Practical Python

MAGNUS LIE HETLAND

Apress™

Practical Python
Copyright ©2002 by Magnus Lie Hetland

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-006-6

Printed and bound in the United States of America 12345678910
Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Karen Watterson
Marketing Manager: Stephanie Rodriguez
Technical Reviewers: Jason Gilmore and Alex Martelli
Project Managers: Erin Mulligan and Tory McLearn
Copy Editor: Nancy Rapoport
Production Editor: Tory McLearn
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Carol Burbo
Cover Designer: Kurt Krames

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany
In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

About the Author xv
About the Technical Reviewers xvi
Acknowledgments xvii
Introduction xix
Chapter 1 Instant Hacking:The Basics 1
Chapter 2 Lists and Tuples 37
Chapter 3 Working with Strings 63
Chapter 4 Dictionaries: When Indices Won't Do 79
Chapter 5 Conditionals, Loops, and Some Other Statements 96
Chapter 6 Abstraction 129
Chapter 7 More Abstraction 167
Chapter 8 Exceptions 193
Chapter 9 Magic Methods, Properties, and Iterators 209
Chapter 10 Batteries Included 245
Chapter 11 Files and Stuff 303
Chapter 12 Graphical User Interfaces 325
Chapter 13 Playful Programming 349
Chapter 14 Project 1: Instant Markup 365
Chapter 15 Project 2: Painting a Pretty Picture 387
Chapter 16 Project 3: XML for All Occasions 401
Chapter 17 Project 4: In the News 421

Chapter 18 Project 5: A Virtual Tea Party 439

Chapter 19 Project 6: Remote Editing with CGI 461

Chapter 20 Project 7: Your Own Bulletin Board 481

Chapter 21 Project 8:File Sharing with XML-RPC 501

Chapter 22 Project 9: File Sharing II—Now with GUI! 525

Chapter 23 Project 10: Do-It-Yourself Arcade Game 535

Appendix A The Short Version 559

Appendix B Python Reference 571

Appendix C Online Resources 589

Index 593

CHAPTER 5

Conditionals, Loops, and Some Other Statements

By now, I'm sure you are getting a bit impatient. All right—all these data types are just dandy, but you can't really *do* much with them, can you?

Let's crank up the pace a bit. You've already encountered a couple of statement types (`print` statements, `import` statements, assignments). Let's first take a look at some more ways to use these before diving into the world of *conditionals* and *loops*. Then, you'll see how *list comprehensions* work almost like conditionals and loops, even though they are expressions, and finally you take a look at `pass`, `del`, and `exec`.

More About `print` and `import`

As you learn more about Python, you may notice that some aspects of Python that you thought you knew have hidden features just waiting to pleasantly surprise you. Let's take a look at a couple of such nice features in `print` and `import`.

Printing with Commas

You've seen how `print` can be used to print an expression, which is either a string or is automatically converted to one. But you can actually print more than one expression, as long as you separate them with commas:

```
>>> print 'Age:', 42
Age: 42
```

As you can see, a space character is inserted between each argument.

NOTE *The arguments of* `print` *do* not *form a tuple, as one might expect:*

```
>>> 1, 2, 3
(1, 2, 3)
>>> print 1, 2, 3
1 2 3
>>> print (1, 2, 3)
(1, 2, 3)
```

This behavior can be very useful if you want to combine text and variable values without using the full power of string formatting:

```
>>> name = 'Gumby'
>>> salutation = 'Mr.'
>>> greeting = 'Hello,'
>>> print greeting, salutation, name
Hello, Mr. Gumby
```

NOTE *If the* `greeting` *string had no comma, how would you get the comma in the result? You couldn't just use*

```
print greeting, ',', salutation, name
```

because that would introduce a space before the comma. One solution would be the following:

```
print greeting + ',', salutation, name
```

Here the comma is simply added to the greeting.

If you add a comma at the end, your next `print` statement will continue printing on the same line. For instance, the statements

```
print 'Hello,',
print 'world!'
```

print out `Hello, world!`

Importing Something as Something Else

Usually when you import something from a module you either use

```
import somemodule
```

or

```
from somemodule import somefunction
```

or

```
from somemodule import *
```

The latter should only be used when you are certain that you want to import *everything* from the given module. But what if you have two modules each containing a function called `open`, for instance—what do you do then? You could simply import the modules using the first form, and then use the functions as follows:

```
module1.open(...)
module2.open(...)
```

But there is another option: You can add an `as` clause to the end and supply the name you want to use, either for the entire module:

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0
```

or for the given function:

```
>>> from math import sqrt as foobar
>>> foobar(4)
2.0
```

For the `open` functions you might use the following:

```
from module1 import open as open1
from module2 import open as open2
```

Assignment Magic

The humble assignment statement also has a few tricks up its sleeve.

Sequence Unpacking

You've seen quite a few examples of assignments, both for variables and for parts of data structures (such as positions and slices in a list, or slots in a dictionary), but there is more. You can perform several different assignments *simultaneously:*

```
>>> x, y, z = 1, 2, 3
>>> print x, y, z
1 2 3
```

Doesn't sound useful? Well, you can use it to switch the contents of two variables:

```
>>> x, y = y, x
>>> print x, y, z
2 1 3
```

Actually, what I'm doing here is called "sequence unpacking"—I have a sequence of values, and I unpack it into a sequence of variables. Let me be more explicit:

```
>>> values = 1, 2, 3
>>> values
(1, 2, 3)
>>> x, y, z = values
>>> x
1
```

This is particularly useful when a function or method returns a tuple; let's say that you want to retrieve (and remove) a random key-value pair from a dictionary. You can then use the `popitem` method, which does just that, returning the pair as a tuple. Then you can unpack the returned tuple directly into two variables:

```
>>> scoundrel = {'name': 'Robin', 'girlfriend': 'Marion'}
>>> key, value = scoundrel.popitem()
>>> key
```

```
'girlfriend'
>>> value
'Marion'
```

This allows functions to return more than one value, packed as a tuple, easily accessible through a single assignment. The sequence you unpack must have exactly as many items as the targets you list on the left of the `=` sign; otherwise Python raises an exception when the assignment is performed.

Chained Assignments

Chained assignments are used as a shortcut when you want to bind several variables to the same value. This may seem a bit like the simultaneous assignments in the previous section, except that here you are only dealing with one value:

```
x = y = somefunction()
```

is the same as

```
y = somefunction()
x = y
```

Note that the statements above may *not* be the same as

```
x = somefunction()
y = somefunction()
```

For more information, see the section about the identity operator (`is`), later in this chapter.

Augmented Assignments

Instead of writing `x = x + 1` you can just put the expression operator (in this case `+`) before the assignment operator (`=`) and write `x += 1`.This is called an augmented assignment, and it works with all the standard operators, such as `*`, `/`, `%`, and so on:

```
>>> x = 2
>>> x += 1
>>> x *= 2
>>> x
6
```

It also works with other data types:

```
>>> fnord = 'foo'
>>> fnord += 'bar'
>>> fnord
'foobar'
```

Augmented assignments can make your code more compact and concise, yet some argue that it can also make it harder to read.

TIP *In general, you should not use* += *with strings,* especially *if you are building a large string piece by piece in a loop (see the section "Loops" later in this chapter for more information about loops). Each addition and assignment needs to create a new string, and that takes time, making your program slower. A much better approach is to append the small strings to a list, and use the string method* `join` *to create the big string when your list is finished.*

Blocks: The Joy of Indentation

This isn't really a type of statement but something you're going to need when you tackle the next two sections.

A block is a *group* of statements that can be executed if a condition is true (conditional statements), or executed several times (loops), and so on. A block is created by *indenting* a part of your code; that is, putting spaces in front of it.

NOTE *You can use tab characters to indent your blocks as well. Python interprets a tab as moving to the next tab stop, with one tab stop every eight spaces, but the standard and preferable style is to use spaces only, no tabs, and specifically four spaces per each level of indentation.*

Each line in a block must be indented by the *same amount*. The following is pseudocode (not real Python code) but shows how the indenting works:

```
this is a line
this is another line:
    this is another block
    continuing the same block
    the last line of this block
phew, there we escaped the inner block
```

In many languages a special word or character (for example, "`begin`" or "`{`") is used to start a block, and another (such as "`end`" or "`}`") is used to end it. In Python, a colon ("`:`") is used to indicate that a block is about to begin, and then every line in that block is indented (by the same amount). When you go back to the same amount of indentation as some enclosing block, you know that the current block has ended.

Now I'm sure you are curious to know how to use these blocks. So, without further ado, let's have a look.

Conditions and Conditional Statements

Until now you've only written programs in which each statement is executed, one after the other. It's time to move beyond that and let your program choose whether or not to execute a block of statements.

So That's What Those Boolean Values Are For

Now you are finally going to need those *truth values* we've been bumping into repeatedly.

NOTE *If you've been paying close attention, you noticed the sidebar in Chapter 1, "Sneak Peek: The* `if` *Statement," which describes the* `if` *statement. I haven't really introduced it formally until now, and as you'll see, there is a bit more to it than what I've told you so far.*

The following values are considered by the interpreter to mean *false*:

```
None     0     ""     ()     []     {}
```

That is, the standard value `None`, numeric zero of all types (including float, long, and so on), all empty sequences (such as empty strings, tuples, and lists), and empty dictionaries. *Everything else* is interpreted as *true*. Laura Creighton describes this as discerning between *something* and *nothing*, rather than *true* and *false*.

Got it? This means that every value in Python can be interpreted as a truth value, which can be a bit confusing at first, but it can also be extremely useful. And even though you have all these truth values to choose from, the “standard” truth values are 0 (for *false*) and 1 (for *true*).

NOTE *In Python 2.3 a separate Boolean type is introduced, with the values* `True` *and* `False`*, which are basically equivalent to* `1` *and* `0`*, but will eventually replace them as “standard” truth values.*

Conditional Execution and the `if` *Statement*

Truth values can be combined (which you’ll see in a while), but let’s first see what you can use them for. Try running the following script:

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
    print 'Hello, Mr. Gumby'
```

This is the `if` statement, which lets you do *conditional execution*. That means that if the *condition* (the expression after `if` but before the colon) evaluates to *true* (as defined previously), the following block (in this case, a single `print` statement) is executed. If the condition is *false*, then the block is *not* executed (but you guessed that, didn’t you?).

NOTE *In the sidebar “Sneak Peek: The* `if` *Statement” in Chapter 1, the statement was written on a single line. That is equivalent to using a single-line block, as in the preceding example.*

`else` *Clauses*

In the example from the previous section, if you enter a name that ends with “Gumby,” the method `name.endswith` returns 1, making the `if` statement enter the block, and the greeting is printed. If you want, you can add an alternative, with the `else` clause (called a “clause” because it isn’t really a separate statement, just a part of the `if` statement):

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
    print 'Hello, Mr. Gumby'
else:
    print 'Hello, stranger'
```

Here, if the first block isn't executed (because the condition evaluated to false), you enter the second block instead. This really makes you see how easy it is to read Python code, doesn't it? Just read the code aloud (from `if`) and it sound just like a normal (or perhaps not *quite* normal) sentence.

`elif` *Clauses*

If you want to check for several conditions, you can use `elif`, which is short for "else if." It is a combination of an `if` clause and an `else` clause—an `else` clause with a condition:

```
num = input('Enter a number: ')
if num > 0:
    print 'The number is positive'
elif num < 0:
    print 'The number is negative'
else:
    print 'The number is zero'
```

Nesting Blocks

Let's throw in a few bells and whistles. You can have `if` statements inside other `if` statement blocks, as follows:

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
    if name.startswith('Mr.'):
        print 'Hello, Mr. Gumby'
    elif name.startswith('Mrs.'):
        print 'Hello, Mrs. Gumby'
    else:
        print 'Hello, Gumby'
else:
    print 'Hello, stranger'
```

Here, if the name ends with "Gumby," you check the start of the name as well—in a separate `if` statement inside the first block. Note the use of `elif` here. The last alternative (the `else` clause) has no condition—if no other alternative is chosen, you use the last one. If you want to, you can leave out either of the `else` clauses. If you leave out the inner `else` clause, names that don't start with either "Mr." or "Mrs." are ignored (assuming the name was "Gumby"). If you drop the outer `else` clause, strangers are ignored.

More Complex Conditions

That's really all there is to know about `if` statements. Now let's return to the conditions themselves because they are the really interesting part of conditional execution.

Comparison Operators

Perhaps the most basic operators used in conditions are the *comparison operators*. They are used (surprise, surprise) to compare things. The comparison operators are summed up in Table 5-1.

Table 5-1. The Python Comparison Operators

EXPRESSION	DESCRIPTION
x `==` *y*	*x* equals *y*
x `<` *y*	*x* is less than *y*
x `>` *y*	*x* is greater than *y*
x `>=` *y*	*x* is greater than or equal to *y*
x `<=` *y*	*x* is less than or equal to *y*
x `!=` *y*	*x* is not equal to *y*
x `is` *y*	*x* and *y* are the same object
x `is not` *y*	*x* and *y* are different objects
x `in` *y*	*x* is a member of the container (e.g., sequence) *y*
x `not in` *y*	*x* is not a member of the container (e.g., sequence) *y*

Comparisons can be *chained* in Python, just like assignments—you can put several comparison operators in a chain, like this: `0 < age < 100`.

TIP *When comparing things, you can also use the built-in function* `cmp` *as described in Chapter 2.*

Some of these operators deserve some special attention and will be described in the following sections.

The Equality Operator

If you want to know if two things are equal, you use the equality operator, written as a double equality sign, `==`:

```
>>> "foo" == "foo"
1
>>> "foo" == "bar"
0
```

Double? Why can't you just use a *single* equality sign, like they do in mathematics? I'm sure you're clever enough to figure this out for yourself, but let's try it:

```
>>> "foo" = "foo"
SyntaxError: can't assign to literal
```

The single equality sign is the assignment operator, which is used to *change* things, which is *not* what you want to do when you compare things.

`is`*: The Identity Operator*

The `is` operator is interesting. It seems to work just like `==`, but it doesn't:

```
>>> x = y = [1, 2, 3]
>>> z = [1, 2, 3]
>>> x == y
1
>>> x == z
1
>>> x is y
1
>>> x is z
0
```

Until the last example, this looks fine, but then you get that strange result, that `x` is not `z` even though they are equal. Why? Because `is` tests for *identity*, rather than *equality*. The variables `x` and `y` have been bound to the *same list*, while `z` is simply bound to another list that happens to contain the same values in the same order. They may be equal, but they aren't the *same object*.

Does that seem unreasonable? Consider this example:

```
>>> x = [1, 2, 3]
>>> y = [2, 4]
>>> x is not y
1
>>> del x[2]
>>> y[1] = 1
>>> y.reverse()
```

In this example, I start with two different lists, `x` and `y`. As you can see, `x is not y` (just the inverse of `x is y`), which you already know. I change the lists around a bit, and though they are now equal, they are still two separate lists:

```
>>> x == y
1
>>> x is y
0
```

Here it is obvious that the two lists are equal but not identical.

To summarize: Use `==` to see if two objects are *equal*, and use `is` to see if they are *identical* (the same object).

CAUTION *Avoid the use of* `is` *with basic, immutable values such as numbers and strings. The result is unpredictable because of the way Python handles these internally.*

`in`*: The Membership Operator*

I have already introduced the `in` operator (in Chapter 2, in the section "Membership"). It can be used in conditions, just like all the other comparison operators:

```
name = raw_input('What is your name? ')
if 's' in name:
    print 'Your name contains the letter "s".'
else:
    print 'Your name does not contain the letter "s".'
```

Comparing Strings and Sequences

Strings are compared according to their order when sorted alphabetically:

```
>>> "alpha" < "beta"
1
```

If you throw in capital letters, things get a bit messy. (Actually, characters are sorted by their ordinal values. The ordinal value of a letter can be found with the `ord` function, whose inverse is `chr`.) To avoid this, use the string methods `upper` or `lower`:

```
>>> 'FnOrD'.lower() == 'Fnord'.lower()
1
```

Other sequences are compared in the same manner, except that instead of letters you have other types of elements:

```
>>> [1, 2] < [2, 1]
1
```

If the sequences contain lists as elements, the same rule applies to these sublists:

```
>>> [2, [1, 4]] < [2, [1, 5]]
1
```

Boolean Operators

Now, you've got plenty of things that return truth values. (Given the fact that all values can be interpreted as truth values, *all* expressions return them.) But you may want to check for more than one condition. For instance, let's say you want to write a program that reads a number and checks whether it's between 1 and 10 (inclusive). You *can* do it like this:

```
number = input('Enter a number between 1 and 10: ')
if number <= 10:
    if number >= 1:
        print 'Great!'
    else:
        print 'Wrong!'
else:
    print 'Wrong!'
```

This will work, but it's clumsy. The fact that you have to write `print 'Wrong!'` in two places should alert you to this clumsiness. Duplication of effort is not a good thing. So what do you do? It's so simple:

```
if number <= 10 and number >= 1:
    print 'Great!'
else:
    print 'Wrong!'
```

NOTE *In this example, you could (and quite probably should) have made this even simpler by using the following chained comparison:*

```
1 <= number <= 10
```

The `and` operator is a so-called Boolean operator (named after George Boole, who did a lot of smart stuff on truth values, also called *logical* or *Boolean* values). It takes two truth values, and returns true if both are true, and false otherwise. You have two more of these operators, `or` and `not`. With just these three, you can combine truth values in any way you like:

```
if ((cash > price) or customer_has_good_credit) and not out_of_stock:
    give_goods()
```

Short-Circuit Logic

The Boolean operators have one interesting property: They only evaluate what they need to. For instance, the expression `x and y` requires both `x` and `y` to be true; so if `x` is false, the expression returns false immediately, without worrying about `y`. Actually, if `x` is false, it returns `x`—otherwise it returns `y`. (Can you see how this gives the expected meaning?) This behavior is called *short-circuit logic:* the Boolean operators are often called logical operators, and as you can see, the second value is sometimes "short-circuited." This works with `or`, too. In the expression `x or y`, if `x` is true, it is returned, otherwise `y` is returned. (Can you see how this makes sense?)

So, how is this useful? Let's say a user is supposed to enter his or her name, but may opt to enter nothing, in which case you want to use the default value `'<unknown>'`. You could use an `if` statement, but you could also state things very succinctly:

```
name = raw_input('Please enter your name: ') or '<unknown>'
```

In other words, if the return value from `raw_input` is true (not an empty string) it is assigned to `name` (nothing changes); otherwise, the default `'<unknown>'` is assigned to `name`.

This sort of short-circuit logic can be used to implement the so-called "ternary operator" (or conditional operator), found in languages such as C and Java. For a thorough explanation, see Alex Martelli's recipe on the subject in the Python Cookbook (`http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52310`).

Assertions

There is a useful relative of the `if` statement, which works more or less like this (pseudocode):

```
if not condition:
    crash program
```

Now, why on earth would you want something like that? Simply because it's better that your program crashes when an error condition emerges than at a much later time. Basically, you can require that certain things be true. The keyword used in the statement is `assert`:

```
>>> age = 10
>>> assert 0 < age < 100
>>> age = -1
>>> assert 0 < age < 100
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

It can be useful to put the `assert` statement in your program as a checkpoint, if you know something *has* to be true for your program to work correctly.

A string may be added after the condition, to explain the assertion:

```
>>> age = -1
>>> assert 0 < age < 100, 'The age must be realistic'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError: The age must be realistic
```

Loops

Now you know how to do something if a condition is true (or false), but how do you do something several times? For instance, you might want to create a program that reminds you to pay the rent every month, but with the tools we have looked at until now, you'd have to write the program like this (pseudocode):

```
send mail
wait one month
send mail
wait one month
send mail
wait one month
(...and so on)
```

But what if you wanted it to continue doing this until you stopped it? Basically, you want something like this (again, pseudocode):

```
while we aren't stopped:
    send mail
    wait one month
```

Or, let's take a simpler example. Let's say that you want to print out all the numbers from 1 to 100. Again, you could do it the stupid way:

```
print 1
print 2
print 3
```

…and so on. But you didn't start using Python because you wanted to do stupid things, right?

`while` *Loops*

In order to avoid the cumbersome code of the preceding example, it would be useful to be able to do something like this:

```
x = 1
while x <= 100:
    print x
    x += 1
```

Now, how do you do that in Python? You guessed it—you do it just like that. Not that complicated is it? You could also use a loop to ensure that the user enters a name, as follows:

```
name = ''
while not name:
    name = raw_input('Please enter your name: ')
print 'Hello, %s!' % name
```

Try running this, and then just pressing the Enter key when asked to enter your name: the question appears again because `name` is still an empty string, which evaluates to *false*.

TIP *What would happen if you entered just a space character as your name? Try it. It is accepted because a string with one space character is not empty, and therefore not* false. *This is definitely a flaw in our little program, but easily corrected: just change* `while not name` *to* `while name.isspace()`.

`for` *Loops*

The `while` statement is very flexible. It can be used to repeat a block of code while *any condition* is true. While this may be very nice in general, sometimes you may want something tailored to your specific needs. One such need is to perform a block of code *for each* element of a set (or, actually, sequence) of values. You can do this with the `for` statement:

```
words = ['this', 'is', 'an', 'ex', 'parrot']
for word in words:
    print word
```

Or...

```
range = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in range:
    print number
```

Because iterating (another word for "looping") over a range of numbers is a common thing to do, there is a built-in function to make ranges for you:

```
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ranges work like slices. They include the first limit (in this case 0), but not the last (in this case 10). Quite often, you want the ranges to start at 0, and this is actually assumed if you only supply one limit (which will then be the last):

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

TIP *There is also another function called* `xrange` *that works just like* `range` *in loops, but where* `range` *creates the whole sequence at once,* `xrange` *creates only one number at a time. This can be useful when iterating over* huge *sequences more efficiently, but in general you needn't worry about it.*

The following program writes out the numbers from 1 to 100:

```
for number in range(1,101):
    print number
```

Notice that this is much more compact than the `while` loop I used earlier.

TIP *If you* can *use a* `for` *loop rather than a* `while` *loop, you should probably do so.*

Iterating over Dictionaries

To loop over the keys of a dictionary, you can use a plain `for` statement, just as you can with sequences:

```
d = {'x': 1, 'y': 2, 'z': 3}
for key in d:
    print key, 'corresponds to', d[key]
```

In Python versions before 2.2, you would have used a dictionary method such as `keys` to retrieve the keys (since direct iteration over dictionaries wasn't allowed). If only the values were of interest, you could have used `d.values` instead of `d.keys`. You may remember that `d.items` returns key-value pairs as tuples. One great thing about `for` loops is that you can use sequence unpacking in them:

```
for key, value in d.items():
    print key, 'corresponds to', value
```

To make your iteration more efficient, you can use the methods `iterkeys` (equivalent to the plain `for` loop), `itervalues`, or `iteritems`. (These don't return lists, but iterators. Iterators are explained in Chapter 9, "Magic Methods and Iterators.")

NOTE *As always, the order of dictionary elements is undefined. In other words, when iterating over either the keys or the values of a dictionary, you can be sure that you'll process all of them, but you can't know in which order. If the order is important, you can store the keys or values in a separate list and sort it before iterating over it.*

Parallel Iteration

Sometimes you want to iterate over two sequences at the same time. Let's say that you have the following two lists:

```
names = ['anne', 'beth', 'george', 'damon']
ages = [12, 45, 32, 102]
```

If you want to print out names with corresponding ages, you *could* do the following:

```
for i in range(len(names)):
    print names[i], 'is', ages[i], 'years old'
```

Here I use `i` as a standard variable name for loop indices (as these things are called).

A useful tool for parallel iteration is the built-in function `zip`, which "zips" together the sequences, returning a list of tuples:

```
>>> zip(names, ages)
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
```

Now I can unpack the tuples in my loop:

```
for name, age in zip(names, ages):
    print name, 'is', age, 'years old'
```

The `zip` function works with as many sequences as you want. It's important to note what `zip` does when the sequences are of different lengths: it stops when the shortest sequence is "used up":

```
>>> zip(range(5), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

I wouldn't recommend using `range` instead of `xrange` in the preceding example—although only the first five numbers are needed, `range` calculates all the numbers, and that may take a lot of time. With `xrange`, this isn't a problem because it calculates only those numbers needed.

Breaking Out of Loops

Usually, a loop simply executes a block until its condition becomes false, or until it has used up all sequence elements—but sometimes you may want to interrupt the loop, to start a new iteration (one "round" of executing the block), or to simply end the loop.

break

To end (break out of) a loop, you use break. Let's say you wanted to find the largest square (an integer that is the square of another integer) below 100. Then you start at 100 and iterate downwards to 0. When you've found a square, there's no need to continue, so you simply break out of the loop:

```
from math import sqrt
for n in range(99, 0, -1):
    root = sqrt(n)
    if root == int(root):
        print n
        break
```

If you run this program, it will print out 81, and stop. Notice that I've added a third argument to range—that's the *step*, the difference between every pair of adjacent numbers in the sequence. It can be used to iterate downwards as I did here, with a negative step value, and it can be used to skip numbers:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

continue

The continue statement is used less often than break. It causes the current iteration to end, and to "jump" to the beginning of the next. It basically means "skip the rest of the loop body, but don't end the loop." This can be useful if you have a large and complicated loop body and several possible reasons for skipping it—in that case you can use continue as follows:

```
for x in seq:
    if condition1: continue
    if condition2: continue
    if condition3: continue
    do_something()
    do_something_else()
    do_another_thing()
    etc()
```

In many cases, however, simply using an `if` statement is just as good:

```
for x in seq:
    if not (condition1 or condition2 or condition3):
        do_something()
        do_something_else()
        do_another_thing()
        etc()
```

Even though `continue` can be a useful tool, it is not essential. The `break` statement, however, is something you should get used to because it is used quite often in concert with `while 1`, as explained in the next section.

The `while 1/break` *Idiom*

The `while` and `for` loops in Python are quite flexible, but every once in a while you may encounter a problem that makes you wish you had more functionality. For instance, let's say you want to do something while a user enters words at a prompt, and you want to end the loop when no word is provided. One way of doing that would be

```
word = 'dummy'
while word:
    word = raw_input('Please enter a word: ')
    # do something with the word:
    print 'The word was ' + word
```

Here is an example session:

```
Please enter a word: first
The word was first
Please enter a word: second
The word was second
Please enter a word:
```

This works just like you want it to. (Presumably you'd do something more useful with the word than print it out, though.) However, as you can see, this code is a bit ugly. To enter the loop in the first place, you have to assign a dummy (unused) value to `word`. Dummy values like this are usually a sign that you aren't doing things quite right. Let's try to get rid of it:

```
word = raw_input('Please enter a word: ')
while word:
    # do something with the word:
    print 'The word was ' + word
    word = raw_input('Please enter a word: ')
```

Here the dummy is gone, but I have repeated code (which is also a bad thing): I have to use the same assignment and call to `raw_input` in two places. How can I avoid that? I can use the `while 1`/`break` idiom:

```
while 1:
    word = raw_input('Please enter a word: ')
    if not word: break
    # do something with the word:
    print 'The word was ' + word
```

NOTE *An idiom is a common way of doing things that people who know the language are assumed to know.*

The `while 1` part gives you a loop that will never terminate by itself. Instead you put the condition in an `if` statement inside the loop, which calls `break` when the condition is fulfilled. Thus you can terminate the loop anywhere inside the loop instead of only at the beginning (as with a normal `while` loop). The `if`/`break` line splits the loop naturally in two parts: The first takes care of setting things up (the part that would be duplicated with a normal `while` loop), and the other part makes use of the initialization from the first part, provided that the loop condition is true.

Although you should be wary of using `break` too often (because it can make your loops harder to read), this specific technique is so common that most Python programmers (including yourself) will probably be able to follow your intentions.

else *Clauses in Loops*

When you use break statements in loops, it is often because you have "found" something, or because something has "happened." It's easy to do something when you break out (like print n), but sometimes you may want to do something if you *didn't* break out. But how do you find out? You could use a Boolean variable, set it to 0 before the loop, and set it to 1 when you break out. Then you can use an if statement afterwards to check whether you did break out or not:

```
broke_out = 0
for x in seq:
    do_something(x)
    if condition(x):
        broke_out = 1
        break
    do_something_else(x)
if not broke_out:
    print "I didn't break out!"
```

A simpler way is to add an else clause to your loop—it is only executed if you didn't call break. Let's reuse the example from the preceding section on break:

```
from math import sqrt
for n in range(99, 81, -1):
    root = sqrt(n)
    if root == int(root):
        print n
        break
else:
    print "Didn't find it!"
```

Notice that I changed the lower (exclusive) limit to 81 to test the else clause. If you run the program, it prints out "Didn't find it!" because (as you saw in the section on break) the largest square below 100 is 81. You can use continue, break, and else clauses both with for loops and while loops.

List Comprehension—Slightly Loopy

List comprehension is a way of making lists from other lists (similar to *set comprehension*, if you know that term from mathematics). It works in a way similar to for loops, and is actually quite simple:

```
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The list is composed of `x*x` for each `x` in `range(10)`. Pretty straightforward? What if you only want to print out those squares that are divisible by 3? Then you can use the modulo operator—`y % 3` returns zero when `y` is divisible by 3. (Note that `x*x` is divisible by 3 only if `x` is divisible by 3.) You put this into your list comprehension by adding an `if` part to it:

```
>>> [x*x for x in range(10) if x % 3 == 0]
[0, 9, 36, 81]
```

You can also add more `for` parts:

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

This can be combined with an `if` clause, just like before:

```
>>> girls = ['alice', 'bernice', 'clarice']
>>> boys = ['chris', 'arnold', 'bob']
>>> [b+'+'+g for b in boys for g in girls if b[0] == g[0]]
['chris+clarice', 'arnold+alice', 'bob+bernice']
```

This gives the pairs of boys and girls who have the same initial letter in their first name.

A Better Solution

The boy/girl pairing example isn't particularly efficient because it checks every possible pairing. There are many ways of solving this problem in Python. The following was suggested by Alex Martelli:

```
girls = ['alice', 'bernice', 'clarice']
boys = ['chris', 'arnold', 'bob']
letterGirls = {}
for girl in girls:
    letterGirls.setdefault(girl[0], []).append(girl)
print [b+'+'+g for b in boys for g in letterGirls[b[0]]]
```

This program constructs a dictionary called `letterGirl` where each entry has a single letter as its key and a list of girls' names as its value. (The `setdefault` dictionary method is described in the previous chapter.) After this dictionary has been constructed, the list comprehension loops over all the boys and looks up all the girls whose name begins with the same letter as the current boy. This way the list comprehension doesn't have to try out every possible combination of boy and girl and check whether the first letters match.

And Three for the Road

To end the chapter, let's take a quick look at three more statements: `pass`, `del`, and `exec`.

Nothing Happened!

Sometimes you need to do nothing. This may not be very often, but when it happens, it's good to know that you have the `pass` statement:

```
>>> pass
>>>
```

Not much going on here.

Now, why on earth would you want a statement that does nothing? It can be useful as a placeholder while you are writing code. For instance, you may have written an `if` statement and you want to try it, but you lack the code for one of your blocks. Consider the following:

```
if name == 'Ralph Auldus Melish':
    print 'Welcome!'
elif name == 'Enid':
    # Not finished yet...
elif name == 'Bill Gates':
    print 'Access Denied'
```

This code won't run because an empty block is illegal in Python. To fix this, simply add a `pass` statement to the middle block:

```
if name == 'Ralph Auldus Melish':
    print 'Welcome!'
elif name == 'Enid':
    # Not finished yet...
    pass
```

```
elif name == 'Bill Gates':
    print 'Access Denied'
```

NOTE *An alternative to the combination of a comment and a* `pass` *statement is to simply insert a string. This is especially useful for unfinished functions (see Chapter 6) and classes (see Chapter 7) because they will then act as "docstrings" (explained in Chapter 6).*

Deleting with `del`

In general, Python deletes objects that you don't use anymore:

```
>>> scoundrel = {'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = scoundrel
>>> scoundrel
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> scoundrel = None
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = None
```

At first, `robin` and `scoundrel` both contain (or "point to") the same dictionary. So when I assign `None` to `scoundrel`, the dictionary is still available through `robin`. But when I assign `None` to `robin` as well, the dictionary suddenly floats around in the memory of the computer with no name attached to it. There is no way I can retrieve it or use it, so the Python interpreter (in its infinite wisdom) simply deletes it. (This is called "garbage collection.") Note that I could have used any value other than `None` as well. The dictionary would be just as gone.

Another way of doing this is to use the `del` statement (which we used to delete sequence and dictionary elements in Chapters 2 and 4, remember?). This not only removes a reference to an object, it also removes the name itself:

```
>>> x = 1
>>> del x
>>> x
Traceback (most recent call last):
  File "<pyshell#255>", line 1, in ?
    x
NameError: name 'x' is not defined
```

This may seem easy, but it can actually be a bit tricky to understand at times. For instance, in the following example, x and y refer to the same list:

```
>>> x = ["Hello", "world"]
>>> y = x
>>> y[1] = "Python"
>>> x
['Hello', 'Python']
```

You might assume that by deleting x, you would also delete y, but that is *not* the case:

```
>>> del x
>>> y
['Hello', 'Python']
```

Why is this? x and y referred to the *same* list, but deleting x didn't affect y at all. The reason for this is that you only delete the *name*, not the list itself (the value). In fact, there is no way to delete values in Python (and you don't really need to because the Python interpreter does it by itself whenever you don't use the value anymore).

Executing and Evaluating Strings with exec *and* eval

Sometimes you may want to create Python code "on the fly" and execute it as a statement or evaluate it as an expression. This may border on dark magic at times—consider yourself warned.

CAUTION *In this section, you learn to execute Python code stored in a string. This is a potential security hole of great dimensions. If you execute a string where parts of the contents have been supplied by a user, you have little or no control over what code you are executing. This is especially dangerous in network applications, such as CGI scripts, which you will learn about in Chapter 19.*

exec

The statement for executing a string is exec:

```
>>> exec "print 'Hello, world!'"
Hello, world!
```

However, using this simple form of the exec statement is rarely a good thing; in most cases you want to supply it with a *namespace*, a place where it can put its variables. You want to do this so that the code doesn't corrupt *your* namespace (that is, change your variables). For instance, let's say that the code uses the name sqrt:

```
>>> from math import sqrt
>>> exec "sqrt = 1"
>>> sqrt(4)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in ?
    sqrt(4)
TypeError: object is not callable: 1
```

Well, why would you do something like that in the first place, you ask? The exec statement is mainly useful when you build the code string on the fly. And if the string is built from parts that you get from other places, and possibly from the user, you can rarely be certain of exactly what it will contain. So to be safe, you give it a dictionary, which will work as a namespace for it.

NOTE *The concept of namespaces, or* scopes, *is a very important one. You will look at it in depth in the next chapter, but for now you can think of a namespace as a place where you keep your variables, much like an invisible dictionary. So when you execute an assignment like* `x = 1`, *you store the key* `x` *with the value* `1` *in the* current namespace, *which will often be the global namespace (which we have been using, for the most part, up until now), but doesn't have to be.*

You do this by adding `in scope`, where `scope` is some dictionary that will function as the namespace for your code string:

```
>>> from math import sqrt
>>> scope = {}
>>> exec 'sqrt = 1' in scope
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

Now you have full control over the variables that are changed by the code because they are all kept inside `scope`. If you try to print out `scope`, you see that it contains a *lot* of stuff because the dictionary called `__builtins__` is automatically added and contains all built-in functions and values:

```
>>> len(scope)
2
>>> scope.keys()
['sqrt', '__builtins__']
```

eval

A built-in function that is similar to `exec` is `eval` (for "evaluate"). Just as `exec` executes a series of Python *statements*, `eval` evaluates a Python *expression* (written in a string) and returns the value. (`exec` doesn't return anything because it is a statement itself.) For instance, you can use the following to make a Python calculator:

```
>>> eval(raw_input("Enter an arithmetic expression: "))
Enter an arithmetic expression: 6 + 18 * 2
42
```

NOTE *The expression* `eval(raw_input(...))` *is, in fact, equivalent to* `input(...)`.

You can supply a namespace with `eval`, just as with `exec`, although expressions rarely rebind variables in the way statements usually do.

CAUTION *Even though expressions don't rebind variables* as a rule, *they certainly can (for instance by calling functions that rebind global variables). Therefore, using* `eval` *with an untrusted piece of code is no safer than using* `exec`*. For a more secure alternative, see the standard library modules* `rexec` *and* `Bastion`*, mentioned in Chapter 10.*

Priming the Scope

When supplying a namespace for `exec` or `eval`, you can also put some values in before actually using the namespace:

```
>>> scope = {}
>>> scope['x'] = 2
>>> scope['y'] = 3
>>> eval('x * y', scope)
6
```

In the same way, a scope from one `exec` or `eval` call can be used again in another one:

```
>>> scope = {}
>>> exec 'x = 2' in scope
>>> eval('x*x', scope)
4
```

Actually, `exec` and `eval` are not used all that often, but they can be nice tools to keep in your back pocket (figuratively, of course).

A Quick Summary

In this chapter you've seen several kinds of statements:

Printing. You can use the `print` statement to print several values by separating them with commas. If you end the statement with a comma, later `print` statements will continue printing on the same line.

Importing. Sometimes you don't like the name of a function you want to import—perhaps you've already used the name for something else. You can use the `import...as...` statement, to locally rename a function.

Assignments. You've seen that through the wonder of sequence unpacking and chained assignments, you can assign values to several variables at once, and that with augmented assignments you can change a variable in place.

Blocks. Blocks are used as a means of grouping statements through indentation. They are used in conditionals and loops, and as you see later in the book, in function and class definitions, among other things.

Conditionals. A conditional statement either executes a block or not, depending on a condition (Boolean expression). Several conditionals can be strung together with `if/elif/else`.

Assertions. An assertion simply asserts that something (a Boolean expression) is true, optionally with a string explaining why it has to be so. If the expression happens to be false, the assertion brings your program to a halt (or actually raises an exception—more on that in Chapter 8). It's better to find an error early than to let it sneak around your program until you don't know where it originated.

Loops. You either can execute a block for each element in a sequence (such as a range of numbers) or continue executing it while a condition is true. To skip the rest of the block and continue with the next iteration, use the `continue` statement; to break out of the loop, use the `break` statement. Optionally, you may add an `else` clause at the end of the loop, which will be executed if you didn't execute any `break` statements inside the loop.

List comprehension. These aren't really statements—they are expressions that look a lot like loops, which is why I grouped them with the looping statements. Through list comprehension you can build new lists from old ones, applying functions to the elements, filtering out those you don't want, and so on. The technique is quite powerful, but in many cases using plain loops and conditionals (which will always get the job done) may be more readable.

`pass`, `del`, `exec`, and `eval`. The `pass` statement does nothing, which can be useful as a placeholder, for instance. The `del` statement is used to delete variables or parts of a datastructure, but cannot be used to delete values. The `exec` statement is used to execute a string as if it were a Python program. The built-in function `eval` evaluates an expression written in a string and returns the result.

New Functions in This Chapter

FUNCTION	**DESCRIPTION**
`chr(`*n*`)`	Returns a one-character string with ordinal *n* (0 ≤ *n* ≤ 256)
`eval(`*source*`[, `*globals*`[, `*locals*`]])`	Evaluates a string as an expression and returns the value
`ord(`*c*`)`	Returns the integer ordinal value of a one-character string
`range([`*start*`,] `*stop*`[, `*step*`])`	Creates a list of integers
`xrange([`*start*`,] `*stop*`[, `*step*`])`	Creates an xrange object, used for iteration
`zip(`*seq1*`, `*seq2*`,...)`	Creates a new sequence suitable for parallel iteration

What Now?

Now you've cleared the basics. You can implement any algorithm you can dream up; you can read in parameters and print out the results. In the next couple of chapters, you learn about something that will help you write larger programs without losing the big picture. That something is called *abstraction*.

Index

Numbers and Symbols

= (assignment operator, using in variables, 20
* (asterisk)
  using as the width or precision (or both), 67
  using when collecting parameters, 144–147
+= (augmented assignment), making code more compact and concise with, 99–100
\ (backslash) character
  using in Python code, 246
  using to escape special characters, 280
  using to escape the quotes in a string, 27
\ \; (backticks), converting a Python value to a string with, 29
" " (blank space), for putting a blank in front of positive numbers, 68
^ (caret), using for pattern matching, 282
: (colon)
  separating the start and stop index of results with, 561
  use of in Python statements, 17
  using to indicate a block is to begin, 101
, (commas), printing with, 95–96
%s (conversion specifiers)
  as modulus operator, 11
  in strings, 64
$ (dollar sign), using to mark the end of a string for pattern matching, 282
"." (dot) conversion specifier
  for specifying number of decimal places in strings, 65
  as wildcard in regular expressions, 279, 280
** (double asterisk) operator
  gathering keyword arguments with, 146–147
  using as exponentiation operator to calculate powers, 12
__ (double underscores), making methods and attributes private with, 175–176
== (equality operator), using in Python, 17
/ (forward slash) character,
  using as division operator, 10–11, 18
  using in file paths in Python code, 246
# (hash sign), beginning comments with, 25, 132, 560
- (minus sign),
  conversion specifier, 65
  as subtraction operator, 11
% (percent), using as string formatting operator, 63–64
%%, using to specify a percent sign, 64
| (pipe) character
  linking several commands together with, 307
  using for character set alternatives, 281
+ (plus sign)
  conversion specifier, 65
  using for concatenating lists, 565
'+' value, for mode argument of the open function, 304
#! (pound bang or shebang) character sequence
  adding to your CGI script, 463
  using, 23
>>> (prompt). *See* Python prompt (>>>)
? (question mark) character
  using after a subpattern in character sets, 281–282
  using to make repetition operator non-greedy, 290
0 (zero), conversion specifier, 65

A

'a' value, for mode argument of the open function, 304
abs function, using, 18

abspath function, getting the absolute path of the data directory with, 475
abstraction, 129–166
  defined, 165
  and structure, 130–131
accessor methods, Rectangle class code example, 223
Acrobat Reader (Adobe), Web site address for downloading, 387
ActivePython, Web site address, 7, 589
adding, as a sequence operation, 38
addition (plus sign) operator, concatenating sequences with, 43
address book, using Python dictionaries for, 79–80
algorithm, 9, 34
alignment, conversion specifier, 68
__all__ variable, public interface for a module defined by, 256–257
allow_reuse_address attribute, setting for SimpleXMLRPCServer, 515
Anygui GUI toolkit
  downloading and installing, 329
  getting started with, 329–330
  Web site address for documentation, 336
  what it is, 328
anygui.link function, linking components with event handlers with, 347
append method, appending an object to end of list with, 51
Application class, 347
Application.add method, adding a window to an application with, 347
Application.run method, starting the main event loop with, 347
apply function, function of, 164–165
arcade game project
  basic design process, 535
  creating, 535–556
  first implementation, 540–545
  preparations for creating, 540
  second implementation, 545–556
  specific goals for creating, 536
  useful tools for creating, 536–539
arithmetic sequence, code example of implementation of, 218
ArithmeticSequence class, using, 219–220
as clause, using to import something from a module, 97
assert keyword, used in Python programs as a checkpoint, 109–110
assertions, using, 109–110, 126
assignment, 14
assignment statements
  function of in Python, 581
  tricks, 98–100
assignments, assigning values to several variables at once with, 126
asterisk (*), using as the width or precision (or both), 67
async_chat class, overriding methods in, 445–446
asynchat module, using to create a chat server, 439
asyncore module
  subclassing the dispatcher class from, 443
  using to create a chat server, 439
AttributeError class, built-in exception, 196
attributes
  making private, 175–176
  setting to customize component in Anygui, 332–333
attributes, functions, and methods, 178–179
augmented assignments, using, 99–100
augmented assignment statements, function of in Python, 582
autoexec.bat file, editing PYTHONPATH in, 252–253
automated tests, using to catch bugs in programs, 356

B
'b' value, for mode argument of the open function, 304
backslash (\) character
  using in Python code, 246
  using to escape the quotes in a string, 27
backticks (\\), converting objects to string representation with, 29, 565
backtracking, using to solve problems, 235–236
base case
  of recursive function, 155
  using to solve the Eight Queens problem, 237–239
  using with recursion, 231
__bases__ attribute
  accessing for classes, 185
  how it works, 186–187
basic (literal) values, 571
Bastion module, function of, 299

Beck, Ken, extreme programming (XP) created by, 349
binary files, adding arguments to code for processing, 304
binary mode, reasons for using, 305
binary search
    definition, 158
    implementing, 159–160
bind method, calling to bind a socket to a specific port, 443–444
bisect module, that implements binary search, 160
blank space (" "), for putting a blank in front of positive numbers, 68
blitting, defined, 537
blocks
    creating and using, 100–101
    grouping statements through indentation with, 126
Boolean operators
    checking for more than one condition with, 107–108
Boolean values
    as truth values, 45
    using in conditional statements, 101–102
bottom keyword argument, in Placer layout manager, 336
bound methods
    example of, 214–215
    using like functions, 178–179
brackets ([ ]), using to create a character set, 280
break statement
    using to break out of a loop, 115
buffering, open function optional parameter for, 305
built-in functions, 18
    list of some important in Python, 573–576
bulletin board project
    building your own, 481–499
    edit.cgi script for, 493–495
    first implementation, 484–489
    main.cgi script for, 489–491
    save.cgi script for, 495–496
    second implementation, 489–496
    specific goals for creating, 482
    structuring the system, 489–490
    trying it out after creating, 497–498
    useful tools for creating, 482–483
    view.cgi script for, 492–493
Button class, 347
buttons
    adding to a window, 331
    setting the text of, 332–333

C
callable function, defined, 131
Canvas class, methods for drawing, 390
caret (^), using to mark the beginning of a string for pattern matching, 282
catching an exception object, 200–201
ceil function, 19
CGI (Common Gateway Interface), remote editing of Web pages with, 461–480
CGI editor
    editing page of, 479
    opening page of, 478
    running, 478–479
cgi module
    using, 467–468
    Web site address for description of, 462
CGI programs, putting in a directory for accessibility via the Web, 463
CGI script
    adding a pound bang (#!) line to, 463
    for creating a simple HTML form, 469–470
    first implementation, 465–473
    invoking without forms, 469
    preparations for making accessible and runnable, 462–465
    process for getting the threaded look for replies, 487
    second implementation, 473–479
    setting the file permissions in, 463–464
    specific goals for creating for remote editing, 462
    splitting the functionality into several scripts, 473–478
CGI security risks, 465
cgitb module
    example of a CGI traceback from, 467
    using for development of CGI scripts, 462
chained assignments, using, 99
character sets
    alternatives and subpatterns, 281
    creating and inverting, 280
    optional and repeated subpatterns, 281–282
    using special characters in, 280–281
chat server
    adding support for identity and command interpretation, 449–450
    code for one that accepts connections, 443–444

chat server (*continued*)
  commands available in, 458
  final version of, 454–458
  first implementation, 443–449
  main additions to new, 453
  pointers for a multithreaded, 441–442
  preparations for creating, 442–443
  second implementation, 449–458
  specific goals for creating, 440
  useful tools for creating, 440–442
chmod command, for changing UNIX file permissions, 464
chr function, 127
clamp rectangle method, in arcade game project, 545
class, defined, 177
__class__ attribute, finding which class an object belongs to with, 186
class definitions, function of in Python, 587
class keyword, defining classes with, 564
class namespace, using, 182–184
class scope, defining a variable in, 183
class statement
  indicating a superclass in, 184
  method definitions written inside of, 178–179
classes
  accessing __bases__ attribute for, 185
  code example for calling, 564–565
  defined, 190
  making your own, 178–179
classes and types, 177–189
clear method, removing all items from the dictionary with, 85
client, needed to test your chat server, 442
client interface, for the XML-RPC file sharing program, 515–516
client.py, code example, 520–522
close database connection method, 485
closure, defined, 153
cmath (complex math) and complex numbers, 20–21
cmath.sqrt function, 35
cmd module, function of, 299
cmp function, 61
code example
  adding a method to the TestHandler class, 406
  adding a test to a recursive generator, 232
  adding buttons to your text editor, 331
  adding emphasis handler to markup program, 372
  adding line numbers to a Python script, 267
  adding PolyLines to your drawing, 394
  adding some default arguments to recursive case, 240
  adding two buttons with customized text, 333
  adding your module directory to PYTHONPATH, 252
  addSource and addDestination methods, 427
  of a basic chat server with some cleanup, 444–445
  calling a class, 564–565
  of a CGI script that invokes a traceback (faulty.cgi), 466
  of a CGI script that retrieves a single value from a FieldStorage, 468
  of a chat server program with ChatSession class, 446–447
  of a chat server that accepts connections, 443–444
  for checking for usable SAX parser, 403
  of a configuration file using ConfigParser module, 354
  for connecting to a peer, 513
  of Content-type header code, 466
  for converting an iterator to a list, 229
  of a create SQL command, 483
  for creating a destination, 428
  for creating a fortune cookie program, 272
  for creating an electronic deck of cards, 272–274
  for creating an electronic die-throwing machine, 272
  of defaultStart and defaultEnd methods, 414–415
  of Dispatcher mixin class dispatch method, 413
  for distribute method, 428
  of edit.cgi script, 475
  for escaping all characters in a string, 286
  for fetching the body of an article, 425
  of a file containing some nonsensical text, 308
  of the files and directories created for the XML project, 418
  of the final editor program, 340–341
  of final version of chat server, 454–458

for finding the search path in the sys module, 251
for finding yesterday's time, 424
of the finished GUI client (guiclient.py), 531–534
of flatten generator rewritten as a plain function, 234
for generating a random date in a given range, 271–272
of generator, 230–231
of generic superclass for chat rooms, 451
for getting a list of sunspot data rows, 396
of a greeting script with an HTML form (simple3.cgi), 470
of a GUI implemented with Jython and Swing, 345
of a GUI implemented with Tkinter, 344
of a GUI implemented with wxPython, 344
of a GUI program with only a window, 330
of handlers (handlers.py), 381–383
for implementation of the NewsItem class, 427
for implementing the simple_markup.py program, 370
for importing MySQLdb, 483
for importing psycopg, 483
of index.html file, 474
of iterating over a file without assigning file object to a variable, 317
of iterating over characters with read, 315
iterating over lines with fileinput, 316
of iterating over lines with readlines, 315
iterating over lines with xreadlines, 316
iterator for a sequence of Fibonacci numbers, 227
of a line numbering program with line numbers added, 267
of linking several commands together with pipes, 307
of a list with an access counter, 221–222
of load function, 340
of loop that finds the subject within NNTP headers, 425
of looping over characters with read method, 314
of the main bulletin board (simple_main.cgi), 488–489
markup program (markup.py), 385–386
of a minimal chat server program, 443
for a minimal XML parser to parse website.xml, 406
of a modified text file, 312
of a module with conditional test code, 250
of a new Node implementation (server.py), 517–520
of newsagent2.py, 432–436
of Node class pseudocode, 507
of a Node controller interface (client.py), 520–522
for placing all components with basic layout, 334–335
for printing the subject and body of an article, 425
of a program for finding the sender of an e-mail, 292–293
of a program using the Loggerclass, 362
of property function, 224–225
of pseudocode for the headline rule, 375–376
for putting program state information into a file, 361
for re match objects, 287–288
for read file method, 311–312
for readline file method, 312
for readlines file method, 312
of Rectangle class, 223
of Rectangle example with magic methods, 226
for retrieving all messages in the bulletin board database, 486
for reversing and printing command-line arguments, 262
for the Rule class in the rules module, 376
of rules (rules.py), 383–384
for running a GUI-enabled file sharing program, 530
for running the Squish arcade game, 546
for running weights.py, 543
of a sample plain text document (test_input.txt), 367–368
of save function, 340
of the saving script (save.cgi), 477–478

code example (*continued*)
- for sequence and mapping protocol, 218–220
- of a set of fictitious e-mail headers, 291
- for setting file permissions, 464
- showing the file object being closed automatically, 312
- of a simple Anygui version of a text editor, 342–343
- of a simple application that uses the shelve module, 275–277
- of a simple chat server (simple_chat.py), 448–449
- of a simple configuration file, 353–354
- of simple default header and footer methods, 414
- for a simple falling weights animation (weights.py), 541–544
- of a simple logger class, 361
- of a simple markup program (simple_markup.py), 369–371
- of a simple module, 245
- of a simple module containing a function, 248
- of a simple module with some problematic test code, 249
- of a simple news gathering agent (newsagent1.py), 426
- for a simple Node implementation, 510–512
- of a simple page maker script, 409–410
- of a simple ReportLab program, 391
- of simple script that counts the words in sys.stdin, 308
- of a simple template, 297
- of a simple text file, 311
- of a simple Web editor (simple_edit.cgi), 472
- of a simple Web site represented as an XML file, 404–405
- of simplest possible CGI script, 465
- of the Squish configuration file, 547
- of the Squish game objects file, 547–549
- of the Squish main game module, 550–556
- of statement for logging, 360
- for telephone book, 83–84
- of a template, 298
- of template definitions, 297
- for a template system, 295
- of test program for computing area of a rectangle, 355
- of test program for my_math module, 359
- of a text block generator (util.py), 368–369
- that uses the Meerkat to find some Python articles, 504–505
- useful for testing an SQL database, 486–487
- for using a load function as an event handler, 339
- using CounterList class, 222
- of using for loops to iterate over a file (in Python 2.2), 317
- of using recursive case to solve Eight Queens problem, 239–240
- of using the default Anygui layout scheme, 337–338
- of using the getvalue method of a FieldStorage, 468
- of the Web site constructor, 416–418
- of writing a read method loop with while 1/break, 314
- for the XML project page handlers, 416

code reuse, using modules to facilitate, 248–249

colon (:)
- use of in Python statements, 17
- using to indicate a block is to begin, 101

columns, creating a polyline for each in a data set, 393–394

command-line interpreter, using cmd module to write, 299

command prompt, running Python scripts from, 23

CommandHandler class, code for simple implementation of, 450

commas (,), printing with, 95–96

comments
- defined, 25
- documenting your functions with, 132–133

commit database connection method, 485

Common Gateway Interface (CGI). *See* CGI (Common Gateway Interface)

comparison function, defining your own for advanced sorting, 57

comparison operators, 104–109

compile function, in re module, 283

complex numbers, cmath and, 20–21

components
- adding to a window, 332
- code example for placing with basic layout, 334–335

setting attributes to lay out in Anygui application, 334
compound statements, function of in Python, 586–587
computer game. *See* arcade game project
concatenating strings, 27–28
concatenation vs. extend method, 52
conditional execution, and the if statement, 102
conditional statements and conditions, 101–110
conditionals, loops, and other statements, 95–127
conditions, and conditional statements, 101–110
ConfigParser module, Web site address for information about, 353
config.py file
  in arcade game project, 545
  code example, 547
configuration
  extracting symbolic constants from your code, 351–352
  in your programs, 350
configuration files, creating, 353–354
conflict function, using to find conflicts in Eight Queens problem, 237
connect database method, connecting to your database with, 485
constants, writing and placing in global variables, 352
constructors
  calling with parameters, 211
  creating, 210–211
  overriding, 211–214
continue statement
  function of in Python, 584
  using in loops, 115–116
conversion flags, 65
conversion specifiers
  anatomy of basic, 65
  in string formatting, 64
conversion types, in string formatting, 66
copy function
  vs. the deepcopy function, 258
  using the help function on, 257–258
copy method, copying a dictionary with, 86–87
count method, counting element occurrences in a list with, 52
CounterList class, code example of using, 222
cracking vs. hacking, 1
create SQL command
  code example using MySQL, 484
  code example using PostgreSQL, 483
create_socket method, calling to create a socket, 443–444
Ctrl-D, exiting interactive interpreter with, 4
Ctrl+Break, using as a keyboard interrupt in DOS, 444
Ctrl+C, using as a keyboard interrupt in UNIX, 444
Ctrl+F5, using to run programs, 22
cursor database connection method, using, 485
custom exception classes, creating by subclassing Exception, 208

D
Dalke, Andrew, Web site address for "Sorting Mini-HOWTO", 58
data structures, defined, 37–61
database
  code example for retrieving all messages in, 486
  core functionality needed for bulletin board project, 485–486
  creating, 483–484
  fields created by the SQL statements, 484
  protecting with a password, 485
  setting permissions in, 484
database application
  interaction between program and user, 278
  interesting features of program that uses shelve module, 277
database modules, Web site address for a list of, 482
date tuples, fields of, 268
deck of cards, code example for creating an electronic, 272–274
deepcopy function
  benefits of using instead of shallow copy, 87
  vs. the copy function, 258
  from copy module, 87
def keyword, using to define functions, 562
def (or function definition) statement, example, 131–132
__del__ method, 211
del statement, 126
  deleting elements from lists with, 49
  function of in Python, 582
  using, 121–122
__delattr__ method, function of, 225

deleting elements from lists, 48–49, 50
__delitem__ method, function of in sequences and mapping protocol, 217
destructor, __del__ method as, 211
dict function, 93
__dict__ attribute, checking all values stored in an object with, 189
dict function, using, 81
dictfetchone database cursor method, using, 486
dictionaries, 79–93
  basic operations of, 82–83
  function of in Python, 562
  iterating over keys of with for loops, 113
  string formatting with, 84–85
  syntax for, 81
dictionary methods, 85–92, 92
  example of, 91–92
  list of, 577–578
die-throwing machine, code example for creating an electronic, 272
difflib library, function of, 299
digital telephone/address book, using Python dictionaries for, 79–80
dir function
  listing the contents of a module with, 195–196, 256
  returning an alphabetized list of attribute names, 301
direction keyword argument, in Placer layout manager, 336
discussion forum
  building your own, 481–499
  first implementation, 484–489
  main.cgi script for, 489–491
  second implementation, 489–496
  specific goals for creating, 482
  structuring the system, 489–490
  useful tools for creating, 482–483
dispatcher class, subclassing from asyncore, 443
division operator, additional available in Python 2.2, 11
division program, printing more instructive error message in, 203–204
DocBook XML, Web site address for information about, 402
docstring
  adding to a function, 132–133
  defined, 258
doctest module
  support for testing in standard libraries, 356–358
  Web site address for information about, 358
documentation, for modules, 258–260
dollar sign ($), using to mark the end of a string for pattern matching, 282
DOM (Document Object Model), Web site address for information about, 405
double-clicking, using, 24–25
double quotes ("), use of in Python strings, 26–27
downloading and installing
  Python on a Windows machine, 1–3
  Python on Linux and UNIX machines, 3–6

E

edit.cgi script
  for the bulletin board project, 490, 493–495
  as part of a CGI script, 473–476
  password handling in, 476
Eight Queens
  problem, 236
  using a tuple to represent a possible solution, 236–237
  using recursive case to solve the problem, 239–240
  using the base case to solve the problem, 237–239
electronic die-throwing machine, code example for creating, 272
elements, using slice assignments to insert into lists, 50
elif (else-if) clauses, using to check for several conditions, 103
else clauses
  function of, 208
  in loops, 118
  using to stop execution when something goes wrong, 202–204
  using with if statements, 102–103
empty lists, creating with None value, 43
encapsulation
  defined, 190
  hiding unnecessary details from the world with, 172–176
encoding, possible problems you may encounter, 419
end match object method, function of, 287
environment variable, including your module directory in, 252–253
equality operator (==)

checking if two things are equal with, 105
using, 17
error message (traceback), returned if exception object is not handled, 193
escape function, in re module, 283
escaping quotes, single-quoted strings and, 26–34
eval function
evaluating a Python expression with, 124–125
function of, 126
event handling, 338–339
events, default for some common Anygui components, 338
example
of dictionary method, 91–92
of indexing, 40–41
of sequence membership, 46
of sequence (string) multiplication, 44
of slicing, 41–42
of string formatting, 68–69
of supplying and receiving parameters, 149–150
except clause
adding more than one to the try/except statement, 199–200
catching exceptions with, 208
omitting exception class from, 201–202
Exception built-in exception class, 196
exception classes, building your own custom, 196–197
exception object, catching with the except clause, 200–201
exception objects, 193, 207
exception traceback, seeing when file does not exist, 303–304
exceptions, 193–208
catching, 197–199
catching all in a piece of code, 201–202
catching two with one block, 200
creating and raising your own, 194–197
and functions, 205
list of most important built in, 196
the Zen of, 206–207
exceptions and functions, function of, 208
exclusive indices, 41–42
exec statement, 126
executing a string with, 123–124
function of in Python, 585
execute database cursor method, using, 485
exponentiation (or power) operator, using, 12
expression statements, function of in Python, 581
expressions, 34, 571–580
vs. statements in Python, 15
extend method
appending several values at once with, 52–53
vs. concatenation, 52
extending and embedding the Python interpreter, Web site address for, 590
extreme programming (XP), created by Ken Beck, 349

F
factorial function, recursive implementation, 155–156
falling weights animation, code example, 541–544
faulty.cgi script, code example, 466
fetchone database cursor method, using, 486
Fibonacci numbers, 129
Fibs object, making and using in a for loop, 228
fields (or key-value pairs), using the FieldStorage class to retrieve, 467–468
FieldStorage class, using to retrieve fields from the cgi module, 467–468
FIFO (First-In-First-Out), 55
file function, 322
file handling, adding to your simple3.cgi script, 471–473
file iterators, new in Python 2.2, 317–318
file-like objects, defined, 320
file methods, basic, 306–313
file name, specifying for files downloaded with urlretrieve, 321
file permissions
setting in your CGI script, 463–464
UNIX command for changing, 464
__file__ property, examining to find Python modules source code, 259–260
file sharing
with GUI, 525–534
with XML-RPC, 501–523

file sharing program
  definition of peer-to-peer, 501
  preparations for creating with XML-RPC, 505
file type, calling as a function, 303
fileinput module
  included in standard library, 300
  iterating over all the lines in a series of text files with, 265–268
  lazy line iteration with, 316
  list of some important functions in, 265
  using to find the sender of an e-mail, 292–293
fileinput.close function, closing the entire chain of files with, 266
fileinput.filelineno function, returning number of current line within current file with, 266
fileinput.filename function, returning name of current file with, 266
fileinput.input function, using, 266
fileinput.isfirstline function, using, 266
fileinput.isstdin function, using, 266
fileinput.lineno function, returning number of the current line with, 266
fileinput.nextfile function, closing current file and skipping to next with, 266, 293
file.readline method, reading lines with, 309
files, 303–323
  closing, 310
  converting in lists of strings, 317–318
  iterating over contents, 313–318
  opening, 303–305
Filter class, filtering sequences with, 184–185
filter function, function of, 161–162
filterwarnings function, filtering warnings so they don't print with, 194
finally clause
  function of, 208
  using to do housekeeping after a possible exception, 204–205
find method, finding a substring within a larger string with, 70–71
findall function, in re module, 283
flags, using before width and precision numbers, 67–68
flags parameter, Web site address for information about, 286
flip function, in pygame.display module, 538
float function, 19
floating-point numbers (floats), use of, 10
floor function, 18
flush method, using to update files while you are working on them, 311
FoodExpert class, creating and initializing its taste buds, 180–182
for loops
  looping over the keys of a dictionary with, 113
  using, 111–113
  using to iterate over file lines in Python 2.2, 317–318
  vs. while loops, 112–113
fortune cookie program, code example for creating, 272
forward slash (/) character, using in Python code, 246
found_terminator method, of async_chat objects, 447
from . . . import . . . statements, using, 20
function calls, 18
function definitions
  defined, 165
  function of in Python, 587
functional programming
  defined, 165
  useful functions for, 160–165
functions, 34
  adding a docstring to, 132–133
  creating your own, 131–134
  documenting, 132–133
  and exceptions, 205
  vs. generators, 230
  making one to initialize a data structure, 138–139
  that don't return anything, 133–134
  using, 17–18
  using pow in place of exponentiation operator, 17–18
  writing one for getting names, 139
  writing one that stores names in your structure, 139–140
__future__ module, using, 21

G

garbage collection, 121
generators, 229–233
  components of, 233
  defined, 233, 243
  vs. functions, 230

making, 230–231
simulating with normal functions, 234
geometry attribute, setting for component layout in Anygui application, 334
get method, using to access dictionary items, 87–88
__getattr__ method, function of, 225
getattr function, using to access an attribute, 188
__getitem__ method, function of in sequences and mapping protocol, 217
get_surface function, in pygame.display module, 538
getvalue method, using to fetch values in a CGI script, 468
global scope, function of, 151
global statement, function of in Python, 585
global variables
encapsulating in objects, 173–174
rebinding, 152
GNU bash, linking several commands together with pipes in, 307
Gnutella, Web site address for, 501
graphical user interfaces (GUIs), making for your Python programs, 325–347
graphics, creating, 387–399
greedy and non-greedy patterns, 290
greeting script, with an HTML form (simple3.cgi), 470
grok, defined, 255
Group class, function of, 539
group numbers and functions, using in substitutions, 288–289
group match object method, function of, 287
groups, in regular expressions, defined, 286
GUI application
code example of one with only a window, 330
how elements of the interface can be used, 326–327
requirements for, 325
writing a basic for text editing, 325–347
GUI client
code example for, 527–529
code for getting a listing from a Node, 530
code for running, 530
files needed to run, 529
first implementation, 526–530
preparations for creating, 526
second implementation, 530–534
specific goals for in file sharing system, 525
using to make file sharing system easier to use, 525
GUI toolkits
available for Python, 325, 327–328
selecting one to use, 346
Web site addresses for some well-known, 591

H
hacking
the basics, 1–36
vs. cracking, 1
handlers.py, code example, 381–383
hasattr function, 191
hash sign (#), using, 25
has_key method, using to check a dictionary for a given key, 88
height attribute, setting in Anygui application, 334
help function
getting help with, 257–258
using in the interactive interpreter, 133
interactive help or help about a specific object, 301
hexadecimals and octals, 13–14
hmove keyword argument, in Placer layout manager, 336
Holden, Steve, *Python Web Programming* by, 440
hstretch keyword argument, in Placer layout manager, 336
HTML, marking up a file with for display in a Web browser, 365–366
HTML forms, 469
HyperDriveError class, building, 197

I
ICQ chat service, 439
identity operator, is operator as, 105–106
if __name__ == "__main__" statement, using, 249–250
if statement
conditional execution and, 102
function of, 17
function of in Python, 586
if statement blocks, nesting if statements inside of, 103–104

imaginary number, square root of a negative number as, 20
immutable parameter, 141
import statement
  importing functions from external modules with, 245
  using to import modules into Python, 19
import-only-once behavior, of modules, 247
import . . . as . . . statement, locally renaming a function with, 126
importing, something from a module, 97
in operator
  checking for a value in a sequence with, 45–46
  as the membership operator, 106–107
inclusive indices, 41–42
index, as assigned number in a sequence, 37
index method, finding the index of first occurrence of a value with, 53
IndexError class, built-in exception, 196
indexing
  example, 40–41
  function of in Python, 561
  as a sequence operation, 38–41
indexing notation, using, 48
indices, as limits for your slice, 41–42
infinite recursion, defined, 155
inflate rectangle method, in arcade game project, 545
information-gathering agent
  adding methods for distributing news item, 427–428
  first implementation, 423–426
  methods for adding sources and destinations to, 427
  second implementation, 426–436
  specific goals for creating, 422
  useful tools for writing, 422
ingredients, in algorithms, 9
inheritance
  defined, 190
  investigating, 185–186
  using when creating a new class, 176
init function, in pygame module, 537
__init__ method, creating constructors with, 210
__init__.py file (module), adding to a package, 254–255
initialization, of empty lists, 43
inplace parameter, care needed when using, 267
input function, 16
  vs. raw_input function, 30
  using to get user input, 560
insert method, inserting an object into a list with, 53–54
inspect module, function of, 189
instances, of a class, 177
Instant Python Web tutorial, Web site address, 559
instructions, in algorithms, 9
int function, 19
integers, using large, 12–13
IDLE
  running, 2
  saving and executing programs in, 22
  Web site address for information and documentation, 2
__iter__ method, using, 227–228
interactive interpreter, 7–9
  code for telling where to look for a module, 246
  exploring for built-in exception classes, 195–196
  investigating modules in, 256
  keyboard shorcut for exiting, 4
  problem with writing programs in, 21
  using as a calculator, 10–14
  using the help function in, 133
interfaces
  checking to see if required methods are present in, 188
  and introspection, 187–189, 191
invisible dictionary, defined, 151
IOError class, built-in exception, 196
IRC chat service, 439
is operator, testing for identity with, 105–106
isinstance function
  for checking if an object is an instance of a class, 185–186
  using, 169
issubclass function, checking to see if a class is a subclass of another with, 185
item assignments, changing lists with, 48
items, for dictionaries, 81
items method, returning all items of a dictionary with, 88
iter function, extracting an iterator from an object with, 228, 243
iterator protocol, 227–228
iterators, 227–229
  code for converting to a list, 229
  defined, 243
  making sequences from, 229
  special (or magic), 209

iteritems method, returning an iterator instead of a list with, 88–89
iterkeys method, returning an iterator over the keys with, 89
itervalues method, returning an iterator of the values in the dictionary with, 91

J
Java Swing GUI toolkit, available for Python, 327
join method, joining the elements of a sequence with, 71–72
ju-jitsu of programming, 349–350
Jython, Web site address for, 7, 345, 589
Jython and Swing, 345

K
KaZaA, Web site address for, 501
key-value pairs (or fields), using the FieldStorage class to retrieve, 467–468
key-value pairs, constructing dictionaries from, 81
keyboard interrupts, in UNIX and DOS, 444
KeyError class, built-in exception, 196
keys method, returning a list of keys in the dictionary with, 89
keyword arguments, *see* keyword parameters
  for default Anygui layout scheme, 336
keyword parameters and defaults, 142–144
Kofler, Michael, *MySQL* by (Apress, 2001), 483

L
lambda expressions, 160–165
  using with the filter function, 161–162
  using with the map function, 161
  using with the reduce function, 162–164
LaTeX, 366
layout
  advanced for Anygui application, 336–338
  code example of using the default scheme, 337–338
  problems with in Anygui application, 334–335
layout managers, for arranging components when window is resized, 336–338
left keyword argument, in Placer layout manager, 336
len function, getting length of a sequence with, 46–47
__len__ method, function of in sequences and mapping protocol, 217
LIFO (Last-In-First-Out), 54
Line-by-Line Chimp Tutorial, Web site address for, 537
lines, reading and writing, 309–310
link function, linking components to event handlers with, 338–339
Linux machines, downloading and installing Python on, 3–6
list comprehension
  using, 126
  using to filter dir function results, 256
  using with loops, 118–120

list, dict, and str types, subclassing, 220–222
list function, 61
  using to convert an iterator to a list, 229

list methods, 50–58
  list of in Python, 577
listen method, calling to listen for incoming connections, 443–444
lists
  deleting elements from, 48–49
  expression for converting back to a string, 47
  vs. tuples, 37
  and tuples, 37–61
  ways you can change, 47–58
  the Python workhorse, 47–58
local scope, storing parameters in, 136
local variables vs. global variables, 151
logger class, code examples of a program using, 361–362
logging, importance of for discovering problems and bugs, 360–362
long function, 19
long integers (longs), 12–13
long strings, using triple quotes with, 31
longs (long integers), 12–13
loops
  breaking out of, 115–117
  else clauses in, 118
  using, 110–118, 126
lower method, returning a lowercase version of a string with, 72–73

M

Macintosh, installing Python on, 7
Macintosh Library modules, Web site address for, 590
MacPython, Web site address, 589
magic methods, 242
  for advanced use, 222–223
  implementing properties with old-style classes with, 225–226
main.cgi script, for the bulletin board project, 489–491
maketrans function, for making a translation table, 74–75
map function, function of, 161
mappings, 92
  vs. sequences, 38
markup program (simple_markup.py)
  basic steps for creating, 369
  code example of, 369–371
markup.py, code example, 385–386
Martelli, Alex, *Python Cookbook*, 109
match objects and groups, returned by re functions, 286–299
match function, in re module, 283
math module, importing into Python, 19
math.ceil function, 35
math.floor function, 35
math.sqrt function, 35
max function, 61
  returning largest element of a sequence with, 46–47
  using with the reduce function, 163
MAX_HISTORY_LENGTH, setting, 509
maximum field width, for converted strings, 65
md5 module, function of, 299
Meerkat news service, code for using to find Python articles, 504–505
membership
  checking for as a sequence operation, 38
  function of, 60
message editor, code example for, 494–495
message viewer, code example for, 492–493
methods, 60. *See also* dictionary methods; list methods; string methods
  calling, 51
  making private, 175–176
  polymorphism in, 170
  special (or magic), 209
  throwing around, 180–182
MIME (Multipurpose Internet Mail Extensions) types, Web site address for, 466
min function, 61
  returning smallest element of a sequence with, 46–47
minimum field width, for converted strings, 65
minus sign (-), conversion specifier, 65
mixin class, implementing to handle administrative details, 412–414
mode argument, possible values for, 304
module.function, writing to use functions from a module, 19
modules, 35
  code for making both importable and runnable, 568
  creating your own, 245–247
  exploring, 255–260, 300
  finding on the Web, 255
  function of, 300
  importing into Python, 19
  making yours available, 250–254
  naming in Python, 254
  other interesting standard, 299
  saving, 245–246
  in standard Python installation, 245–260
  structuring by grouping into packages, 254–255
  telling the interpreter where to look for them, 252–253
  using __file__ property to examine source code for, 259–260
  using __name__ variable in, 249–250
  using to define things, 248–250
  using to facilitate code reuse, 248–249
  Web site addresses for third party used in projects, 591
  what they contain, 256–257
modulus (remainder or %) operator, using, 11–12
multiple inheritance, using, 186–187
multiplication
  example, 44
  of sequences, 43–44
multiplying, as a sequence operation, 38
multithreaded chat server, pointers for, 441–442
music, Web site address for information about using XML to represent, 402
my_math module
  code example of part of, 359
  executing the test program for, 359

MySQL (Michael Kofler, Apress 2001), 483
MySQL database, Web site address for, 482
MySQLdb module, Web site address for, 482

N
name-storing program
  collecting parameters in, 144–147
  writing one that stores names in your structure, 139–141
NameError class, built-in exception, 196
namespace, 123
nested scopes, support for in Python 2.2, 153–154
nesting blocks, 103–104
network programming, standard library modules for, 318–319
new-style classes, introduced in Python 2.2, 242
NEWNEWS command, importance of NNTP server support for, 422
newsagent1.py, code example, 426
newsagent2.py
  code example, 432–436
  code for running, 431
newsgroups and mailing lists, Web site addresses for, 592
newsreader. *See* NNTP client (newsreader)
NNTP (Network News Transfer Protocol), defined, 421
NNTP client (newsreader), 421
NNTP server
  code for fetching the body of an article, 425
  code for instantiating and calling the newnews method, 424
  importance of support for NEWNEWS command, 422
  testing, 423
nntplib library
  introduction to, 421–437
  using to write an information-gathering agent, 421–437
non-greedy patterns, 290
None value
  using for empty lists, 43
  using to leave a list position empty, 48
numbering, the lines of a Python script, 266–268
numbers and expressions, 10–14
Numerical Python, Web site address for information about, 536

O
object-oriented design, 189–191
object-oriented model, sketch of, 189–190
object-oriented programming, benefits of, 167–168
objects
  accessing attributes of from the outside, 174–175
  changes in how they work in Python 2.2, 209–210
  defined, 190
  encapsulating global variables in, 173–174
  the magic of, 167–176
  most important benefits of, 167–168
objects.py file
  in arcade game project, 545
  code example, 547–549
octal numbers, how they are written, 14
official Python distribution, Web site address, 589
official Python tutorial, Web site address for, 590
online resources
  directory of, 589–592
  for finding third party modules, 388
open function, opening files with, 303–305
operators, list of in Python, 572–573
ord function, 127
os module
  getting access to operating services with, 262–264
  included in standard library, 300
  some important functions and variables in, 263
os.environ mapping, environment variables contained in, 263
os.linesep variable, line separator string used in text files, 263
os.makedirs function, making necessary directories in a given path with, 415
os.path.isdir function, checking if a path is a directory with, 415
os.path.join function, joining several paths with the correct separator with, 415
os.pathsep variable, using when grouping several paths, 263
os.sep variable, separator used in pathnames, 263
os.startfile function, using to start a Web browser, 264

os.system function
  example of starting a Web browser with, 264
  using to run external programs, 263
overriding 242–243

P
package directory, adding modules to, 254–255
package layout, example of simple, 254
packages
  function of, 300
  grouping modules into, 254–255
pagemaker.py, code example, 409–410
parallel execution, handling, 441–442
parallel iteration, for iterating over two sequences at the same time, 114
parameter gathering, reversing the process, 147–149
parameters
  changing inside a function, 135–137
  collecting, 144–150
  combining positional and keyword, 143
  defined, 165
  distributing, 147–149
  example of supplying and receiving, 149–150
  keyword, 142–144
  the magic of, 134–150
  modifying by wrapping your value in a list, 141
  reasons for wanting to modify, 137–141
  where the values come from, 134–135
parse function, using to parse an XML file, 405–408
pass statement, 126
  function of in Python, 582
  using, 120–121
password, protecting your database with, 485
password handling, in edit.cgi script, 476
path configuration file, filename extension for, 253
pattern matching, the beginning and end of a string, 282
(*pattern*)* operator, for repeating a pattern zero or more times, 282
(*pattern*)+ operator, for repeating a pattern one or more times, 282
(*pattern*){*m, n*} operator, for repeating a pattern from *m* to *n* times, 282
patterns, greedy and non-greedy, 290
PDF (Portable Document Format), 387
  basic module for generating, 390
pdfgen module, for creating PDF files, 390
peek_sum, peek_max, using with the reduce function, 164
peer-to-peer file sharing program
  creating peers in, 513
  defined, 501
  requirements program must satisfy, 502–503
  useful tools for creating, 503–505
peers
  code for introducing the first to the second, 514
  connecting to, 513
  creating in a peer-to-peer file sharing program, 513
percent (%), using as string formatting operator, 63–64
period ("dot") character, as regexp wildcard, 279
Piddle project page, Web site address for, 399
pipe (|) character
  linking several commands together with, 307
  using for character set alternatives, 281
pipeline, command example, 307
Placer layout manager, layout arguments for, 336
playful programming, 349–364
plus sign (+)
  conversion specifier, 65
  using for concatenating lists, 565
PolyLines
  code example for adding to your drawing, 394
  creating, 392–394
polymorphism
  creating an example with, 172–173
  defined, 168, 190
  the many forms of, 171–172
  using to create an online payment system, 168–172
pop method, modifying a list and returning a value with, 54–55
popitem method
  removing and processing dictionary items with, 89
port numbers
  choosing for your chat server, 442
position attribute, setting for component layout in Anygui application, 334

positional parameters, 142
PostgreSQL database, Web site address for, 482
pound bang (#!) line, adding to your CGI script, 463
pow function, using in place of exponentiation operator, 17–18
power function, recursive implementation, 156–157
pprint function, using instead of the print statement, 251
precision and width, for conversion specifiers, 67
prettyprint, adding a helper function to, 241
principles of abstraction, polymorphism and encapsulation as, 172
print statement
  function of in Python, 583
  using to print several values, 95–96, 125
printing, with commas, 95–96
programming
  aspects of practice of flexibility in, 350
  planning for change, 356
programs, saving and executing, 21–25
projects
  basic structure of all, 363
  building your own bulletin board, 481–499
  creating a chat server, 439–460
  file sharing with XML-RPC, 501–524
  file sharing II, with GUI, 525–534
  information-gathering agent, 421–437
  instant markup, 365–386
  in the news, 421–437
  painting a pretty picture (graphics), 387–399
  remote editing with CGI, 461–480
  XML for all occasions, 401
prompt (>>>). *See* Python prompt (>>>)
properties, 223–227
  defined, 224
  special (or magic), 209
property function, using, 224–225
protocol, use of, 216
prototyping your programs, 350–351
pseudo-code, defined, 157
psycopg front-end, for PostgreSQL, 482
.pth filename extension, for a path configuration file, 253
PuTTY terminal emulator, Web site address for information about, 442
.py filename extension
  importance of using for created files, 21
  using .pyw instead of in Windows, 254
.pyc filename extension, defined, 246
Pydoc Online, Web site address for, 590
Pygame
  contents of module, 537
  Web site address for documentation, 535
  Web site address for downloading, 536
pygame.display module, contents of, 537–538
pygame.event module, function of, 539
pygame.font module, function of, 538
pygame.image module, function of, 539
pygame.locals module, contents of, 537
pygame.mouse module, function of, 539
pygame.sprite module, contents of, 538
PyGTK GUI toolkit, available for Python, 327
PyQt GUI toolkit, available for Python, 327
PyRXP XML parser, Web site address for, 405
Python
  assignment in, 14
  the basics, 559
  checking for on Linux and UNIX systems, 3
  compiling from sources, 5–6
  dictionaries, 79–93
  downloading and installing, 1–7
  downloading and installing on a Windows machine, 1–3
  hexadecimals and octals in, 13–14
  how large integers behave in older versions, 13
  installing on a Macintosh, 7
  list of dictionary methods in, 577–578
  list of list methods in, 577
  list of operators in, 572–573
  list of some important built-in functions in, 573–576
  list of string methods in, 578–580
  listing of basic (literal) values in, 571
  long integers (longs) in, 12–13
  a minimal introduction to, 559–569
  numbers and expressions, 10–14
  objects and stuff, 564–568
  online resources for distributions, 589
  other distributions, 7
  programming in, 9
  prompt (>>>), 8

Python (*continued*)
reference, 571–580
saving and executing programs in, 21–25
statements in, 14–15
summary of statements in, 581–587
taking control of your computer with, 1–36
useful tools and modules, 591
useful tools for XML support, 403
using equality operator (==) in, 17
using exponentiation (or power) operator in, 12
using large integers in, 12–13
using remainder (modulus or %) operator in, 11–12
using the input function in, 16
using the interactive interpreter as a calculator, 10–14
variables in, 14
version implemented in Java, 7
Web site address for information about setting up in Mac OS, 253
Web site address for information about using DOM in, 405
Python 2.2
additional division operator available in, 11
file iterators as new feature in, 317–318
the help function in, 257–258
new-style classes introduced in, 242
subclassing built-in types in, 221
types and classes in, 178
using super function in, 215
Python Cookbook, Web site address, 109
Python Database Topic Guide, Web site address for information about, 482
Python date tuples, fields of, 268
Python DB-API, using to interface with the server, 482
Python dictionaries, some arbitrary uses of, 79–80
Python documentation, Web site address for, 590–591
Python Enhancement Proposal number 245
Web site address for, 188
Python Enhancement Proposals, Web site address for, 591
Python interpreter, Web site address for extending and embedding, 590
Python Language Reference, Web site address for, 590
Python Library Reference
for description of string module, 69
finding built-in classes available in, 195
finding module information in, 259
for information about using DOM in Python, 405
slice objects described in, 220
Web site address, 301, 590
Python programs. *See* scripts
Python prompt (>>>), 8
Python reference, 571–587
Python Reference Manual, Web site address, 220, 223
Python scripts
example for numbering the lines of, 266–268
running from a command prompt, 23
Python shell, IDLE interactive, 3
Python Web Programming (Steve Holden), for information about asyncore and asynchat, 440
Python Web site address, for downloading Python documentation, 259
PYTHONPATH, code for adding your module directory to, 252
PythonWare, Web site address, 7, 589
PythonWin GUI toolkit, available for Python, 327
.pyw filename extension, using instead of .py, 254
PyXML package, Web site address for downloading, 403

Q
queens generator, solutions given by, 240
question mark (?) character, using after a subpattern in character sets, 281–282
quote function, urllib module utility, 321
quote_plus function, urllib module utility, 321

R
r prefix, use of in raw strings, 33–34
'r' value, for mode argument of the open function, 304
Raggett, Dave, Getting Started with HTML by, 365
raise statement
function of, 208, 584

using with an argument to raise an exception, 195–196
using without arguments, 198–199
random access, file-object methods used for, 308–309
random module
included in standard library, 300
some important functions in, 270
random numbers, functions in random module that return, 270–271
random.choice function, 191, 271
returning a random element from the sequence *seq* with, 270
random.random function, 270
random.randrange function, using, 271
random.shuffle function, shuffling the elements of a sequence randomly with, 271
random.uniform function,
getting a random real number with, 270
using, 271
range function, 127
for iterating over a range of numbers, 112
raw strings, using, 32–33
raw_input function, 35
vs. input function, 30
re functions, using VERBOSE flag in, 289
re match objects, some important methods of, 287
re module
contents of, 283–286
included in standard library, 301
some important functions in, 283
support for regular expressions contained in, 279–299
read method
iterating over characters with, 315
looping over file characters with, 314
reading and writing files, 306–307
readline method, using in a while loop, 315
readlines method, iterating over lines with, 315
re.compile function, transforming a regexp to a pattern object with, 283
Rectangle class, code example for using, 223
recursion, defined, 154–155, 165
recursive case
of recursive function, 155
using to solve the Eight Queens problem, 239–240
using with recursion, 231
recursive functions
parts of, 155
two classics, 155–157
recursive generator
adding a test to beginning of, 232
creating, 231–233
Red Hat Linux Package Manager (RPM), steps for installing Python RPM packages, 4–5
reduce function, 166
function of, 162–164
re.escape function, escaping all characters in a string with, 285–286
refactoring, defined, 421
re.findall function, returning a list of all occurrences of a pattern with, 285
regexp (regular expression)
defined, 279
period (“dot”) character as wildcard in, 279
Web site address for complete listing of operators, 282
register_function method of SimpleXMLRPCServer, 506
register_instance method of SimpleXMLRPCServer, 505–506
regular expressions, using VERBOSE flag to make more readable, 289

reload function
function of, 301
using to reload modules, 247
remainder (modulus or %) operator, using, 11–12
re.match function, matching a regexp at beginning of a string with, 284
remote editing, of documents with CGI via the Web, 461–480
remote files
accessing with urllib module, 318–321
opening with urlopen from the urllib module, 319–320
retrieving, 320–321
remove method, removing the first occurrence of an value in a list with, 55
repetition operators, making nongreedy, 290
replace method, using, 73
reportlab module, code for importing, 389
ReportLab package
drawing with, 390–392

ReportLab package (*continued*)
  learning the basics of for a graphics project, 388
  PolyLine class in, 392–394
  Web site address for downloading, 388
reportlab.graphics package, creating objects with, 390
reportlab.graphics.charts.lineplots module, using the LinePlot class from, 397–398
repr function, 35
  using, 28–29
  polymorphic use, 171–172
requirement specifications, importance of writing precise, 354–355
re.search function, searching a string with, 283–284
re.split function, splitting a string by occurrences of a pattern with, 284–285
re.sub function, 283
  substituting occurrences of a pattern with a replacement with, 285
  using group numbers in the substitution string, 288
return statement
  function of in Python, 583
  using, 132
  using to end a function, 134
  using to return a value from a function, 562–563
reverse method, reversing elements in a list with, 55–56
rexec module, function of, 299
right keyword argument, in Placer layout manager, 336
round function, 35
  using, 18
RPC (Remote Procedure Call), 501
RPM packages, locating on for your system, 5
rules module, code example for Rule class in, 376
rules.py, code example, 383–384

S

save.cgi script
  for the bulletin board project, 490, 495–496
  as part of a CGI script, 473, 476–478
saving script
  code example for, 495–496
  code example of, 477–478
SAX (Simple API for XML), processing XML files with, 401, 405
scopes
  defined, 165
  nested, 153–154
scoping, 150–154
scripts
  making executable before running, 24
  making them behave like normal programs, 23–25
  running from a command prompt, 23
search path, code example for finding in the sys module, 251
seek method, for random access of files, 308–309
self parameter, 178–179
sequence and mapping protocol
  basic, 217–220
  code example for, 218–220
sequence types, built-in, 37
sequence unpacking, for performing several assignments simultaneously, 98
sequences, 60
  as basic data structures in Python, 37
  common operations, 38–47
  comparing, 107
  concatenating with the addition operator, 43
  making from iterators, 229
  vs. mappings, 38
  multiplying, 43–44
sequences and mapping protocol, 243
  extra requirements imposed upon methods, 217
serve_forever method, calling, 506, 510
server.py, code example, 517–520
set_caption function, in pygame.display module, 538
setattr function, setting the attributes of an object with, 188
__setattr__ method, function of, 225
setdefault dictionary method, 90
__setitem__ method, function of in sequences and mapping protocol, 217
set_mode function, in pygame.display module, 538
set_terminator method, setting the line terminator with, 447
SHA (Secure Hash Algorithm), 476
sha module
  code example of how you can use, 477
  function of, 299
  using in password handling, 476–478
shadowing, the problem of, 152

shallow copy, 86
Shelf object, using to store things, 274–275
shelve module
  code example of a simple database application that uses, 275–277
  included in standard library, 301
  using for a simple storage solution, 274–275
shelve.open function, potential trap when using, 274–275
short-circuit logic, 107–108
signs, alignment, and zero-padding, 67–68
Simple API for XML (SAX). *See* SAX (Simple API for XML)
simple generators. *See* generators
simple statements, function of in Python, 581
simple2.cgi, code example, 468
simple3.cgi, code example, 470
simple_chat.py, code example, 448–449
simple_edit.cgi, code example, 472
simple_guiclient.py, code example for, 527–529
simple_main.cgi, code example, 488–489
simple_markup.py program
  adding components to in second implementation, 371–386
  adding handlers (renderers) to, 371–372
  code example for implementing, 370
  making rules a separate object for, 375–376
simple_node.py
  code example for, 510–512
  example of how it is used, 513–514
single quote ('), use of in Python strings, 26–27
single-quoted strings and escaping quotes, 26–34
site-packages directory, storing modules in, 251–252
six degrees of separation hypothesis, Web site address for, 509
size attribute, setting for component layout in Anygui application, 334
Slashdot discussion forum, Web site address, 481
slice assignments, using, 49–50
slices
  assigning, 49–50
  deleting from lists, 50
slicing
  function of in Python, 561
  as a sequence operation, 38
  shortcut for, 41–42
  using to access ranges of elements, 41–42
socket module, using functions in to create a chat server, 440–441
sockets, defined, 439
SolarWolf game (Pete Shinners), Web site address for, 557
sort list method
  sorting lists in place with, 56–57
sorting, defining your own comparison function for advanced, 57
Sorting Mini-HOWTO, Web site address for, 58
sorting order, for upper- and lowercase letters, 293
source code, reading to better understand Python, 259–260
space character, conversion specifier, 65
Space Environment Center, Web site address for, 389
space keyword argument, in Placer layout manager, 336
SPAMFilter, important points in definition of, 185
span match object method, function of, 287
special characters
  escaping, 280
  rules for using in character sets, 280–281
Sping graphics and plotting package, Web site address for, 399
split method, splitting a string into a sequence with, 73
split function, in re module, 283
Sprite class, function of, 538
SQL databases
  creating, 483–484
  fields created by the SQL statements, 483–484
  setting permissions in, 484
  two freely available, 482
SQL programming, Web site address for practicing, 483
sqrt function, 19–20
square root, using sqrt function to calculate, 20–21
Squish arcade game
  basic design process, 535
  first implementation, 540–545
  preparations for creating, 540
  second implementation, 545–556

Squish arcade game (*continued*)
some screenshots of, 546
specific goals for creating, 536
steps for creating animation of falling16-ton weight, 540–541
useful tools for creating, 536–539
squish.py file
in arcade game project, 545
code example, 550–556
code for executing to run the game, 546
stack trace, example of, 205
standard libraries, support for testing in, 356–360
standard library, included in standard Python installation, 260–300
standard library reference, Web site address for time module information, 270
star. *See* * (asterisk); asterisk (*)
start match object method, function of, 287
statements, 14–15, 34
vs. expressions in Python, 15
summary of, 581–587
use of colon character in, 17
StopIteration exception, raising, 227
store function
steps performed by, 140
testing, 140–141
str function, using, 28–29
streams, three standard, 306
strftime function, formatting time as strings with, 424
string formatting
advanced template mechanism, 294
conversion types, 66
with dictionaries, 84–85, 92
example, 68–69
the long version, 64–69
operator, 63–64
the short version, 63–64
a simple conversion, 66
for working with strings, 76
string literals, indexing, 39
string methods, 69–75
list of in Python, 578–580
using to check membership, 45
for working with strings, 76
string module
useful values from, 70
Web site address for description of, 69
string representations, 28–29
string.maketrans function, making a translation table with, 77
strings, 26–34, 35
basic operations, 63
comparing, 107
concatenating, 27–28
statements for executing and evaluating, 122–125
working with, 63–78
strip method, returning a string where whitespace has been stripped, 74
sub method, in Handler class, 374
subclass, defined, 177
subpatterns
and alternatives in character sets, 281
making optional, 281–282
operators that allow you to repeat more than once, 282
optional and repeated in character sets, 281–282
sunspot graph program
code example for final, 397–398
first prototype for, 394–395
second implementation, 395–398
sunspots data
downloading from the Space Environment Center Web site, 389–390
using urllib to fetch files across the Internet, 396
super function
using in Python 2.2, 215
why it is better, 216
superclass
defined, 177
specifying, 184–185
superclass constructor, calling the unbound, 214–215
superclasses, multiple, 186–187
SyntaxError class, built-in exception, 196
sys module
included in standard library, 300
some important functions and variables in, 261
sys.argv variable, contents of, 261
sys.exit function, exiting the current program with, 261
sys.modules mapping, mapping module names to actual modules with, 261
sys.path variable, using, 261
sys.platform variable, 261
sys.stderr variable, 262
sys.stdin variable
a simple script that counts the words in, 308

using to iterate over lines in standard input, 317
where Python gets its input from, 262
sys.stdout, where Python prints to, 262

T
tab characters, using to indent blocks, 100
telephone book
code example for, 83–84
using Python dictionaries for, 79–80
tell method, for random access of files, 308–309
telnet, using to connect to any socket server, 442
template system
code example, 295
code example of output from, 298
code example of template definitions, 297
making, 294–299
a simple template example, 297
what the program does, 296
terminal emulator software, 442–443
ternary operator, implementing, 109
test document, code example, 367–368
test program, for computing the area of a rectangle, 355
TestCase class, writing a test program for my_math module with, 359
testing
automated in you Python programs, 350
setting up for your programs, 354–362
testmod function, running from doctest, 357
$\mathrm{T_EX}$ typesetting program, $\mathrm{L^{A}T_EX}$ markup system based on, 366
$\mathrm{T_EX}$ Users Group home page, Web site address, 366
text areas vs. text fields, 327
text editor
adding buttons to, 331
adding components to, 332
creating windows and components, 330–332
the finished program, 339–342
sketch of, 326
steps for trying out the finished program, 341
writing a basic, 325–347
text fields vs. text areas, 327
text files, writing a basic program for editing, 325–347
TextArea class, 347
TextField class, 347
The Covers Project, code example for connecting to its server, 504
third party modules, Web site addresses for those used in the ten projects, 591
threading module, information sources, 441–442
threads, alternative approach to creating a chat server, 439
time module
checking the standard library reference for information about, 270
fields of Python date tuples, 268
functions needed from for the information-gathering agent, 423
included in standard library, 300
list of some important functions in, 269
using, 268–270
time stamps, getting for your sunspot graph program, 393–394
time.asctime function, formatting the current time as a string with, 269
time.gmtime function, displaying universal time with, 269
time.localtime function, converting seconds to a date tuple, local time with, 269
time.mktime function, converting time tuple to local time with, 269
time.sleep function, makes interpreter wait for a given number of seconds, 269
time.strptime function, parsing a string into a time tuple with, 269
time.time function, returning the current time as seconds since epoch with, 269
TinyFugue terminal emulator, Web site address for, 443
Tkinter GUI toolkit
available for Python, 327–328
Web site addresses for information about, 343
tools
deciding which you need to write your programs, 366–367
useful for the graphic-generating package, 388–389
top keyword argument, in Placer layout manager, 336

traceback (error message), returned if exception object is not handled, 193
translate method, replacing parts of a string with, 74–75
translation table
  contents of, 75
  making before you can use translate method, 74–75
triple quotes (''' or """), using with long strings, 31
truth values, 101–102
try/except statement
  avoiding program termination with, 206–207
  checking for object attributes with, 207
  using to catch exceptions, 197–199
try/finally statement, avoiding program termination with, 206–207
tuple function
  function of, 59
  using in Python 2.2 and later, 169
tuples
  basic operations of, 59
  as immutable sequences, 58–59
  important reasons for using, 60
  and lists, 37–61
  vs. lists, 37
type checking, reasons to use, 220
type function, using, 169
TypeError class, built-in exception, 196
types and classes, in Python 2.2, 178

U
u prefix, use of in Unicode strings, 34
unbound methods, example of, 214–215
unbound superclass constructor, calling, 214–215
Unicode, Web site address for information about, 33
Unicode strings, using, 33–34
unittest module
  support for testing in standard libraries, 358–360
  Web site address for information about, 359
universal time, displaying with the time.gmtime function, 269
UNIX, code for making an executable script in, 569
UNIX machines
  downloading and installing Python on, 3–6
  making Python scripts behave like normal programs on, 23–25
update function, in pygame.display module, 538
update method, updating one dictionary with items of another with, 90–91
urlencode(*query*[, *doweq*]) function, urllib module utility, 321
urllib module
  accessing remote files with, 318–321
  utilities available in, 321
urllib.encode function, 322
urllib.quote_plus function, 322
urllib.quote function, 322
urllib.unquote function, 322
urllib.unquote_plus function, 322
urllib.urlcleanup function, 321
urllib.urlopen opening remote files with, 319–320
urllib.urlretrieve retrieving remote files with, 320–321
Usenet, Web site address for information about, 421
UserDict module, in the Python standard library, 221
UserList module, in the Python standard library, 221
UserString module, in the Python standard library, 221

V
-v switch, using to get more input from doctest.testmod function, 357–358
ValueError class, built-in exception, 196
values, getting for your sunspot graph program, 393–394
values method, returning a list of values in the dictionary with, 91
van Rossum, Guido, Python's Benevolent Dictator For Life (BDFL), 590–591
variables, 14, 34
  making global, 152
  naming, 14
  storing from an enclosing scope, 154
  switching the contents of two with sequence unpacking, 98
  using to refer to functions in Python, 20
vars function, using to return a namespace or scope, 150–151
Vaults of Parnassus
  finding a graphics-generating package in, 388
  searching for GUI toolkits in, 327
VERBOSE flag, making regular expressions more readable with, 289

view.cgi script, for the bulletin board project, 489, 492–493
vmove keyword argument, in Placer layout manager, 336
vstretch keyword argument, in Placer layout manager, 336

W
'w' value, for mode argument of the open function, 304
W3Schools Web site, for a tutorial on XML, 401
warnings, function of, 194
warnings.filterwarnings function, using to filter out warnings, 194
Web browser, example of starting with the os.system function, 264
Web editor, code example of a simple, 472
Web pages, editing remotely with CGI, 461–480
Web Programming Topic Guide, Web site address for information about, 461
Web security, Web site address for information about, 465
Web server, putting your programs on, 462–463
Web site address
  for ActivePython, 7
  ActivePython, 589
  for Andrew Dalke's "Sorting Mini-HOWTO", 58
  Anygui GUI toolkit, 328–329
  for Anygui GUI toolkit documentation, 336
  for Cameron Laird's Web page of GUI toolkits, 327
  for a complete listing of regexp operators, 282
  for concise description of what XML is, 401
  for The Covers Project, 504
  for Dave Raggett's Getting Started with HTML guide, 365
  for description of six degrees of separation hypothesis, 509
  for description of string module, 69
  for detailed information about asyncore, 440
  for downloading Acrobat Reader, 387
  for downloading Instant Python Web tutorial, 559
  for downloading new version of SimpleXMLRPCServer.py, 503
  for downloading Pygame, 536
  for downloading PyXML package, 403
  for downloading ReportLab package, 388
  for finding NNTP server resources, 422–423
  for game development Web sites, 557
  for GNU bash information, 307
  for Gnutella, 501
  for Gordon McMillan's Socket Programming HOWTO, 439
  for IDLE information and documentation, 2
  for information about DocBook XML, 402
  for information about encodings, 419
  for information about extreme programming (XP), 349
  for information about graphs and trees, 235
  for information about HTML and HTML forms, 469
  for information about SAX, 401
  for information about setting up Python in Mac OS, 253
  for information about the ConfigParser module, 353
  for information about the doctest module, 358
  for information about the unittest module, 359
  for information about Usenet, 421
  for information about using DOM in Python, 405
  for information about using XML for representing music, 402
  for information about using XML to describe any physical object, 402
  for information about Web security, 465
  for information about XHTML, 402
  for information about XML-RPC, 501
  for Jython, 7, 589
  for KaZaA, 501
  for the Line-by-Line Chimp Tutorial, 537
  for a list of chat services, 439
  for a list of database modules, 482
  for a list of official MIME types, 466
  for Macintosh Library modules, 590
  MacPython, 589
  for MySQL database, 482
  for MySQLdb module, 482
  for newsgroups and mailing lists, 592
  for the official Python distribution, 589
  for the official Python tutorial, 590

Web site address (*continued*)
for PostgreSQL database, 482
for practicing SQL programming, 483
for psycopg front-end for PostgreSQL, 482
for PuTTY terminal emulator, 442
for Pydoc Online, 590
for Pygame documentation, 535
for the Python Database Topic Guide, 482
for Python documentation, 590–591
for the Python Enhancement Proposal number 245, 188
for Python Enhancement Proposals, 591
for the Python Library Reference, 259, 301
for Python Web page, 319
for the Python win32all Windows extensions, 589
for PythonWare, 7, 389
for Slashdot discussion forum, 481
for the SolarWolf game by Pete Shinners, 557
for Space Environment Center, 389
for the standard SocketServer module documentation, 441
for the standard threading module documentation, 441
$\mathrm{T_EX}$ Users Group home page, 366
for tutorial about Meerkat XML-RPC API, 505
for Unicode information, 33
for using XML to represent human moods, emotions, and character traits, 402
for Vaults of Parnassus, 388
for the Web Programming Topic Guide, 461
for the World Wide Web Consortium, 365
webbrowser module, using to start a Web browser, 264–265
website.xml
code example, 404–405
code for a minimal XML parser to parse, 406
code that uses methods to build a list of the h1 elements for, 407–408
weight.png image file, in arcade game project, 545
weights.py
code example, 541–544
code example for running, 541–544
explanation of important points, 544–545
while 1/break idiom
using, 116–117
using to iterate over file contents, 314
while loops
vs. for loops, 112–113
using, 111
using to iterate over file contents, 315
while statement, function of in Python, 586
width and precision, for conversion specifiers, 67
width attribute, setting for component layout in Anygui application, 334
win32all, Web site address for the Python Windows extensions, 589
window
adding to your text editor, 330–331
example of one with layout problems, 333
Window class, Window
instantiating to create a text editor window, 330–332
Window.add method, adding a component to a window with, 347
Windows machines, downloading and installing Python on, 1–3
wrap function, defined, 432
write file method, writing lines with, 310
wxPython, using, 344
wyPython GUI toolkit, available for Python, 327

X

XHTML, Web site address for information about, 402
XML
using to represent human moods, emotions, and character traits, 402
Web site address for description of what it does, 401
Web site address for sampling of existing applications, 402
XML Cover Pages, Web site address, 402
XML file, using to generate a complete Web site, 402
XML for all occasions project, defining the problem, 401–402
XML format, main concepts for designing your Web page, 403–404
XML parsing, looking at how it works, 405

XML-RPC
  defining result codes for, 508
  file sharing with, 501–523
  Web site address for information about, 501
XML-RPC file sharing program
  calling the `fetch` method in, 508
  calling the `query` method in, 508
  client interface, 515–516
  exceptions handling in, 516
  first implementation, 505–514
  implementing the `hello` method, 508
  second implementation, 514–522
  some flaws in the first implementation, 514–515
  starting the program, 517
  testing first implementation, 506
  testing the second implementation, 517–522
  validating file names in, 516–517
`xmlrpclib` module, connecting to XML-RPC servers with, 503–505
`xrange` function, 127
  using, 112
`xreadlines` object, lazy line iteration with, 316

Y
`yield` statement, 230–231

Z
zero (0), conversion specifier, 65
zero padding, of numbers in strings, 67
`ZeroDivisionError` class, built-in exception, 196
`zip` function, for zipping sequences together, 114