account_balance and return the resulting document
ret = dbh.users.find_and_modify({"username":username},
    {"$inc":{"account_balance":20}}, safe=True, new=True)
new_account_balance = ret["account_balance"]
```

Fast Accounting Pattern

Many of the applications people are building today are realtime with very large data sets. That is to say, users expect changes they make to be reflected within the application instantly. For example if a user wins a new high score in a multiplayer game, they expect the high score table in the game to be updated immediately. However, it may not be a single high score table which must be updated. Perhaps you are also ranking by high score this week, or this month, or even this year. Furthermore, as the application developer you may wish to keep a detailed log of each change—including when it occured,

what the client IP address was, the software version of the client, etc.—per user for analytics purposes.

This pattern isn't limited to high scores. Similar high performance accounting requirements exist for in-app social activity feeds, billing systems which charge per byte, and so on. Not only do these counts need to be fast to read from the database, they needs to be quick to write. Additionally, with potentially millions of users, the data set can grow very large, very quickly.

You might be tempted to keep only a detailed log, with one document per change. Totals for the various time periods can then be calculated by an aggregate query across the collection. This may work well initially, with only hundreds or thousands of documents to be aggregated to compute the result. However when the number of these documents grows into the millions or even billions—which they may easily do in a successful application—this approach will quickly become intractable.

Of course, as with many problems in Computer Science, the solution is ultimately a form of caching. MongoDB and its document-oriented data model gives us a nice idiom for this kind of period-based accounting, however. Given that we are counting on a per-user basis, we can utilize embedded sub-documents containing property names derived from time period. Consider for example a high score table supporting resolutions of week, month and total (across all time).

For the weekly resolution score counts, we can name the properties after the current week number. To disambiguate over multiple years, we can include the four-digit year in the key:

```
# Store weekly scores in sub-document
user_doc = {
    "scores_weekly":{
        "2011-01":10,
        "2011-02":3,
        "2011-06":20
     }
}
```

To fetch the score for this week, we simply execute the following simple dictionary lookup:

```
# Fetch the score for the current week
import datetime
now = datetime.datetime.utcnow()
current_year = now.year
current_week = now.isocalendar()[1]
# Default missing keys to a score of zero
user_doc["scores_weekly"].get("%d-%d" %(current_year, current_week), 0)
```

Such a lookup is incredibly fast. There is no aggregation to perform whatsoever. With this pattern, we can also write very quickly and safely. Because we are counting, we can take advantage of MongoDB's atomic increment and decrement update modifiers, `$inc` and `$dec`. Atomic update operators are great because they ensure the underlying data is in a consistent state and help to avoid nasty race conditions. Especially when dealing with billing, accurate counts are very important.

Imagine we wish to increment the user's score for this week by 24. We can do so with the following query:

```
# Update the score for the current week
import datetime
username = "foouser"
now = datetime.datetime.utcnow()
current_year = now.year
current_week = now.isocalendar()[1]
# Use atomic update modifier to increment by 24
dbh.users.update({"username":username},
    {"$inc":{"scores_weekly.%s-%s" %(current_year, current_week):24}},
    safe=True)
```

If the application needs to track multiple time-periods, these can be represented as additional sub-documents:

```
# Store daily, weekly, monthly and total scores in user document
user_doc = {
    "scores_weekly":{
        "2011-01":10,
        "2011-02":3,
        "2011-06":20
     },
     "scores_daily":{
        "2011-35":2,
        "2011-59":7,
        "2011-83":15
     },
     "scores_monthly":{
         "2011-09":30,
         "2011-10":43,
         "2011-11":24
     },
     "score_total":123
}
```

Of course, in your writes, you should increment the counts for each time period:

```
# Update the score for the current week
import datetime
username = "foouser"
now = datetime.datetime.utcnow()
current_year = now.year
current_month = new.month
current_week = now.isocalendar()[1]
current_day = now.timetuple().tm_yday
# Use atomic update modifier to increment by 24
dbh.users.update({"username":username},
    {"$inc":{
        "scores_weekly.%s-%s" %(current_year, current_week):24,
        "scores_daily.%s-%s" %(current_year, current_day):24,
        "scores_monthly.%s-%s" %(current_year, current_month):24,
        "score_total":24,
        }
},
safe=True)
```

In cases where you want to report the count immediately after the update, that can be achieved by using the findAndModify command (described in previous section) to return the new document after the update has been applied.

This pattern can help greatly with high speed counting. If more detailed logs are still needed—such as when each action took place—feel free to maintain those in a separate collection. This summary data is most useful for extremely fast reads and writes.

CHAPTER 4

MongoDB with Web Frameworks

While MongoDB can be used in all sorts of applications, its most obvious role is as the database backend for a web application. These days, a great many mobile and tablet applications are functioning as "fat clients" to the same HTTP-based API's as browser-based web applications; hence mobile and tablet apps need the same sort of backend database infrastructure as more traditional web apps.

Many organizations and engineers are finding the advantages of MongoDB's document-oriented architecture compelling enough to migrate parts or even entire applications from traditional RDBMS such as MySQL to MongoDB. Numerous well-known companies have built their whole application from the ground up on MongoDB.

It is my opinion that for the vast majority of web, mobile and tablet applications, MongoDB is a better starting point than RDBMS technology such as MySQL. This chapter is an attempt to get you off the ground using MongoDB with three common Python web frameworks: Pylons, Pyramid and Django.

Pylons 1.x and MongoDB

Pylons is one of the older WSGI-based Python web frameworks, dating back to September 2005. Pylons reached version 1.0 in 2010 and is considered very stable at this point. In fact, not much development is planned for Pylons 1.x any more; all new development is happening in Pyramid (see "Pyramid and MongoDB" on page 49 for details). The Pylons philosophy is the precise opposite of "one-size-fits-all." Application developers are free to choose from the various database, templating, session store options available. This kind of framework is excellent when you aren't exactly sure what pieces you will need when you are starting work on your application. If it turns out you need to use an XML-based templating system, you are free to do so.

The existence of Pyramid aside, Pylons 1.x is a very capable and stable framework. As Pylons is so modular, it is easy to add MongoDB support to it.

First you need to create a virtual environment for your project. These instructions assume you have the `virtualenv` tool installed on your system. Install instructions for the `virtualenv` tool are provided in the first chapter of this book.

To create the virtual environment and install Pylons along with its dependencies, run the following commands:

```
virtualenv --no-site-packages myenv
cd myenv
source bin/activate
easy_install pylons
```

Now we have Pylons installed in a virtual environment. Create another directory named whatever you like in which to create your Pylons 1.x project, change your working directory to it, then execute:

```
paster create -t pylons
```

You will be prompted to enter a name for your project, along with which template engine you want to use and whether or not you want the SQLAlchemy Object-Relational Mapper (ORM). The defaults ("mako" for templating engine, False to SQLAlchemy) are fine for our purposes—not least since we are demonstrating a NoSQL database!

After I ran the `paster create` command, a "pylonsfoo" directory (I chose "pylonsfoo" as my project name) was created with the following contents:

```
MANIFEST.in
README.txt
development.ini
docs
ez_setup.py
pylonsfoo
pylonsfoo.egg-info
setup.cfg
setup.py
test.ini
```

Next you need to add the PyMongo driver as a dependency for your project. Change your working directory to the just-created directory named after your project. Open the `setup.py` file present in it with your favourite editor. Change the `install_requires` list to include the string `pymongo`. Your file should look something like this:

```
try:
    from setuptools import setup, find_packages
except ImportError:
    from ez_setup import use_setuptools
    use_setuptools()
    from setuptools import setup, find_packages

setup(
    name='pylonsfoo',
    version='0.1',

    description='',
    author='',
    author_email='',
    url='',
    install_requires=[
        "Pylons>=1.0", "pymongo",
    ],
    setup_requires=["PasteScript>=1.6.3"],
    packages=find_packages(exclude=['ez_setup']),
    include_package_data=True,
    test_suite='nose.collector',
    package_data={'pylonsfoo': ['i18n/*/LC_MESSAGES/*.mo']},
    #message_extractors={'pylonsfoo': [
    #        ('**.py', 'python', None),
    #        ('templates/**.mako', 'mako', {'input_encoding': 'utf-8'}),
    #        ('public/**', 'ignore', None)]},
    zip_safe=False,
    paster_plugins=['PasteScript', 'Pylons'],
    entry_points="""
    [paste.app_factory]
    main = pylonsfoo.config.middleware:make_app

    [paste.app_install]
    main = pylons.util:PylonsInstaller
    """,
)
```

Now you need to fetch the PyMongo driver into your virtual environment. It is easy to do this by executing:

```
python setup.py develop
```

Your Pylons app is now ready to be configured with a MongoDB connection. First, we shall create a config file for development

```
cp development.ini.sample development.ini
```

Next open the file `development.ini` in your favourite editor. Underneath the section `[app:main]` add the following two variables, changing the URI and database names to whatever works for your set up:

```
mongodb.url = mongodb://localhost
mongodb.db_name = mydb
```

You can now try starting your project with the following command:

```
paster serve --reload development.ini
```

You should see the following output:

```
Starting subprocess with file monitor
Starting server in PID 82946.
serving on http://127.0.0.1:5000
```

If you open the URL *http://localhost:5000/* in a web browser, you should see the default Pylons page. This means that you have correctly set up your project. However, we do not yet have a way to talk to MongoDB.

Now that the configuration is in place, we can tell Pylons how to connect to MongoDB and where to make the PyMongo connection available to our application. Pylons provides a convenient place for this in `<project_name>/lib/app_globals.py`. Edit this file and change the contents to the following:

```
from beaker.cache import CacheManager
from beaker.util import parse_cache_config_options
from pymongo import Connection
from pylons import config

class Globals(object):
    """Globals acts as a container for objects available throughout the
    life of the application

    """

    def __init__(self, config):
        """One instance of Globals is created during application
        initialization and is available during requests via the
        'app_globals' variable

        """
        mongodb_conn = Connection(config['mongodb.url'])
        self.mongodb = mongodb_conn[config['mongodb.db_name']]
        self.cache = CacheManager(**parse_cache_config_options(config))
```

Once this has been set up, a PyMongo `Database` instance will be available to your Pylons controller actions through the globals object. To demonstrate, we will create a new controller named "mongodb" with the following command:

```
paster controller mongodb
```

You should see a file named mongodb.py in the `<project_name>/controllers` directory. For demonstration purposes, we shall modify it to increment a counter document in MongoDB every time the controller action is run.

Open this file with your editor. Modify it to look like the following (remembering to change the `from pylonsfoo` import line into whatever you named your project):

```
import logging

from pylons import app_globals as g, request, response, session, tmpl_context as c, url
from pylons.controllers.util import abort, redirect

from pylonsfoo.lib.base import BaseController, render

log = logging.getLogger(__name__)

class MongodbController(BaseController):

    def index(self):
        new_doc = g.mongodb.counters.find_and_modify({"counter_name":"test_counter"},
            {"$inc":{"counter_value":1}}, new=True, upsert=True , safe=True)
        return "MongoDB Counter Value: %s" % new_doc["counter_value"]
```

Once you have saved these changes, in a web browser open the URL *http://localhost:5000/mongodb/index*. Each time you load this page, you should see a document in the counters collection be updated with its `counter_value` property incremented by 1.

Pyramid and MongoDB

Pyramid is an unopinionated web framework which resulted from the merge of the `repoz.bfg` framework into the Pylons umbrella project (not to be confused with Pylons 1.x, the web framework). Pyramid can be considered to be a bit like a Pylons 2.0; it is a clean break, a completely new codebase with no code-level backwards compatibility with Pylons 1.x.

However, many of the concepts are very similar to the older Pylons 1.x. Pyramid is where all the new development is happening, and it has fantastic code test coverage and documentation. This section is only intended to be a brief introduction to setting up a Pyramid project with a MongoDB connection. To learn more, refer to the excellent Pyramid book and other resources available free online at *http://docs.pylonsproject.org/*.

On its own, Pyramid is just a framework, a set of libraries you can use. Projects are most easily started from a what is known as a scaffold. A scaffold is like a project skeleton which sets up plumbing and placeholders for your code.

A number of different scaffolds are included with Pyramid, offering different persistence options, URL mappers and session implementations. Conveniently enough, there is a scaffold called `pyramid_mongodb` which will build out a skeleton project with MongoDB support for you. `pyramid_mongodb` eliminates the need for you to worry about writing the glue code to make a MongoDB connection available for request processing in Pyramid.

As with Pylons 1.x, to start using Pyramid you first need to create a virtual environment for your project. These instructions assume you have the `virtualenv` tool installed on your system. Install instructions for the `virtualenv` tool are provided in the first chapter of this book.

To create the virtual environment and install Pyramid and its dependencies, run the following commands:

```
virtualenv --no-site-packages myenv
cd myenv
source bin/activate
easy_install pyramid
```

Take note of the line sourcing the `bin/activate` script. It is important to remember to do this once in every shell to make the virtual environment active. Without this step, your default system Python install will be invoked, which does not have Pyramid installed.

Now your virtual environment has Pyramid and all its dependencies installed. However, you still need `pyramid_mongodb` and its dependencies like PyMongo etc. Run the following command to install `pyramid_mongodb` in your virtual environment:

```
easy_install pyramid_mongodb
```

With Pyamid and `pyramid_mongodb` installed in your virtual environment, you are ready to create a Pyramid project with MongoDB support. Decide upon a project directory and a project name. From that project directory execute in the shell:

```
paster create -t pyramid_mongodb <project_name>
```

After I ran the `paster create` command, a "mongofoo" directory (I chose "mongofoo" as my project name) was created with the following contents:

```
README.txt
development.ini
mongofoo
mongofoo.egg-info
production.ini
setup.cfg
setup.py
```

The default configuration files tell Pyramid to connect to a MongoDB server on `localhost`, and a database called "mydb". If you need to change that, simply edit the `mongodb.url` and `mongodb.db_name` settings in the INI-files. Note that if you do not have a MongoDB server running at the address configured in the INI-file, your Pyramid project will fail to start.

Before you can run or test your app, you need to execute:

```
python setup.py develop
```

This will ensure any additional dependencies are installed. To run your project in debug mode, simply execute:

```
paster serve --reload development.ini
```

If all went well, you should see output like the following:

```
Starting subprocess with file monitor
Starting server in PID 54019.
serving on 0.0.0.0:6543 view at http://127.0.0.1:6543
```

You can now open *http://localhost:6543/* in a web browser and see your Pyramid project, with the default template. If you made it this far, Pyramid is correctly installed and `pyramid_mongodb` was able to successfully connect to the configured MongoDB server.

The `pyramid_mongodb` scaffold sets up your Pyramid project in such a way that there is a PyMongo `Database` object attached to each `request` object. To demonstrate how to use this, open the file `<project_name>/views.py` in your favourite editor. There should be a skeletal Python function named `my_view`:

```
def my_view(request):
    return {'project':'mongofoo'}
```

This is a very simple Pyramid view callable. Pyramid view callables are similar to controller actions in Pylons 1.x, and are where much of the application-defined request processing occurs. Since view callables are passed an instance of a `request` object, which in turn has a property containing the PyMongo `Database` object, this is an ideal place to interact with MongoDB.

Imagine a somewhat contrived example whereby we wish to insert a document into a collection called "page_hits" each time the `my_view` view callable is executed. We could do the following:

```
import datetime
def my_view(request):
    new_page_hit = {"timestamp":datetime.datetime.utcnow(), "url":request.url}
    request.db.page_hits.insert(new_page_hit, safe=True)
    return {"project":"mongofoo"}
```

If you now reload the web page at *http://localhost:6543* you should see a collection called "page_hits" in the MongoDB database you configured in your INI-file. In this collection there should be a single document for each time the view has been called.

From here, you should be well on your way to building web applications with Pyramid and MongoDB.

Django and MongoDB

Django is proabably the most widely-used Python web framework. It has an excellent community and many plugins and extension modules. The Django philosophy is the opposite of Pylons or Pyramid; it offers one well-integrated package including its own database and ORM layer, templating system, URL mapper, admin interface and so on.

There are a number of options for running Django with MongoDB. Since the Django ORM is such an integral part of Django, there is a project known as Django MongoDB Engine which attempts to provide a MongoDB backend for the Django ORM. However, this approach heavily abstracts the underlying query language and data model, along with many of the low-level details discussed in the course of the book. If you are already familiar with the Django ORM, enjoy using it, and are willing to use a fork of Django, Django MongoDB Engine is worth a look. You can find more information at the website *http://django-mongodb.org/*.

Our recommended approach for now is to use the PyMongo driver directly with Django. Be aware, however, that with this method, the Django components which depend on the Django ORM (admin interface, session store etc) will not work with MongoDB. There is another project called Mango which attempts to provide MongoDB-backed session and authentication support for Django. You can find Mango at *https://github.com/vpulim/mango*.

10gen have made a sample Django app with PyMongo integration available. This sample app can be found at *https://github.com/mdirolf/DjanMon*. We shall step through running the sample Django + MongoDB app on your local machine, and examine how it sets up the MongoDB connection.

First, download the sample Django project. If you already have the git command line tools installed, you can run `git clone https://github.com/mdirolf/DjanMon.git`. Otherwise, simply click the "Download" button at *https://github.com/mdirolf/DjanMon*.

In order to successfully run the sample app, you will need to build a Python virtual environment with Django, pymongo and PIL installed. As with Pylons and Pyramid, you will first need to have the `virtualenv` tool installed on your system—details on how to do this are covered in the first chapter of this book. Once you have `virtualenv` installed, chose a directory in which to store virtual env, then execute the following shell commands in it:

```
virtualenv --no-site-packages djangoenv
cd djangoenv
source bin/activate
pip install django pymongo PIL
```

This will create your virtual environment, activate it and then install Django, the PyMongo driver and the PIL image manipulation library (required by the demo app) into it. Assuming this all succeeded, you are ready to start the sample app development server. Note that the sample app expects a MongoDB server to be running on `localhost`.

Now we can run 10gen's Django demonstration app. Change your current working directory to your copy of the "DjanMon" project. There should be a file called `manage.py` in the current working directory. The app can be run with the Django development server with the command:

```
python manage.py runserver
```

You should see output on the console like the following:

```
Validating models...

0 errors found
Django version 1.3, using settings 'DjanMon.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Now you can open a web browser and visit *http://localhost:8000/* and see the demonstration app! The app lets you create simple messages (optionally with attached images) which are persisted in MongoDB.

Let us examine how the sample app works. Take a look at the file `status/view.py`. This is where the MongoDB connection is created, and where most of the application logic is stored. In their Django + MongoDB integration example, 10gen take a different approach from the others outlined in this chapter. They create a PyMongo `Database` in the global scope of the views module, rather than attaching it to request objects as in Pyramid or making it a framework-wide global as in Pylons 1.x:

```
import datetime
import string
import random
import mimetypes
import cStringIO as StringIO

from PIL import Image
from django.http import HttpResponse
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from pymongo.connection import Connection
from pymongo import DESCENDING
import gridfs

db = Connection().sms
```

This approach is simple and works fine for a demo. However, in larger Django projects with multiple installed applications (in this sample, there is a single installed app—it is named "status") this would require a separate PyMongo connection pool to be maintained for each app. This results in wasted MongoDB connections and duplicated code. Instead, it would be recommended to create the connection in a single place and import it in any other modules which need access.

This should be enough information to get you started building your Django MongoDB application.

Going Further

In this book we have tried to give you a solid grasp of how to leverage MongoDB in real-world applications. You should have a decent understanding of how to go about modeling your data, writing effective queries and avoiding concurrency problems such as race conditions and deadlocks. There are a number of other advanced topics which we didn't have space for in this book but are nonetheless worth looking into as you build your application. Notably, map-reduce enables computing aggregates efficiently. Sharding permits you to scale your application beyond the available memory of a single machine. GridFS allows you to store binary data in MongoDB. Capped Collections are a special type of collection, which look like a circular buffer and are great for log data. With these features at your disposal, Python and MongoDB are extremely powerful tools to have in your toolbox when developing an application.