

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal featured on the cover of *Programming Python, Second Edition* is an African rock python, one of approximately 18 species of python. Pythons are nonvenomous constrictor snakes that live in tropical regions of Africa, Asia, Australia, and some Pacific Islands. Pythons live mainly on the ground, but they are also excellent swimmers and climbers. Both male and female pythons retain vestiges of their ancestral hind legs. The male python uses these vestiges, or spurs, when courting a female. The python kills its prey by suffocation. While the snake's sharp teeth grip and hold the prey in place, the python's long body coils around its victim's chest, constricting tighter each time it breathes out. They feed primarily on mammals and birds. Python attacks on humans are extremely rare.

Emily Quill was the production editor for *Programming Python, Second Edition*. Clairemarie Fisher O'Leary, Nicole Arigo, and Emily Quill copyedited the book. Matt Hutchinson, Colleen Gorman, Rachel Wheeler, Mary Sheehan, and Jane Ellin performed quality control reviews. Gabe Weiss, Lucy Muellner, Deborah Smith, Molly Shangraw, Matt Hutchinson, and Mary Sheehan provided production assistance. Nancy Crumpton wrote the index.

Edie Freedman designed the cover of this book. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with Quark™XPress 4.1 using Adobe's ITC Garamond font.

David Futato and Melanie Wang designed the interior layout, based on a series design by Nancy Priest. Cliff Dyer converted the files from Microsoft Word to FrameMaker 5.5.6, using tools created by Mike Sierra. The text and heading fonts are ITC Garamond Light and Garamond Book; the code font is Constant Willison. The illustrations that appear in the book were produced by Robert Romano using Macromedia FreeHand 8 and Adobe Photoshop 5. This colophon was written by Nicole Arigo.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal featured on the cover of *Programming Python, Second Edition* is an African rock python, one of approximately 18 species of python. Pythons are nonvenomous constrictor snakes that live in tropical regions of Africa, Asia, Australia, and some Pacific Islands. Pythons live mainly on the ground, but they are also excellent swimmers and climbers. Both male and female pythons retain vestiges of their ancestral hind legs. The male python uses these vestiges, or spurs, when courting a female. The python kills its prey by suffocation. While the snake's sharp teeth grip and hold the prey in place, the python's long body coils around its victim's chest, constricting tighter each time it breathes out. They feed primarily on mammals and birds. Python attacks on humans are extremely rare.

Emily Quill was the production editor for *Programming Python, Second Edition*. Clairemarie Fisher O'Leary, Nicole Arigo, and Emily Quill copyedited the book. Matt Hutchinson, Colleen Gorman, Rachel Wheeler, Mary Sheehan, and Jane Ellin performed quality control reviews. Gabe Weiss, Lucy Muellner, Deborah Smith, Molly Shangraw, Matt Hutchinson, and Mary Sheehan provided production assistance. Nancy Crumpton wrote the index.

Edie Freedman designed the cover of this book. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with Quark™XPress 4.1 using Adobe's ITC Garamond font.

David Futato and Melanie Wang designed the interior layout, based on a series design by Nancy Priest. Cliff Dyer converted the files from Microsoft Word to FrameMaker 5.5.6, using tools created by Mike Sierra. The text and heading fonts are ITC Garamond Light and Garamond Book; the code font is Constant Willison. The illustrations that appear in the book were produced by Robert Romano using Macromedia FreeHand 8 and Adobe Photoshop 5. This colophon was written by Nicole Arigo.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

Copyright © 2001 O'Reilly & Associates, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of an African rock python and the topic of Python programming is a trademark of O'Reilly & Associates, Inc.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Foreword

Less than five years ago, I wrote the Foreword for the first edition of Programming Python. Since then, the book has changed about as much as the language and the Python community! I no longer feel the need to defend Python: the statistics and developments listed in Mark's Preface speak for themselves.

In the past year, Python has made great strides. We released Python 2.0, a big step forward, with new standard library features such as Unicode and XML support, and several new syntactic constructs, including augmented assignment: you can now write `x += 1` instead of `x = x+1`. A few people wondered what the big deal was (answer: instead of `x`, imagine `dict[key]` or `list[index]`), but overall this was a big hit with those users who were already used to augmented assignment in other languages.

Less warm was the welcome for the extended print statement, `print>>file`, a shortcut for printing to a different file object than standard output. Personally, it's the Python 2.0 feature I use most frequently, but most people who opened their mouths about it found it an abomination. The discussion thread on the newsgroup berating this simple language extension was one of the longest ever-apart from the never-ending Python versus Perl thread.

Which brings me to the next topic. (No, not Python versus Perl. There are better places to pick a fight than a Foreword.) I mean the speed of Python's evolution, a topic dear to the heart of the author of this book. Every time I add a feature to Python, another patch of Mark's hair turns gray-there goes another chapter out of date! Especially the slew of new features added to Python 2.0, which appeared just as he was working on this second edition, made him worry: what if Python 2.1 added as many new things? The book would be out of date as soon as it was published!

Relax, Mark. Python will continue to evolve, but I promise that I won't remove things that are in active use! For example, there was a lot of worry about the string module. Now that string objects have methods, the string module is mostly redundant. I wish I could declare it obsolete (or deprecated) to encourage Python programmers to start using string methods instead. But given that a large majority of existing Python code-even many standard library modules-imports the string module, this change is obviously not going to happen overnight. The first likely opportunity to remove the string module will be when we introduce Python 3000; and even at that point, there will probably be a string module in the backwards compatibility library for use with old code.

Python 3000?! Yes, that's the nickname for the next generation of the Python interpreter. The name may be considered a pun on Windows 2000, or a reference to Mystery Science Theater 3000, a suitably Pythonesque TV show with a cult following. When will Python 3000 be released? Not for a loooooong time-although you won't quite have to wait until the year 3000.

Originally, Python 3000 was intended to be a complete rewrite and redesign of the language. It would allow me to make incompatible changes in order to fix problems with the language design that weren't solvable in a backwards compatible way. The current plan, however, is that the necessary changes will be introduced gradually into the current Python 2.x line of development, with a clear transition path that includes a period of backwards compatibility support.

Take, for example, integer division. In line with C, Python currently defines x/y with two integer arguments to have an integer result. In other words, $1/2$ yields 0! While most dyed-in-the-wool programmers expect this, it's a continuing source of confusion for newbies, who make up an ever-larger fraction of the (exponentially growing) Python user population. From a numerical perspective, it really makes more sense for the $/$ operator to yield the same value regardless of the type of the operands: after all, that's what all other numeric operators do. But we can't simply change Python so that $1/2$ yields 0.5, because (like removing the string module) it would break too much existing code. What to do?

The solution, too complex to describe here in detail, will have to span several Python releases, and involves gradually increasing pressure on Python programmers (first through documentation, then through deprecation warnings, and eventually through errors) to change their code. By the way, a framework for issuing warnings will be introduced as part of Python 2.1. Sorry, Mark!

So don't expect the announcement of the release of Python 3000 any time soon. Instead, one day you may find that you are already using Python 3000-only it won't be called that, but rather something like Python 2.8.7. And most of what you've learned in this book will still apply! Still, in the meantime, references to Python 3000 will abound; just know that this is intentionally vaporware in the purest sense of the word. Rather than worry about Python 3000, continue to use and learn more about the Python version that you do have.

I'd like to say a few words about Python's current development model. Until early 2000, there were hundreds of contributors to Python, but essentially all contributions had to go through my inbox. To propose a change to Python, you would mail me a context diff, which I would apply to my work version of Python, and if I liked it, I would check it into my CVS source tree. (CVS is a source code version management system, and the subject of several books.) Bug reports followed the same path, except I also ended up having to come up with the patch. Clearly, with the increasing number of contributions, my inbox became a bottleneck. What to do?

Fortunately, Python wasn't the only open source project with this problem, and a few smart people at VA Linux came up with a solution: SourceForge! This is a dynamic web site with a complete set of distributed project management tools available: a public CVS repository, mailing lists (using Mailman, a very popular Python application!), discussion forums, bug and patch managers, and a download area, all made available to any open source project for the asking.

We currently have a development group of 30 volunteers with SourceForge checkin privileges, and a development mailing list comprising twice as many folks. The privileged volunteers have all sworn their allegiance to the BDFL (Benevolent Dictator For Life-that's me :-). Introduction of major new features is regulated via a lightweight system of proposals and feedback called Python Enhancement Proposals (PEPs). Our PEP system proved so successful that it was copied almost verbatim by the Tcl community when they made a similar transition from Cathedral to Bazaar.

So, it is with confidence in Python's future that I give the floor to Mark Lutz. Excellent job, Mark. And to finish with my favorite Monty Python quote: Take it away, Eric, the orchestra leader!

Guido van Rossum
Reston, Virginia, January 2001

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[Active Scripting](#)

[Active Server Pages](#) [See ASP]

[ActiveState 2nd](#)

[ActiveX](#) [See COM]

adding

[C components to Python](#)

[frontend to FTP client 2nd](#)

[graphics to web pages](#)

[GUIs to command lines](#)

[input devices to HTML forms](#)

[relational algebra to sets](#)

[tables to web pages](#)

[tree interpreter to parsers](#)

[user interaction to CGI scripts](#)

administering

[databases](#)

[backing up](#)

[displaying](#)

[state changes](#)

[web sites](#)

[_zz](#) [See also Zope][See also Zope]

[AF_INET variable, socket module](#)

[after method](#)

[after__idle tools](#)

[animation techniques](#)

[anydbm module 2nd](#)

[shelve module and](#)

[Apache](#)

APIs (application programming interfaces)

[embedded-call Python](#)

[GC](#)

[object model](#)

[ppembed](#)

[code strings, running with](#)

[customizable validations, running](#)

[objects, running](#)

[Python C](#)

[documentation](#)

[vs. JPython](#)

[Python versionschanges](#)

[Python integration](#)

[Python Interpreter](#)

[running Python from Java](#)

[SQL](#)

[append\(\)](#)

- [lists](#)
- applets
 - [browser, coding](#)
 - [Grail](#)
 - [writing in JPython](#)
- [applications for Python](#)
- [apply\(\), call syntax used instead of argument lists](#)
- arrow option
 - [canvas widget](#)
- arrowshape option
 - [canvas widget](#)
- [ASCII module](#)
- [ASP \(Active Server Pages\)](#)
- [assert statement, added in v1.5](#)
- [assignment operators](#)
- [asynchat module](#)
- [asyncore module](#)
- attributes 2nd
 - [class](#)
 - [COM servers and](#)
- [automatic GUI construction](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[background activities](#)

[backups, database of comments/errata](#)

[base64 module 2nd](#)

[BaseHTTPServer module](#)

[BASIC, Python compared to](#)

[bastion module 2nd](#)

[BigGui example](#)

[binary](#)

[files](#)

[as email attachments](#)

[distinguishing from text files](#)

[downloading](#)

[module](#)

[operators](#)

[search trees](#)

[compared to dictionaries](#)

[BinaryTree class](#)

[binascii module](#)

[binhex module](#)

[bison system](#)

[browse module 2nd](#)

[browsers](#)

[Active Scripting support](#)

[applets, coding](#)

[email client](#)

[complexity of](#)

[deleting mail](#)

[forwarding mail](#)

[implementing](#)

[performance](#)

[portability](#)

[replying to mail](#)

[retrieving mail](#)

[root page](#)

[security protocols](#)

[selecting mail](#)

[sending mail](#)

[utility modules](#)

[viewing mail](#)

[examples in book, running on](#)

[HTML, languages embedded in](#)

[interoperability issues](#)

[JPython and 2nd](#)

[Python-based \[See Grail browser\]](#)

[restricted file access](#)

[server files, displaying on
surfing the Internet with
browsing](#)
[comment reports
implementing
interface for](#)
[errata reports
implementing
interface for](#)
[buttons](#)
[bytecodes, precompiling strings to](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[C modules](#)

[C types 2nd](#)

[wrapping in classes](#)

[zzz \[See also SWIG\]\[See also SWIG\]](#)

[compiling](#)

[string stacking](#)

[timing implementations](#)

[wrapping in classes](#)

[C# compiler](#)

[C/C++ \[See also extensions\]](#)

[API](#)

[documentation](#)

[vs. JPython](#)

[Python version changes](#)

[classes](#)

[embedding Python and](#)

[using in Python](#)

[wrapping C types in](#)

[data conversion](#)

[codes](#)

[embedding Python in](#)

[code strings, running](#)

[exceptions and](#)

[extensions](#)

[zz \[See also modules, C extension SWIG\]\[See also modules, C extension SWIG\]](#)

[components, adding](#)

[Python compared to](#)

[Python modules](#)

[JPython and](#)

[translating to](#)

[Python scripts integration 2nd](#)

[calculator, GUI 2nd](#)

[__call__ method, deprecated in v1.4](#)

[callable objects](#)

[callback handlers 2nd](#)

[CGI and](#)

[Grail](#)

[GUI, reloading](#)

[JPython and](#)

[registering, embedding Python code and](#)

[reload\(\) and](#)

[scheduled callbacks](#)

[callbacks, select\(\) and](#)

[canvas widgets](#)

[scrolling](#)

[CGI \(Common Gateway Interface\)](#)

- [HTML and module](#) [See [cgi module](#)]
- [Python versions](#)
- [PYTHONPATH, configuring scripts](#)
 - [as callback handlers](#)
 - [coding for maintainability](#)
 - [converting strings in debugging](#)
 - [email browser](#) [See [browser](#), [email client](#)]
 - [HTML and 2nd HTMLgen and installing](#)
 - [missing/invalid inputs, checking for Python and state information, saving web pages](#)
 - [web sites](#) [See [web sites](#), from [CGI scripts](#)]
 - [zz](#) [See also [Zope](#)][See also [Zope](#)]
- [user interaction, adding to cgi module 2nd](#)
 - [web pages, parsing user input](#)
- [cgi.escape\(\) 2nd](#)
- [cgi.FieldStorage\(\), form inputs, mocking up](#)
- [cgi.FieldStorage\(\)](#)
- [cgi.print__form\(\), debugging CGI scripts](#)
- [cgi.text\(\), debugging CGI scripts](#)
- [CGIHTTPServer module 2nd](#)
- [checkboxbuttons](#)
 - [configuring](#)
 - [dialogs](#)
 - [Entry widgets](#)
 - [Message widgets](#)
 - [variables and windows, top-level](#)
- [checkboxbuttons, adding to HTML forms](#)
- [child processes](#)
 - [exiting from](#)
 - [forking servers and](#)
- [circle\(\), clock widget](#)
- [__class__ attribute](#)
- [classes](#)
 - [as functions](#)
 - [of stored objects, changing application-specific tool set](#)
 - [assignment operators and C/C++](#) [See also [ppembed API](#)]
 - [embedding Python code and using in Python](#)

- [wrapping C types in database interface](#)
 - [DBM and form layout, for FTP client GUI](#)
 - [graphs to mixin](#)
 - [multiple clients, handling with pickled objects and set](#)
 - [shelves and stack](#)
 - [storage-specific subclasses](#)
 - [widget](#)
- [client function](#)
- [client/server architecture on the Web](#)
- [clients 2nd COM](#)
 - [using servers from connecting to closing establishing](#)
- [email](#)
 - [browser \[See browsers, email client\]](#)
 - [command-line](#)
 - [forwarding mail](#)
 - [implementing](#)
 - [interacting with](#)
 - [loading mail](#)
 - [replying to mail](#)
 - [sending mail](#)
 - [threading](#)
 - [viewing mail 2nd](#)
- [multiple, handling](#)
 - [with forking servers](#)
 - [with multiplexing servers](#)
 - [with threading servers](#)
 - [with classes](#)
- [path formats](#)
- [scripts](#)
 - [email](#)
 - [files, transferring over Internet](#)
 - [newsgroups](#)
 - [web sites, accessing](#)
 - [zzz \[See also CGI, scripts\]\[See also CGI, scripts\]](#)
- [sending files to](#)
- [socket calls](#)
- [socket programs, running locally](#)
- [spawning in parallel](#)

- [VB, using servers from](#)
- [cmath module](#)
- [code](#)
 - [GUI \[See GUIs\]](#)
 - [HTML, escaping](#)
 - [URLs embedded in](#)
 - [legacy, migration](#)
 - [maintainable](#)
 - [sharing objects between web pages](#)
 - [restricted execution mode, module](#)
 - [URLs, escaping](#)
- [code reuse](#)
 - [C extension modules and](#)
 - [data structures and](#)
 - [designing databases for](#)
 - [email clients and](#)
 - [browsers](#)
 - [form layout class](#)
 - [GUI calculator](#)
 - [web forms](#)
- [code strings, embedding Python code and 2nd](#)
 - [calling Python objects](#)
- [compiling](#)
 - [to bytcodes](#)
- [running](#)
 - [in dictionaries](#)
 - [with results and namespaces](#)
- [columns, summing](#)
- [COM \(Component Object Model\)](#)
 - [clients](#)
 - [early/late dispatch binding](#)
 - [using servers from](#)
- [extensions](#)
 - [distributed](#)
 - [installing](#)
 - [integration with Python](#)
 - [interfaces in Windows ports](#)
 - [servers](#)
 - [constraints](#)
 - [GUIDs](#)
- [combo function](#)
- [command lines](#)
 - [adding GUIs to](#)
- [email](#)
 - [client](#)
 - [sending from](#)
- [JPython](#)
 - [running Python programs from](#)
- [command option \(buttons\)](#)
- [comment reports, web site for](#)

- [browsing](#)
 - [implementation](#)
- [implementing](#)
- [submitting](#)
 - [implementation](#)
 - [interface for](#)
- [Common Gateway Interface](#) [See CGI]
- [Common Object Request Broker \(CORBA\)](#)
- [commonhtml module](#)
 - [email, viewing](#)
 - [state information in URL parameters, passing](#)
- [communication endpoints](#) [See sockets]
- [compiling](#)
 - [C extension](#)
 - [files](#)
 - [modules](#)
 - [code strings in embedded Python code](#)
- [Component Object Model](#) [See COM]
- [concurrent updates](#)
- [configuring](#)
 - [checkboxbuttons](#)
 - [email client](#)
 - [environment for Python](#)
 - [shell variables](#)
 - [PYTHONPATH, CGI scripts and](#)
 - [radiobuttons](#)
- [connections](#)
 - [client](#)
 - [closing](#)
 - [establishing](#)
 - [reserved ports and](#)
 - [database](#)
 - [server](#)
 - [closing 2nd](#)
 - [establishing](#)
 - [opening](#)
 - [POP](#)
- [connectivity](#)
 - [creating](#)
 - [Internet, Python and](#)
- [const modifier \(C++\)](#)
- [constants](#)
 - [raw strings](#)
- [constructors](#)
- [__contains__ method](#)
- [conventions, naming](#)
- [converting](#)
 - [end-of-lines for examples in book](#)
 - [objects to strings, pickled objects and](#)
 - [Python objects to/from C data types](#)

- [return values](#)
- [Python objects to/from C datatypes](#)
- [strings 2nd](#)
 - [in CGI scripts](#)
- [cookies 2nd](#)
- [CORBA \(Common Object Request Broker\)](#)
- [cPickle module 2nd](#)
- [create__filehandler tool](#)
- [creating](#)
 - [connectivity](#)
 - [objects](#)
 - [servers](#)
 - [with Python code](#)
 - [Apache](#)
 - [Mailman](#)
 - [tools for](#)
- [csh \(C shell\)](#)
- [customizing](#)
 - [by users, Python and](#)

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]
[[Y](#)] [[Z](#)]

[data conversion](#)

[codes](#)

[data storage, persistent](#)

[data types](#)

[Unicode](#)

[database cursor object](#)

[Database Management](#) [See DBM]

[database object](#)

[databases](#)

[of comments/errata](#)

[_zzz](#) [See also comment reports, web site for errata reports, web site for][See also
comment reports, web site for errata reports, web site for]

[concurrent updates](#)

[interfaces for](#)

[accessing reports](#)

[of comments/errata](#)

[administering](#)

[reusability](#)

[scalability](#)

[backup tools](#)

[display tools](#)

[state changes](#)

[connecting to](#)

[DBM](#) [See DBM]

[modules](#)

[Python and](#)

[servers](#)

[SQL](#)

[storage structures](#)

[flat-file 2nd](#)

[shelve](#) [See shelves]

[Unicode 3.0, access to](#)

[datatypes](#)

[binary trees](#) [See binary, search trees]

[C, converting to/from Python objects](#)

[objects and](#)

[sets](#)

[classes](#)

[functions](#)

[moving to dictionaries](#)

[relational algebra, adding to](#)

[stacks](#)

[as lists](#)

[optimizing](#)

[date/time formatting](#)

- [dbase\(\) 2nd](#)
- [dbhash module](#)
- [DBM \(Database Management\)](#)
 - [files 2nd](#)
 - [compability of](#)
 - [file operations](#)
 - [shelves and](#)
- [dbm module](#)
- [dbswitch module](#)
- [deadlocks](#)
 - [FTP and](#)
 - [pipes](#)
- [debugging](#)
 - [assert statement in v1.5](#)
 - [CGI scripts](#)
 - [using IDLE for](#)
- [deleting](#)
 - [email 2nd 3rd](#)
 - [from browser](#)
 - [files, when downloading web sites](#)
 - [objects](#)
- [delimiter.join\(\)](#)
 - [string.join\(\) and](#)
- [DHTML \(Dynamic Hypertext Markup Language\)](#)
- [dialogs](#)
 - [__dict__ attribute](#)
- [dictionaries 2nd](#)
 - [code strings, running in](#)
 - [sets as](#)
- [dictionary methods](#)
- [Digital Creations](#)
- [dir\(\), objects and](#)
- [directories](#)
 - [CGI scripts and](#)
 - [Examples](#)
 - [tools](#)
 - [walking](#)
 - [one directory](#)
- [Dispatch\(\)](#)
- [distutils module](#)
- [documentation](#)
 - [Python](#)
 - [Python C API](#)
- [domain names](#)
- [dump\(\), pickled objects and](#)
- [Dynamic Hypertext Markup Language \(DHTML\)](#)

[[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]
[[Y](#)] [[Z](#)]

[email](#)

- [attachments, decoding](#)

 - [base64 data](#)

- [client](#)

 - [browser as](#) [See browsers, email client]

 - [code reuse](#)

 - [command-line](#)

 - [configuration module](#)

 - [forwarding mail](#)

 - [help text](#)

 - [implementing](#)

 - [interacting with](#)

 - [loading mail](#)

 - [replying to mail](#)

 - [sending mail](#)

 - [status messages](#)

 - [threading](#)

 - [viewing mail 2nd](#)

- [deleting 2nd 3rd](#)

 - [from browser](#)

- [forwarding](#)

 - [from browser](#)

- [loading 2nd 3rd 4th](#)

- [mailboxes](#)

 - [accessing](#)

 - [encapsulating fetches](#)

 - [modules](#)

 - [unlocking](#)

- [Mailman](#)

- [modules 2nd](#)

 - [headers](#)

- [parsing](#)

- [passwords](#)

 - [escaping in HTML](#)

 - [POP page 2nd](#)

- [POP message numbers](#)

- [reading](#)

 - [from browser](#)

- [replying to](#)

 - [from browser](#)

- [saving 2nd](#)

- [selecting from browser](#)

- [sending 2nd](#)

 - [from browser](#)

 - [comment/errata reports](#)

- [text, escaping in HTML](#)
- [viewing](#)
 - [from browser](#)
- [viruses](#)
- [embedding Python code](#) [2nd](#) [3rd](#)
- [code strings](#)
 - [compiling](#)
 - [running in dictionaries](#)
 - [running with objects](#)
 - [running with results and namespaces](#)
- [precompiling strings to bytecode](#)
- [registering callback handler objects](#)
- [using Python classes in C](#)
 - [ppembed API](#)
- [encapsulation](#)
- [encrypting](#)
 - [passwords](#)
 - [Python tools for](#)
- [end-of-line characters](#)
 - [CGI scripts and](#)
 - [converting, for examples in book](#)
- [endpoints, communication](#) [See sockets]
- [environment, configuring for Python](#)
 - [shell variables](#)
- [errata reports, web site for](#)
 - [browsing](#)
 - [implementation](#)
 - [directories on](#)
 - [implementing](#)
 - [root page](#)
 - [submitting](#)
 - [implementation](#)
 - [interface for](#)
- [eval\(\)](#)
 - [input expressions](#)
 - [security and](#)
- [examples in book](#)
 - [converting end-of-lines](#)
 - [Internet](#)
 - [launching](#)
 - [PYTHONPATH and](#)
 - [security](#)
 - [server-side scripts](#)
 - [changing](#)
 - [running](#)
 - [viewing](#)
 - [source code distribution](#)
 - [updates to](#)
- [exceptions](#)
 - [C and](#)

- [CGI scripts and class](#)
- [local name oddities](#)
- [sockets and exec statement](#)
- [input expressions](#)
- [JPython](#)
- [security and execfile\(\), JPython](#)
- [Expat XML parser](#)
- [extend\(\)](#)
- [Extensible Markup Language](#) [See XML]
- [extensions 2nd](#)
 - [for COM, installing](#)
 - [C types 2nd](#)
 - [compiling](#)
 - [string stacking](#)
 - [timing implementations](#)
 - [wrapping in classes](#)
 - [C/C++ 2nd](#)
 - [components, adding](#)
 - [zz](#) [See also modules, C extension SWIG][See also modules, C extension SWIG]
- [NumPy](#)
- [Python interface](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[fastset](#) [See sets]

[fcntl module 2nd](#)

[fcntl.flock\(\)](#)

[mutexes](#)

[fifos](#)

[File Transfer Protocol](#) [See FTP]

[filenames](#)

[CGI scripts and](#)

[conventions, accessing databases and](#)

[DOS](#)

[rewriting](#)

[files](#)

[from web server, displaying on client](#)

[binary](#)

[distinguishing from text files](#)

[downloading](#)

[as email attachments](#)

[module](#)

[C header, searching](#)

[classes for, storing](#)

[client, uploading to web server](#)

[DBM 2nd](#)

[compatibility of](#)

[shelves and](#)

[deleting, when downloading web sites](#)

[downloading 2nd](#)

[frontend for, adding](#)

[GDBM](#)

[header, Python](#)

[HTML, permissions](#)

[libpython1.5.a](#)

[locking](#)

[databases](#)

[exclusiveAction\(\)](#)

[sharedAction\(\)](#)

[shelves](#)

[Python header](#)

[reading from](#)

[remote](#)

[deleting](#)

[retrieving](#)

[restricted, accessing on browsers](#)

[shelve](#) [See shelves]

[storing](#)

[text](#)

- [distinguishing from binary](#)
 - [uploading](#)
- [transferring over Internet](#)
 - [to clients and servers](#)
 - [downloading](#)
 - [frontend for](#)
 - [uploading](#)
 - [using various means](#)
 - [with urllib](#)
- [float\(\)](#)
- [flushes](#)
 - [pipes](#)
 - [streams](#)
- [forking servers](#)
 - [child processes, exiting from](#)
 - [zombies](#)
 - [killing](#)
 - [preventing](#)
- [Form class \(HTMLgen\)](#)
- [forms](#)
 - [numbers in, security and](#)
 - [web](#)
 - [adding input devices to](#)
 - [changing](#)
 - [hidden fields in](#)
 - [inputs, checking for missing/invalid](#)
 - [inputs, mocking up](#)
 - [laying out with tables](#)
 - [reusable](#)
 - [selection list on](#)
 - [tags](#)
 - [Zope and](#)
- [frame widgets](#)
- [frames, inheritance and](#)
- [freeze tool](#)
- [FTP \(File Transfer Protocol\)](#)
 - [deadlock and](#)
 - [files, transferring over Internet](#)
 - [downloading](#)
 - [mirroring web sites](#)
 - [uploading](#)
 - [with urllib](#)
- [ftp object](#)
 - [quit\(\)](#)
 - [retrbinary\(\) 2nd](#)
 - [storlines\(\)](#)
- [ftplib module 2nd](#)
- [functions](#)
 - [as published objects](#)
- [C](#)

[SWIG and
comparison
initialization
method
overloading
re module
sets
multiple operands, supporting
timing](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[garbage collection 2nd](#)

[reference count management](#)

[GC API](#)

[gc module](#)

[gdbm module](#)

[generate\(\)](#)

[geometry manager](#)

[grid](#)

[widget.grid\(\)](#)

[geometry managers](#)

[pack](#)

[alternatives to](#)

[columns, spanning](#)

[layout system](#)

[rows, spanning](#)

[widgets, resizing 2nd 3rd 4th 5th 6th](#)

[get\(\)](#)

[getfile\(\), FTP](#)

[__getitem__ method](#)

[getpass.getpass\(\), FTP](#)

[GIF \(Graphics Interchange Format\) images, displaying on web pages with HTMLgen](#)

[global interpreter lock](#)

[globally unique identifier \(GUID\)](#)

[gopherlib module](#)

[Grail browser 2nd](#)

[graphics](#)

[on web pages](#)

[adding](#)

[moving](#)

[painting](#)

[slideshow program](#)

[on web pages](#)

[displaying with HTMLgen](#)

[Graphics Interchange Format \[See GIF\]](#)

[graphs](#)

[moving to classes](#)

[searching](#)

[grep utility](#)

[grid geometry manager](#)

[FTP client frontend](#)

[widget.grid\(\)](#)

[grids](#)

[GUID \(globally unique identifier\)](#)

[GuiInput class](#)

[GuiMaker class](#)

- menus
- testing
- toolbars
- GuiMixin class
- GuiOutput class 2nd
- GUIs (graphical user interfaces)
 - analog/digital clock widget
 - animation techniques
 - automating
 - calculator 2nd
 - callback handlers, reloading
 - canvas widgets
 - checkboxbuttons
 - adding to HTML forms
 - configuring
 - dialogs
 - Entry widgets
 - Message widgets
 - variables and
 - windows, top-level
 - command lines, adding to
 - comment/errata database
 - browsing reports
 - concurrent updates
 - submitting reports 2nd
- email client
 - browser as
 - deleting mail
 - forwarding mail
 - implementing
 - loading mail
 - replying to mail
 - saving mail
 - sending mail
 - threading
 - viewing mail
- frame widgets
- FTP client frontend
- Grail and
- grids
- GUI code, running
- inheritance
- JPython, interface automation
- listboxes
- menus 2nd
 - GuiMaker class
- object viewers
 - persistent
- paint program
- radiobuttons

- [adding to HTML forms](#)
- [configuring](#)
- [dialogs](#)
- [Entry widgets](#)
- [Message widgets](#)
- [variables and](#)
- [windows, top-level](#)
- [scrollbars](#)
- [sliders](#)
 - [variables and](#)
- [slideshow program](#)
- [streams to widgets, redirecting](#)
- [text](#)
 - [editing](#)
- [threads and](#)
- [Tic-Tac-Toe game 2nd](#)
- [toolbars](#)
 - [GuiMaker class](#)
- [GuiStreams example](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[handleClient\(\)](#)

[threading](#)

[header files](#)

[searching for patterns](#)

[history of Python](#)

[holmes system](#)

[HTML \(Hypertext Markup Language\)](#)

[browsing](#)

[database reports](#)

[email, complexity of](#)

[CGI scripts and 2nd 3rd](#)

[embedding in](#)

[document templates, Zope and](#)

[email passwords](#)

[escaping](#)

[escaping](#)

[conventions](#)

[email passwords](#)

[embedded URLs](#)

[text](#)

[file permissions](#)

[forms and](#)

[Grail and](#)

[hidden input fields, passing state information in](#)

[HTMLgen and](#)

[hyperlinks \[See hyperlinks\]](#)

[JavaScript embedded in](#)

[library of tools](#)

[module \[See htmllib module\]](#)

[passwords](#)

[tags](#)

[_input](#)

[, Grail and](#)

[forms](#)

[HTMLgen and](#)

[tables 2nd](#)

[VBScript embedded in](#)

[web pages and](#)

[HTMLgen 2nd](#)

[GIF images, displaying](#)

[hyperlinks and](#)

[PYTHONPATH setting](#)

[HTMLgen module](#)

[htmllib module 2nd](#)

[HTTP \(Hypertext Transfer Protocol\)](#)

- [CGI scripts and cookies 2nd module](#) [See [httplib module](#)]
- [requests, Zope and servers](#)
 - [CGI scripts and Python implementations](#)
- [httplib module 2nd 3rd hybrid systems](#)
- [hyperlinks](#)
 - [CGI and scripts](#)
 - [state information](#)
 - [comment/errata reports](#)
 - [encrypted passwords in](#)
 - [escaping URLs and HTML and HTMLgen and smart links](#)
- [URLs](#)
 - [embedded in 2nd embedded in, updating syntax](#)
- [Zope and Hypertext Markup Language](#) [See [HTML](#)]
- [Hypertext Transfer Protocol](#) [See [HTTP](#)]

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[IANA \(Internet Assigned Numbers Authority\)](#)

[Icon, Python compared to
identifiers for machines on Internet](#)

[IDEs, running Python from](#)

[IDLE](#)

[Python versions 2nd](#)

[running Python from](#)

[IETF \(Internet Engineering Task Force\)](#)

[images \[See graphics\]](#)

[IMAP \(Internet Message Access Protocol\) module](#)

[imaplib module](#)

[import statements 2nd](#)

[importing Python programs](#)

[IndexError exception](#)

[inheritance 2nd](#)

[frames and](#)

[virtual](#)

[widgets and](#)

[__init__ method](#)

[initialization function](#)

[input/output, redirection](#)

[zzz \[See also redirection, streams to widgets\]\[See also redirection, streams to widgets\]](#)

[installing](#)

[CGI scripts](#)

[automating](#)

[Python](#)

[on Linux](#)

[on Macintosh](#)

[on Unix](#)

[on Windows 2nd](#)

[Tcl/Tk](#)

[on Windows](#)

[int\(\) 2nd](#)

[integration 2nd 3rd 4th](#)

[vs. optimization](#)

[zz \[See also extensions, C/C++ embedding Python code\]\[See also extensions, C/C++](#)

[embedding Python code\]](#)

[COM and Python](#)

[CORBA and Python](#)

[examples](#)

[JPython and](#)

[limitations of](#)

[v2.0 and](#)

[interfaces \[See GUIs\]](#)

[internationalizing](#)

Internet

[addresses](#) [See URLs]

[applications](#)

[client/server](#) [See client/server architecture]

[clients](#) [See clients]

[files, transferring over](#)
[to clients and servers](#)

[downloading](#)

[frontend for](#)

[uploading](#)

[various means of](#)
[with urllib](#)

[identifiers for machines connected to](#)

[machine names](#)

[message formats](#)

[modules](#)

[port numbers](#) [See port numbers]

[protocols 2nd](#)

[message formats](#)

[modules](#)

[structures](#)

[resources for Python](#)

[running Python over](#)

[scripting](#)

[zzz](#) [See also clients, scripts servers, scripts][See also clients, scripts servers, scripts]

[servers](#) [See servers]

[Internet Assigned Numbers Authority \(IANA\)](#)

[Internet Engineering Task Force \(IETF\)](#)

[Internet Explorer](#)

[HTML](#)

[JavaScript embedded in](#)

[VBScript embedded in](#)

[registering Python with](#)

[Internet Message Access Protocol \(IMAP\) module](#)

[Internet Protocol \(IP\)](#)

[Internet Service Provider](#) [See ISP]

[IP \(Internet Protocol\)](#)

[IP addresses](#)

[socket module](#)

[ISPs \(Internet Service Providers\)](#)

[Python-friendly](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)
[\[X\]](#) [\[Y\]](#) [\[Z\]](#)

Java

- [applets](#) [See applets]
- classes
 - [accessing with JPython](#)
 - [testing with JPython](#)
- [end-user customization](#)
- [libraries, JPython and Python and](#)
- [zzz](#) [See also JPython][See also JPython]
- [Java virtual machine](#) [See JVM]
- [JPython 2nd 3rd](#)
 - [vs. Python C API](#)
 - [API](#)
 - [applets, writing](#)
 - [browsers and](#)
 - [callback handlers](#)
 - [command lines](#)
 - [compatibility with Python 2nd](#)
 - [components](#)
 - [integration and](#)
 - [interface automations 2nd](#)
- Java classes
 - [accessing](#)
 - [testing](#)
- [Java libraries and](#)
- [object model](#)
- [performance issues](#)
- [Python-to-Java compiler](#)
- [PYTHONPATH](#)
- [scripts, compared to Java](#)
- [trade-offs in using](#)

[jpython program](#)
[jpythone program](#)

JVM (Java virtual machine)

- [JPython and](#)
- [Python scripts executed by](#)

[lython](#) [See JPython]

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[kwParsing system](#)

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[lambda statement](#)

[LEFT constant](#)

[libpython1.5.a file](#)

[libraries](#)

[Java](#)

[JPython](#)

[libpython1.5.a](#)

[Python version changes 2nd 3rd](#)

[linking, C extension modules](#)

[static vs. dynamic binding](#)

[Linux](#)

[C extension modules](#)

[compiling](#)

[wrapping environment calls](#)

[end-of-lines, CGI scripts and](#)

[portability in v2.0](#)

[Python programs, launching](#)

[Python, installing on](#)

[servers](#)

[killing processes on](#)

[web, finding Python on](#)

[showinfo\(\)](#)

[SWIG on](#)

[Lisp, Python compared to](#)

[listboxes](#)

[adding to HTML forms](#)

[lists](#)

[Python version changes 2nd 3rd](#)

[stacks as](#)

[load\(\), pickled objects and](#)

[loading Python programs](#)

[loadmail module](#)

[email, viewing](#)

[POP mail interface](#)

[loadmail.loadmail\(\)](#)

[long\(\)](#)

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[machine names](#)

[Macintosh, installing Python on](#)

[mailbox module](#)

[mailconfig module](#)

[Mailman program](#)

[__main__ module](#)

[mainloop\(\)](#)

[recursive calls](#)

[makefile\(\), sockets and](#)

[makeKey\(\)](#)

[map function](#)

[map\(\)](#)

[list\(\) and](#)

[zip\(\) and](#)

[mapping Tk/Tkinter](#)

[marshal module](#)

[math module](#)

[Medusa](#)

[memory](#)

[threads and](#)

[menus 2nd](#)

[GuiMaker class 2nd](#)

[message function](#)

[messages](#)

[error, CGI scripts and](#)

[POP message numbers](#)

[status, email](#)

[metaclass protocols](#)

[method functions](#)

[method pointers](#)

[methods](#)

[as class attributes](#)

[dictionary](#)

[list](#)

[registering](#)

[Set class](#)

[socket](#)

[stack module/Stack class](#)

[string objects](#)

[virtual](#)

[mllib module 2nd](#)

[MIME \(Multimedia Internet Mail Extensions\)](#)

[module](#)

[mimetools module](#)

[mimify module](#)

- [mirroring web sites](#)
- [mixin classes](#)
- [mod__python package](#)
- [modules](#)
 - [archives](#)
 - [arithmetic operations](#)
 - [binary data, encoding](#)
 - [C extension](#)
 - [code reuse and](#)
 - [compiling](#)
 - [linking](#)
 - [string stacking](#)
 - [structure of](#)
 - [wrapping environment calls](#)
- [CGI scripts](#)
- [code execution](#)
- [creating servers](#)
 - [with Medusa](#)
 - [with Zope](#)
- [data, encoding](#)
- [distributing](#)
- [email 2nd 3rd](#)
 - [attachments, decoding](#)
 - [client configuration](#)
- [expression operators, implementing](#)
- [FTP \[See ftplib module\]](#)
- [Gopher](#)
- [HTML \[See htmllib module\]](#)
- [HTMLgen \[See HTMLgen module\]](#)
- [HTTP \[See httplib module\]](#)
- [IMAP](#)
- [Internet](#)
- [MIME](#)
- [network communications](#)
- [NNTP \[See nntplib module\]](#)
- [pickled objects and](#)
- [POP \[See poplib module\]](#)
- [Python extensions in C/C++, JPython and](#)
- [regular expressions 2nd](#)
- [servers](#)
- [SGML](#)
- [SMTP \[See smtp lib module\]](#)
- [stacks as](#)
- [string, as object methods](#)
- [Telnet](#)
- [Tk v8.08.3, support for](#)
- [translating to C](#)
- [URLs](#)
- [utility](#)
 - [for browsing comment/errata reports](#)

[for email browser](#)
[for submitting comment/errata reports](#)
[text-processing](#)
[web pages 2nd](#)
[Windows Registry access](#)
[XML](#)
[mouseclicks, running Python programs from](#)
[multifile module](#)
[Multimedia Internet Mail Extensions module](#)
[multiplexing servers, with select\(\)](#)
[mutex class](#)
[mutextcntl module](#)

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[named pipes \(fifos\)](#)

[namespaces, running code strings with](#)

[naming conventions](#)

[nested frame widgets](#)

[Netscape, Active Scripting and](#)

[Network News Transfer Protocol \(NNTP\)](#)

[newsgroups, accessing](#)

[ni module, replaced between v1.3 and v1.5.2](#)

[NNTP \(Network News Transfer Protocol\)](#)

[nntplib module 2nd](#)

[numbers](#)

[converting strings from](#)

[NumPy](#)

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[Object Linking and Embedding \(OLE\)](#)

[object model API](#)

[Object Request Broker](#) [See ORB]

[object viewers](#)

[persistent](#)

[object-oriented databases \(OODBs\)](#)

[object-oriented programming \(OOP\), Python and](#)

[objects](#)

[assignment operators and](#)

[attributes and](#)

[callable, embedded Python code and](#)

[callback handler, registering](#)

[class](#)

[code strings, running with](#)

[converting](#)

[to/from C datatypes](#)

[to strings, pickled objects and](#)

[creating](#)

[cyclic](#)

[references between](#)

[database connection](#)

[database cursor](#)

[databases and](#)

[flat-file](#)

[shelve](#)

[datatypes and](#)

[DBM file](#)

[deleting](#)

[dir\(\) and](#)

[files, grep utility](#)

[ftp](#)

[quit\(\)](#)

[retrbinary\(\) 2nd](#)

[storlines\(\)](#)

[graphs](#)

[HTMLgen](#)

[JPython](#)

[mapping URLs into calls on](#)

[persistent, shelves and](#)

[pickled](#)

[zzz](#) [See also shelves][See also shelves]

[publishing](#)

[COM and](#)

[sequences](#)

[permutations of](#)

- [reversing](#) 2nd
 - [sorting](#) 2nd
- [sharing between web pages](#)
- socket
 - [accept\(\)](#)
 - [bind\(\)](#)
 - [close\(\)](#) 2nd
 - [connect\(\)](#)
 - [listen\(\)](#)
 - [recv\(\)](#) 2nd
 - [send\(\)](#) 2nd
 - [setblocking\(\)](#)
 - [socket\(\)](#) 2nd
- stacks
- [stored, changing classes of](#)
- [string, methods of](#)
- [Toplevel](#)
- Zope
 - [as database for storing](#)
 - [ORB and](#)
- [OLE \(Object Linking and Embedding\)](#)
- [OODBs \(object-oriented databases\)](#)
- [OOP \(object-oriented programming\), Python and](#)
- [open source software, compared to commercial](#)
- [OpenSSL](#)
 - [Python support for](#) 2nd
- [operator module](#)
- operators
 - [binary](#)
 - [in, overriding](#)
 - [overloading](#)
 - [scope](#)
 - [stack versus module](#)
- [optimization](#) [See also [performance](#)]
 - [vs. integration](#)
 - [C extension files](#)
 - [sets](#) 2nd
 - [stacks](#)
- ORB (Object Request Broker)
 - [Zope](#) 2nd
 - [URLs, mapping into calls on Python objects](#)
- [os module, security and](#)
- [os.__exit\(\), forking servers and](#)
- [os.chmod\(\), uploading client files to web servers](#)
- os.environ
 - [configuring Python environment](#)
- [os.execl\(\), os.fork\(\) and](#)
- os.fork()
 - [os.execl\(\) and](#)
 - [servers, forking](#)

- [os.listdir\(\)](#)
- [os.path.isdir\(\)](#)
- [os.path.join\(\)](#)
- [os.path.samefile\(\), restricted file access](#)
- [os.path.split\(\)](#)
- [os.path.walk\(\)](#)
 - [directories, renaming](#)
 - [directory trees 2nd 3rd 4th 5th 6th](#)
 - [DOS filenames, rewriting](#)
 - [files, renaming](#)
 - [web sites, moving](#)
- [os.popen\(\)](#)
 - [comment/errata reports, submitting](#)
 - [directory trees](#)
 - [email, sending](#)
 - [Linux](#)
 - [os.listdir\(\) and](#)
 - [shell commands 2nd](#)
 - [streams](#)
 - [redirection](#)
- [os.popen2\(\) 2nd](#)
- [os.popen3\(\) 2nd](#)
- [os.putenv\(\)](#)
- [os.system\(\)](#)
 - [FTP client frontend](#)
 - [security and](#)
- [os.waitpid\(\), zombies, killing](#)
- [overloading](#)
 - [functions](#)
 - [operators](#)

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[pack geometry manager](#)

[alternatives to](#)

[columns, spanning](#)

[layout system](#)

[rows, spanning](#)

[widgets, resizing 2nd](#)

[anchor option](#)

[clipping](#)

[expansion](#)

[pack\(\)](#)

[clock widget](#)

[pack__forget, clock widget](#)

[packages, importing](#)

[PackDialog class](#)

[packer, FTP client frontend and](#)

[parent processes, forking servers and](#)

[parse tree interpreter](#)

[adding to parsers](#)

[exploring](#)

[structure](#)

[parsing rule strings](#)

[passwords](#)

[in hyperlinks, encrypted](#)

[on POP page 2nd](#)

[email client and](#)

[encrypting](#)

[utility module](#)

[escaping in HTML](#)

[PATH](#)

[CGI scripts and](#)

[performance 2nd](#)

[C extension files](#)

[email browser](#)

[forking and](#)

[HTMLgen and](#)

[CGI and](#)

[JPython](#)

[Python](#)

[profiler](#)

[scripts integrated with C/C++](#)

[reading from files](#)

[sets](#)

[stacks](#)

[threads and](#)

[Perl, Python compared to 2nd](#)

- permissions
 - [Administrator](#)
 - [CGI scripts and HTML files](#)
- [permute\(\)](#)
- [persistence](#)
 - [_zzz \[See also databases\]\[See also databases\]](#)
 - [object viewer](#)
 - [pickled objects](#)
 - [Python and](#)
- [Persistent class](#)
- [pickle module 2nd](#)
 - [shelve module and](#)
- [zz \[See also cPickle module\]\[See also cPickle module\]](#)
- [pickle.dump\(\)](#)
- [pickle.dumps\(\)](#)
- [pickle.load\(\)](#)
- [pickle.loads\(\)](#)
- [pickle.Pickler\(\)](#)
- [PickleDictionary class](#)
- [Pickler class](#)
 - [constraints](#)
- [PIL](#)
- [ping command](#)
- [pipes](#)
 - [anonymous](#)
 - [bidirectional IPC with fifos, named pipes](#)
- [platforms, installing Python on various](#)
- [PMW \(Python Mega Widgets\)](#)
- [point\(\), clock widget](#)
- [POP \(Post Office Protocol\)](#)
 - [mail interface, utility modules](#)
 - [message numbers](#)
 - [module \[See poplib module\]](#)
 - [passwords 2nd](#)
 - [encrypting](#)
 - [retrieving email 2nd](#)
 - [from browser](#)
 - [servers, connecting to](#)
- [pop\(\)](#)
- [popen4\(\)](#)
- [poplib module 2nd](#)
 - [email client](#)
- [poplib.POP3\(\)](#)
- [port numbers](#)
 - [clients](#)
 - [reserved 2nd](#)
 - [client connections and talking to](#)

- [servers](#)
- portability
- email
 - [browser](#)
 - [clients and sending](#)
 - [forking and JPython](#)
 - [Python versions](#)
 - [v2.0](#)
 - [select\(\) and signal handlers and threading and threads and](#)
- Post Office Protocol [See POP]
- ppembed API
 - [running code strings with](#)
 - [running customizable validations](#)
 - [running objects with](#)
- [print statement](#)
- [privacy constraints](#)
- [private members](#)
- processes
 - child
 - [exiting from forking servers and parent, forking servers and](#)
 - zombies
 - [killing](#)
 - [preventing](#)
- processing
 - [hand-coded parsers](#)
 - [expression grammar](#)
 - [tree interpreter, adding](#)
 - [parser generators](#)
 - [rule strings](#)
 - [text](#)
 - [summing columns](#)
- programming
 - [mathematical operations](#)
 - [sockets](#)
 - [client calls](#)
 - [server calls](#)
- programs
 - [analog/digital clock widget](#)
 - [GUI \[See GUIs\]](#)
 - [importing/reloading](#)
 - [paint](#)
 - running from
 - [command line](#)

- [mouseclicks](#)
- [slideshow](#)
- [socket](#)
 - [running locally](#)
 - [running remotely](#)
- [text editor](#)
- [Tic-Tac-Toe game 2nd](#)
- [protocols](#)
 - [Internet 2nd 3rd 4th](#)
 - [message formats](#)
 - [modules](#)
 - [structures of](#)
 - [standards](#)
 - [prototyping, Python and](#)
 - [PSA \(Python Software Activity\)](#)
 - [PSP \(Python Server Pages\)](#)
 - [Py_BuildValue\(\) 2nd](#)
 - [Py_CompileString\(\)](#)
 - [Py_DECREF\(\)](#)
 - [Py_INCREF\(\)](#)
 - [Py_Initialize\(\)](#)
 - [Py_XDECREF\(\)/Py_XINCREASE\(\)](#)
 - [PyApache](#)
 - [PyArg_Parse\(\) 2nd 3rd](#)
 - [PyArg_ParseTuple\(\)](#)
 - [PyCalc example](#)
 - [components](#)
 - [adding buttons to](#)
 - [using PyCalc as](#)
 - [source code](#)
 - [PyClock example](#)
 - [source code](#)
 - [PyDict_GetItemString\(\)](#)
 - [PyDict_New\(\)](#)
 - [PyDict_SetItemString\(\)](#)
 - [PyDraw example](#)
 - [source code](#)
 - [PyEdit example](#)
 - [source code](#)
 - [PyErr_SetString\(\)](#)
 - [PyErrata example](#)
 - [browsing reports](#)
 - [database interfaces](#)
 - [implementing](#)
 - [root page](#)
 - [submitting reports](#)
 - [PyEval_CallObject\(\)](#)
 - [PyEval_EvalCode\(\)](#)
 - [PyEval_GetBuiltins\(\)](#)
 - [pyexpat module](#)

[PyForm example](#)

[GUI code](#)

[limitations](#)

[PyFtp](#)

[PyFtpGui example](#)

[PyImport__ImportModule\(\) 2nd](#)

[PyMailCGI example](#)

[root page](#)

[PyMailGui example](#)

[implementing](#)

[interacting with](#)

[PyModule__GetDict\(\)](#)

[PyObject type](#)

[PyObject__GetAttrString\(\)](#)

[PyObject__SetAttrString\(\)](#)

[PyRun__SimpleString\(\)](#)

[PyRun__String\(\) 2nd](#)

[Python](#)

[as executable pseudocode](#)

[applets \[See applets Grail browser\]](#)

[C# compiler](#)

[C/C++ and](#)

[comparing](#)

[integration](#)

[CGI scripts and](#)

[class model](#)

[compared to other languages 2nd](#)

[compatibility with JPython 2nd](#)

[databases and](#)

[DBM \[See DBM\]](#)

[development cycle](#)

[documentation](#)

[embedded-call API](#)

[embedding \[See embedding Python code\]](#)

[encryption tools](#)

[features of 2nd](#)

[freeze tool](#)

[history of](#)

[installing](#)

[configuring environment for](#)

[on Windows](#)

[integrating with CORBA](#)

[Internet connections and](#)

[Internet uses for](#)

[interpreter](#)

[Java and](#)

[OOP and](#)

[OpenSSL](#)

[persistent data and](#)

[profiler, performance and](#)

- programs
 - [importing](#)
 - [loading](#)
 - [running from command line](#)
 - [running from IDEs](#)
 - [running from mouseclicks](#)
 - [running from Pythonwin](#)
 - [registering with Internet Explorer](#)
 - [restricted execution mode](#)
 - [rexec module](#)
 - [Unix](#)
 - [Windows](#)
 - [scripts run on startup](#)
 - [versions](#)
 - [changes in](#)
 - [web servers, finding on](#)
 - [XML support](#)
- Python C API
 - [documentation](#)
 - [Python version changes](#)
- [Python Mega Widgets \(PMW\)](#)
- [Python Server Pages \(PSP\)](#)
- [Python Software Activity \(PSA\)](#)
- [Python.h header file 2nd](#)
- [pythoncom.CreateGuid\(\)](#)
- [PYTHONHOME setting](#)
- [PythonInterpreter API](#)
- [PythonInterpreter class \(JPython\)](#)
- [PYTHONPATH 2nd](#)
 - [C extension module](#)
 - [CGI scripts and examples in book and HTMLgen and JPython](#)
 - [Pickler class and Python version changes](#)
 - [Windows](#)
- [PYTHONSTARTUP](#)
- [PythonWare](#)
- [PythonWin IDE](#)
- [Pythonwin, running Python from](#)
- [PythonWorks, running Python from](#)
- [PyToe example](#)
 - [source code](#)
- [PyTree example](#)
 - [parse trees](#)
 - [source code 2nd](#)
- [PyView example](#)
 - [source code](#)

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[quopri module](#)

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[radiobuttons](#)

- [configuring](#)
- [dialogs](#)
- [Entry widgets](#)
- [HTML forms, adding to](#)
- [Message widgets](#)
- [variables and](#)
- [windows, top-level](#)

[raise statement](#)

[rand module, replaced by random module](#)

[random module](#)

[raw__input\(\)](#)

[re module 2nd 3rd](#)

- [functions](#)

[re.compile\(\) 2nd](#)

[re.match\(\)](#)

[re.search\(\)](#)

[re.split\(\)](#)

[re.sub\(\)](#)

[re.subn\(\)](#)

[read\(\), Unpickler class](#)

[readlines\(\)](#)

[reapChildren\(\)](#)

[redirectedGuiFunc\(\)](#)

[redirection](#)

- [streams to widgets](#)
- [using for packing scripts](#)

[ref parameters \(C++\)](#)

[reference count management](#)

[regex module](#)

- [migrating code using](#)

[Register__Handler\(\)](#)

[registering methods](#)

[Registry \[See Windows, Registry access\]](#)

[regression test script](#)

[regular expressions 2nd](#)

- [vs. string module](#)
- [compiled pattern objects](#)
- [match objects](#)
- [patterns](#)
- [re module](#)
- [SRE engine, compatibility of](#)

[relational algebra, adding to sets](#)

[reload\(\)](#)

- [callback handlers and](#)

[repr\(\)](#)
[reserved words](#)
[resource module](#)
[resources for Python, Internet](#)
[return values](#)
[rexec module 2nd](#)
[rfc module, getaddr\(\)](#)
[rfc822 module](#)
 [bug in](#)
 [email client](#)
[RIGHT constant](#)
[rotor module](#)
[Route__Event\(\)](#)
[rule strings](#)
[runtime type information](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[sandbox model](#)

scalability

- [designing databases for multiple shelfe field indexing](#)

[scheduled callbacks](#)

[Scheme, Python compared to](#)

[scope operators](#)

scripts

- [CGI](#) [\[See CGI, scripts\]](#)

- [client-side](#)

 - [email](#) [\[See email, client\]](#)

 - [files, transferring over Internet](#)

 - [newsgroups](#)

 - [web sites, accessing](#)

- [JPython, compared to Java](#)

launching

- [on Windows](#)

- [examples in book](#)

- [Python in Java applications](#)

- [run on startup](#)

server-side

- [databases](#)

- [examples in book](#)

- [zzz](#) [\[See also URLs Web web pages web sites\]\[See also URLs Web web pages web](#)

sites]

- [site.py](#)

- [user.py](#)

[scrollbars](#)

[ScrolledText class](#)

[ScrolledText widget](#)

[search path](#)

searching

- [binary search trees](#)

- [C header files](#)

- [graphs](#)

- [grep utility](#)

[secret module](#)

security

- [CGI scripts and](#)

 - [HTTP servers running](#)

- [examples in book](#)

- [os module](#)

passwords

- [in encrypted hyperlinks](#)

- [on POP page 2nd](#)

- email client
- encrypting 2nd
- escaping in HTML
- restricted execution mode
- sandbox model
- viruses, email
- web forms, numbers in
- web server files, displaying on browsers
- select module
- select(), servers, multiplexing
- sendmail program
- sequences
 - assigning
 - objects
 - permutations of
 - reversing 2nd
 - sorting 2nd
 - comparison functions
- serial ports
- SeriesDocument class (HTMLgen)
- server function
- server.getfile(), web sites, accessing
- server.sendmail()
- servers
 - asynchronous
 - COM
 - constraints
 - GUIDs
 - using from clients
 - using from VB client
- connecting to
- creating
 - with Python code
 - Apache
 - Mailman
 - tools for
- database
- email
- file
 - frontend, adding 2nd
- forking
 - multiple clients, handling with
 - zombies, killing
 - zombies, preventing
- FTP
 - closing connection
 - opening connection to
- HTTP
 - CGI scripts and
 - multiple clients, handling

- [with classes](#)
- [multiplexing](#)
- [POP, connecting to](#)
- [scripts](#)
 - [examples in book](#)
 - [zzz \[See also CGI, scripts\]\[See also CGI, scripts\]](#)
- [sending files to](#)
- [socket calls](#)
- [socket programs, running locally](#)
- [threading](#)
- [web](#)
 - [.zz \[See also Zope\]\[See also Zope\]](#)
 - [email client and](#)
 - [finding Python on](#)
 - [uploading client files to](#)
- [Set class](#)
 - [methods](#)
- [set\(\)](#)
- [__setattr__ method](#)
- [setops\(\)](#)
- [sets](#)
 - [classes](#)
 - [functions](#)
 - [moving to dictionaries](#)
 - [relational algebra, adding to](#)
- [SGML \(Standard Graphic Markup Language\) module](#)
- [sgmlib module 2nd](#)
- [shell scripts](#)
- [shell variables](#)
 - [configuring](#)
 - [faking inputs on forms with](#)
- [ShellGui class](#)
- [ShellGui example](#)
- [shellgui module](#)
- [shelve module 2nd](#)
 - [concurrent updates and](#)
 - [cPickle module and](#)
 - [pickle module and](#)
- [shelves 2nd](#)
 - [constraints](#)
 - [file operations](#)
 - [multiple, indexing](#)
 - [mutual exclusion for](#)
 - [file locking](#)
 - [mutexes](#)
- [OODBs and](#)
- [storage](#)
 - [classes 2nd](#)
 - [databases](#)
 - [object types](#)

- [objects, changing classes of](#)
 - [showinfo\(\), Linux and](#)
 - [signal handlers, zombies, preventing with](#)
 - [signal.signal\(\)](#)
 - [zombies and](#)
 - [zzz \[See also signals\]\[See also signals\]](#)
- [signals](#)
- [Simple Mail Transfer Protocol \[See SMTP\]](#)
- [SimpleDocument class \(HTMLgen\)](#)
- [SimpleHTTPServer module 2nd](#)
- [Simplified Wrapper and Interface Generator \[See SWIG\]](#)
- [site.py script](#)
- [slide presentation program](#)
- [sliders](#)
- [variables and](#)
- [Smalltalk, Python compared to](#)
- [smart links \[See hyperlinks\]](#)
- [SMTP \(Simple Mail Transfer Protocol\)](#)
- [date formatting standard](#)
 - [module \[See sgmlib module\]](#)
 - [sending mail](#)
 - [from browser](#)
- [smtpplib module 2nd](#)
- [email, sending from browser](#)
- [smtpplib.SMTP\(\)](#)
- [SOCK__STREAM variable, socket module](#)
- [socket module 2nd 3rd](#)
- [support for OpenSSL](#)
 - [variables](#)
- [socket object](#)
- [accept\(\)](#)
 - [bind\(\)](#)
 - [close\(\) 2nd](#)
 - [connect\(\)](#)
 - [listen\(\)](#)
 - [recv\(\) 2nd](#)
 - [send\(\) 2nd](#)
 - [setblocking\(\)](#)
 - [socket\(\) 2nd](#)
- [socket objects](#)
- [socket.bind\(\)](#)
- [sockets 2nd 3rd](#)
- [blocking/unblocking](#)
 - [calls](#)
 - [client](#)
 - [server](#)
- [CGI scripts and](#)
- [connect\(\)](#)
- [identifiers for machines](#)
- [IP addresses](#)

- [machine names](#)
- [message formats](#)
- [methods](#)
- [multiplexing servers and](#)
- [port numbers](#) [See port numbers]
- [programming](#)
- [programs](#)
 - [running locally](#)
 - [running remotely](#)
- [select\(\) and](#)
- [SocketServer module](#)
- [SocketServer.TCPServer class](#)
- [sort\(\)](#)
- [spam](#) 2nd
- [speed](#) [See performance]
- [split\(\)](#)
- [splitpath\(\)](#)
- [SQL \(Structured Query Language\)](#)
- [Stack class](#)
 - [optimizing](#)
 - [performance](#)
- [stack module](#)
 - [methods](#)
- [stacks](#)
 - [as lists](#)
 - [optimizing](#)
- [Standard Graphic Markup Language \(SGML\) module](#)
- [static binding](#)
- [static members](#)
- [storage](#)
 - [databases](#)
 - [flat-file](#) 2nd
 - [shelve](#) [See shelves]
 - [object types](#)
 - [persistent](#) 2nd
 - [pickled objects](#)
 - [zzz](#) [See also DBM][See also DBM]
- [__str__](#)
- [str\(\)](#)
- [str.lower\(\)](#)
- [streams](#)
 - [CGI and](#)
 - [pickled](#)
 - [redirecting](#)
 - [to widgets](#)
- [string module](#) 2nd
 - [vs. regular expressions](#)
 - [deprecation of](#)
- [string.atoi\(\)](#) 2nd
- [string.atol\(\)](#)

- [string.find\(\)](#)
- [string.join\(\)](#)
 - [delimiter.join\(\) and text processing](#)
- [string.replace\(\)](#)
 - [email client](#)
- [string.split\(\) 2nd](#)
 - [text processing](#)
- [string.strip\(\)](#)
- [string.upper\(\)](#)
- [strings](#)
 - [converting](#)
 - [in CGI scripts](#)
 - [to numbers 2nd](#)
 - [objects to, pickled objects and raw](#)
 - [regular expressions](#)
 - [compiled pattern objects](#)
 - [match objects](#)
 - [patterns](#)
 - [re module](#)
 - [rule](#)
 - [Unicode](#)
- [Structured Query Language \(SQL\)](#)
- [submit module](#)
- [subset\(\)](#)
- [summing columns](#)
- [SWIG \(Simplified Wrapper and Interface Generator\) 2nd](#)
 - [C extension module string stack](#)
 - [C structs](#)
 - [C variables and constants](#)
 - [C++ class integration](#)
 - [wrapping C environment calls](#)
 - [wrapping C++ classes](#)
- [sys.exec__info\(\)](#)
- [sys.exit\(\), vs. os.__exit\(\)](#)
- [sys.modules attribute](#)
- [sys.path](#)
 - [shelve module](#)
- [sys.path.append\(\)](#)
- [sys.stderr, error messages, trapping](#)
- [sys.stdout, error message, trapping](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[Table class \(HTMLgen\)](#)

tables on web pages

[adding](#)

[laying out forms](#)

[tags 2nd](#)

[Tcl language](#)

Tcl/Tk

[on Windows](#)

[Python and](#)

[TCP \(Transmission Control Protocol\)](#)

[socket module](#)

[TCP objects](#)

[TCP/IP](#)

[socket module](#)

[Telnet](#)

[telnetlib module](#)

[templates](#)

testing

[assert statement in v1.5](#)

[GuiMaker class](#)

[mixin methods](#)

[sequence permutations](#)

[set classes](#)

[set functions](#)

text

[editing](#)

[editors, program](#)

files

[distinguishing from binary](#)

[uploading](#)

processing

[parser generators](#)

[rule strings](#)

[summing columns](#)

[utilities](#)

[zz \[See also regular expressions\]\[See also regular expressions\]](#)

[ScrolledText class](#)

[widgets](#)

[this instance pointer \(C++\)](#)

thread module

[global objects, synchronizing access to](#)

[thread exits, waiting for](#)

[threading module](#)

[threads 2nd](#)

[email transfers and](#)

- [exiting, waiting for](#)
- [global interpreter lock and](#)
- [global objects, synchronizing access to](#)
- [GUIs and](#)
- [support for](#)
- [threading servers](#)
- [time-slicing \[See multiplexing servers\]](#)
- [time.sleep\(\)](#)
 - [client requests](#)
 - [servers, multiplexing](#)
- [time.strftime\(\)](#)
- [time/date formatting](#)
- [timer module 2nd](#)
- [Tk library](#)
- [Tkinter](#)
 - [animation techniques](#)
 - [canvas widgets](#)
 - [checkbuttons](#)
 - [adding to HTML forms](#)
 - [configuring](#)
 - [dialogs](#)
 - [Entry widget](#)
 - [Message widget](#)
 - [variables and](#)
 - [windows, top-level](#)
 - [Grail and](#)
 - [graphics](#)
 - [grids](#)
 - [listboxes](#)
 - [menus 2nd](#)
 - [GuiMaker class](#)
 - [object viewers](#)
 - [persistent](#)
 - [radiobuttons](#)
 - [adding to HTML forms](#)
 - [configuring](#)
 - [dialogs](#)
 - [Entry widget](#)
 - [Message widget](#)
 - [variables and](#)
 - [windows, top-level](#)
 - [scrollbars](#)
 - [sliders](#)
 - [variables and](#)
 - [text](#)
 - [editing](#)
 - [toolbars](#)
 - [GuiMaker class](#)
- [Tkinter module](#)
- [__tkinter module](#)

- [Tkinter module](#)
 - [documentation for email client](#)
 - [browser as deleting mail](#)
 - [forwarding mail](#)
 - [implementing loading mail](#)
 - [replying to mail](#)
 - [saving mail](#)
 - [sending mail](#)
 - [threading](#)
 - [viewing mail](#)
 - [FTP client frontend](#)
 - [PMW](#)
 - [Windows](#)
- [toolbars](#)
 - [GuiMaker class](#)
- [Toplevel object](#)
- [translating](#)
 - [conversion codes](#)
 - [Tcl/Tk to Python/Tkinter](#)
- [Transmission Control Protocol](#) [See TCP]
- [Trigger__Event\(\)](#)
- [try/finally statements](#)
 - [mailboxes, unlocking](#)
 - [shelves, mutual exclusion for](#)

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[UDP \(Uniform Datagram Protocol\), socket module](#)

[UnboundLocalError exception](#)

[Unicode 3.0 database, accessing](#)

[Unicode strings](#)

[unicodedata module](#)

[Uniform Datagram Protocol \(UDP\), socket module](#)

[Unix](#)

[end-of-lines, CGI scripts and](#)

[installing Python on](#)

[OpenSSL](#)

[Python programs, launching](#)

[running Python in restricted execution mode](#)

[web servers, finding Python on](#)

[unparsing \[See parsing rule strings\]](#)

[Unpickler class](#)

[urllib module 2nd 3rd](#)

[comment/errata reports, browsing](#)

[files, FTPing](#)

[state information in URL parameters](#)

[URLs, escaping](#)

[web sites, accessing](#)

[urllib.quote\(\), URLs, escaping](#)

[urllib.quote_plus\(\), URLs, escaping](#)

[urllib.urlencode\(\)](#)

[urllib.urlretrieve\(\)](#)

[Urlparse module](#)

[URLs \(Uniform Resource Locators\) 2nd](#)

[comment/errata reports](#)

[browsing](#)

[explicit URLs with](#)

[components of](#)

[minimal](#)

[embedded in hyperlinks](#)

[updating](#)

[escaping](#)

[conventions](#)

[form tags, embedded in](#)

[hardcoded, passing parameters in](#)

[hyperlinks, embedded in](#)

[module](#)

[parameters](#)

[passing](#)

[passing state information](#)

[parsing](#)

[passwords encrypted in](#)

- [text in, escaping](#)
- [web sites, accessing](#)
- Zope
 - [invoking functions through](#)
 - [mapping into calls on objects by](#)
 - [user interaction, adding to CGI scripts](#)
 - [user.py script](#)
- utilities
 - comment/errata reports
 - [browsing](#)
 - [submitting](#)
 - [email browser](#)
 - [external components](#)
 - [POP interface](#)
 - [POP password encryption](#)
 - [text processing](#)
- [uu module](#)

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

variables

[global flag](#)

[shell](#)

[configuring](#)

[faking inputs on forms with](#)

[versions of Python](#)

[changes in 2nd](#)

[virtual methods](#)

[Visual Python, running Python from](#)

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

Web

- [applications, trade-offs](#)
- [CGI scripts and](#) [See web sites, from CGI scripts]
- [client/server architecture](#)
- [servers](#) [See web servers Zope]
- [web browsers](#) [See browsers]
- [web pages](#)
 - [cgi module, parsing user input](#)
 - [CGI scripts and](#)
- email
 - [deleting](#)
 - [forwarding](#)
 - [passwords 2nd](#)
 - [replying to](#)
 - [selecting](#)
 - [sending](#)
 - [viewing](#)
- forms on
 - [changing](#)
 - [hidden fields in](#)
 - [laying out with tables](#)
 - [mocking up inputs](#)
 - [reusable](#)
 - [selection list](#)
 - [tags](#)
- [graphics on, adding](#)
- [HTMLgen](#) [See HTMLgen]
- [modules](#)
- [sharing objects between](#)
- tables on
 - [adding](#)
 - [tags](#)
- [templates, Zope and](#)
- [web servers](#)
 - [email client and](#)
 - [finding Python on](#)
 - [uploading client files to](#)
- web sites
 - from CGI scripts
 - [email](#) [See browsers, email client]
 - [implementing](#)
 - [root page](#)
 - [databases](#)
 - [accessing](#)
 - [httplib module](#)

- [urllib module](#)
- from CGI scripts
 - [error-reporting system](#)
- [downloading](#)
 - [deleting files when](#)
- [mirroring](#)
- [relocating, automating edits](#)
- [uploading](#)
 - [with subdirectories](#)
- [user interaction, adding](#)
- [zz](#) [See also Zope][See also Zope]
- [webbrowser module](#)
- [whichdb module](#)
- widgets
 - [analog/digital clock](#)
 - [canvas](#)
 - [arrow option](#)
 - [arrowshape option](#)
 - [scrolling](#)
 - [frame, nested](#)
 - [gridded](#)
 - [input](#)
 - [adding to HTML forms](#)
 - [missing/invalid, checking for](#)
 - [PMW](#)
 - [redirecting streams to](#)
 - [resizing 2nd 3rd](#)
 - [anchor option](#)
 - [ScrolledText](#)
 - [text](#)
 - [editing text](#)
 - [Tic-Tac-Toe game 2nd](#)
 - [zzz](#) [See also GUIs][See also GUIs]
- [win32all package 2nd 3rd](#)
- [win32COM extensions](#)
- Windows
 - [client requests and](#)
 - [COM and 2nd](#)
 - [DBM and](#)
- windows
 - [email client status messages](#)
- Windows
 - [email client, threading](#)
 - [forking servers and](#)
 - [input](#)
 - [installing Python on 2nd](#)
 - [os.popen\(\) and](#)
 - [portability in v2.0](#)
 - [Python programs, launching](#)
 - [Python version changes](#)

[Python versions](#)
[PYTHONPATH](#)
[Registry access](#)
[restricted execution mode](#)
[serial ports on](#)
[server processes, killing](#)
[Tcl/Tk on](#)
[Tkinter support](#)
[web scripting extensions](#)
 [Active Scripting](#)
 [ASP](#)
 [COM](#)
[zzz](#) [See also Internet Explorer][See also Internet Explorer]
[__winreg module](#)
[write\(\)](#), [Pickler class](#)

I l@ve RuBoard

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)
[\[Y\]](#) [\[Z\]](#)

[xdrlib module](#)

XML (Extensible Markup Language)

[module](#) [See [xmllib module](#)]

[parsing](#), [Expat XML](#)

[processing tools](#)

[support for](#)

[xml module](#)

[xmllib module](#) [2nd](#)

I l@ve RuBoard

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[yacc system](#)

[YAPPS \(Yet Another Python Parser System\)](#)

I l@ve RuBoard

I l@ve RuBoard

[A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X]
[Y] [Z]

[zip\(\) 2nd](#)

[zipfile module](#)

[zombies](#)

[killing](#)

[preventing](#)

[Zope 2nd 3rd 4th](#)

[components](#)

[forms and](#)

[HTML document templates](#)

[hyperlinks and](#)

[object database](#)

[ORB 2nd](#)

[Python objects, publishing](#)

I l@ve RuBoard



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)

Programming Python, 2nd Edition

By [Mark Lutz](#)

Publisher : O'Reilly
Pub Date : March 2001
ISBN : 0-596-00085-5
Pages : 1256

Programming Python focuses on advanced applications of Python. Endorsed by Python creator Guido van Rossum, it demonstrates advanced Python techniques, and addresses software design issues such as reusability and object-oriented programming. The enclosed platform-neutral CD-ROM has book examples and various Python-related packages, including the full Python Version 2.0 source code distribution.

A.1 Major Changes in 2.0

This section lists changes introduced in Python release 2.0. Note that third-party extensions built for Python 1.5.x or 1.6 cannot be used with Python 2.0; these extensions must be rebuilt for 2.0. Python bytecode files (**.pyc* and **.pyo*) are not compatible between releases either.

A.1.1 Core Language Changes

The following sections describe changes made to the Python language itself.

A.1.1.1 Augmented assignment

After nearly a decade of complaints from C programmers, Guido broke down and added 11 new C-like assignment operators to the language:

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

The statement `A += B` is similar to `A = A + B` except that `A` is evaluated only once (useful if it is a complex expression). If `A` is a mutable object, it may be modified in place; for instance, if it is a list, `A += B` has the same effect as `A.extend(B)`.

Classes and built-in object types can override the new operators in order to implement the in-place behavior; the non-in-place behavior is automatically used as a fallback when an object does not implement the in-place behavior. For classes, the method name is the method name for the corresponding non-in-place operator prepended with an "i" (e.g., `__iadd__` implements in-place `__add__`).

A.1.1.2 List comprehensions

A new expression notation was added for lists whose elements are computed from another list (or lists):

```
[<expression> for <variable> in <sequence>]
```

For example, `[i**2 for i in range(4)]` yields the list `[0, 1, 4, 9]`. This is more efficient than using `map` with a lambda, and at least in the context of scanning lists, avoids some scoping issues raised by lambdas (e.g., using defaults to pass in information from the enclosing scope). You can also add a condition:

```
[<expression> for <variable> in <sequence> if <condition>]
```

For example, `[w for w in words if w == w.lower()]` yields the list of words that contain no uppercase characters. This is more efficient than `filter` with a lambda. Nested `for` loops and more than one `if` is supported as well, though using this seems to yield code that is as complex as nested maps and lambdas (see Python manuals for more details).

A.1.1.3 Extended import statements

Import statements now allow an "as" clause (e.g., `import mod as name`), which saves an

assignment of an imported module's name to another variable. This works with `from` statements and package paths too (e.g., `from mod import var as name`). The word "as" was not made a reserved word in the process. (To import odd filenames that don't map to Python variable names, see the `__import__` built-in function.)

A.1.1.4 Extended print statement

The print statement now has an option that makes the output go to a different file than the default `sys.stdout`. For instance, to write an error message to `sys.stderr`, you can now write:

```
print >> sys.stderr, "spam"
```

As a special case, if the expression used to indicate the file evaluates to `None`, the current value of `sys.stdout` is used (like not using `>>` at all). Note that you can always write to file objects such as `sys.stderr` by calling their `write` method; this optional extension simply adds the extra formatting performed by the print statement (e.g., string conversion, spaces between items).

A.1.1.5 Optional collection of cyclical garbage

Python is now equipped with a garbage collector that can hunt down cyclical references between Python objects. It does not replace reference counting (and in fact depends on the reference counts being correct), but can decide that a set of objects belongs to a cycle if all their reference counts are accounted for in their references to each other. A new module named `gc` lets you control parameters of the garbage collection; an option to the Python "configure" script lets you enable or disable the garbage collection. (See the 2.0 release notes or the library manual to check if this feature is enabled by default or not; because running this extra garbage collection step periodically adds performance overheads, the decision on whether to turn it on by default is pending.)

A.1.2 Selected Library Changes

This is a partial list of standard library changes introduced by Python release 2.0; see 2.0 release notes for a full description of the changes.

A.1.2.1 New zip function

A new function `zip` was added: `zip(seq1, seq2, ...)` is equivalent to `map(None, seq1, seq2, ...)` when the sequences have the same length. For instance, `zip([1, 2, 3], [10, 20, 30])` returns `[(1, 10), (2, 20), (3, 30)]`. When the lists are not all the same length, the shortest list defines the result's length.

A.1.2.2 XML support

A new standard module named `pyexpat` provides an interface to the Expat XML parser. A new standard module package named `xml` provides assorted XML support code in (so far) three subpackages: `xml.dom`, `xml.sax`, and `xml.parsers`.

A.1.2.3 New web browser module

The new `webbrowser` module attempts to provide a platform-independent API to launch a web browser. (See also the `LaunchBrowser` script at the end of [Chapter 4](#).)

A.1.3 Python/C Integration API Changes

Portability was ensured to 64-bit platforms under both Linux and Win64, especially for the new Intel Itanium processor. Large file support was also added for Linux64 and Win64.

The garbage collection changes resulted in the creation of two new slots on an object, `tp_traverse` and `tp_clear`. The augmented assignment changes result in the creation of a new slot for each in-place operator. The GC API creates new requirements for container types implemented in C extension modules. See *Include/objimpl.h* in the Python source distribution.

A.1.4 Windows Changes

New `popen2`, `popen3`, and `popen4` calls were added in the `os` module.

The `os.popen` call is now much more usable on Windows 95 and 98. To fix this call for Windows 9x, Python internally uses the `w9xpopen.exe` program in the root of your Python installation (it is not a standalone program). See Microsoft Knowledge Base article Q150956 for more details.

Administrator privileges are no longer required to install Python on Windows NT or Windows 2000. The Windows installer also now installs by default in `\Python20\` on the default volume (e.g., `C:\Python20`), instead of the older-style `\Program Files\Python-2.0\`.

The Windows installer no longer runs a separate Tcl/Tk installer; instead, it installs the needed Tcl/Tk files directly in the Python directory. If you already have a Tcl/Tk installation, this wastes some disk space (about 4 MB) but avoids problems with conflicting Tcl/Tk installations and makes it much easier for Python to ensure that Tcl/Tk can find all its files.

Python 2.1 Alpha Features

Like the weather in Colorado, if you wait long enough, Python's feature set changes. Just before this edition went to the printer, the first alpha release of Python 2.1 was announced. Among its new weapons are these:

- Functions can now have arbitrary attributes attached to them; simply assign to function attribute names to associate extra information with the function (something coders had been doing with formatted documentation stings).
- A new rich comparison extension now allows classes to overload individual comparison operators with distinct methods (e.g., `__lt__` overloads `<` tests), instead of trying to handle all tests in the single `__cmp__` method.
- A warning framework provides an interface to messages issued for use of deprecated features (e.g., the `regex` module).
- The Python build system has been revamped to use the `Distutils` package.

- A new `sys.displayhook` attribute allows users to customize the way objects are printed at the interactive prompt.
- Line-by-line file input/output (the file `readline` method) was made much faster, and a new `xreadlines` file method reads just one line at a time in `for` loops.
- Also: the numeric coercion model used in C extensions was altered, modules may now set an `__all__` name to specify which names they export for `from * imports`, the `ftplib` module now defaults to "passive" mode to work better with firewalls, and so on.
- Other enhancements, such as statically nested scopes and weak references, were still on the drawing board in the alpha release.

As usual, of course, you should consult this book's web page (<http://www.rmi.net/~lutz/about-pp.html>) and Python 2.1 and later release notes for Python developments that will surely occur immediately after I ship this insert off to my publisher.

A.2 Major Changes in 1.6

This section lists changes introduced by Python release 1.6; by proxy, most are part of release 2.0 as well.

A.2.1 Incompatibilities

The `append` method for lists can no longer be invoked with more than one argument. This used to append a single tuple made out of all arguments, but was undocumented. To append a tuple, write `l.append((a, b, c))`.

The `connect`, `connect_ex`, and `bind` methods for sockets require exactly one argument. Previously, you could call `s.connect(host, port)`, but this was not by design; you must now write `s.connect((host, port))`.

The `str` and `repr` functions are now different more often. For long integers, `str` no longer appends an "L"; `str(1L)` is "1", which used to be "1L", and `repr(1L)` still returns "1L". For floats, `repr` now gives 17 digits of precision to ensure that no precision is lost (on all current hardware).

Some library functions and tools have been moved to the deprecated category, including some widely used tools such as `find`. The `string` module is now simply a frontend to the new string methods, but given that this module is used by almost every Python module written to date, it is very unlikely to go away.

A.2.2 Core Language Changes

The following sections describe changes made to the Python language itself.

A.2.2.1 Unicode strings

Python now supports Unicode (i.e., 16-bit wide character) strings. Release 1.6 added a new fundamental datatype (the Unicode string), a new built-in function `unicode`, and numerous C APIs to deal with Unicode and encodings. Unicode string constants are prefixed with the letter "u", much like raw strings (e.g., `u"..."`). See the file *Misc/unicode.txt* in your Python distribution for details, or visit web site <http://starship.python.net/crew/lemburg/unicode-proposal.txt>.

A.2.2.2 String methods

Many of the functions in the `string` module are now available as methods of string objects. For instance, you can now say `str.lower()` instead of importing the `string` module and saying `string.lower(str)`. The equivalent of `string.join(sequence, delimiter)` is `delimiter.join(sequence)`. (That is, you use `" ".join(sequence)` to mimic `string.join(sequence)`).

A.2.2.3 New (internal) regular expression engine

The new regular expression engine, SRE, is fully backward-compatible with the old engine, and is invoked using the same interface (the `re` module). That is, the `re` module's interface remains the way to write matches, and is unchanged; it is simply implemented to use SRE. You can explicitly invoke the old engine by importing `pre`, or the SRE engine by importing `sre`. SRE is faster than `pre`, and supports Unicode (which was the main reason to develop yet another underlying regular expression engine).

A.2.2.4 apply-like function calls syntax

Special function call syntax can be used instead of the `apply` function: `f(*args, **kws)` is equivalent to `apply(f, args, kws)`. You can also use variations like `f(a1, a2, *args, **kws)`, and can leave one or the other out (e.g., `f(*args)`, `f(**kws)`).

A.2.2.5 String to number conversion bases

The built-ins `int` and `long` take an optional second argument to indicate the conversion base, but only if the first argument is a string. This makes `string.atoi` and `string.atol` obsolete. (`string.atof` already was.)

A.2.2.6 Better errors for local name oddities

When a local variable is known to the compiler but undefined when used, a new exception `UnboundLocalError` is raised. This is a class derived from `NameError`, so code that catches `NameError` should still work. The purpose is to provide better diagnostics in the following example:

```
x = 1
def f( ):
    print x
    x = x+1
```

This used to raise a confusing `NameError` on the `print` statement.

A.2.2.7 Membership operator overloading

You can now override the `in` operator by defining a `__contains__` method. Note that it has its arguments backward: `x in a` runs `a.__contains__(x)` (that's why the name isn't `__in__`).

A.2.3 Selected Library Module Changes

This section lists some of the changes made to the Python standard library.

`distutils`

New; tools for distributing Python modules.

`zipfile`

New; read and write zip archives (module `gzip` does gzip files).

unicodedata

New; access to the Unicode 3.0 database.

_winreg

New; Windows registry access (one without the _ is in progress).

socket , httplib, urllib

Expanded to include optional OpenSSL secure socket support (on Unix only).

_tkinter

Support for Tk versions 8.0 through 8.3.

string

This module no longer uses the built-in C `strop` module, but takes advantage of the new string methods to provide transparent support for both Unicode and ordinary strings.

A.2.4 Selected Tools Changes

This section lists some of the changes made to Python tools.

IDLE

Completely overhauled. See the IDLE home page at <http://www.python.org> for more information.

Tools/i18n/pygettext.py

Python equivalent of `xgettext` message text extraction tool used for internationalizing applications written in Python.

A.3 Major Changes Between 1.3 and 1.5.2

This section describes significant language, library, tool, and C API changes in Python between of this book (Python 1.3) and Python release 1.5.2.

A.3.1 Core Language Changes

The following sections describe changes made to the Python language itself.

A.3.1.1 Pseudo-private class attributes

Python now provides a name-mangling protocol that hides attribute names used by classes. Inside a class statement, a name of the form `_x` is automatically changed by Python to `_Class_x`, where *Class* is the name of the class being defined by the statement. Because the enclosing class name is prepended to the attribute name, this mechanism limits the possibilities of name clashes when you extend or mix existing classes. Note that this is not a name-mangling mechanism at all, just a class name localization feature to minimize name clashes in hierarchies. The attribute name is localized to the instance object's namespace at the bottom of the attribute inheritance links chain.

A.3.1.2 Class exceptions

Exceptions may now take the form of class (and class instance) objects. The intent is to support broad categories. Because an `except` clause will now match a raised exception if it names the raised class or its superclass, specifying superclasses allows `try` statements to catch broad categories without listing members explicitly (e.g., catching a numeric-error superclass exception will also catch specific numeric errors). Python's standard built-in exceptions are now classes (instead of strings) and have been organized into a shallow class hierarchy; see the library manual for details.

A.3.1.3 Package imports

Import statements may now reference directory paths on your computer by dotted-path syntax. For example,

```
import directory1.directory2.module           # and use path
from directory1.directory2.module import name # and use "name"
```

Both load a module nested two levels deep in packages (directories). The leftmost package name in the path (`directory1`) must be a directory within a directory that is listed in the Python module search path (`sys.path` initialized from `PYTHONPATH`). Thereafter, the `import` statement's path denotes subdirectories to follow. Paths prevent module name conflicts when installing multiple Python systems on the same machine. Each system is expected to find their own version of the same module name (otherwise, only the first on `PYTHONPATH` would be used).

Unlike the older `ni` module that this feature replaces, the new package support is always available (without running special imports) and requires each package directory along an import path to contain a `__init__.py` module file to identify the directory as a package, and serve as its namespace if imported. Packages tend to work better with `from` than with `import`, since the full path must be repeated to import objects after an `import`.

A.3.1.4 New assert statement

Python 1.5 added a new statement:

```
assert test [, value]
```

which is the same as:

```
if __debug__:
    if not test:
        raise AssertionError, value
```

Assertions are mostly meant for debugging, but can also be used to specify program constraints (on entry to functions).

A.3.1.5 Reserved word changes

The word "assert" was added to the list of Python reserved words; "access" was removed (it has been deprecated in earnest).

A.3.1.6 New dictionary methods

A few convenience methods were added to the built-in dictionary object to avoid the need for more. `D.clear()`, `D.copy()`, `D.update()`, and `D.get()`. The first two methods empty and copy `D` respectively. `D1.update(D2)` is equivalent to the loop:

```
for k in D2.keys(): D1[k] = D2[k]
```

`D.get(k)` returns `D[k]` if it exists, or `None` (or its optional second argument) if the key does not exist.

A.3.1.7 New list methods

List objects have a new method, `pop`, to fetch and delete the last item of the list:

```
x = s.pop() ...is the same as the two statements... x = s[-1]; del s[-1]
```

and `extend`, to concatenate a list of items on the end, in place:

```
s.extend(x) ...is the same as... s[s[len(s)]:] += x
```

The `pop` method can also be passed an index to delete (it defaults to -1). Unlike `append`, `extend` appends the entire list and adds each of its items at the end.

A.3.1.8 "Raw" string constants

In support of regular expressions and Windows, Python allows string constants to be written in the form `r"..."`, which works like a normal string except that Python leaves any backslashes in the string as-is. They remain as literal `\` characters rather than being interpreted as special escape codes by Python.

A.3.1.9 Complex number type

Python now supports complex number constants (e.g., `1+3j`) and complex arithmetic operations. It also includes `cmath` operators, plus a `cmath` module with many of the `math` module's functions for complex numbers.

A.3.1.10 Printing cyclic objects doesn't core dump

Objects created with code like `L.append(L)` are now detected and printed specially by the interpreter. In the past, trying to print cyclic objects caused the interpreter to loop recursively (which eventually led to a core dump).

A.3.1.11 raise without arguments: re-raise

A `raise` statement without any exception or extra-data arguments now makes Python re-raise the last raised uncaught exception.

A.3.1.12 raise forms for class exceptions

Because exceptions can now either be string objects or classes and class instances, you can use a variety of the following `raise` statement forms:

```
raise string          # matches except with same string object
raise string, data    # same, with optional data

raise class, instance # matches except with class or its superclass
raise instance        # same as: raise instance.__class__, instance

raise                 # reraise last exception
```

You can also use the following three forms, which are for backwards-compatibility with earlier Python versions where all built-in exceptions were strings:

```
raise class          # same as: raise class( ) (and: raise class, instance)
raise class, arg     # same as: raise class(arg)
raise class, (arg,...) # same as: raise class(args...)
```

A.3.1.13 Power operator X ** Y

The new `**` binary operator computes the left operand raised to the power of the right operand. It is similar to the built-in `pow` function.

A.3.1.14 Generalized sequence assignments

In an assignment (`=` statements and other assignment contexts), you can now assign any sort of object to a list or tuple on the left (e.g., `(A, B) = seq`, `[A, B] = seq`). In the past, the sequence type had to be a list or tuple.

A.3.1.15 It's faster

Python 1.5 has been clocked at almost twice the speed of its predecessors on the *Lib/test/pystone* benchmark (I've seen almost a threefold speedup in other tests.)

A.3.2 Library Changes

The following sections describe changes made to the Python standard library.

A.3.2.1 dir(X) now works on more objects

The built-in `dir` function now reports attributes for modules, classes, and class instances, as well as objects such as lists, dictionaries, and files. You don't need to use members like `__methods__` (but you can).

A.3.2.2 New conversions: `int(X)`, `float(X)`, `list(S)`

The `int` and `float` built-in functions now accept string arguments, and convert from strings to numbers like `string.atoi/atof`. The new `list(S)` built-in function converts any sequence to a list, much like the obscure `map(None, S)` trick.

A.3.2.3 The new `re` regular expression module

A new regular expression module, `re`, offers full-blown Perl-style regular expression matching. For details, see the `re` module documentation. The older `regex` module described in the first edition is still available, but considered deprecated.

A.3.2.4 `splitfields/joinfields` became `split/join`

The `split` and `join` functions in the `string` module were generalized to do the same work as the old `splitfields` and `joinfields`.

A.3.2.5 Persistence: unpickler no longer calls `__init__`

Beginning in Python 1.5, the `pickle` module's unpickler (loader) no longer calls class `__init__` to recreate pickled class instance objects. This means that classes no longer need defaults for all constructor arguments to be used for persistent objects. To force Python to call the `__init__` method (as it always does for non-persistent objects), classes must provide a special `__getinitargs__` method; see the library manual for details.

A.3.2.6 Object pickler coded in C: `cPickle`

An implementation of the `pickle` module in C is now a standard part of Python. It's called `cPickle` and is reportedly many times faster than the original `pickle`. If present, the `shelve` module loads it instead of the original `pickle` module automatically.

A.3.2.7 `anydbm.open` now expects a "c" second argument for prior behavior

To open a DBM file in "create new or open existing for read+write" mode, pass a "c" in argument to `anydbm.open`. This changed as of Python 1.5.2; passing a "c" now does what passing no second argument used to do (the second argument now defaults to "r" -- read-only). This does not impact `shelve.open`.

A.3.2.8 `rand` module replaced by `random` module

The `rand` module is now deprecated; use `random` instead.

A.3.2.9 Assorted Tkinter changes

Tkinter became portable to and sprouted native look-and-feel for all major platforms (Windows, Mac OS, etc.). There has been a variety of changes in the Tkinter GUI interface:

StringVar objects can't be called

The `__call__` method for `StringVar` class objects was dropped in Python 1.4; that means you must explicitly call their `get()`/`set()` methods, instead of calling them with or without arguments.

ScrolledText changed

The `ScrolledText` widget went through a minor interface change in Python 1.4, which was backed out in release 1.5 due to code breakage (so never mind).

Gridded geometry manager

Tkinter now supports Tk's new `grid` geometry manager. To use it, call the `grid` method of a widget (much like `pack`, but passes row and column numbers, not constraints).

New Tkinter documentation site

Fredrik Lundh now maintains a nice set of Tkinter documentation at <http://www.pythonw.com>; it provides references and tutorials.

A.3.2.10 CGI module interface change

The CGI interface changed. An older `FormContent` interface was deprecated in favor of the `Field` object's interface. See the library manual for details.

A.3.2.11 site.py, user.py, and PYTHONHOME

These scripts are automatically run by Python on startup, used to tailor initial paths configuration. See the library manual for details.

A.3.2.12 Assignment to os.environ[key] calls putenv

Assigning to a key in the `os.environ` dictionary now updates the corresponding environment variable. It triggers a call to the C library's `putenv` routine such that the changes are reflected in the C code layers as well as in the environment of any child processes spawned by the Python program. This is now exposed in the `os` module too (`os.putenv`).

A.3.2.13 New sys.exc_info() tuple

The new `exc_info()` function in the `sys` module returns a tuple of values corresponding to `sys.exc_value`. These older names access a single global exception; `exc_info` is specific to the current exception.

A.3.2.14 The new operator module

There is a new standard module called `operator`, which provides functions that implement most Python expression operators. For instance, `operator.add(X, Y)` does the same thing as `X+Y`, but `operator` module exports are functions, they are sometimes handy to use in things like `map`, so you can create a function or use a lambda form.

A.3.3 Tool Changes

The following sections describe major Python tool-related changes.

A.3.3.1 JPython (a.k.a. Jython): a Python-to-Java compiler

The new JPython system is an alternative Python implementation that compiles Python program Virtual Machine (JVM) bytecode and provides hooks for integrating Python and Java programs [15](#).

A.3.3.2 MS-Windows ports: COM, Tkinter

The COM interfaces in the Python Windows ports have evolved substantially since the first edition descriptions (it was "OLE" back then); see [Chapter 15](#). Python also now ships as a self-installer with built-in support for the Tkinter interface, DBM-style files, and more; it's a simple double-click today.

A.3.3.3 SWIG growth, C++ shadow classes

The SWIG system has become a primary extension writers' tool, with new "shadow classes" for classes. See [Chapter 19](#).

A.3.3.4 Zope (formerly Bobo): Python objects for the Web

This system for publishing Python objects on the Web has grown to become a popular tool for C programmers and web scripters in general. See the Zope section in [Chapter 15](#).

A.3.3.5 HTMLgen: making HTML from Python classes

This tool for generating correct HTML files (web page layouts) from Python class object trees has matured. See [Chapter 15](#).

A.3.3.6 PMW: Python mega-widgets for Tkinter

The PMW system provides powerful, higher-level widgets for Tkinter-based GUIs in Python. See [Chapter 15](#).

A.3.3.7 IDLE: an integrated development environment GUI

Python now ships with a point-and-click development interface named IDLE. Written in Python Tkinter GUI library, IDLE either comes in the source library's Tools directory or is automatically installed with Python itself (on Windows, see IDLE's entry in the Python menu within your Start button menus). IDLE provides a syntax-coloring text editor, a graphical debugger, an object browser, and more. If you have Python support enabled and are accustomed to more advanced development interfaces, IDLE provides an alternative to the traditional Python command line. IDLE does not provide a GUI builder today.

A.3.3.8 Other tool growth: PIL, NumPy, Database API

The PIL image processing and NumPy numeric programming systems have matured considerably. The portable database API for Python has been released. See [Chapter 6](#) and [Chapter 16](#).

A.3.4 Python/C Integration API Changes

The following sections describe changes made to the Python C API.

A.3.4.1 A single Python.h header file

All useful Python symbols are now exported in the single *Python.h* header file; no other header files are imported in most cases.

A.3.4.2 A single libpython*.a C library file

All Python interpreter code is now packaged in a single library file when you build Python. For Python 1.5, you need only link in *libpython1.5.a* when embedding Python (instead of the older *libpython* libraries plus *.o*'s).

A.3.4.3 The "Great (Grand?) Renaming" is complete

All exposed Python symbols now start with a "Py" prefix.

A.3.4.4 Threading support, multiple interpreters

A handful of new API tools provide better support for threads when embedding Python. For instance, `Py_Finalize` is a new tool for finalizing Python (`Py_Finalize`) and for creating "multiple interpreters" (`Py_NewInterpreter`).

Note that spawning Python language threads may be a viable alternative to C-level threads, and namespaces are often sufficient to isolate names used in independent system components; both are easier to manage than multiple interpreters and threads. But in some threaded programs, it's also necessary to have one copy of system modules and structures per thread, and this is where multiple interpreters come in (e.g., without one copy per thread, imports might find an already-loaded module in the `sys.modules` dictionary that was imported by a different thread). See the new C API documentation manuals for details.

A.3.4.5 New Python C API documentation

There is a new reference manual that ships with Python and documents major C API tools and functions. It's not fully fleshed out yet, but it's a useful start.

Appendix A. Recent Python Changes

This appendix summarizes prominent changes introduced in Python releases since the first edition of this book. It is divided into three sections, mostly because the sections on 1.6 and 2.0 changes were adapted from release note documents:

- Changes introduced in Python 2.0 (and 2.1)
- Changes introduced in Python 1.6
- Changes between the first edition and Python 1.5.2

Python 1.3 was the most recent release when the first edition was published (October 1996), and Python 1.6 and 2.0 were released just before this second edition was finished. 1.6 was the last release posted by CNRI, and 2.0 was released from BeOpen (Guido's two employers prior to his move to Digital Creations); 2.0 adds a handful of features to 1.6.

With a few notable exceptions, the changes over the last five years have introduced new features to Python, but have not changed it in incompatible ways. Many of the new features are widely useful (e.g., module packages), but some seem to address the whims of Python gurus (e.g., list comprehensions) and can be safely ignored by anyone else. In any event, although it is important to keep in touch with Python evolution, you should not take this appendix too seriously. Frankly, application library and tool usage is much more important in practice than obscure language additions.

For information on the Python changes that will surely occur after this edition's publication, consult either the resources I maintain at this book's web site (<http://rmi.net/~lutz/about-pp.html>), the resources available at Python's web site (<http://www.python.org>), or the release notes that accompany Python releases.

B.1 Installing Python

This section gives an overview of install-related details -- instructions for putting the Python interpreter on your computer.

B.1.1 Windows

Python install details vary per platform and are described in the resources just listed. But as an overview, Windows users will find a Python self-installer executable at <http://examples.oreilly.com/python2> (see the top-level Python 1.5.2 and 2.0 directories). Simply double-click the installer program and answer "yes," "next," or "default" to step through a default Windows install. Be sure to install Tcl/Tk too, if you are asked about it along the way.

After the install, you will find an entry for Python in your Start button's Programs menu; it includes options for running both the IDLE integrated GUI development interface and the command-line console session, viewing Python's standard manuals, and more. Python's manuals are installed with the interpreter in HTML form, and open locally in a web browser when selected.

Python also registers itself to open Python files on Windows, so you can simply click on Python scripts in a Windows file explorer window to launch them. You can also run Python scripts by typing `python file.py` command lines at any DOS command-line prompt, provided that the directory containing the `python.exe` Python interpreter program is added to your PATH DOS shell variable (see the configuration and running sections later).

Note that the standard Python package for Windows includes full Tkinter support. You do not need to install other packages or perform any extra install steps to run Tkinter GUIs on Windows; simply install Python. All necessary Tkinter components are installed by the Python self-installer, and Python automatically finds the necessary components without extra environment settings. The Windows install also includes the `bsddb` extension to support DBM-style files.

If you plan on doing any Windows-specific work such as COM development, you will probably want to install the extra `win32all` extensions package (available at <http://examples.oreilly.com/python2> as well as at <http://www.python.org>). This package registers Python for Active Scripting, provides MFC wrappers and COM integration, and more (see [Chapter 15](#)). Also note that Python distributions available from other sources (e.g., the ActivePython distribution from ActiveState, <http://www.activestate.com>) may include both Python and the Windows extensions package.

B.1.2 Unix and Linux

Python may already be available on these platforms (it's often installed as a standard part of Linux these days); check your `/usr/bin` and `/usr/local/bin` directories to see if a Python interpreter is lurking there. If not, Python is generally installed on these platforms from either an `rpm` package (which installs Python executables and libraries automatically) or the source

code distribution package (which you unpack and compile locally on your computer). Compiling Python from its source on Linux is a trivial task -- usually just a matter of typing two or three simple command lines. See the Python source distribution's top-level README files and Linux `rpm` documentation for more details.

B.1.3 Macintosh and Others

Please see the documentation associated with the Macintosh ports for install and usage details. For other platforms, you will likely need to find ports at <http://www.python.org> and consult the port's install notes or documentation.

B.2 Book Examples Distribution

This section briefly discusses the book's example source code distribution, and covers example usage details.

B.2.1 The Book Examples Package

The *Examples\PP2E* CD directory is a Python module package that contains source code files for all examples presented in this book (and more). The *PP2E* package in turn contains nested module packages that partition the example files into subdirectories by topic. You can either run files straight off the CD, or copy the *PP2E* directory onto your machine's hard drive (copying over allows you to change the files, and lets Python store their compiled bytecode for faster startups).

Either way, the directory that contains the *PP2E* root must generally be listed on the Python module search path (normally, the PYTHONPATH environment variable). This is the only entry that you must add to the Python path, though; import statements in book examples are always package import paths relative to the *PP2E* root directory unless the imported module lives in the same directory as the importer.

Also in the examples package, you'll find scripts for converting example files' line-feeds to and from Unix format (they are in DOS format on the CD--see <http://examples.oreilly.com/python2>), making files writable (useful after a drag-and-drop on Windows), and more. See the README files at the top of the *Examples* and *PP2E* directory trees for more details on package tree usage and utilities.

B.2.2 Running the Demo Launcher Scripts

The top level of the CD's *Examples\PP2E* package (see <http://examples.oreilly.com/python2>) includes Python self-configuring scripts that can be run to launch major book examples, even if you do not configure your environment. That is, they should work even if you don't set your PATH or PYTHONPATH shell variables. These two scripts, PyDemos and PyGadgets, are presented in [Chapter 8](#), and described more fully in both this book's Preface and the CD's README files (see <http://examples.oreilly.com/python2>). In most cases, you should be able to run these scripts right off the book's CD by double-clicking on them in a file explorer GUI (assuming Python has been installed, of course).

B.3 Environment Configuration

This section introduces Python environment setup details and describes settings that impact Python programs.

B.3.1 Shell Variables

The following shell environment variables (among others) are usually important when using Python:

PYTHONPATH

Python's module file search path. If set, it is used to locate modules at run-time when they're imported by a Python program -- Python looks for an imported module file or package directory in each directory listed on PYTHONPATH, from left to right. Python generally searches the home directory of a script as well as the Python standard source code library directory automatically, so you don't need to add these. Hint: check `sys.path` interactively to see how the path is truly set up.

PYTHONSTARTUP

An optional Python initialization file. If used, set this variable to the full path-name of a file of Python code (a module) that you want to be run each time the Python interactive command-line interpreter starts up. This is a convenient way to import modules you use frequently when working interactively.

PATH

The operating system's executable search path variable. It is used to locate program files when you type their names at a command line without their full directory paths. Add the directory containing the `python` interpreter executable file (or else you must retype its directory path each time).

In addition, users on platforms other than Windows may need to set variables to use Tkinter if Tcl/Tk installations cannot be found normally. Set `TK_LIBRARY` and `TCL_LIBRARY` variables to point to the local Tk and Tcl library file directories.

B.3.2 Configuration Settings

The *Examples\PP2E\Config* directory on the CD (see <http://examples.oreilly.com/python2>) contains example configuration files with comments for Python variable settings. On Windows NT, you can set these variables in the system settings GUI (more on this in a minute); on Windows 98, you can set them from DOS batch files, which can be run from your *C:\autoexec.bat* file to make sure they are set every time you start your compute. For example, my *autoexec* file includes this line:

```
C:\PP2ndEd\examples\PP2E\Config\setup-pp.bat
```

which in turn invokes a file that contains these lines to add Python to the system PATH, and the book examples package root to PYTHONPATH:

```
REM PATH %PATH%;c:\Python20
PATH %PATH%;c:\"program files"\python

set PP2EHOME=C:\PP2ndEd\examples
set PYTHONPATH=%PP2EHOME%;%PYTHONPATH%
```

Pick (i.e., remove the `REM` from) one of the first two lines, depending upon your Python install -- the first line assumes a Python 2.0 default install, and the second assumes Python 1.5.2. Also change the `PP2EHOME` setting here to the directory that contains the *PP2E* examples root on your machine (the one shown works on my computer). On Linux, my `~/.cshrc` startup file sources a *setup-pp.csh* file that looks similar:

```
setenv PATH $PATH:/usr/bin
setenv PP2EHOME /home/mark/PP2ndEd/examples
setenv PYTHONPATH $PP2EHOME:$PYTHONPATH
```

But the syntax used to set variables varies per shell (see the *PP2E\Config* CD directory for more details). Setting the PYTHONPATH shell variable to a list of directories like this works on most platforms, and is the typical way to configure your module search path. On some platforms, there are other ways to set the search path. Here are a few platform-specific hints:

Windows port

The Windows port allows the Windows registry to be used in addition to setting PYTHONPATH in DOS. On some versions of Windows, rather than changing *C:\autoexec.bat* and rebooting, you can also set your path by selecting the Control Panel, picking the System icon, clicking in the Environment Settings tab, and typing PYTHONPATH and the path you want (e.g., *C:\mydir*) in the resulting dialog box. Such settings are permanent, just like adding them to *autoexec.bat*.

JPython

Under JPython, the Java implementation of Python, the path may take the form of `-Dpath` command-line arguments on the Java command used to launch a program, or `python.path` assignments in Java registry files.

B.3.3 Configuring from a Program

In all cases, `sys.path` represents the search path to Python scripts and is initialized from path settings in your environment plus standard defaults. This is a normal Python list of strings that may be changed by Python programs to configure the search path dynamically. To extend your search path within Python, do this:

```
import sys
sys.path.append('mydirpath')
```

Because shell variable settings are available to Python programs in the built-in `os.environ` dictionary, a Python script may also say something like `sys.path.append(os.environ['MYDIR'])` to add the directory named by the MYDIR shell variable to the Python module search path at runtime. Because `os.pathsep` gives the character

used to separate directory paths on your platform, and `string.split` knows how to split up strings around delimiters, this sequence:

```
import sys, os, string
path = os.environ['MYPYTHONPATH']
dirs = string.split(path, os.pathsep)
sys.path = sys.path + dirs
```

adds all names in the MYPYTHONPATH list setting to the module search path in the same way that Python usually does for PYTHONPATH. Such `sys.path` changes can be used to dynamically configure the module search path from within a script. They last only as long as the Python program or session that made them, though, so you are usually better off setting PYTHONPATH in most cases.

B.4 Running Python Programs

Python code can be typed at a `>>>` interactive prompt, run from a C program, or placed in text files and run. There is a variety of ways to run code in files:

Running from a command line

Python files can always be run by typing a command of the form `python file.py` in your system shell or console box, as long as the Python interpreter program is on your system's search path. On Windows, you can type this command in an MS-DOS console box; on Linux, use an xterm.

Running by clicking

Depending on your platform, you can usually start Python program files by double-clicking on their icons in a file explorer user interface. On Windows, for instance, `.py` Python files are automatically registered such that they can be run by being clicked (as are `.pyc` and `.pyw` files).

Running by importing and reloading

Files can also be run by importing them, either interactively or from within another module file. To rerun a module file's code again without exiting Python, be sure to run a call like `reload(module)`.

Running files in IDLE

For many, running a console window and one or more separate text editor windows constitutes an adequate Python development environment. For others, IDLE -- the Python Integrated Development Environment (but really named for Monty Python's Eric Idle) -- is a development environment GUI for Python. It can also be used to run existing program files or develop new systems from scratch. IDLE is written in Python/Tkinter, and thus is portable across Windows, X Windows (Unix), and Macintosh. It ships (for free) as a standard tool with the Python interpreter. On Windows, IDLE is installed automatically with Python; see [Section B.1.1](#) under [Section B.1](#) earlier in this appendix.

IDLE lets you edit, debug, and run Python programs. It does syntax coloring for edited Python code, sports an object browser that lets you step through your system's objects in parallel with its source code, and offers a point-and-click debugger interface for Python. See IDLE's help text and page at <http://www.python.org> for more details. Or simply play with it on your machine; most of its interfaces are intuitive and easy to learn. The only thing IDLE seems to lack today is a point-and-click GUI builder (but Tkinter's simplicity tends to make such builders less important in Python work).

Running files in Pythonwin

Pythonwin is another freely available, open source IDE for Python, but is targeted at Windows platforms only. It makes use of the MFC integration made available to Python programmers in the `win32all` Windows-specific Python extensions package described in [Chapter 15](#). In fact, Pythonwin is something of an example application of these Windows tools. Pythonwin supports source code editing and launching much like IDLE does (and there has been some cross-pollination between these systems). It doesn't sport all the features or portability of IDLE, but offers tools all its own for Windows developers. Fetch and install the `win32all` Windows extensions package to experiment with Pythonwin. You can also find this package on this book's CD (see <http://examples.oreilly.com/python2>).

Running Python from other IDEs

If you are accustomed to more sophisticated development environments, see the Visual Python products from Active State (<http://www.activestate.com>), and the PythonWorks products from PythonWare (<http://www.pythonware.com>). Both are emerging as I write this, and promise to provide advanced integrated development tool suites for Python programmers. For instance, ActiveState's plans include support for developing Python programs under both Microsoft's Visual Studio and the Mozilla application environment, as well as a Python port to the C#.NET environment. PythonWare's products support visual interface development and a suite of development tools.

Other platforms have additional ways to launch Python programs (e.g., dropping files on Mac icons). Here are a few more hints for Unix and Windows users:

Unix and Linux users

You can also make Python module files directly executable by adding the special `#!/usr/bin/python` type line at the top of the file and giving the file executable permissions with a `chmod` command. If you do, you can run Python files simply by typing their names (e.g., `file.py`), as though they were compiled executables. See [Chapter 2](#), for more details.

Windows users

If you see a flash when you click on a Python program in the file explorer, it's probably the console box that Python pops up to represent the program's standard input/output streams. If you want to keep the program's output up so that you can see it, add a call `raw_input()` to the bottom of your program; this call pauses until you press the Enter key. If you write your own GUI and don't want to see the console pop-up at all, name your source files with a `.pyw` extension instead of `.py`.

Windows NT and 2000

You can also launch a Python script file from a command-line prompt simply by typing the name of the script's file (e.g., `file.py`). These platforms correctly run the script with the Python interpreter without requiring the special `#!` first line needed to run files directly in Unix. To run Python command lines on Windows 9x platforms, you'll need to add the word "python" before the filename and make sure the `python.exe` executable is on your PATH setting (as described earlier). On all Windows platforms, you may also click on Python filenames in a Windows explorer to start them.

B.5 Python Internet Resources

Finally, [Table B-1](#) lists some of the most useful Internet sites for Python information and resources. Nearly all of these are accessible from Python's home page (<http://www.python.org>) and most are prone to change over time, so be sure to consult Python's home page for up-to-date details.

Table B-1. Python Internet Links

Resource	Address
Python's main web site	http://www.python.org
Python's FTP site	ftp://ftp.python.org/pub/python
Python's newsgroup	comp.lang.python (python-list@cwil.nl)
O'Reilly's main web site	http://www.oreilly.com
O'Reilly Python DevCenter	http://www.oreillynet.com/python
Book's web site	http://www.rmi.net/~lutz/about-pp2e.html
Author's web site	http://www.rmi.net/~lutz
Python support mail-list	mailto:python-help@python.org
Python online manuals	http://www.python.org/doc
Python online FAQ	http://www.python.org/doc/FAQ.html
Python special interest groups	http://www.python.org/sigs
Python resource searches	http://www.python.org/search
Starship (library)	http://starship.python.net
Vaults of Parnassus (library)	http://www.vex.net/parnassus
JPython's site	http://www.jython.org
SWIG's site	http://www.swig.org
Tk's site	http://www.scriptics.com
Zope's site	http://www.zope.org
ActiveState (tools)	http://www.activestate.com
PythonWare (tools)	http://www.pythonware.com

Appendix B. Pragmatics

This appendix is a very brief introduction to some install-level details of Python use, and contains a list of Python Internet resources. More information on topics not covered fully here can be found at other resources:

- For additional install details, consult the various README text files in the examples distribution on this book's CD (view CD-ROM content online at <http://examples.oreilly.com/python2>), as well as the README files and other documentation that accompany the Python distributions and other packages on the CD. In particular, the README files in the *Examples* and *Examples\PP2E* directories contain book example tree documentation and Python install details not repeated in this appendix.
- For more background information on running Python programs in general, see the Python manuals included on this book's CD, or the introductory-level O'Reilly text *Learning Python*.
- For more background information on the core Python language itself, refer to the Python standard manuals included on this book's CD, and the O'Reilly texts *Learning Python* and *Python Pocket Reference*.
- For more information about all things Python, see <http://www.python.org>. This site has online Python documentation, interpreter downloads, search engines, and links to just about every other relevant Python site on the Web. For links to information about this book, refer back to the Preface.

Appendix C. Python Versus C++

This appendix briefly summarizes some of the differences between Python and C++ classes. Python's `class` system can be thought of as a subset of C++'s. Although the comparison to Modula 3 may be closer, C++ is the dominant OOP language today. But in Python, things are intentionally simpler -- classes are simply objects with attached attributes that may have links to other class objects. They support generation of multiple instances, customization by attribute inheritance, and operator overloading, but the object model in Python is comparatively uncluttered. Here are some specific differences between Python and C++:

Attributes

There is no real distinction between data members and methods in Python; both simply designate named attributes of instances or classes, bound to functions or other kinds of objects. Attributes are names attached to objects, and accessed by qualification:

`object.attribute`. Methods are merely class attributes assigned to functions normally created with nested `def` statements; members are just attribute names assigned to other kinds of objects.

Class object generation

Class statements create class objects and assign them to a name. Statements that assign names within a `class` statement generate class attributes, and classes inherit attributes from all other classes listed in their `class` statement header line (multiple inheritance is supported; this is discussed in a moment).

Instance object creation

Calling a class object as though it were a function generates a new class instance object. An instance begins with an empty namespace that inherits names in the class's namespace; assignments to instance attributes (e.g., to `self` attributes within class method functions) create attributes in the instance.

Object deletion

Both classes and instances (and any data they embed) are automatically reclaimed when no longer held. There is no `new` (classes are called instead) and Python's `del` statement removes just one reference, unlike C++'s `delete`.

Member creation

Class and instance attributes, like simple variables, spring into existence when assigned, are not declared ahead of time, and may reference any type of object (they may even reference different object datatypes at different times).

Inheritance

Python inheritance is generally kicked off to search for an attribute name's value: given an expression of the form `object.attribute`, Python searches the namespace object

tree at `object` and above for the first appearance of `name` attribute. An inheritance search also occurs when expression operators and type operations are applied to objects. A new, independent inheritance search is performed for every `object.attribute` expression that is evaluated -- even `self.attr` expressions within a method function search anew for `attr` at the instance object referenced by `self` and above.

Runtime type information

Python classes are objects in memory at runtime -- they can be passed around a program to provide a sort of runtime type resource (e.g., a single function can generate instances of arbitrary classes passed in as an argument). Both class and instance objects carry interpreter information (e.g., a `__dict__` attribute dictionary), and Python's `type` function allows object type testing. Instance objects' `__class__` attributes reference the class they were created from, and class objects' `__bases__` attributes give class superclasses (base classes).

"this" pointer

Python's equivalent of the C++ `this` instance pointer is the first argument added to method function calls (and usually called `self` by convention). It is usually implicit in a call but is used explicitly in methods: there is no hidden instance scope for unqualified names. Python methods are just functions nested in a `class` statement that receive the implied instance objects in their leftmost parameters.

Virtual methods

In Python, all methods and data members are `virtual` in the C++ sense: there is no notion of a compile-time resolution of attributes based on an object's type. Every attribute qualification (`object.name`) is resolved at runtime, based on the qualified object's type.

Pure virtuals

Methods called by a superclass but not defined by it correspond to C++'s concept of "pure virtual" methods: methods that must be redefined in a subclass. Since Python is not statically compiled, there is no need for C++'s special syntax to declare this case. Calls to undefined methods raise a name error exception at runtime, which may or may not be caught with `try` statements.

Static members

There is no `static` class data declaration; instead, assignments nested in a `class` statement generate attribute names associated with the class and shared by all its instances.

Private members

There is no notion of true access restrictions for attributes; every member and method is public in the C++ sense. Attribute hiding is a matter of convention rather than syntax: C++'s `public`, `private`, and `protected` constraints don't apply (but see also the new `__x` class name localization feature in [Appendix A](#)).

Const interfaces

Objects may be immutable, but names are not -- there is no equivalent of C++'s `const` modifier. Nothing prevents a name or object from being changed in a method, and methods can change mutable arguments (the `self` object, for example). Convention and common sense replaces extra syntax.

Reference parameters

There is no direct analogue for C++'s reference parameters. Python methods may return multiple values in a tuple and can change passed-in objects if they're mutable (for instance, by assigning to an object's attributes or changing lists and dictionaries in place). But there is no aliasing between names at the call and names in a function header: arguments are passed by assignment, which creates shared object references.

Operator overloading

Special method names overload operators: there is no `operator+` -like syntax but the effects are similar. For instance, a class attribute named `__add__` overloads (intercepts and implements) application of the `+` operator to instances of the class; `__getattr__` is roughly like C++ `->` overloading. Arbitrary expressions require coding right-side methods (e.g., `__radd__`).

Templates

Python is dynamically typed -- names are references to arbitrary objects, and there is no notion of type declarations. C++ templates are neither applicable nor necessary. Python classes and functions can generally be applied to any object type that implements the interface protocols (operations and operators) expected by the class's code. Subjects need not be of a certain datatype.

Friends

Everything is friendly in Python. Because there is no notion of privacy constraints, any class can access the internals of another.

Function overloading

Python polymorphism is based on virtual method calls: the type of a qualified object determines what its methods do. Since Python arguments' types are never declared (dynamically typed), there is nothing like C++'s function overloading for dispatching to different versions of a function based on the datatypes of its arguments. You can explicitly test types and argument list lengths in methods instead of writing separate functions for each type combination (see `type` built-in function and `*args` function argument form).

Multiple inheritance

Multiple inheritance is coded by listing more than one superclass in parentheses in a class statement header line. When multiple inheritance is used, Python simply uses the first appearance of an attribute found during a depth-first, left-to-right search through

the superclass tree. Python resolves multiple inheritance conflicts this way instead of treating them as errors.

Virtual inheritance

C++'s notion of virtual base classes doesn't quite apply in Python. A Python class instance is a single namespace dictionary (with a class pointer for access to inherited attributes). Classes add attributes to the class instance dictionary by assignment. Because of this structure, each attribute exists in just one place -- the instance dictionary. For inherited class attributes, the search of the superclass tree resolves references unambiguously.

Constructors

Python only runs the one `__init__` method found by the inheritance object tree search. It doesn't run all accessible classes' constructors automatically; if needed, we have to call other class constructors manually. But this is no harder than specifying superclass constructor arguments in C++. Python destructors (`__del__`) run when an instance is garbage-collected (i.e., deallocated), not in response to delete calls.

Scope operators

C++ scope operators of the form `Superclass::Method` are used to extend inherited methods and disambiguate inheritance conflicts. Python's closest equivalent is `Superclass.Method`, a class object qualification. It isn't required for inheritance conflict resolution, but can be used to override the default search rule and to call back to superclasses in method extensions.

Method pointers

Instead of special syntax, Python method references are objects; they may be passed, stored in data structures, and so on. Method objects come in two flavors: bound methods (when an instance is known) are instance/method pairs called later like simple functions, and unbound methods are simply references to a method function object and require an instance to be passed explicitly when called.

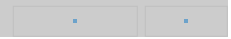
Naturally, Python has additional class features not found in C++, such as metaclass protocols: `__setattr__` can be used to implement alternative interfaces, and an instance's `__class__` pointer can be reset to change the class type of an object dynamically. Moreover, class attributes can be modified arbitrarily at runtime; classes are merely objects with attached attribute names.

In addition, Python differs from C++ in numerous ways besides its class model. For instance, there are neither type declarations nor compile and linking steps in Python; you cannot overload `=` in Python as you can in C++ (assignment isn't an operator in Python); and pointers, central to much C and C++ programming, are completely absent in Python (though object references can have some of the same effects). Instead of pointers, Python programs use first-class objects, which are automatically allocated and reclaimed.

Most of these differences stem from the fact that Python was designed for speed of development, not speed of execution; much of C++'s extra syntax would interfere with Python's purpose. See the O'Reilly text *Learning Python* for a complete introduction to Python

classes and the remainder of the core Python language.

I l@ve RuBoard



1.1 "And Now for Something Completely Different"

This book is about using Python, a very high-level, object-oriented, open source^[1] programming language, designed to optimize development speed. Although it is completely general-purpose, Python is often called an object-oriented scripting language, partly because of its sheer ease of use, and partly because it is commonly used to orchestrate or "glue" other software components in an application.

[1] Open source systems are sometimes called freeware, in that their source code is freely distributed and community-controlled. Don't let that concept fool you, though; with roughly half a million users in that community today, Python is very well supported.

If you are new to Python, chances are you've heard about the language somewhere, but are not quite sure what it is about. To help you get started, this chapter provides a nontechnical introduction to Python's features and roles. Most of it will make more sense once you have seen real Python programs, but let's first take a quick pass over the forest before wandering among the trees.

In the preface, I mentioned that Python emphasizes concepts such as quality, productivity, portability, and integration. Since these four terms summarize most of the reasons for using Python, I'd like to define them in a bit more detail:

Quality

Python makes it easy to write software that can be reused and maintained. It was deliberately designed to raise development quality expectations in the scripting world. Python's clear syntax and coherent design almost forces programmers to write readable code -- a critical feature for software that may be changed by others. The Python language really does look like it was designed, not accumulated. Python is also well tooled for modern software reuse methodologies. In fact, writing high-quality Python components that may be applied in multiple contexts is almost automatic.

Productivity

Python is optimized for speed of development. It's easy to write programs fast in Python, because the interpreter handles details you must code explicitly in lower-level languages. Things like type declarations, memory management, and build procedures are nowhere to be found in Python scripts. But fast initial development is only one component of productivity. In the real world, programmers must write code both for a computer to execute and for other programmers to read and maintain. Because Python's syntax resembles executable pseudocode, it yields programs that are easy to understand long after they have been written. In addition, Python supports (but does not impose) advanced paradigms such as object-oriented programming, which further boost developer productivity and shrink development schedules.

Portability

Most Python programs run without change on almost every computer system in use

today. In fact, Python programs run today on everything from IBM mainframes and Cray Supercomputers to notebook PCs and handheld PDAs. Although some platforms offer nonportable extensions, the core Python language and libraries are platform-neutral. For instance, most Python scripts developed on Linux will generally run on Windows immediately, and vice versa -- simply copy the script over. Moreover, a graphical user interface (GUI) program written with Python's standard Tkinter library will run on the X Windows system, Microsoft Windows, and the Macintosh, with native look-and-feel on each, and without modifying the program's source code at all.

Integration

Python is designed to be integrated with other tools. Programs written in Python can be easily mixed with and script (i.e., direct) other components of a system. Today, for example, Python scripts can call out to existing C and C++ libraries, talk to Java classes, integrate with COM and CORBA components, and more. In addition, programs written in other languages can just as easily run Python scripts by calling C and Java API functions, accessing Python-coded COM servers, and so on. Python is not a closed box.

In an era of increasingly short development schedules, faster machines, and heterogeneous applications, these strengths have proven to be powerful allies in both small and large development projects. Naturally, there are other aspects of Python that attract developers, such as its simple learning curve for developers and users alike, libraries of precoded tools to minimize up-front development, and completely free nature that cuts product development and deployment costs.

But Python's productivity focus is perhaps its most attractive and defining quality. As I write this, the main problem facing the software development world is not just writing programs quickly, but finding developers with time to write programs at all. Developers' time has become paramount -- much more critical than execution speed. There are simply more projects than programmers to staff them.

As a language optimized for developer productivity, Python seems to be the right answer to the questions being asked by the development world. Not only can Python developers implement systems quickly, but the resulting systems will be maintainable, portable, and easily integrated with other application components.

1.2 The Life of Python

Python was invented around 1990 by Guido van Rossum, when he was at CWI in Amsterdam. Despite the reptiles, it is named after the BBC comedy series Monty Python's Flying Circus, of which Guido is a fan (see the following silly sidebar). Guido was also involved with the Amoeba distributed operating system and the ABC language. In fact, the original motivation for Python was to create an advanced scripting language for the Amoeba system.

But Python's design turned out to be general enough to address a wide variety of domains. It's now used by hundreds of thousands of engineers around the world, in increasingly diverse roles. Companies use Python today in commercial products, for tasks such as testing chips and boards, developing GUIs, searching the Web, animating movies, scripting games, serving up maps and email on the Internet, customizing C++ class libraries, and much more.^[2] In fact, because Python is a completely general-purpose language, its target domains are only limited by the scope of computers in general.

[2] See the preface for more examples of companies using Python in these ways, and see <http://www.python.org> for a more comprehensive list of commercial applications.

Since it first appeared on the public domain scene in 1991, Python has continued to attract a loyal following, and spawned a dedicated Internet newsgroup, comp.lang.python, in 1994. And as the first edition of this book was being written in 1995, Python's home page debuted on the WWW at <http://www.python.org> -- still the official place to find all things Python.

What's in a Name?

Python gets its name from the 1970s British TV comedy series, Monty Python's Flying Circus. According to Python folklore, Guido van Rossum, Python's creator, was watching reruns of the show at about the same time he needed a name for a new language he was developing. And, as they say in show business, "the rest is history."

Because of this heritage, references to the comedy group's work often show up in examples and discussion. For instance, the name "Spam" has a special connotation to Python users, and confrontations are sometimes referred to as "The Spanish Inquisition." As a rule, if a Python user starts using phrases that have no relation to reality, they're probably borrowed from the Monty Python series or movies. Some of these phrases might even pop up in this book. You don't have to run out and rent *The Meaning of Life* or *The Holy Grail* to do useful work in Python, of course, but it can't hurt.

While "Python" turned out to be a distinctive name, it's also had some interesting side effects. For instance, when the Python newsgroup, comp.lang.python, came online in 1994, its first few weeks of activity were almost entirely taken up by people wanting to discuss topics from the TV show. More recently, a special Python supplement in the Linux Journal magazine featured photos of Guido garbed in an

obligatory "nice red uniform."

There's still an occasional post from fans of the show on Python's news list. For instance, one poster innocently offered to swap Monty Python scripts with other fans. Had he known the nature of the forum, he might have at least mentioned whether they ran under DOS or Unix.

To help manage Python's growth, organizations aimed at supporting Python developers have taken shape over the years: among them, Python Software Activity (PSA) was formed to help facilitate Python conferences and web sites, and the Python Consortium was formed by organizations interested in helping to foster Python's growth. Although the future of the PSA is unclear as I write these words, it has helped to support Python through the early years.

Today, Guido and a handful of other key Python developers, are employed by a company named Digital Creations to do Python development on a full-time basis. Digital Creations, based in Virginia, is also home to the Python-based Zope web application toolkit (see <http://www.zope.org>). However, the Python language is owned and managed by an independent body, and remains a true open source, community-driven system.

Other companies have Python efforts underway as well. For instance, ActiveState and PythonWare develop Python tools, O'Reilly (the publisher of this book) and a company named Foretech both organize annual Python conferences, and O'Reilly manages a supplemental Python web site (see the O'Reilly Network's Python DevCenter at <http://www.oreillynet.com/python>). The O'Reilly Python Conference is held as part of the annual Open Source Software Convention. Although the world of professional organizations and companies changes more frequently than do published books, it seems certain that the Python language will continue to meet the needs of its user community.

1.3 The Compulsory Features List

One way to describe a language is by listing its features. Of course, this will be more meaningful after you've seen Python in action; the best I can do now is speak in the abstract. And it's really how Python's features work together, that make it what it is. But looking at some of Python's attributes may help define it; [Table 1-1](#) lists some of the common reasons cited for Python's appeal.

Table 1-1. Python Language Features

Features	Benefits
No compile or link steps	Rapid development cycle turnaround
No type declarations	Simpler, shorter, and more flexible programs
Automatic memory management	Garbage collection avoids bookkeeping code
High-level datatypes and operations	Fast development using built-in object types
Object-oriented programming	Code reuse, C++, Java, and COM integration
Embedding and extending in C	Optimization, customization, system "glue"
Classes, modules, exceptions	Modular "programming-in-the-large" support
A simple, clear syntax and design	Readability, maintainability, ease of learning
Dynamic loading of C modules	Simplified extensions, smaller binary files
Dynamic reloading of Python modules	Programs can be modified without stopping
Universal "first-class" object model	Fewer restrictions and special-case rules
Runtime program construction	Handles unforeseen needs, end-user coding
Interactive, dynamic nature	Incremental development and testing
Access to interpreter information	Metaprogramming, introspective objects
Wide interpreter portability	Cross-platform programming without ports
Compilation to portable bytecode	Execution speed, protecting source code
Standard portable GUI framework	Tkinter scripts run on X, Windows, and Macs
Standard Internet protocol support	Easy access to email, FTP, HTTP, CGI, etc.
Standard portable system calls	Platform-neutral system scripting
Built-in and third-party libraries	Vast collection of precoded software components
True open source software	May be freely embedded and shipped

To be fair, Python is really a conglomeration of features borrowed from other languages. It includes elements taken from C, C++, Modula-3, ABC, Icon, and others. For instance, Python's modules came from Modula, and its slicing operation from Icon (as far as anyone can seem to remember, at least). And because of Guido's background, Python borrows many of ABC's ideas, but adds practical features of its own, such as support for C-coded extensions.

1.4 What's Python Good For?

Because Python is used in a wide variety of ways, it's almost impossible to give an authoritative answer to this question. In general, any application that can benefit from the inclusion of a language optimized for speed of development is a good target Python application domain. Given the ever-shrinking schedules in software development, this a very broad category.

A more specific answer is less easy to formulate. For instance, some use Python as an embedded extension language, while others use it exclusively as a standalone programming tool. And to some extent, this entire book will answer this very question -- it explores some of Python's most common roles. For now, here's a summary of some of the more common ways Python is being applied today:

System utilities

- Portable command-line tools, testing systems

Internet scripting

- CGI web sites, Java applets, XML, ASP, email tools

Graphical user interfaces

- With APIs such as Tk, MFC, Gnome, KDE

Component integration

- C/C++ library front-ends, product customization

Database access

- Persistent object stores, SQL database system interfaces

Distributed programming

- With client/server APIs like CORBA, COM

Rapid-prototyping /development

- Throwaway or deliverable prototypes

Language-based modules

- Replacing special-purpose parsers with Python

And more

- Image processing, numeric programming, AI, etc.

"Buses Considered Harmful"

The PSA organization described earlier was originally formed in response to an early thread on the Python newsgroup, which posed the semiserious question: "What would happen if Guido was hit by a bus?"

These days, Guido van Rossum is still the ultimate arbiter of proposed Python changes, but Python's user base helps support the language, work on extensions, fix bugs, and so on. In fact, Python development is now a completely open process -- anyone can inspect the latest source-code files or submit patches by visiting a web site (see <http://www.python.org> for details).

As an open source package, Python development is really in the hands of a very large cast of developers working in concert around the world. Given Python's popularity, bus attacks seem less threatening now than they once did; of course, I can't speak for Guido.

On the other hand, Python is not really tied to any particular application area at all. For example, Python's integration support makes it useful for almost any system that can benefit from a frontend, programmable interface. In abstract terms, Python provides services that span domains. It is:

- A dynamic programming language, for situations in which a compile/link step is either impossible (on-site customization), or inconvenient (prototyping, rapid development, system utilities)
- A powerful but simple programming language designed for development speed, for situations in which the complexity of larger languages can be a liability (prototyping, end-user coding)
- A generalized language tool, for situations where we might otherwise need to invent and implement yet another "little language" (programmable system interfaces, configuration tools)

Given these general properties, Python can be applied to any area we're interested in by extending it with domain libraries, embedding it in an application, or using it all by itself. For instance, Python's role as a system tools language is due as much to its built-in interfaces to operating system services as to the language itself. In fact, because Python was built with integration in mind, it has naturally given rise to a growing library of extensions and tools, available as off-the-shelf components to Python developers. [Table 1-2](#) names just a few; you can find more about most of these components in this book or on Python's web site.

Table 1-2. A Few Popular Python Tools and Extensions

Domain	Extensions
Systems programming	Sockets, threads, signals, pipes, RPC calls, POSIX bindings
Graphical user interfaces	Tk, PMW, MFC, X11, wxPython, KDE, Gnome
Database interfaces	Oracle, Sybase, PostGRES, mSQL, persistence, dbm
Microsoft Windows tools	MFC, COM, ActiveX, ASP, ODBC, .NET
Internet tools	JPython, CGI tools, HTML/XML parsers, email tools, Zope

Distributed objects	DCOM, CORBA, ILU, Fnorb
Other popular tools	SWIG, PIL, regular expressions, NumPy, cryptography

1.5 What's Python Not Good For?

To be fair again, some tasks are outside of Python's scope. Like all dynamic languages, Python (as currently implemented) isn't as fast or efficient as static, compiled languages like C. In many domains, the difference doesn't matter; for programs that spend most of their time interacting with users or transferring data over networks, Python is usually more than adequate to meet the performance needs of the entire application. But efficiency is still a priority in some domains.

Because it is interpreted today,^[3] Python alone usually isn't the best tool for delivery of performance-critical components. Instead, computationally intensive operations can be implemented as compiled extensions to Python, and coded in a low-level language like C. Python can't be used as the sole implementation language for such components, but it works well as a frontend scripting interface to them.

[3] Python is "interpreted" in the same way that Java is: Python source code is automatically compiled (translated) to an intermediate form called "bytecode," which is then executed by the Python virtual machine (that is, the Python runtime system). This makes Python scripts more portable and faster than a pure interpreter that runs raw source code or trees. But it also makes Python slower than true compilers that translate source code to binary machine code for the local CPU. Keep in mind, though, that some of these details are specific to the standard Python implementation; the JPython (a.k.a. "Jython") port compiles Python scripts to Java bytecode, and the new C#.NET port compiles Python scripts to binary *.exe* files. An optimizing Python compiler might make most of the performance cautions in this chapter invalid (we can hope).

For example, numerical programming and image processing support has been added to Python by combining optimized extensions with a Python language interface. In such a system, once the optimized extensions have been developed, most of the programming occurs at the higher-level Python scripting level. The net result is a numerical programming tool that's both efficient and easy to use.

Moreover, Python can still serve as a prototyping tool in such domains. Systems may be implemented in Python first, and later moved in whole or piecemeal to a language like C for delivery. C and Python have distinct strengths and roles; a hybrid approach, using C for compute-intensive modules, and Python for prototyping and frontend interfaces, can leverage the benefits of both.

In some sense, Python solves the efficiency/flexibility tradeoff by not solving it at all. It provides a language optimized for ease of use, along with tools needed to integrate with other languages. By combining components written in Python and compiled languages like C and C++, developers may select an appropriate mix of usability and performance for each particular application. While it's unlikely that it will ever be as fast as C, Python's speed of development is at least as important as C's speed of execution in most modern software projects.

On Truth in Advertising

In this book's conclusion we will return to some of the bigger ideas introduced in this chapter, after we've had a chance to study Python in action. I want to point out up front, though, that my background is in Computer Science, not marketing. I plan to be brutally honest in this book, both about Python's features and its downsides. Despite the fact that Python is one of the most easy-to-use programming languages ever created, there are indeed some pitfalls, which we will examine in this book.

Let's start now. Perhaps the biggest pitfall you should know about is this one: Python makes it incredibly easy to throw together a bad design quickly. It's a genuine problem. Because developing programs in Python is so simple and fast compared to traditional languages, it's easy to get wrapped up in the act of programming itself, and pay less attention to the problem you are really trying to solve.

In fact, Python can be downright seductive -- so much so that you may need to consciously resist the temptation to quickly implement a program in Python that works, and is arguably "cool," but leaves you as far from a maintainable implementation of your original conception as you were when you started. The natural delays built in to compiled language development -- fixing compiler error messages, linking libraries, and the like -- aren't there to apply the brakes in Python work.

This isn't necessarily all bad. In most cases, the early designs that you throw together fast are stepping stones to better designs that you later keep. But be warned: even with a rapid development language like Python, there is no substitute for brains -- it's always best to think before you start typing code. To date, at least, no computer programming language has managed to make intelligence obsolete.

Chapter 1. Introducing Python

[Section 1.1. "And Now for Something Completely Different"](#)

[Section 1.2. The Life of Python](#)

[Section 1.3. The Compulsory Features List](#)

[Section 1.4. What's Python Good For?](#)

[Section 1.5. What's Python Not Good For?](#)

10.1 "Tune in, Log on, and Drop out"

In the last few years, the Internet has virtually exploded onto the mainstream stage. It has rapidly grown from a simple communication device used primarily by academics and researchers into a medium that is now nearly as pervasive as the television and telephone. Social observers have likened the Internet's cultural impact to that of the printing press, and technical observers have suggested that all new software development of interest occurs only on the Internet. Naturally, time will be the final arbiter for such claims, but there is little doubt that the Internet is a major force in society, and one of the main application contexts for modern software systems.

The Internet also happens to be one of the primary application domains for the Python programming language. Given Python and a computer with a socket-based Internet connection, we can write Python scripts to read and send email around the world, fetch web pages from remote sites, transfer files by FTP, program interactive web sites, parse HTML and XML files, and much more, simply by using the Internet modules that ship with Python as standard tools.

In fact, companies all over the world do: Yahoo, Infoseek, Hewlett-Packard, and many others rely on Python's standard tools to power their commercial web sites. Many also build and manage their sites with the Zope web application server, which is itself written and customizable in Python. Others use Python to script Java web applications with JPython (a.k.a. "Jython") -- a system that compiles Python programs to Java bytecode, and exports Java libraries for use in Python scripts.

As the Internet has grown, so too has Python's role as an Internet tool. Python has proven to be well-suited to Internet scripting for some of the very same reasons that make it ideal in other domains. Its modular design and rapid turnaround mix well with the intense demands of Internet development. In this part of the book, we'll find that Python does more than simply support Internet scripts; it also fosters qualities such as productivity and maintainability that are essential to Internet projects of all shapes and sizes.

10.1.1 Internet Scripting Topics

Internet programming entails many topics, so to make the presentation easier to digest, I've split this subject over the next six chapters of this book. This chapter introduces Internet fundamentals and explores sockets, the underlying communications mechanism of the Internet. From there, later chapters move on to discuss the client, the server, web site construction, and more advanced topics.

Each chapter assumes you've read the previous one, but you can generally skip around, especially if you have any experience in the Internet domain. Since these chapters represent a big portion (about a third) of this book at large, the following sections go into a few more details about what we'll be studying.

10.1.1.1 What we will cover

In conceptual terms, the Internet can roughly be thought of as being composed of multiple functional layers:

Low-level networking layers

Mechanisms such as the TCP/IP transport mechanism, which deal with transferring bytes between machines, but don't care what they mean

Sockets

The programmer's interface to the network, which runs on top of physical networking layers like TCP/IP

Higher-level protocols

Structured communication schemes such as FTP and email, which run on top of sockets and define message formats and standard addresses

Server-side web scripting (CGI)

Higher-level client/server communication protocols between web browsers and web servers, which also run on top of sockets

Higher-level frameworks and tools

Third-party systems such as Zope and JPython, which address much larger problem domains

In this chapter and [Chapter 11](#), our main focus is on programming the second and third layers: sockets and higher-level protocols. We'll start this chapter at the bottom, learning about the socket model of network programming. Sockets aren't strictly tied to Internet scripting, but they are presented here because this is their primary role. As we'll see, most of what happens on the Internet happens through sockets, whether you notice or not.

After introducing sockets, the next chapter makes its way up to Python's client-side interfaces to higher-level protocols -- things like email and FTP transfers, which run on top of sockets. It turns out that a lot can be done with Python on the client alone, and [Chapter 11](#) will sample the flavor of Python client-side scripting. The next three chapters then go on to present server-side scripting (programs that run on a server computer and are usually invoked by a web browser). Finally, the last chapter in this part, [Chapter 15](#), briefly introduces even higher-level tools such as JPython and Zope.

Along the way, we will also put to work some of the operating-system and GUI interfaces we've studied earlier (e.g., processes, threads, signals, and Tkinter), and investigate some of the design choices and challenges that the Internet presents.

That last statement merits a few more words. Internet scripting, like GUIs, is one of the sexier application domains for Python. As in GUI work, there is an intangible but instant gratification in seeing a Python Internet program ship information all over the world. On the other hand, by its very nature, network programming imposes speed overheads and user interface limitations. Though it may not be a fashionable stance these days, some applications are still better off not being deployed on the Net. In this part of the book, we will take an honest look at the Net's trade-offs as they arise.

The Internet is also considered by many to be something of an ultimate proof of concept for open source tools. Indeed, much of the Net runs on top of a large number of tools, such as Python, Perl, the Apache web server, the sendmail program, and Linux. Moreover, new tools and technologies for programming the Web sometimes seem to appear faster than developers can absorb.

The good news is that Python's integration focus makes it a natural in such a heterogeneous world. Today, Python programs can be installed as client-side and server-side tools, embedded within HTML code, used as applets and servlets in Java applications, mixed into distributed object systems like CORBA and DCOM, integrated with XML-coded objects, and more. In more general terms, the rationale for using Python in the Internet domain is exactly the same as in any other: Python's emphasis on productivity, portability, and integration make it ideal for writing Internet programs that are open, maintainable, and delivered according to the ever-shrinking schedules in this field.

10.1.1.2 What we won't cover

Now that I've told you what we will cover in this book, I should also mention what we won't cover. Like Tkinter, the Internet is a large domain, and this part of the book is mostly an introduction to core concepts and representative tasks, not an exhaustive reference. There are simply too many Python Internet modules to include each in this text, but the examples here should help you understand the library manual entries for modules we don't have time to cover.

I also want to point out that higher-level tools like JPython and Zope are large systems in their own right, and they are best dealt with in more dedicated documents. Because books on both topics are likely to appear soon, we'll merely scratch their surfaces here. Moreover, this book says almost nothing about lower-level networking layers such as TCP/IP. If you're curious about what happens on the Internet at the bit-and-wire level, consult a good networking text for more details.

10.1.1.3 Running examples in this part of the book

Internet scripts generally imply execution contexts that earlier examples in this book have not. That is, it usually takes a bit more to run programs that talk over networks. Here are a few pragmatic notes about this part's examples up front:

- You don't need to download extra packages to run examples in this part of the book. Except in [Chapter 15](#), all of the examples we'll see are based on the standard set of Internet-related modules that come with Python (they are installed in Python's library directory).

- You don't need a state-of-the-art network link or an account on a web server to run most of the examples in this and the following chapters; a PC and dial-up Internet account will usually suffice. We'll study configuration details along the way, but client-side programs are fairly simple to run.
- You don't need an account on a web server machine to run the server-side scripts in later chapters (they can be run by any web browser connected to the Net), but you will need such an account to change these scripts.

When a Python script opens an Internet connection (with the `socket` or protocol modules), Python will happily use whatever Internet link exists on your machine, be that a dedicated T1 line, a DSL line, or a simple modem. For instance, opening a socket on a Windows PC automatically initiates processing to create a dial-up connection to your Internet Service Provider if needed (on my laptop, a Windows modem connection dialog automatically pops up). In other words, if you have a way to connect to the Net, you likely can run programs in this chapter.

Moreover, as long as your machine supports sockets, you probably can run many of the examples here even if you have no Internet connection at all. As we'll see, a machine name "localhost" or "" usually means the local computer itself. This allows you to test both the client and server sides of a dialog on the same computer without connecting to the Net. For example, you can run both socket-based clients and servers locally on a Windows PC without ever going out to the Net.

Some later examples assume that a particular kind of server is running on a server machine (e.g., FTP, POP, SMTP), but client-side scripts work on any Internet-aware machine with Python installed. Server-side examples in [Chapter 12](#), [Chapter 13](#), and [Chapter 14](#) require more: you'll need a web server account to code CGI scripts, and you must download advanced third-party systems like JPython and Zope separately (or find them by viewing <http://examples.oreilly.com/python2>).

In the Beginning There Was Grail

Besides creating the Python language, Guido van Rossum also wrote a World Wide Web browser in Python, named (appropriately enough) Grail. Grail was partly developed as a demonstration of Python's capabilities. It allows users to browse the Web much like Netscape or Internet Explorer, but can also be programmed with Grail applets -- Python/Tkinter programs downloaded from a server when accessed and run on the client by the browser. Grail applets work much like Java applets in more widespread browsers (more on applets in [Chapter 15](#)).

Grail is no longer under development and is mostly used for research purposes today. But Python still reaps the benefits of the Grail project, in the form of a rich set of Internet tools. To write a full-featured web browser, you need to support a wide variety of Internet protocols, and Guido packaged support for all of these as standard library modules that are now shipped with the Python language.

Because of this legacy, Python now includes standard support for Usenet news (NNTP), email processing (POP, SMTP, IMAP), file transfers (FTP), web pages and interactions (HTTP, URLs, HTML, CGI), and other commonly used protocols (Telnet, Gopher, etc.). Python scripts can connect to all of these Internet components by simply importing the associated library module.

Since Grail, additional tools have been added to Python's library for parsing XML files, OpenSSL secure sockets, and more. But much of Python's Internet support can be traced back to the Grail browser -- another example of Python's support for code reuse at work. At this writing, you can still find the Grail at <http://www.python.org>.

10.2 Plumbing the Internet

Unless you've been living in a cave for the last decade, you are probably already familiar with what the Internet is about, at least from a user's perspective. Functionally, we use it as a communication and information medium, by exchanging email, browsing web pages, transferring files, and so on. Technically, the Internet consists of many layers of abstraction and device -- from the actual wires used to send bits across the world to the web browser that grabs and renders those bits into text, graphics, and audio on your computer.

In this book, we are primarily concerned with the programmer's interface to the Internet. This too consists of multiple layers: sockets, which are programmable interfaces to the low-level connections between machines, and standard protocols, which add structure to discussions carried out over sockets. Let's briefly look at each of these layers in the abstract before jumping into programming details.

10.2.1 The Socket Layer

In simple terms, sockets are a programmable interface to network connections between computers. They also form the basis, and low-level "plumbing," of the Internet itself: all of the familiar higher-level Net protocols like FTP, web pages, and email, ultimately occur over sockets. Sockets are also sometimes called communications endpoints because they are the portals through which programs send and receive bytes during a conversation.

To programmers, sockets take the form of a handful of calls available in a library. These socket calls know how to send bytes between machines, using lower-level operations such as the TCP network transmission control protocol. At the bottom, TCP knows how to transfer bytes, but doesn't care what those bytes mean. For the purposes of this text, we will generally ignore how bytes sent to sockets are physically transferred. To understand sockets fully, though, we need to know a bit about how computers are named.

10.2.1.1 Machine identifiers

Suppose for just a moment that you wish to have a telephone conversation with someone halfway across the world. In the real world, you would probably either need that person's telephone number, or a directory that can be used to look up the number from his or her name (e.g., a telephone book). The same is true on the Internet: before a script can have a conversation with another computer somewhere in cyberspace, it must first know that other computer's number or name.

Luckily, the Internet defines standard ways to name both a remote machine, and a service provided by that machine. Within a script, the computer program to be contacted through a socket is identified by supplying a pair of values -- the machine name, and a specific port number on that machine:

Machine names

A machine name may take the form of either a string of numbers separated by dots called an IP address (e.g., 166.93.218.100), or a more legible form known as a domain name (e.g., starship.python.net). Domain names are automatically mapped into their dotted numeric address equivalent when used, by something called a domain name server -- a program on the Net that serves the same purpose as your local telephone directory assistance service.

Port numbers

A port number is simply an agreed-upon numeric identifier for a given conversation. Because computers on the Net can support a variety of services, port numbers are used to name a particular conversation on a given machine. For two machines to talk over the Net, both must associate sockets with the same machine name and port number when initiating network connections.

The combination of a machine name and a port number uniquely identifies every dialog on the Net. For instance, an Internet Service Provider's computer may provide many kinds of services for customers -- web pages, Telnet, FTP transfers, email, and so on. Each service on the machine is assigned a unique port number to which requests may be sent. To get web pages from a web server, programs need to specify both the web server's IP or domain name, and the port number on which the server listens for web page requests.

If this all sounds a bit strange, it may help to think of it in old-fashioned terms. In order to have a telephone conversation with someone within a company, for example, you usually need to dial both the company's phone number, as well as the extension of the person you want to reach. Moreover, if you don't know the company's number, you can probably find it by looking up the company's name in a phone book. It's almost the same on the Net -- machine names identify a collection of services (like a company), port numbers identify an individual service within a particular machine (like an extension), and domain names are mapped to IP numbers by domain name servers (like a phone book).

When programs use sockets to communicate in specialized ways with another machine (or with other processes on the same machine), they need to avoid using a port number reserved by a standard protocol -- numbers in the range of 0-1023 -- but we first need to discuss protocols to understand why.

10.2.2 The Protocol Layer

Although sockets form the backbone of the Internet, much of the activity that happens on the Net is programmed with protocols,^[1] which are higher-level message models that run on top of sockets. In short, Internet protocols define a structured way to talk over sockets. They generally standardize both message formats and socket port numbers:

[1] Some books also use the term protocol to refer to lower-level transport schemes such as TCP. In this book, we use protocol to refer to higher-level structures built on top of sockets; see a networking text if you are curious about what happens at lower levels.

- Message formats provide structure for the bytes exchanged over sockets during conversations.

- Port numbers are reserved numeric identifiers for the underlying sockets over which messages are exchanged.

Raw sockets are still commonly used in many systems, but it is perhaps more common (and generally easier) to communicate with one of the standard higher-level Internet protocols.

10.2.2.1 Port number rules

Technically speaking, socket port numbers can be any 16-bit integer value between and 65,535. However, to make it easier for programs to locate the standard protocols, port numbers in the range of 0-1023 are reserved and preassigned to the standard higher-level protocols. [Table 10-1](#) lists the ports reserved for many of the standard protocols; each gets one or more preassigned numbers from the reserved range.

Table 10-1. Port Numbers Reserved for Common Protocols

Protocol	Common Function	Port Number	Python Module
HTTP	Web pages	80	httplib
NNTP	Usenet news	119	nntplib
FTP data default	File transfers	20	ftplib
FTP control	File transfers	21	ftplib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Finger	Informational	79	n/a
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib

10.2.2.2 Clients and servers

To socket programmers, the standard protocols mean that port numbers 0-1023 are off-limits to scripts, unless they really mean to use one of the higher-level protocols. This is both by standard and by common sense. A Telnet program, for instance, can start a dialog with any Telnet-capable machine by connecting to its port 23; without preassigned port numbers, each server might install Telnet on a different port. Similarly, web sites listen for page requests from browsers on port 80 by standard; if they did not, you might have to know and type the HTTP port number of every site you visit while surfing the Net.

By defining standard port numbers for services, the Net naturally gives rise to a client/server architecture. On one side of a conversation, machines that support standard protocols run a set of perpetually running programs that listen for connection requests on the reserved ports. On the other end of a dialog, other machines contact those programs to use the services they export.

We usually call the perpetually running listener program a server and the connecting program a client. Let's use the familiar web browsing model as an example. As shown in [Table 10-1](#), the HTTP protocol used by the Web allows clients and servers to talk over sockets on port 80:

Server

A machine that hosts web sites usually runs a web server program that constantly listens for incoming connection requests, on a socket bound to port 80. Often, the server itself does nothing but watch for requests on its port perpetually; handling requests is delegated to spawned processes or threads.

Clients

Programs that wish to talk to this server specify the server machine's name and port 80 to initiate a connection. For web servers, typical clients are web browsers like Internet Explorer or Netscape, but any script can open a client-side connection on port 80 to fetch web pages from the server.

In general, many clients may connect to a server over sockets, whether it implements a standard protocol or something more specific to a given application. And in some applications, the notion of client and server is blurred -- programs can also pass bytes between each other more as peers than as master and subordinate. For the purpose of this book, though, we usually call programs that listen on sockets servers, and those that connect, clients. We also sometimes call the machines that these programs run on server and client (e.g., a computer on which a web server program runs may be called a web server machine, too), but this has more to do with the physical than the functional.

10.2.2.3 Protocol structures

Functionally, protocols may accomplish a familiar task like reading email or posting a Usenet newsgroup message, but they ultimately consist of message bytes sent over sockets. The structure of those message bytes varies from protocol to protocol, is hidden by the Python library, and is mostly beyond the scope of this book, but a few general words may help demystify the protocol layer.

Some protocols may define the contents of messages sent over sockets; others may specify the sequence of control messages exchanged during conversations. By defining regular patterns of communication, protocols make communication more robust. They can also minimize deadlock conditions -- machines waiting for messages that never arrive.

For example, the FTP protocol prevents deadlock by conversing over two sockets: one for control messages only, and one to transfer file data. An FTP server listens for control messages (e.g., "send me a file") on one port, and transfers file data over another. FTP clients open socket connections to the server machine's control port, send requests, and send or receive file data over a socket connected to a data port on the server machine. FTP also defines standard message structures passed between client and server. The control message used to request a file, for instance, must follow a standard format.

10.2.3 Python's Internet Library Modules

If all of this sounds horribly complex, cheer up: Python's standard protocol modules handle all the details. For example, the Python library's `ftplib` module manages all the socket and message-level handshaking implied by the FTP protocol. Scripts that import `ftplib` have access to a much higher-level interface for FTPing files and can be largely ignorant of both the underlying FTP protocol, and the sockets over which it runs. [2]

[2] Since Python is an open source system, you can read the source code of the `ftplib` module if you are curious about how the underlying protocol actually works. See file `ftplib.py` in the standard source library directory in your machine. Its code is complex (since it must format messages and manage two sockets), but with the other standard Internet protocol modules, it is a good example of low-level socket programming.

In fact, each supported protocol is represented by a standard Python module file with a name of the form `xxxlib.py`, where `xxx` is replaced by the protocol's name, in lowercase. The last column in [Table 10-1](#) gives the module name for protocol standard modules. For instance, FTP is supported by module file `ftplib.py`. Moreover, within the protocol modules, the top-level interface object is usually the name of the protocol. So, for instance, to start an FTP session in a Python script, you run `import ftplib` and pass appropriate parameters in a call to `ftplib.FTP()`; for Telnet, create a `telnetlib.Telnet()`.

In addition to the protocol implementation modules in [Table 10-1](#), Python's standard library also contains modules for parsing and handling data once it has been transferred over sockets or protocols. [Table 10-2](#) lists some of the more commonly used modules in this category.

Table 10-2. Common Internet-Related Standard Modules

Python Modules	Utility
<code>socket</code>	Low-level network communications support (TCP/IP, UDP, etc.).
<code>cgi</code>	Server-side CGI script support: parse input stream, escape HTML text, etc.
<code>urllib</code>	Fetch web pages from their addresses (URLs), escape URL text
<code>httplib</code> , <code>ftplib</code> , <code>nntplib</code>	HTTP (web), FTP (file transfer), and NNTP (news) protocol modules
<code>poplib</code> , <code>imaplib</code> , <code>smtplib</code>	POP, IMAP (mail fetch), and SMTP (mail send) protocol modules
<code>telnetlib</code> , <code>gopherlib</code>	Telnet and Gopher protocol modules
<code>htmllib</code> , <code>sgmlib</code> , <code>xmllib</code>	Parse web page contents (HTML, SGML, and XML documents)
<code>xdrlib</code>	Encode binary data portably (also see the <code>struct</code> and <code>socket</code> modules)
<code>rfc822</code>	Parse email-style header lines
<code>mhlib</code> , <code>mailbox</code>	Process complex mail messages and mailboxes
<code>mimertools</code> , <code>mimify</code>	Handle MIME-style message bodies
<code>multifile</code>	Read messages with multiple parts
<code>uu</code> , <code>binhex</code> , <code>base64</code> , <code>binascii</code> , <code>quopri</code>	Encode and decode binary (or other) data transmitted as text
<code>urlparse</code>	Parse URL string into components
<code>SocketServer</code>	Framework for general net servers
<code>BaseHTTPServer</code>	Basic HTTP server implementation
<code>SimpleHTTPServer</code> , <code>CGIHTTPServer</code>	Specific HTTP web server request handler modules

rexec, bastion

Restricted code execution modes

We will meet many of this table's modules in the next few chapters of this book, but not all. The modules demonstrated are representative, but as always, be sure to see Python's standard Library Reference Manual for more complete and up-to-date lists and details.

More on Protocol Standards

If you want the full story on protocols and ports, at this writing you can find a comprehensive list of all ports reserved for protocols, or registered as used by various common systems, by searching the web pages maintained by the Internet Engineering Task Force (IETF) and the Internet Assigned Numbers Authority (IANA). The IETF is the organization responsible for maintaining web protocols and standards. The IANA is the central coordinator for the assignment of unique parameter values for Internet protocols. Another standards body, the W3 (for WWW), also maintains relevant documents. See these web pages for more details:

<http://www.ietf.org>

<http://www.iana.org/numbers.html>

<http://www.iana.org/assignments/port-numbers>

<http://www.w3.org>

It's not impossible that more recent repositories for standard protocol specifications will arise during this book's shelf-life, but the IETF web site will likely be the main authority for some time to come. If you do look, though, be warned that the details are, well, detailed. Because Python's protocol modules hide most of the socket and messaging complexity documented in the protocol standards, you usually don't need to memorize these documents to get web work done in Python.

10.3 Socket Programming

Now that we've seen how sockets figure into the Internet picture, let's move on to explore the tools that Python provides for programming sockets with Python scripts. This section shows you how to use the Python socket interface to perform low-level network communications; in later chapters, we will instead use one of the higher-level protocol modules that hide underlying sockets.

The basic socket interface in Python is the standard library's `socket` module. Like the `os` POSIX module, Python's `socket` module is just a thin wrapper (interface layer) over the underlying C library's socket calls. Like Python files, it's also object-based: methods of a socket object implemented by this module call out to the corresponding C library's operations after data conversions. The `socket` module also includes tools for converting bytes to a standard network ordering, wrapping socket objects in simple file objects, and more. It supports socket programming on any machine that supports BSD-style sockets -- MS Windows, Linux, Unix, etc -- and so provides a portable socket interface.

10.3.1 Socket Basics

To create a connection between machines, Python programs import the `socket` module, create a socket object, and call the object's methods to establish connections and send and receive data. Socket object methods map directly to socket calls in the C library. For example, the script in [Example 10-1](#) implements a program that simply listens for a connection on a socket, and echoes back over a socket whatever it receives through that socket, adding 'Echo=>' string prefixes.

Example 10-1. PP2E\Internet\Sockets\echo-server.py

```
#####
# Server side: open a socket on a port, listen for
# a message from a client, and send an echo reply;
# this is a simple one-shot listen/reply per client,
# but it goes into an infinite loop to listen for
# more clients as long as this server script runs;
#####

from socket import *                # get socket constructor and constants
myHost = ''                         # server machine, '' means local host
myPort = 50007                      # listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP socket object
sockobj.bind((myHost, myPort))        # bind it to server port number
sockobj.listen(5)                   # listen, allow 5 pending connects

while 1:                             # listen until process killed
    connection, address = sockobj.accept() # wait for next client connect
    print 'Server connected by', address # connection is a new socket
    while 1:
        data = connection.recv(1024) # read next line on client socket
        if not data: break           # send a reply line to the client
        connection.send('Echo=>' + data) # until eof when socket closed
    connection.close()
```

As mentioned earlier, we usually call programs like this that listen for incoming connections servers because they provide a service that can be accessed at a given machine and port on the Internet. Programs that connect to such a server to access its service are generally called clients. [Example 10-2](#) shows a simple client implemented in Python.

Example 10-2. PP2E\Internet\Sockets\echo-client.py

```
#####
# Client side: use sockets to send data to the server, and
# print server's reply to each message line; 'localhost'
# means that the server is running on the same machine as
# the client, which lets us test client and server on one
# machine; to test over the Internet, run a server on a remote
# machine, and set serverHost or argv[1] to machine's domain
# name or IP addr; Python sockets are a portable BSD socket
# interface, with object methods for standard socket calls;
#####

import sys
from socket import *          # portable socket interface plus constants
serverHost = 'localhost'     # server name, or: 'starship.python.net'
serverPort = 50007           # non-reserved port used by the server

message = ['Hello network world'] # default text to send to server
if len(sys.argv) > 1:
    serverHost = sys.argv[1]     # or server from cmd line arg 1
    if len(sys.argv) > 2:       # or text from cmd line args 2..n
        message = sys.argv[2:] # one message for each arg listed

sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP/IP socket object
sockobj.connect((serverHost, serverPort)) # connect to server machine and port

for line in message:
    sockobj.send(line)          # send line to server over socket
    data = sockobj.recv(1024)   # receive line from server: up to 1024 bytes
    print 'Client received:', `data`

sockobj.close()                # close socket to send eof to server
```

10.3.1.1 Server socket calls

Before we see these programs in action, let's take a minute to explain how this client and server do their stuff. Both are fairly simple examples of socket scripts, but they illustrate common call patterns of most socket-based programs. In fact, this is boilerplate code: most socket programs generally make the same socket calls that our two scripts do, so let's step through the important points of these scripts line by line.

Programs such as [Example 10-1](#) that provide services for other programs with sockets generally start out by following this sequence of calls:

```
sockobj = socket(AF_INET, SOCK_STREAM)
```

Uses the Python socket module to create a TCP socket object. The names `AF_INET` and `SOCK_STREAM` are preassigned variables defined by and imported from the socket module; using them in combination means "create a TCP/IP socket," the standard communication device for the Internet. More specifically, `AF_INET` means the IP address protocol, and `SOCK_STREAM` means the TCP transfer protocol.

If you use other names in this call, you can instead create things like UDP connectionless sockets (use `SOCK_DGRAM` second) and Unix domain sockets on the local machine (use `AF_UNIX` first), but we won't do so in this book. See the Python library manual for details on these and other socket module options.

```
sockobj.bind((myHost, myPort))
```

Associates the socket object to an address -- for IP addresses, we pass a server machine name and port number on that machine. This is where the server identifies the machine and port associated with the socket. In server programs, the hostname is typically an empty string (""), which means the machine that the script runs on and the port is a number outside the range 0-1023 (which is reserved for standard protocols, described earlier). Note that each unique socket dialog you support must have its own port number; if you try to open a socket on a port already in use, Python will raise an exception. Also notice the nested parenthesis in this call -- for the `AF_INET` address protocol socket here, we pass the host/port socket address to `bind` as a two-item tuple object (pass a string for `AF_UNIX`). Technically, `bind` takes a tuple of values appropriate for the type of socket created (but see the next Note box about the older and deprecated convention of passing values to this function as distinct arguments).

```
sockobj.listen(5)
```

Starts listening for incoming client connections and allows for a backlog of up to five pending requests. The value passed sets the number of incoming client requests queued by the operating system before new requests are denied (which only happens if a server isn't fast enough to process requests before the queues fill up). A value of 5 is usually enough for most socket-based programs; the value must be at least 1.

At this point, the server is ready to accept connection requests from client programs running on remote machines (or the same machine), and falls into an infinite loop waiting for them to arrive

```
connection, address = sockobj.accept()
```

Waits for the next client connection request to occur; when it does, the `accept` call returns a brand new socket object over which data can be transferred from and to the connected client. Connections are accepted on `sockobj`, but communication with a client happens on `connection`, the new socket. This call actually returns a two-item tuple -- `address` is the connecting client's Internet address. We can call `accept` more than one time, to service multiple client connections; that's why each call returns a new, distinct socket for talking to a particular client.

Once we have a client connection, we fall into another loop to receive data from the client in blocks of 1024 bytes at a time, and echo each block back to the client:

```
data = connection.recv(1024)
```

Reads at most 1024 more bytes of the next message sent from a client (i.e., coming across the network), and returns it to the script as a string. We get back an empty string when the client has finished -- end-of-file is triggered when the client closes its end of the socket.

```
connection.send('Echo=>' + data)
```

Sends the latest data block back to the client program, prepending the string 'Echo=>' to it first. The client program can then `recv` what we `send` here -- the next reply line.

```
connection.close()
```

Shuts down the connection with this particular client.

After talking with a given client, the server goes back to its infinite loop, and waits for the next client connection request.

10.3.1.2 Client socket calls

On the other hand, client programs like the one shown in [Example 10-2](#) follow simpler call sequences. The main thing to keep in mind is that the client and server must specify the same port number when opening their sockets, and the client must identify the machine on which the server is running (in our scripts, server and client agree to use port number 50007 for their conversation, outside the standard protocol range):

```
sockobj = socket(AF_INET, SOCK_STREAM)
```

Creates a Python socket object in the client program, just like the server.

```
sockobj.connect((serverHost, serverPort))
```

Opens a connection to the machine and port on which the server program is listening for client connections. This is where the client specifies the name of the service to be contacted. In the client, we can either specify the name of the remote machine as a domain name (e.g., `starship.python.net`) or numeric IP address. We can also give the server name as `localhost` to specify that the server program is running on the same machine as the client; that comes in handy for debugging servers without having to connect to the Net. And again, the client's port number must match the server's exactly. Note the nested parentheses again -- just as in server `bind` calls, we really pass the server's host/port address to `connect` in a tuple object.

Once the client establishes a connection to the server, it falls into a loop sending a message one line at a time and printing whatever the server sends back after each line is sent:

```
sockobj.send(line)
```

Transfers the next message line to the server over the socket.

```
data = sockobj.recv(1024)
```

Reads the next reply line sent by the server program. Technically, this reads up to 1024 bytes of the next reply message and returns it as a string.

```
sockobj.close()
```

Closes the connection with the server, sending it the end-of-file signal.

And that's it. The server exchanges one or more lines of text with each client that connects. The operating system takes care of locating remote machines, routing bytes sent between programs across the Internet, and (with TCP) making sure that our messages arrive intact. That involves a lot of processing, too -- our strings may ultimately travel around the world, crossing phone wires, satellite links, and more along the way. But we can be happily ignorant of what goes on beneath the socket call layer when programming in Python.

In older Python code, you may see the `AF_INET` server address passed to the server-side `bind` and client-side `connect` socket methods as two distinct arguments, instead of a two-item tuple:

```
soc.bind(host,port)      vs soc.bind((host,port))
soc.connect(host,port)  vs soc.connect((host,port))
```

This two-argument form is now deprecated, and only worked at all due to a shortcoming in earlier Python releases (unfortunately, the Python library manual's socket example used the two-argument form too!). The tuple server address form is preferred, and, in a rare Python break with full backward-compatibility, will likely be the only one that will work in future Python releases.

10.3.1.3 Running socket programs locally

Okay, let's put this client and server to work. There are two ways to run these scripts -- either on the same machine or on two different machines. To run the client and the server on the same machine, bring up two command-line consoles on your computer, start the server program in one and run the client repeatedly in the other. The server keeps running and responds to requests made each time you run the client script in the other window.

For instance, here is the text that shows up in the MS-DOS console window where I've started the server script:

```
C:\...\PP2E\Internet\Sockets>python echo-server.py
Server connected by ('127.0.0.1', 1025)
Server connected by ('127.0.0.1', 1026)
Server connected by ('127.0.0.1', 1027)
```

The output here gives the address (machine IP name and port number) of each connecting client. Like most servers, this one runs perpetually, listening for client connection requests. This one receives three, but I have to show you the client window's text for you to understand what this means:

```
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world'

C:\...\PP2E\Internet\Sockets>python echo-client.py localhost spam Spam SPAM
Client received: 'Echo=>spam'
Client received: 'Echo=>Spam'
Client received: 'Echo=>SPAM'

C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Shrubbery
Client received: 'Echo=>Shrubbery'
```


Here, I ran the client script three times, while the server script kept running in the other window. Each client connected to the server, sent it a message of one or more lines of text, and read back the server's reply -- an echo of each line of text sent from the client. And each time a client is run a new connection message shows up in the server's window (that's why we got three).

It's important to notice that clients and server are running on the same machine here (a Windows PC). The server and client agree on port number, but use machine names "" and "localhost" respectively, to refer to the computer that they are running on. In fact, there is no Internet connection to speak of. Sockets also work well as cross-program communications tools on a single machine.

10.3.1.4 Running socket programs remotely

To make these scripts talk over the Internet instead of on a single machine, we have to do some extra work to run the server on a different computer. First, upload the server's source file to a remote machine where you have an account and a Python. Here's how I do it with FTP; your server name and upload interface details may vary, and there are other ways to copy files to a computer (e.g., email, web-page post forms, etc.):^[3]

[3] The FTP command is standard on Windows machines and most others. On Windows, simply type it in a DOS console box to connect to an FTP server (or start your favorite FTP program); on Linux, type the FTP command in an xterm window. You'll need to supply your account name and password to connect to a non-anonymous FTP site. For anonymous FTP, use "anonymous" for the username and your email address for the password (anonymous FTP sites are generally limited).

```
C:\...\PP2E\Internet\Sockets>ftp starship.python.net
Connected to starship.python.net.
User (starship.python.net:(none)): lutz
331 Password required for lutz.
Password:
230 User lutz logged in.
ftp> put echo-server.py
200 PORT command successful.
150 Opening ASCII mode data connection for echo-server.py.
226 Transfer complete.
ftp: 1322 bytes sent in 0.06Seconds 22.03Kbytes/sec.
ftp> quit
```

Once you have the server program loaded on the other computer, you need to run it there. Connect to that computer and start the server program. I usually telnet into my server machine and start the server program as a perpetually running process from the command line.^[4] The `&` syntax in Unix/Linux shells can be used to run the server script in the background; we could also make the server directly executable with a `#!` line and a `chmod` command (see [Chapter 2](#), for details). Here is the text that shows up in a Window on my PC that is running a Telnet session connected to the Linux server where I have an account (less a few deleted informational lines):

[4] Telnet is a standard command on Windows and Linux machines, too. On Windows, type it at a DOS console prompt or in the Start/Run dialog box (it can also be started via a clickable icon). Telnet usually runs in a window of its own.

```
C:\...\PP2E\Internet\Sockets>telnet starship.python.net
```

```
Red Hat Linux release 6.2 (Zoot)
Kernel 2.2.14-5.0smp on a 2-processor i686
login: lutz
Password:
[lutz@starship lutz]$ python echo-server.py &
[1] 4098
```

Now that the server is listening for connections on the Net, run the client on your local computer multiple times again. This time, the client runs on a different machine than the server, so we pass in the server's domain or IP name as a client command-line argument. The server still uses a machine name of "" because it always listens on whatever machine it runs upon. Here is what shows up in the server's Telnet window:

```
[lutz@starship lutz]$ Server connected by ('166.93.68.61', 1037)
Server connected by ('166.93.68.61', 1040)
Server connected by ('166.93.68.61', 1043)
Server connected by ('166.93.68.61', 1050)
```

And here is what appears in the MS-DOS console box where I run the client. A "connected by" message appears in the server Telnet window each time the client script is run in the client window:

```
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world'

C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net ni Ni NI
Client received: 'Echo=>ni'
Client received: 'Echo=>Ni'
Client received: 'Echo=>NI'

C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Shrubber
Client received: 'Echo=>Shrubbery'

C:\...\PP2E\Internet\Sockets>ping starship.python.net
Pinging starship.python.net [208.185.174.112] with 32 bytes of data:
Reply from 208.185.174.112: bytes=32 time=311ms TTL=246
ctrl-C
C:\...\PP2E\Internet\Sockets>python echo-client.py 208.185.174.112 Does she?
Client received: 'Echo=>Does'
Client received: 'Echo=>she?'
```

The "ping" command can be used to get an IP address for a machine's domain name; either machine name form can be used to connect in the client. This output is perhaps a bit understated - a lot is happening under the hood. The client, running on my Windows laptop, connects with and talks to the server program running on a Linux machine perhaps thousands of miles away. It all happens about as fast as when client and server both run on the laptop, and it uses the same library calls; only the server name passed to clients differs.

10.3.1.5 Socket pragmatics

Before we move on, there are three practical usage details you should know. First of all, you can run the client and server like this on any two Internet-aware machines where Python is installed. Of course, to run clients and server on different computers, you need both a live Internet connection and access to another machine on which to run the server. You don't need a big, expensive Internet link, though -- a simple modem and dialup Internet account will do for clients. When sockets are opened, Python is happy to use whatever connectivity you have, be it a dedicated T1 line, or a dialup modem account.

On my laptop PC, for instance, Windows automatically dials out to my ISP when clients are started or when Telnet server sessions are opened. In this book's examples, server-side programs that run remotely are executed on a machine called `starship.python.net`. If you don't have an account of your own on such a server, simply run client and server examples on the same machine, as shown earlier; all you need then is a computer that allows sockets, and most do.

Secondly, the `socket` module generally raises exceptions if you ask for something invalid. For instance, trying to connect to a nonexistent server (or unreachable servers, if you have no Internet link) fails:

```
C:\...\PP2E\Internet\Sockets>python echo-client.py www.nonesuch.com hello
Traceback (innermost last):
  File "echo-client.py", line 24, in ?
    sockobj.connect((serverHost, serverPort)) # connect to server machine...
  File "<string>", line 1, in connect
socket.error: (10061, 'winsock error')
```

Finally, also be sure to kill the server process before restarting it again, or else the port number will be still in use, and you'll get another exception:

```
[lutz@starship uploads]$ ps -x
  PID TTY          STAT       TIME COMMAND
 5570 pts/0        S           0:00 -bash
 5570 pts/0        S           0:00 -bash
 5633 pts/0        S           0:00 python echo-server.py
 5634 pts/0        R           0:00 ps -x
[lutz@starship uploads]$ python echo-server.py
Traceback (most recent call last):
  File "echo-server.py", line 14, in ?
    sockobj.bind((myHost, myPort)) # bind it to server port number
socket.error: (98, 'Address already in use')
```

Under Python 1.5.2, a series of Ctrl-C's will kill the server on Linux (be sure to type *fg* to bring it to the foreground first if started with an `&`):

```
[lutz@starship uploads]$ python echo-server.py
ctrl-c
Traceback (most recent call last):
  File "echo-server.py", line 18, in ?
    connection, address = sockobj.accept() # wait for next client connect
KeyboardInterrupt
```

A Ctrl-C kill key combination won't kill the server on my Windows machine, however. To kill the perpetually running server process running locally on Windows, you may need to type a Ctrl Alt-Delete key combination, and then end the Python task by selecting it in the process listbox that appears. You can usually also kill a server on Linux with a `kill -9 pid` shell command if it is running in another window or in the background, but Ctrl-C is less typing.

10.3.1.6 Spawning clients in parallel

To see how the server handles the load, let's fire up eight copies of the client script in parallel using the script in [Example 10-3](#) (see the end of [Chapter 3](#), for details on the `launchmodes` module used here to spawn clients).

Example 10-3. `PP2E\Internet\Sockets\testecho.py`

```
import sys, string
from PP2E.launchmodes import QuietPortableLauncher

numclients = 8
def start(cmdline): QuietPortableLauncher(cmdline, cmdline())

# start('echo-server.py')           # spawn server locally if not yet started
args = string.join(sys.argv[1:], ' ') # pass server name if running remotely
for i in range(numclients):
    start('echo-client.py %s' % args) # spawn 8? clients to test the server
```

To run this script, pass no arguments to talk to a server listening on port 50007 on the local machine; pass a real machine name to talk to a server running remotely. On Windows, the client output is discarded when spawned from this script:

```
C:\...\PP2E\Internet\Sockets>python testecho.py
```

```
C:\...\PP2E\Internet\Sockets>python testecho.py starship.python.net
```

If the spawned clients connect to a server run locally, connection messages show up in the server's window on the local machine:

```
C:\...\PP2E\Internet\Sockets>python echo-server.py
Server connected by ('127.0.0.1', 1283)
Server connected by ('127.0.0.1', 1284)
Server connected by ('127.0.0.1', 1285)
Server connected by ('127.0.0.1', 1286)
Server connected by ('127.0.0.1', 1287)
Server connected by ('127.0.0.1', 1288)
Server connected by ('127.0.0.1', 1289)
Server connected by ('127.0.0.1', 1290)
```

If the server is running remotely, the client connection messages instead appear in the window displaying the Telnet connection to the remote computer:

```
[lutz@starship lutz]$ python echo-server.py
Server connected by ('166.93.68.61', 1301)
Server connected by ('166.93.68.61', 1302)
Server connected by ('166.93.68.61', 1308)
Server connected by ('166.93.68.61', 1309)
Server connected by ('166.93.68.61', 1313)
Server connected by ('166.93.68.61', 1314)
Server connected by ('166.93.68.61', 1307)
Server connected by ('166.93.68.61', 1312)
```

Keep in mind, however, that this works for our simple scripts only because the server doesn't take a long time to respond to each client's requests -- it can get back to the top of the server script's outer `while` loop in time to process the next incoming client. If it could not, we would probably need to change the server to handle each client in parallel, or some might be denied a connection. Technically, client connections would fail after five clients are already waiting for the server's attention, as specified in the server's `listen` call. We'll see how servers can handle multiple clients robustly in the next section.

10.3.1.7 Talking to reserved ports

It's also important to know that this client and server engage in a proprietary sort of discussion, and so use a port number 50007 outside the range reserved for standard protocols (0-1023). There's nothing preventing a client from opening a socket on one of these special ports, however. For instance, the following client-side code connects to programs listening on the standard email FTP, and HTTP web server ports on three different server machines:

```
C:\...\PP2E\Internet\Sockets>python
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('mail.rmi.net', 110))           # talk to RMI POP mail server
>>> print sock.recv(40)
+OK Cubic Circle's v1.31 1998/05/13 POP3
>>> sock.close()

>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('www.python.org', 21))         # talk to Python FTP server
>>> print sock.recv(40)
220 python.org FTP server (Version wu-2.
>>> sock.close()

>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('starship.python.net', 80))   # starship HTTP web server
>>> sock.send('GET /\r\n')                     # fetch root web page
7
>>> sock.recv(60)
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">\012<HTM'
>>> sock.recv(60)
'L>\012 <HEAD>\012 <TITLE>Starship Slowly Recovering</TITLE>\012 </HE'
```

If we know how to interpret the output returned by these ports' servers, we could use raw socket-like this to fetch email, transfer files, and grab web pages and invoke server-side scripts. Fortunately, though, we don't have to worry about all the underlying details -- Python's `poplib`, `ftplib`, `httplib`, and `urllib` modules provide higher-level interfaces for talking to servers on these ports. Other Python protocol modules do the same for other standard ports (e.g., NNTP, Telnet, and so on). We'll meet some of these client-side protocol modules in the next chapter.^[5]

[5] You might be interested to know that the last part of this example, talking to port 80, is exactly what your web browser does as you surf the Net: followed links direct it to download web pages over this port. In fact, this lowly port is the primary basis of the Web. In [Chapter 12](#), we will meet an entire application environment based upon sending data over port 80 -- CGI server-side scripting.

By the way, it's okay to open client-side connections on reserved ports like this, but you can't install your own server-side scripts for these ports unless you have special permission:

```
[lutz@starship uploads]$ python
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.bind('', 80)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
socket.error: (13, 'Permission denied')
```

Even if run by a user with the required permission, you'll get the different exception we saw earlier if the port is already being used by a real web server. On computers being used as general servers, these ports really are reserved.

10.4 Handling Multiple Clients

The `echo` client and server programs shown previously serve to illustrate socket fundamentals. But the server model suffers from a fairly major flaw: if multiple clients try to connect to the server, and it takes a long time to process a given client's request, the server will fail. More accurately, the cost of handling a given request prevents the server from returning to the code that checks for new clients in a timely manner, it won't be able to keep up with all the requests, and some clients will eventually be denied connections.

In real-world client/server programs, it's far more typical to code a server so as to avoid blocking new requests while handling a current client's request. Perhaps the easiest way to do so is to service each client's request in parallel -- in a new process, in a new thread, or by manually switching (multiplexing) between clients in an event loop. This isn't a socket issue per se, and we've already learned how to start processes and threads in [Chapter 3](#). But since these schemes are so typical of socket server programming, let's explore all three ways to handle client requests in parallel here.

10.4.1 Forking Servers

The script in [Example 10-4](#) works like the original `echo` server, but instead forks a new process to handle each new client connection. Because the `handleClient` function runs in a new process, the `dispatcher` function can immediately resume its main loop, to detect and service a new incoming request.

Example 10-4. PP2E\Internet\Sockets\fork-server.py

```
#####
# Server side: open a socket on a port, listen for
# a message from a client, and send an echo reply;
# forks a process to handle each client connection;
# child processes share parent's socket descriptors;
# fork is less portable than threads--not yet on Windows;
#####

import os, time, sys
from socket import *
myHost = ''
myPort = 50007

# get socket constructor and constant
# server machine, '' means local host
# listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM)
sockobj.bind((myHost, myPort))
sockobj.listen(5)

# make a TCP socket object
# bind it to server port number
# allow 5 pending connects

def now():
    return time.ctime(time.time())

# current time on server

activeChildren = []
def reapChildren():
    while activeChildren:
        pid,stat = os.waitpid(0, os.WNOHANG)
        if not pid: break
        activeChildren.remove(pid)

# reap any dead child processes
# else may fill up system table
# don't hang if no child exists

def handleClient(connection):
    time.sleep(5)

# child process: reply, exit
# simulate a blocking activity
```

```
while 1:                                # read, write a client socket
    data = connection.recv(1024)        # till eof when socket closed
    if not data: break
    connection.send('Echo=>%s at %s' % (data, now()))
connection.close()
os._exit(0)

def dispatcher():                       # listen until process killed
    while 1:                             # wait for next connection,
        connection, address = sockobj.accept() # pass to process for service
        print 'Server connected by', address,
        print 'at', now()
        reapChildren()                  # clean up exited children now
        childPid = os.fork()            # copy this process
        if childPid == 0:                # if in child process: handle
            handleClient(connection)
        else:                             # else: go accept next connect
            activeChildren.append(childPid) # add to active child pid list

dispatcher()
```

10.4.1.1 Running the forking server

Parts of this script are a bit tricky, and most of its library calls work only on Unix-like platforms (not Windows). But before we get into too many details, let's start up our server and handle a few client requests. First off, notice that to simulate a long-running operation (e.g., database updates, other network traffic), this server adds a five-second `time.sleep` delay in its client handler function, `handleClient`. After the delay, the original echo reply action is performed. That means that when we run a server and clients this time, clients won't receive the echo reply until five seconds after they've sent their requests to the server.

To help keep track of requests and replies, the server prints its system time each time a client connect request is received, and adds its system time to the reply. Clients print the reply time sent back from the server, not their own -- clocks on the server and client may differ radically, so to compare apples to apples, all times are server times. Because of the simulated delays, we also usually must start each client in its own console window on Windows (on some platforms, client will hang in a blocked state while waiting for their reply).

But the grander story here is that this script runs one main parent process on the server machine, which does nothing but watch for connections (in `dispatcher`), plus one child process per active client connection, running in parallel with both the main parent process and the other client processes (in `handleClient`). In principle, the server can handle any number of clients without bogging down. To test, let's start the server remotely in a Telnet window, and start three clients locally in three distinct console windows:

```
[server telnet window]
[lutz@starship uploads]$ uname -a
Linux starship ...
[lutz@starship uploads]$ python fork-server.py
Server connected by ('38.28.162.194', 1063) at Sun Jun 18 19:37:49 2000
Server connected by ('38.28.162.194', 1064) at Sun Jun 18 19:37:49 2000
Server connected by ('38.28.162.194', 1067) at Sun Jun 18 19:37:50 2000
```

```
[client window 1]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world at Sun Jun 18 19:37:54 2000'
```

```
[client window 2]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Bruce
```



```
Client received: 'Echo=>Bruce at Sun Jun 18 19:37:54 2000'
```

```
[client window 3]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net The
Meaning of Life
Client received: 'Echo=>The at Sun Jun 18 19:37:55 2000'
Client received: 'Echo=>Meaning at Sun Jun 18 19:37:56 2000'
Client received: 'Echo=>of at Sun Jun 18 19:37:56 2000'
Client received: 'Echo=>Life at Sun Jun 18 19:37:56 2000'
```

Again, all times here are on the server machine. This may be a little confusing because there are four windows involved. In English, the test proceeds as follows:

1. The server starts running remotely.
2. All three clients are started and connect to the server at roughly the same time.
3. On the server, the client requests trigger three forked child processes, which all immediately go to sleep for five seconds (to simulate being busy doing something useful).
4. Each client waits until the server replies, which eventually happens five seconds after their initial requests.

In other words, all three clients are serviced at the same time, by forked processes, while the main parent process continues listening for new client requests. If clients were not handled in parallel like this, no client could connect until the currently connected client's five-second delay expired.

In a more realistic application, that delay could be fatal if many clients were trying to connect at once -- the server would be stuck in the action we're simulating with `time.sleep`, and not get back to the main loop to `accept` new client requests. With `process forks` per request, all clients can be serviced in parallel.

Notice that we're using the same client script here (*echo-client.py*), just a different server; clients simply send and receive data to a machine and port, and don't care how their requests are handled on the server. Also note that the server is running remotely on a Linux machine. (As we learned [Chapter 3](#), the `fork` call is not supported on Windows in Python at the time this book was written.) We can also run this test on a Linux server entirely, with two Telnet windows. It works about the same as when clients are started locally, in a DOS console window, but here "local" means a remote machine you're telnetting to locally:

```
[one telnet window]
[lutz@starship uploads]$ python fork-server.py &
[1] 3379
Server connected by ('127.0.0.1', 2928) at Sun Jun 18 22:44:50 2000
Server connected by ('127.0.0.1', 2929) at Sun Jun 18 22:45:08 2000
Server connected by ('208.185.174.112', 2930) at Sun Jun 18 22:45:50 2000

[another telnet window, same machine]
[lutz@starship uploads]$ python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 22:44:55 2000'

[lutz@starship uploads]$ python echo-client.py localhost niNiNI
Client received: 'Echo=>niNiNI at Sun Jun 18 22:45:13 2000'

[lutz@starship uploads]$ python echo-client.py starship.python.net Say no More!
Client received: 'Echo=>Say at Sun Jun 18 22:45:55 2000'
Client received: 'Echo=>no at Sun Jun 18 22:45:55 2000'
Client received: 'Echo=>More! at Sun Jun 18 22:45:55 2000'
```

Now let's move on to the tricky bits. This server script is fairly straightforward as forking code goes, but a few comments about some of the library tools it employs are in order.

10.4.1.2 Forking processes

We met `os.fork` in [Chapter 3](#), but recall that forked processes are essentially a copy of the process that forks them, and so they inherit file and socket descriptors from their parent process. Because of that, the new child process that runs the `handleClient` function has access to the connection socket created in the parent process. Programs know they are in a forked child process if the fork call returns 0; otherwise, the original parent process gets back the new child's ID.

10.4.1.3 Exiting from children

In earlier fork examples, child processes usually call one of the `exec` variants to start a new program in the child process. Here, instead, the child process simply calls a function in the same program and exits with `os._exit`. It's imperative to call `os._exit` here -- if we did not, each child would live on after `handleClient` returns, and compete for accepting new client requests.

In fact, without the exit call, we'd wind up with as many perpetual server processes as requests served -- remove the exit call and do a `ps` shell command after running a few clients, and you'll see what I mean. With the call, only the single parent process listens for new requests. `os._exit` is like `sys.exit`, but it exits the calling process immediately without cleanup actions. It's normally only used in child processes, and `sys.exit` is used everywhere else.

10.4.1.4 Killing the zombies

Note, however, that it's not quite enough to make sure that child processes exit and die. On systems like Linux, parents must also be sure to issue a `wait` system call to remove the entries for dead child processes from the system's process table. If we don't, the child processes will no longer run, but they will consume an entry in the system process table. For long-running servers, these bogus entries may become problematic.

It's common to call such dead-but-listed child processes "zombies": they continue to use system resources even though they've already passed over to the great operating system beyond. To clean up after child processes are gone, this server keeps a list, `activeChildren`, of the process IDs of all child processes it spawns. Whenever a new incoming client request is received, the server runs its `reapChildren` to issue a `wait` for any dead children by issuing the standard Python `os.waitpid(0,os.WNOHANG)` call.

The `os.waitpid` call attempts to wait for a child process to exit and returns its process ID and exit status. With a PID for its first argument, it waits for any child process. With the `WNOHANG` parameter for its second, it does nothing if no child process has exited (i.e., it does not block or pause the caller). The net effect is that this call simply asks the operating system for the process ID of any child that has exited. If any have, the process ID returned is removed both from the system process table and from this script's `activeChildren` list.

To see why all this complexity is needed, comment out the `reapChildren` call in this script, run on a server, and then run a few clients. On my Linux server, a `ps -f` full process listing command shows that all the dead child processes stay in the system process table (show as `<defunct>`):

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     3270 3264  0  22:33 pts/1        00:00:00 -bash
lutz     3311 3270  0  22:37 pts/1        00:00:00 python fork-server.py
lutz     3312 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3313 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3314 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3316 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3317 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3318 3311  0  22:37 pts/1        00:00:00 [python <defunct>]
lutz     3322 3270  0  22:38 pts/1        00:00:00 ps -f
```

When the `reapChildren` command is reactivated, dead child zombie entries are cleaned up each time the server gets a new client connection request, by calling the Python `os.waitpid` function. A few zombies may accumulate if the server is heavily loaded, but will remain only until the next client connection is received:

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     3270 3264  0  22:33 pts/1        00:00:00 -bash
lutz     3340 3270  0  22:41 pts/1        00:00:00 python fork-server.py
lutz     3341 3340  0  22:41 pts/1        00:00:00 [python <defunct>]
lutz     3342 3340  0  22:41 pts/1        00:00:00 [python <defunct>]
lutz     3343 3340  0  22:41 pts/1        00:00:00 [python <defunct>]
lutz     3344 3270  6  22:41 pts/1        00:00:00 ps -f
[lutz@starship uploads]$
Server connected by ('38.28.131.174', 1170) at Sun Jun 18 22:41:43 2000
```

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     3270 3264  0  22:33 pts/1        00:00:00 -bash
lutz     3340 3270  0  22:41 pts/1        00:00:00 python fork-server.py
lutz     3345 3340  0  22:41 pts/1        00:00:00 [python <defunct>]
lutz     3346 3270  0  22:41 pts/1        00:00:00 ps -f
```

If you type fast enough, you can actually see a child process morph from a real running program into a zombie. Here, for example, a child spawned to handle a new request (process ID 11785) changes to `<defunct>` on exit. Its process entry will be removed completely when the next request is received:

```
[lutz@starship uploads]$
Server connected by ('38.28.57.160', 1106) at Mon Jun 19 22:34:39 2000
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz     11780 11089  0  22:34 pts/2        00:00:00 python fork-server.py
lutz     11785 11780  0  22:34 pts/2        00:00:00 python fork-server.py
lutz     11786 11089  0  22:34 pts/2        00:00:00 ps -f

[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz     11780 11089  0  22:34 pts/2        00:00:00 python fork-server.py
lutz     11785 11780  0  22:34 pts/2        00:00:00 [python <defunct>]
lutz     11787 11089  0  22:34 pts/2        00:00:00 ps -f
```

10.4.1.5 Preventing zombies with signal handlers

On some systems, it's also possible to clean up zombie child processes by resetting the signal handler for the `SIGCHLD` signal raised by the operating system when a child process exits. If a

Python script assigns the `SIG_IGN` (ignore) action as the `SIGCHLD` signal handler, zombies will be removed automatically and immediately as child processes exit; the parent need not issue wait calls to clean up after them. Because of that, this scheme is a simpler alternative to manually reaping zombies (on platforms where it is supported).

If you've already read [Chapter 3](#), you know that Python's standard `signal` module lets scripts install handlers for signals -- software-generated events. If you haven't read that chapter, here is a brief bit of background to show how this pans out for zombies. The program in [Example 10-5](#) installs a Python-coded signal handler function to respond to whatever signal number you type on the command line.

Example 10-5. PP2E\Internet\Sockets\signal-demo.py

```
#####
# Demo Python's signal module; pass signal number as a
# command-line arg, use a "kill -N pid" shell command
# to send this process a signal; e.g., on my linux
# machine, SIGUSR1=10, SIGUSR2=12, SIGCHLD=17, and
# SIGCHLD handler stays in effect even if not restored:
# all other handlers restored by Python after caught,
# but SIGCHLD is left to the platform's implementation;
# signal works on Windows but defines only a few signal
# types; signals are not very portable in general;
#####

import sys, signal, time

def now():
    return time.ctime(time.time())

def onSignal(signum, stackframe):
    print 'Got signal', signum, 'at', now()
    if signum == signal.SIGCHLD:
        print 'sigchld caught'
        #signal.signal(signal.SIGCHLD, onSignal)

signum = int(sys.argv[1])
signal.signal(signum, onSignal)
while 1: signal.pause()
# install signal handler
# sleep waiting for signals
```

To run this script, simply put it in the background and send it signals by typing the `kill -signal number process-id` shell command line. Process IDs are listed in the PID column of `ps` command results. Here is this script in action catching signal numbers 10 (reserved for general use) and 9 (the unavoidable terminate signal):

```
[lutz@starship uploads]$ python signal-demo.py 10 &
[1] 11297
[lutz@starship uploads]$ ps -f
UID          PID  PPID  C  STIME TTY          TIME CMD
lutz         11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz         11297 11089  0  21:49 pts/2        00:00:00 python signal-demo.py 10
lutz         11298 11089  0  21:49 pts/2        00:00:00 ps -f

[lutz@starship uploads]$ kill -10 11297
Got signal 10 at Mon Jun 19 21:49:27 2000

[lutz@starship uploads]$ kill -10 11297
Got signal 10 at Mon Jun 19 21:49:29 2000

[lutz@starship uploads]$ kill -10 11297
```

```
Got signal 10 at Mon Jun 19 21:49:32 2000
```

```
[lutz@starship uploads]$ kill -9 11297
[1]+  Killed                  python signal-demo.py 10
```

And here the script catches signal 17, which happens to be `SIGCHLD` on my Linux server. Signal numbers vary from machine to machine, so you should normally use their names, not their numbers. `SIGCHLD` behavior may vary per platform as well (see the signal module's library manual entry for more details):

```
[lutz@starship uploads]$ python signal-demo.py 17 &
[1] 11320
[lutz@starship uploads]$ ps -f
UID          PID    PPID  C  STIME TTY          TIME CMD
lutz        11089  11088  0  21:13 pts/2        00:00:00 -bash
lutz        11320  11089  0  21:52 pts/2        00:00:00 python signal-demo.py 17
lutz        11321  11089  0  21:52 pts/2        00:00:00 ps -f

[lutz@starship uploads]$ kill -17 11320
Got signal 17 at Mon Jun 19 21:52:24 2000
[lutz@starship uploads] sigchld caught

[lutz@starship uploads]$ kill -17 11320
Got signal 17 at Mon Jun 19 21:52:27 2000
[lutz@starship uploads] sigchld caught
```

Now, to apply all this to kill zombies, simply set the `SIGCHLD` signal handler to the `SIG_IGN` ignore handler action; on systems where this assignment is supported, child processes will be cleaned up when they exit. The forking server variant shown in [Example 10-6](#) uses this trick to manage its children.

Example 10-6. PP2E\Internet\Sockets\fork-server-signal.py

```
#####
# Same as fork-server.py, but use the Python signal
# module to avoid keeping child zombie processes after
# they terminate, not an explicit loop before each new
# connection; SIG_IGN means ignore, and may not work with
# SIG_CHLD child exit signal on all platforms; on Linux,
# socket.accept cannot be interrupted with a signal;
#####

import os, time, sys, signal, signal
from socket import *                               # get socket constructor and constant
myHost = ''                                       # server machine, '' means local host
myPort = 50007                                    # listen on a non-reserved port numbe

sockobj = socket(AF_INET, SOCK_STREAM)           # make a TCP socket object
sockobj.bind((myHost, myPort))                   # bind it to server port numbe
sockobj.listen(5)                                 # up to 5 pending connects
signal.signal(signal.SIGCHLD, signal.SIG_IGN)    # avoid child zombie processes

def now():                                        # time on server machine
    return time.ctime(time.time())

def handleClient(connection):                    # child process replies, exits
    time.sleep(5)                                 # simulate a blocking activity
    while 1:                                     # read, write a client socket
        data = connection.recv(1024)
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()
    os._exit(0)
```

```
def dispatcher():
    while 1:
        connection, address = sockobj.accept()
        print 'Server connected by', address,
        print 'at', now()
        childPid = os.fork()
        if childPid == 0:
            handleClient(connection)

dispatcher()
```

Where applicable, this technique is:

- Much simpler -- we don't need to manually track or reap child processes.
- More accurate -- it leaves no zombies temporarily between client requests.

In fact, there is really only one line dedicated to handling zombies here: the `signal.signal` call near the top, to set the handler. Unfortunately, this version is also even less portable than using `os.fork` in the first place, because signals may work slightly different from platform to platform. For instance, some platforms may not allow `SIG_IGN` to be used as the `SIGCHLD` action at all. On Linux systems, though, this simpler forking server variant works like a charm:

```
[lutz@starship uploads]$
Server connected by ('38.28.57.160', 1166) at Mon Jun 19 22:38:29 2000
```

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz     11827 11089  0  22:37 pts/2        00:00:00 python fork-server-signal.py
lutz     11835 11827  0  22:38 pts/2        00:00:00 python fork-server-signal.py
lutz     11836 11089  0  22:38 pts/2        00:00:00 ps -f
```

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz     11827 11089  0  22:37 pts/2        00:00:00 python fork-server-signal.py
lutz     11837 11089  0  22:38 pts/2        00:00:00 ps -f
```

Notice that in this version, the child process's entry goes away as soon as it exits, even before a new client request is received; no "defunct" zombie ever appears. More dramatically, if we now start up the script we wrote earlier that spawns eight clients in parallel (*testechno.py*) to talk to this server, all appear on the server while running, but are removed immediately as they exit:

```
[lutz@starship uploads]$ ps -f
UID      PID  PPID  C  STIME TTY          TIME CMD
lutz     11089 11088  0  21:13 pts/2        00:00:00 -bash
lutz     11827 11089  0  22:37 pts/2        00:00:00 python fork-server-signal.py
lutz     11839 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11840 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11841 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11842 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11843 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11844 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11845 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11846 11827  0  22:39 pts/2        00:00:00 python fork-server-signal.py
lutz     11848 11089  0  22:39 pts/2        00:00:00 ps -f
```

```
[lutz@starship uploads]$ ps -f
```

```
UID          PID    PPID    C  STIME TTY          TIME CMD
lutz         11089 11088    0  21:13 pts/2        00:00:00 -bash
lutz         11827 11089    0  22:37 pts/2        00:00:00 python fork-server-signal.py
lutz         11849 11089    0  22:39 pts/2        00:00:00 ps -f
```

10.4.2 Threading Servers

But don't do that . The forking model just described works well on some platforms in general, but suffers from some potentially big limitations:

Performance

On some machines, starting a new process can be fairly expensive in terms of time and space resources.

Portability

Forking processes is a Unix device; as we just noted, the fork call currently doesn't work on non-Unix platforms such as Windows.

Complexity

If you think that forking servers can be complicated, you're right. As we just saw, forking also brings with it all the shenanigans of managing zombies -- cleaning up after child processes that live shorter lives than their parents.

If you read [Chapter 3](#), you know that the solution to all of these dilemmas is usually to use threads instead of processes. Threads run in parallel and share global (i.e., module and interpreter) memory, but they are usually less expensive to start, and work both on Unix-like machines and Microsoft Windows today. Furthermore, threads are simpler to program -- child threads die silently on exit, without leaving behind zombies to haunt the server.

[Example 10-7](#) is another mutation of the echo server that handles client request in parallel by running them in threads, rather than processes.

Example 10-7. PP2E\Internet\Sockets\thread-server.py

```
#####
# Server side: open a socket on a port, listen for
# a message from a client, and send an echo reply;
# echos lines until eof when client closes socket;
# spawns a thread to handle each client connection;
# threads share global memory space with main thread;
# this is more portable than fork--not yet on Windows;
#####

import thread, time
from socket import *
myHost = ''
myPort = 50007

sockobj = socket(AF_INET, SOCK_STREAM)
sockobj.bind((myHost, myPort))
sockobj.listen(5)

def now():
    return time.ctime(time.time())

# get socket constructor and constants
# server machine, '' means local host
# listen on a non-reserved port number

# make a TCP socket object
# bind it to server port number
# allow up to 5 pending connections

# current time on the server
```

```
def handleClient(connection):
    time.sleep(5)
    while 1:
        data = connection.recv(1024)
        if not data: break
        connection.send('Echo=>%s at %s' % (data, now()))
    connection.close()

def dispatcher():
    while 1:
        connection, address = sockobj.accept()
        print 'Server connected by', address,
        print 'at', now()
        thread.start_new(handleClient, (connection,))

dispatcher()
```

This `dispatcher` delegates each incoming client connection request to a newly spawned thread running the `handleClient` function. Because of that, this server can process multiple clients at once, and the main dispatcher loop can get quickly back to the top to check for newly arrived requests. The net effect is that new clients won't be denied service due to a busy server.

Functionally, this version is similar to the `fork` solution (clients are handled in parallel), but it works on any machine that supports threads, including Windows and Linux. Let's test it on both. First, start the server on a Linux machine and run clients on both Linux and Windows:

```
[window 1: thread-based server process, server keeps accepting
client connections while threads are servicing prior requests]
[lutz@starship uploads]$ /usr/bin/python thread-server.py
Server connected by ('127.0.0.1', 2934) at Sun Jun 18 22:52:52 2000
Server connected by ('38.28.131.174', 1179) at Sun Jun 18 22:53:31 2000
Server connected by ('38.28.131.174', 1182) at Sun Jun 18 22:53:35 2000
Server connected by ('38.28.131.174', 1185) at Sun Jun 18 22:53:37 2000

[window 2: client, but on same server machine]
[lutz@starship uploads]$ python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 22:52:57 2000'

[window 3: remote client, PC]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net
Client received: 'Echo=>Hello network world at Sun Jun 18 22:53:36 2000'

[window 4: client PC]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Bruce
Client received: 'Echo=>Bruce at Sun Jun 18 22:53:40 2000'

[window 5: client PC]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net The
Meaning of Life
Client received: 'Echo=>The at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>Meaning at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>of at Sun Jun 18 22:53:42 2000'
Client received: 'Echo=>Life at Sun Jun 18 22:53:42 2000'
```

Because this server uses threads instead of forked processes, we can run it portably on both Linux and a Windows PC. Here it is at work again, running on the same local Windows PC as its client again, the main point to notice is that new clients are accepted while prior clients are being processed in parallel with other clients and the main thread (in the five-second sleep delay):

```
[window 1: server, on local PC]
```



```
C:\...\PP2E\Internet\Sockets>python thread-server.py
Server connected by ('127.0.0.1', 1186) at Sun Jun 18 23:46:31 2000
Server connected by ('127.0.0.1', 1187) at Sun Jun 18 23:46:33 2000
Server connected by ('127.0.0.1', 1188) at Sun Jun 18 23:46:34 2000

[window 2: client, on local
PC]
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 23:46:36 2000'

[window 3: client]
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Brian
Client received: 'Echo=>Brian at Sun Jun 18 23:46:38 2000'

[window 4: client]
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Bright side of Lif
Client received: 'Echo=>Bright at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>side at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>of at Sun Jun 18 23:46:39 2000'
Client received: 'Echo=>Life at Sun Jun 18 23:46:39 2000'
```

Recall that a thread silently exits when the function it is running returns; unlike the process `fork` version, we don't call anything like `os._exit` in the client handler function (and we shouldn't -- it may kill all threads in the process!). Because of this, the thread version is not only more portable but is also simpler.

10.4.3 Doing It with Classes: Server Frameworks

Now that I've shown you how to write forking and threading servers to process clients without blocking incoming requests, I should also tell you that there are standard tools in the Python library to make this process easier. In particular, the `SocketServer` module defines classes that implement all flavors of forking and threading servers that you are likely to be interested in. Simply create the desired kind of imported server object, passing in a handler object with a callback method of your own, as shown in [Example 10-8](#).

Example 10-8. PP2E\Internet\Sockets\class-server.py

```
#####
# Server side: open a socket on a port, listen for
# a message from a client, and send an echo reply;
# this version uses the standard library module
# SocketServer to do its work; SocketServer allows
# us to make a simple TCPServer, a ThreadingTCPServer,
# a ForkingTCPServer, and more, and routes each client
# connect request to a new instance of a passed-in
# request handler object's handle method; also supports
# UDP and Unix domain sockets; see the library manual.
#####

import SocketServer, time                # get socket server, handler objects
myHost = ''                              # server machine, '' means local host
myPort = 50007                            # listen on a non-reserved port number
def now():
    return time.ctime(time.time())

class MyClientHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        print self.client_address, now()  # on each client connect
        time.sleep(5)                    # show this client's address
        while 1:                           # simulate a blocking activity
            data = self.request.recv(1024) # self.request is client socket
            # read, write a client socket
```

```
        if not data: break
        self.request.send('Echo=>%s at %s' % (data, now()))
    self.request.close()

# make a threaded server, listen/handle clients forever
myaddr = (myHost, myPort)
server = SocketServer.ThreadingTCPServer(myaddr, MyClientHandler)
server.serve_forever()
```

This server works the same as the threading server we wrote by hand in the previous section, but instead focuses on service implementation (the customized `handle` method), not on threading details. It's run the same way, too -- here it is processing three clients started by hand, plus eight spawned by the `testecho` script shown in [Example 10-3](#):

```
[window1: server, serverHost='localhost' in echo-client.py]
C:\...\PP2E\Internet\Sockets>python class-server.py
('127.0.0.1', 1189) Sun Jun 18 23:49:18 2000
('127.0.0.1', 1190) Sun Jun 18 23:49:20 2000
('127.0.0.1', 1191) Sun Jun 18 23:49:22 2000
('127.0.0.1', 1192) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1193) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1194) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1195) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1196) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1197) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1198) Sun Jun 18 23:49:50 2000
('127.0.0.1', 1199) Sun Jun 18 23:49:50 2000

[window2: client]
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Jun 18 23:49:23 2000'

[window3: client]
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Robin
Client received: 'Echo=>Robin at Sun Jun 18 23:49:25 2000'

[window4: client]
C:\...\PP2E\Internet\Sockets>python echo-client.py localhost Brave Sir Robin
Client received: 'Echo=>Brave at Sun Jun 18 23:49:27 2000'
Client received: 'Echo=>Sir at Sun Jun 18 23:49:27 2000'
Client received: 'Echo=>Robin at Sun Jun 18 23:49:27 2000'

C:\...\PP2E\Internet\Sockets>python testecho.py

[window4: contact remote server instead -- times skewed]
C:\...\PP2E\Internet\Sockets>python echo-client.py starship.python.net Brave
Sir Robin
Client received: 'Echo=>Brave at Sun Jun 18 23:03:28 2000'
Client received: 'Echo=>Sir at Sun Jun 18 23:03:28 2000'
Client received: 'Echo=>Robin at Sun Jun 18 23:03:29 2000'
```

To build a forking server instead, just use class name `ForkingTCPServer` when creating the `serv` object. The `SocketServer` module is more powerful than shown by this example; it also support synchronous (nonparallel) servers, UDP and Unix sockets, and so on. See Python's library manu for more details. Also see the end of [Chapter 15](#) for more on Python server implementation tools.^[6]

[6] Incidentally, Python also comes with library tools that allow you to implement a full-blown HTTP (web) server that knows how to run server-side CGI scripts, in a few lines of Python code. We'll explore those tools in [Chapter 15](#).

10.4.4 Multiplexing Servers with `select`

So far we've seen how to handle multiple clients at once with both forked processes and spawned threads, and we've looked at a library class that encapsulates both schemes. Under both approaches, all client handlers seem to run in parallel with each other and with the main dispatch loop that continues watching for new incoming requests. Because all these tasks run in parallel (i.e., at the same time), the server doesn't get blocked when accepting new requests or when processing a long-running client handler.

Technically, though, threads and processes don't really run in parallel, unless you're lucky enough to have a machine with arbitrarily many CPUs. Instead, your operating system performs a juggler's act -- it divides the computer's processing power among all active tasks. It runs part of one, then part of another, and so on. All the tasks appear to run in parallel, but only because the operating system switches focus between tasks so fast that you don't usually notice. This process of switching between tasks is sometimes called time-slicing when done by an operating system; it is more generally known as multiplexing.

When we spawn threads and processes, we rely on the operating system to juggle the active tasks but there's no reason that a Python script can't do so as well. For instance, a script might divide tasks into multiple steps -- do a step of one task, then one of another, and so on, until all are completed. The script need only know how to divide its attention among the multiple active tasks to multiplex on its own.

Servers can apply this technique to yield yet another way to handle multiple clients at once, a way that requires neither threads nor forks. By multiplexing client connections and the main dispatch with the `select` system call, a single event loop can process clients and accept new ones in parallel (or at least close enough to avoid stalling). Such servers are sometimes called asynchronous because they service clients in spurts, as each becomes ready to communicate. In asynchronous servers, a single main loop runs in a single process and thread decides which clients should get a slice of attention each time through. Client requests and the main dispatcher are each given a small slice of the server's attention if they are ready to converse.

Most of the magic behind this server structure is the operating system `select` call, available in Python's standard `select` module. Roughly, `select` is asked to monitor a list of input sources, output sources, and exceptional condition sources, and tells us which sources are ready for processing. It can be made to simply poll all the sources to see which are ready, wait for a maximum time period for sources to become ready, or wait indefinitely until one or more sources are ready for processing.

However used, `select` lets us direct attention to sockets ready to communicate, so as to avoid blocking on calls to ones that are not. That is, when the sources passed to `select` are sockets, we can be sure that socket calls like `accept`, `recv`, and `send` will not block (pause) the server when applied to objects returned by `select`. Because of that, a single-loop server that uses `select` need not get stuck communicating with one client or waiting for new ones, while other clients are starved for the server's attention.

10.4.4.1 A `select`-based echo server

Let's see how all this translates into code. The script in [Example 10-9](#) implements another echo server, one that can handle multiple clients without ever starting new processes or threads.

Example 10-9. PP2E\Internet\Sockets\select-server.py

```
#####
# Server: handle multiple clients in parallel with select.
# use the select module to multiplex among a set of sockets:
# main sockets which accept new client connections, and
# input sockets connected to accepted clients; select can
# take an optional 4th arg--0 to poll, n.m to wait n.m secs,
# omitted to wait till any socket is ready for processing.
#####

import sys, time
from select import select
from socket import socket, AF_INET, SOCK_STREAM
def now(): return time.ctime(time.time())

myHost = '' # server machine, '' means local host
myPort = 50007 # listen on a non-reserved port number
if len(sys.argv) == 3: # allow host/port as cmdline args too
    myHost, myPort = sys.argv[1:]
numPortSocks = 2 # number of ports for client connects

# make main sockets for accepting new client requests
mainsocks, readsocks, writesocks = [], [], []
for i in range(numPortSocks):
    portsock = socket(AF_INET, SOCK_STREAM) # make a TCP/IP spocket object
    portsock.bind((myHost, myPort)) # bind it to server port number
    portsock.listen(5) # listen, allow 5 pending connect
    mainsocks.append(portsock) # add to main list to identify
    readsocks.append(portsock) # add to select inputs list
    myPort = myPort + 1 # bind on consecutive ports

# event loop: listen and multiplex until server process killed
print 'select-server loop starting'
while 1:
    #print readsocks
    readables, writeables, exceptions = select(readsocks, writesocks, [])
    for sockobj in readables:
        if sockobj in mainsocks: # for ready input sockets
            # port socket: accept new client
            newsock, address = sockobj.accept() # accept should not block
            print 'Connect:', address, id(newsock) # newsock is a new socket
            readsocks.append(newsock) # add to select list, wait
        else:
            # client socket: read next line
            data = sockobj.recv(1024) # recv should not block
            print '\tgot', data, 'on', id(sockobj)
            if not data: # if closed by the clients
                sockobj.close() # close here and remv from
                readsocks.remove(sockobj) # del list else reselected
            else:
                # this may block: should really select for writes too
                sockobj.send('Echo=>%s at %s' % (data, now()))
```

The bulk of this script is the big `while` event loop at the end that calls `select` to find out which sockets are ready for processing (these include main port sockets on which clients can connect, and open client connections). It then loops over all such ready sockets, accepting connections on main port sockets, and reading and echoing input on any client sockets ready for input. Both the `accept` and `recv` calls in this code are guaranteed to not block the server process after `select` returns; because of that, this server can get quickly back to the top of the loop to process newly arrived client requests and already-connected clients' inputs. The net effect is that all new requests and clients are serviced in pseudo-parallel fashion.

To make this process work, the server appends the connected socket for each client to the `readables` list passed to `select`, and simply waits for the socket to show up in the selected input list. For illustration purposes, this server also listens for new clients on more than one port -- on ports 50007 and 50008 in our examples. Because these main port sockets are also interrogated with `select`, connection requests on either port can be accepted without blocking either already-connected clients or new connection requests appearing on the other port. The `select` call returns whatever sockets in list `readables` are ready for processing -- both main port sockets and socket connected to clients currently being processed.

10.4.4.2 Running the select server

Let's run this script locally to see how it does its stuff (the client and server can also be run on different machines, as in prior socket examples). First of all, we'll assume we've already started this server script in one window, and run a few clients to talk to it. The following code is the interaction in two such client windows running on Windows (MS-DOS consoles). The first client simply runs the `echo-client` script twice to contact the server, and the second also kicks off the `testecho` script to spawn eight `echo-client` programs running in parallel. As before, the server simply echoes back whatever text that clients send. Notice that the second client window really runs a script called `echo-client-50008` so as to connect to the second port socket in the server; it's the same as `echo-client`, with a different port number (alas, the original script wasn't designed to input a port):

```
[client window 1]
C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Aug 13 22:52:01 2000'

C:\...\PP2E\Internet\Sockets>python echo-client.py
Client received: 'Echo=>Hello network world at Sun Aug 13 22:52:03 2000'

[client window 2]
C:\...\PP2E\Internet\Sockets>python echo-client-50008.py localhost Sir Lancelot
Client received: 'Echo=>Sir at Sun Aug 13 22:52:57 2000'
Client received: 'Echo=>Lancelot at Sun Aug 13 22:52:57 2000'

C:\...\PP2E\Internet\Sockets>python testecho.py
```

Now, in the next code section is the sort of interaction and output that shows up in the window where the server has been started. The first three connections come from `echo-client` runs; the rest is the result of the eight programs spawned by `testecho` in the second client window. Notice that for `testecho`, new client connections and client inputs are all multiplexed together. If you study the output closely, you'll see that they overlap in time, because all activity is dispatched by the single event loop in the server.^[7] Also note that the server gets an empty string when the client has closed its socket. We take care to close and delete these sockets at the server right away, or else they would be needlessly reselected again and again, each time through the main loop:

[7] And the trace output on the server will probably look a bit different every time it runs. Clients and new connections are interleaved almost at random due to timing differences on the host machines.

```
[server window]
C:\...\PP2E\Internet\Sockets>python select-server.py
```

```
select-server loop starting
Connect: ('127.0.0.1', 1175) 7965520
    got Hello network world on 7965520
    got  on 7965520
Connect: ('127.0.0.1', 1176) 7964288
    got Hello network world on 7964288
    got  on 7964288
Connect: ('127.0.0.1', 1177) 7963920
    got Sir on 7963920
    got Lancelot on 7963920
    got  on 7963920
```

```
    [testecho results]
Connect: ('127.0.0.1', 1178) 7965216
    got Hello network world on 7965216
    got  on 7965216
Connect: ('127.0.0.1', 1179) 7963968
Connect: ('127.0.0.1', 1180) 7965424
    got Hello network world on 7963968
Connect: ('127.0.0.1', 1181) 7962976
    got Hello network world on 7965424
    got  on 7963968
    got Hello network world on 7962976
    got  on 7965424
    got  on 7962976
Connect: ('127.0.0.1', 1182) 7963648
    got Hello network world on 7963648
    got  on 7963648
Connect: ('127.0.0.1', 1183) 7966640
    got Hello network world on 7966640
    got  on 7966640
Connect: ('127.0.0.1', 1184) 7966496
    got Hello network world on 7966496
    got  on 7966496
Connect: ('127.0.0.1', 1185) 7965888
    got Hello network world on 7965888
    got  on 7965888
```

A subtle but crucial point: a `time.sleep` call to simulate a long-running task doesn't make sense the server here -- because all clients are handled by the same single loop, sleeping would pause everything (and defeat the whole point of a multiplexing server). Here are a few additional notes before we move on:

Select call details

Formally, `select` is called with three lists of selectable objects (input sources, output sources, and exceptional condition sources), plus an optional timeout. The timeout argument may be a real wait expiration value in seconds (use floating-point numbers to express fractions of a second), a zero value to mean simply poll and return immediately, or be omitted to mean wait until at least one object is ready (as done in our server script earlier). The call returns a triple of ready objects -- subsets of the first three arguments -- any or all of which may be empty if the timeout expired before sources became ready.

Select portability

The `select` call works only for sockets on Windows, but also works for things like files a pipes on Unix and Macintosh. For servers running over the Internet, of course, sockets are the primary devices we are interested in.

Nonblocking sockets

`select` lets us be sure that socket calls like `accept` and `recv` won't block (pause) the caller but it's also possible to make Python sockets nonblocking in general. Call the `setblocking` method of socket objects to set the socket to blocking or nonblocking mode. For example, given a call like `sock.setblocking(flag)`, the socket `sock` is set to nonblocking mode if the flag is zero, and set to blocking mode otherwise. All sockets start out in blocking mode initially, so socket calls may always make the caller wait.

But when in nonblocking mode, a `socket.error` exception is raised if a `recv` socket call doesn't find any data, or if a `send` call can't immediately transfer data. A script can catch the exception to determine if the socket is ready for processing. In blocking mode, these calls always block until they can proceed. Of course, there may be much more to processing client requests than data transfers (requests may also require long-running computations), nonblocking sockets don't guarantee that servers won't stall in general. They are simply another way to code multiplexing servers. Like `select`, they are better suited when client requests can be serviced quickly.

The asyncore module framework

If you're interested in using `select`, you will probably also be interested in checking out the `asyncore.py` module in the standard Python library. It implements a class-based callback model, where input and output callbacks are dispatched to class methods by a precoded `select` event loop. As such, it allows servers to be constructed without threads or forks. We'll learn more about this tool at the end of [Chapter 15](#).

10.4.5 Choosing a Server Scheme

So when should you use `select` to build a server instead of threads or forks? Needs vary per application, of course, but servers based on the `select` call are generally considered to perform very well when client transactions are relatively short. If they are not short, threads or forks may be a better way to split processing among multiple clients. Threads and forks are especially useful if clients require long-running processing above and beyond socket calls.

It's important to remember that schemes based on `select` (and nonblocking sockets) are not completely immune to blocking. In the example earlier, for instance, the `send` call that echoes text back to a client might block, too, and hence stall the entire server. We could work around that blocking potential by using `select` to make sure that the output operation is ready before we attempt it (e.g., use the `writesocks` list and add another loop to send replies to ready output sockets), albeit at a noticeable cost in program clarity.

In general, though, if we cannot split up the processing of a client's request in such a way that it can be multiplexed with other requests and not block the server's loop, `select` may not be the best way to construct the server. Moreover, `select` also seems more complex than spawning either processes or threads, because we need to manually transfer control among all tasks (for instance, compare the threaded and `select` versions of this server, even without write selects). As usual, though, the degree of that complexity may vary per application.

10.5 A Simple Python File Server

Time for something more realistic. Let's conclude this chapter by putting some of these socket ideas to work in something a bit more useful than echoing text back and forth. [Example 10-10](#) implements both the server-side and client-side logic needed to ship a requested file from server to client machines over a raw socket.

In effect, this script implements a simple file download system. One instance of the script is run on the machine where downloadable files live (the server), and another on the machines you wish to copy files to (the clients). Command-line arguments tell the script which flavor to run and optionally name the server machine and port number over which conversations are to occur. A server instance can respond to any number of client file requests at the port on which it listens, because it serves each in a thread.

Example 10-10. PP2E\Internet\Sockets\getfile.py

```
#####
# implement client and server side logic to transfer an
# arbitrary file from server to client over a socket;
# uses a simple control-info protocol rather than
# separate sockets for control and data (as in ftp),
# dispatches each client request to a handler thread,
# and loops to transfer the entire file by blocks; see
# ftplib examples for a higher-level transport scheme;
#####

import sys, os, thread, time
from socket import *
def now(): return time.ctime(time.time())

blksz = 1024
defaultHost = 'localhost'
defaultPort = 50001

helptext = """
Usage...
server=> getfile.py -mode server          [-port nnn] [-host hhh|localhost]
client=> getfile.py [-mode client] -file fff [-port nnn] [-host hhh|localhost]
"""

def parsecommandline():
    dict = {}
    args = sys.argv[1:]
    while len(args) >= 2:
        dict[args[0]] = args[1]
        args = args[2:]
    return dict

def client(host, port, filename):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))
    sock.send(filename + '\n')
    dropdir = os.path.split(filename)[1]
    file = open(dropdir, 'wb')
    while 1:
        data = sock.recv(blksz)
        if not data: break
        file.write(data)
```



```
sock.close()
file.close()
print 'Client got', filename, 'at', now()

def serverthread(clientsock):
    sockfile = clientsock.makefile('r')           # wrap socket in dup file obj
    filename = sockfile.readline()[:-1]          # get filename up to end-line
    try:
        file = open(filename, 'rb')
        while 1:
            bytes = file.read(blksz)             # read/send 1K at a time
            if not bytes: break                  # until file totally sent
            sent = clientsock.send(bytes)
            assert sent == len(bytes)
    except:
        print 'Error downloading file on server:', filename
    clientsock.close()

def server(host, port):
    serversock = socket(AF_INET, SOCK_STREAM)     # listen on tcp/ip socket
    serversock.bind((host, port))                # serve clients in threads
    serversock.listen(5)
    while 1:
        clientsock, clientaddr = serversock.accept()
        print 'Server connected by', clientaddr, 'at', now()
        thread.start_new_thread(serverthread, (clientsock,))

def main(args):
    host = args.get('-host', defaultHost)        # use args or defaults
    port = int(args.get('-port', defaultPort))   # is a string in argv
    if args.get('-mode') == 'server':           # None if no -mode: client
        if host == 'localhost': host = ''       # else fails remotely
        server(host, port)
    elif args.get('-file'):                      # client mode needs -file
        client(host, port, args['-file'])
    else:
        print helptext

if __name__ == '__main__':
    args = parsecommandline()
    main(args)
```

This script doesn't do much different than the examples we saw earlier. Depending on the command-line arguments passed, it invokes one of two functions:

- The `server` function farms out each incoming client request to a thread that transfers the requested file's bytes.
- The `client` function sends the server a file's name and stores all the bytes it gets back in a local file of the same name.

The most novel feature here is the protocol between client and server: the client starts the conversation by shipping a filename string up to the server, terminated with an end-of-line character, and including the file's directory path in the server. At the server, a spawned thread extracts the requested file's name by reading the client socket, and opens and transfers the requested file back to the client, one chunk of bytes at a time.

10.5.1 Running the File Server and Clients

Since the server uses threads to process clients, we can test both client and server on the same Windows machine. First, let's start a server instance, and execute two client instances on the same machine while the server runs:

```
[server window, localhost]

C:\...\PP2E\Internet\Sockets>python getfile.py -mode server
Server connected by ('127.0.0.1', 1089) at Thu Mar 16 11:54:21 2000
Server connected by ('127.0.0.1', 1090) at Thu Mar 16 11:54:37 2000

[client window, localhost]

C:\...\Internet\Sockets>ls
class-server.py  echo.out.txt      testdir           thread-server.py
echo-client.py   fork-server.py    testecho.py
echo-server.py   getfile.py        testechowait.py

C:\...\Internet\Sockets>python getfile.py -file testdir\python15.lib -port 5000
Client got testdir\python15.lib at Thu Mar 16 11:54:21 2000

C:\...\Internet\Sockets>python getfile.py -file testdir\textfile
Client got testdir\textfile at Thu Mar 16 11:54:37 2000
```

Clients run in the directory where you want the downloaded file to appear -- the client instance code strips the server directory path when making the local file's name. Here the "download" simply copied the requested files up to the local parent directory (the DOS *fc* command compare file contents):

```
C:\...\Internet\Sockets>ls
class-server.py  echo.out.txt      python15.lib      testechowait.py
echo-client.py   fork-server.py    testdir           textfile
echo-server.py   getfile.py        testecho.py       thread-server.py

C:\...\Internet\Sockets>fc /B python1.lib testdir\python15.lib
Comparing files python15.lib and testdir\python15.lib
FC: no differences encountered

C:\...\Internet\Sockets>fc /B textfile testdir\textfile
Comparing files textfile and testdir\textfile
FC: no differences encountered
```

As usual, we can run server and clients on different machines as well. Here the script is being used to run a remote server on a Linux machine and a few clients on a local Windows PC (I added line breaks to some of the command lines to make them fit). Notice that client and server machine times are different now -- they are fetched from different machine's clocks and so may be arbitrarily skewed:

```
[server telnet window: first message is the python15.lib request
in client window]
[lutz@starship lutz]$ python getfile.py -mode server
Server connected by ('166.93.216.248', 1185) at Thu Mar 16 16:02:07 2000
Server connected by ('166.93.216.248', 1187) at Thu Mar 16 16:03:24 2000
Server connected by ('166.93.216.248', 1189) at Thu Mar 16 16:03:52 2000
Server connected by ('166.93.216.248', 1191) at Thu Mar 16 16:04:09 2000
Server connected by ('166.93.216.248', 1193) at Thu Mar 16 16:04:38 2000

[client window 1: started first, runs in thread while other client
requests are made in client window 2, and processed by other threads]
```

```
C:\...\Internet\Sockets>python getfile.py -mode client
                        -host starship.python.net
                        -port 50001 -file python15.lib
Client got python15.lib at Thu Mar 16 14:07:37 2000

C:\...\Internet\Sockets>fc /B python15.lib testdir\python15.lib
Comparing files python15.lib and testdir\python15.lib
FC: no differences encountered

[client window 2: requests made while client window 1 request downloading]
C:\...\Internet\Sockets>python getfile.py
                        -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:02:29 2000

C:\...\Internet\Sockets>python getfile.py
                        -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:04:11 2000

C:\...\Internet\Sockets>python getfile.py
                        -host starship.python.net -file textfile
Client got textfile at Thu Mar 16 14:04:21 2000

C:\...\Internet\Sockets>python getfile.py
                        -host starship.python.net -file index.html
Client got index.html at Thu Mar 16 14:06:22 2000

C:\...\Internet\Sockets>fc textfile testdir\textfile
Comparing files textfile and testdir\textfile
FC: no differences encountered
```

One subtle security point here: the server instance code is happy to send any server-side file whose pathname is sent from a client, as long as the server is run with a username that has read access to the requested file. If you care about keeping some of your server-side files private, you should add logic to suppress downloads of restricted files. I'll leave this as a suggested exercise here, but will implement such filename checks in the `getfile` download tool in [Example 12-1](#).^[8]

[8] We'll see three more `getfile` programs before we leave Internet scripting. The next chapter's `getfile.py` fetches a file with the higher-level FTP interface rather than using raw socket calls, and its `http-getfile` scripts fetch files over the HTTP protocol. [Example 12-1](#) presents a `getfile.cgi` script that transfers file contents over the HTTP port in response to a request made in a web browser client (files are sent as the output of a CGI script). All four of the download schemes presented in this text ultimately use sockets, but only the version here makes that use explicit.

Making Sockets Look Like Files

For illustration purposes, `getfile` uses the socket object `makefile` method to wrap the socket in a file-like object. Once so wrapped, the socket can be read and written using normal file methods; `getfile` uses the file `readline` call to read the filename line sent by the client.

This isn't strictly required in this example -- we could have read this line with the socket `recv` call, too. In general, though, the `makefile` method comes in handy any time you need to pass a socket to an interface that expects a file.

For example, the `pickle` module's `load` and `dump` methods expect an object with a

file-like interface (e.g., `read` and `write` methods), but don't require a physical file. Passing a TCP/IP socket wrapped with the `makefile` call to the pickler allows us to ship serialized Python objects over the Internet. See [Chapter 16](#), for more details on object serialization interfaces.

More generally, any component that expects a file-like method protocol will gladly accept a socket wrapped with a socket object `makefile` call. Such interfaces will also accept strings wrapped with the built-in `StringIO` module, and any other sort of object that supports the same kinds of method calls as built-in file objects. As always in Python, we code to protocols -- object interfaces -- not to specific datatypes.

10.5.2 Adding a User-Interface Frontend

You might have noticed that we have been living in the realm of the command line for all of this chapter -- our socket clients and servers have been started from simple DOS or Linux shells. There is nothing stopping us from adding a nice point-and-click user interface to some of these scripts, though; GUI and network scripting are not mutually exclusive techniques. In fact, they can be arguably sexy when used together well.

For instance, it would be easy to implement a simple Tkinter GUI frontend to the client-side portion of the `getfile` script we just met. Such a tool, run on the client machine, may simply pop up a window with `Entry` widgets for typing the desired filename, server, and so on. Once download parameters have been input, the user interface could either import and call the `getfile.client` function with appropriate option arguments, or build and run the implied `getfile.py` command line using tools such as `os.system`, `os.fork`, `thread`, etc.

10.5.2.1 Using Frames and command lines

To help make this all more concrete, let's very quickly explore a few simple scripts that add a Tkinter frontend to the `getfile` client-side program. The first, in [Example 10-11](#), creates a dialog for inputting server, port, and filename information, and simply constructs the corresponding `getfile` command line and runs it with `os.system`.

Example 10-11. PP2E\Internet\Sockets\getfilegui-1.py

```
#####
# launch getfile script client from simple Tkinter GUI;
# could also or os.fork+exec, os.spawnv (see Launcher);
# windows: replace 'python' with 'start' if not on path;
#####

import sys, os
from Tkinter import *
from tkMessageBox import showinfo

def onReturnKey():
    cmdline = ('python getfile.py -mode client -file %s -port %s -host %s' %
              (content['File'].get(),
               content['Port'].get(),
               content['Server'].get()))
    os.system(cmdline)
    showinfo('getfilegui-1', 'Download complete')

box = Frame(Tk())
```

```
box.pack(expand=YES, fill=X)
lcol, rcol = Frame(box), Frame(box)
lcol.pack(side=LEFT)
rcol.pack(side=RIGHT, expand=Y, fill=X)

labels = ['Server', 'Port', 'File']
content = {}
for label in labels:
    Label(lcol, text=label).pack(side=TOP)
    entry = Entry(rcol)
    entry.pack(side=TOP, expand=YES, fill=X)
    content[label] = entry

box.master.title('getfilegui-1')
box.master.bind('<Return>', (lambda event: onReturnKey()))
mainloop()
```

When run, this script creates the input form shown in [Figure 10-1](#). Pressing the Enter key (<Return>) runs a client-side instance of the `getfile` program; when the generated `getfile` command line is finished, we get the verification pop-up displayed in [Figure 10-2](#).

Figure 10-1. getfilegui-1 in action

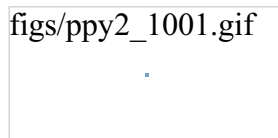
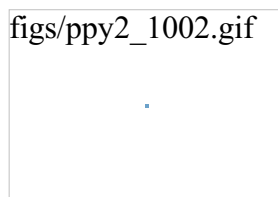


Figure 10-2. getfilegui-1 verification pop-up



10.5.2.2 Using grids and function calls

The first user-interface script ([Example 10-11](#)) uses the `pack` geometry manager and `Frames` to layout the input form, and runs the `getfile` client as a stand-alone program. It's just as easy to use the `grid` manager for layout, and `import` and call the client-side logic function instead of running a program. The script in [Example 10-12](#) shows how.

Example 10-12. PP2E\Internet\Sockets\getfilegui-2.py

```
#####
# same, but with grids and import+call, not packs and cmdline;
# direct function calls are usually faster than running files;
#####

import getfile
from Tkinter import *
from tkMessageBox import showinfo

def onSubmit():
    getfile.client(content['Server'].get(),
                  int(content['Port'].get()),
```

```
        content['File'].get())
    showinfo('getfilegui-2', 'Download complete')

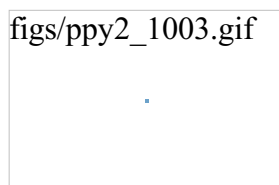
box    = Tk()
labels = ['Server', 'Port', 'File']
rownum = 0
content = {}
for label in labels:
    Label(box, text=label).grid(col=0, row=rownum)
    entry = Entry(box)
    entry.grid(col=1, row=rownum, sticky=E+W)
    content[label] = entry
    rownum = rownum + 1

box.columnconfigure(0, weight=0) # make expandable
box.columnconfigure(1, weight=1)
Button(text='Submit', command=onSubmit).grid(row=rownum, col=0, columnspan=2)

box.title('getfilegui-2')
box.bind('<Return>', (lambda event: onSubmit()))
mainloop()
```

This version makes a similar window ([Figure 10-3](#)), but adds a button at the bottom that does the same thing as an Enter key press -- it runs the `getfile` client procedure. Generally speaking, importing and calling functions (as done here) is faster than running command lines, especially if done more than once. The `getfile` script is set up to work either way -- as program or function library.

Figure 10-3. getfilegui-2 in action



10.5.2.3 Using a reusable form-layout class

If you're like me, though, writing all the GUI form layout code in those two scripts can seem a bit tedious, whether you use packing or grids. In fact, it became so tedious to me that I decided to write a general-purpose form-layout class, shown in [Example 10-13](#), that handles most of the GUI layout grunt work.

Example 10-13. PP2E\Internet\Sockets\form.py

```
# a reusable form class, used by getfilegui (and others)

from Tkinter import *
entrysize = 40

class Form:
    def __init__(self, labels, parent=None):
        # add non-modal form box
        # pass field labels list
        box = Frame(parent)
        box.pack(expand=YES, fill=X)
        rows = Frame(box, bd=2, relief=GROOVE)
        # box has rows, button
        lcol = Frame(rows)
        # rows has lcol, rcol
        rcol = Frame(rows)
        # button or return key,
        rows.pack(side=TOP, expand=Y, fill=X)
        # runs onSubmit method
        lcol.pack(side=LEFT)
```

```
    rcol.pack(side=RIGHT, expand=Y, fill=X)
    self.content = {}
    for label in labels:
        Label(lcol, text=label).pack(side=TOP)
        entry = Entry(rcol, width=entrysize)
        entry.pack(side=TOP, expand=YES, fill=X)
        self.content[label] = entry
    Button(box, text='Cancel', command=self.onCancel).pack(side=RIGHT)
    Button(box, text='Submit', command=self.onSubmit).pack(side=RIGHT)
    box.master.bind('<Return>', (lambda event, self=self: self.onSubmit()))

def onSubmit(self):
    for key in self.content.keys():
        print key, '\t=>\t', self.content[key].get()

def onCancel(self):
    Tk().quit()

class DynamicForm(Form):
    def __init__(self, labels=None):
        import string
        labels = string.split(raw_input('Enter field names: '))
        Form.__init__(self, labels)
    def onSubmit(self):
        print 'Field values...'
        Form.onSubmit(self)
        self.onCancel()

if __name__ == '__main__':
    import sys
    if len(sys.argv) == 1:
        Form(['Name', 'Age', 'Job'])
    else:
        DynamicForm()
    mainloop()
```

Running this module standalone triggers its self-test code at the bottom. Without arguments (and when double-clicked in a Windows file explorer), the self-test generates a form with canned input fields captured in [Figure 10-4](#), and displays the fields' values on Enter key presses or Submit button clicks:

```
C:\...\PP2E\Internet\Sockets>python form.py
Job      =>      Educator, Entertainer
Age      =>      38
Name     =>      Bob
```

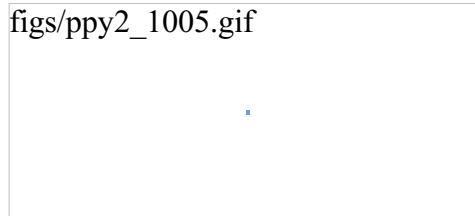
Figure 10-4. Form test, canned fields



With a command-line argument, the form class module's self-test code prompts for an arbitrary set of field names for the form; fields can be constructed as dynamically as we like. [Figure 10-5](#) shows the input form constructed in response to the following console interaction. Field names could be accepted on the command line, too, but `raw_input` works just as well for simple tests like this. In this mode, the GUI goes away after the first submit, because `DynamicForm.onSubmit` says so:

```
C:\...\PP2E\Internet\Sockets>python form.py -
Enter field names: Name Email Web Locale
Field values...
Email   =>      lutz@rmi.net
Locale  =>      Colorado
Web     =>      http://rmi.net/~lutz
Name    =>      mel
```

Figure 10-5. Form test, dynamic fields



And last but not least, [Example 10-14](#) shows the `getfile` user interface again, this time constructed with the reusable form layout class. We need to fill in only the form labels list, and provide an `onSubmit` callback method of our own. All of the work needed to construct the form comes "for free," from the imported and widely reusable `Form` superclass.

Example 10-14. PP2E\Internet\Sockets\getfilegui.py

```
#####
# launch getfile client with a reusable gui form class;
# os.chdir to target local dir if input (getfile stores in cwd);
# to do: use threads, show download status and getfile prints;
#####

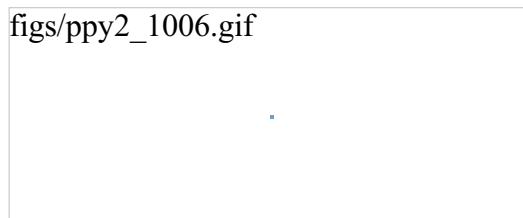
from form import Form
from Tkinter import Tk, mainloop
from tkMessageBox import showinfo
import getfile, os

class GetfileForm(Form):
    def __init__(self, oneshot=0):
        root = Tk()
        root.title('getfilegui')
        labels = ['Server Name', 'Port Number', 'File Name', 'Local Dir?']
        Form.__init__(self, labels, root)
        self.oneshot = oneshot
    def onSubmit(self):
        Form.onSubmit(self)
        localdir = self.content['Local Dir?'].get()
        portnumber = self.content['Port Number'].get()
        servername = self.content['Server Name'].get()
        filename = self.content['File Name'].get()
        if localdir:
            os.chdir(localdir)
        portnumber = int(portnumber)
        getfile.client(servername, portnumber, filename)
        showinfo('getfilegui', 'Download complete')
        if self.oneshot: Tk().quit() # else stay in last localdir

if __name__ == '__main__':
    GetfileForm()
    mainloop()
```


The form layout class imported here can be used by any program that needs to input form-like data; when used in this script, we get a user-interface like that shown in [Figure 10-6](#) under Windows (and similar on other platforms).

Figure 10-6. getfilegui in action



Pressing this form's Submit button or the Enter key makes the `getfilegui` script call the imported `getfile.client` client-side function as before. This time, though, we also first change to the local directory typed into the form, so that the fetched file is stored there (`getfile` stores in the current working directory, whatever that may be when it is called). As usual, we can use this interface to connect to servers running locally on the same machine, or remotely. Here is some of the interaction we get for both modes:

```
[talking to a local server]
C:\...\PP2E\Internet\Sockets>python getfilegui.py
Port Number      =>      50001
Local Dir?       =>      temp
Server Name      =>      localhost
File Name        =>      testdir\python15.lib
Client got testdir\python15.lib at Tue Aug 15 22:32:34 2000

[talking to a remote server]
[lutz@starship lutz]$ /usr/bin/python getfile.py -mode server -port 51234
Server connected by ('38.28.130.229', 1111) at Tue Aug 15 21:48:13 2000

C:\...\PP2E\Internet\Sockets>python getfilegui.py
Port Number      =>      51234
Local Dir?       =>      temp
Server Name      =>      starship.python.net
File Name        =>      public_html/index.html
Client got public_html/index.html at Tue Aug 15 22:42:06 2000
```

One caveat worth pointing out here: the GUI is essentially dead while the download is in progress (even screen redraws aren't handled -- try covering and uncovering the window and you'll see what I mean). We could make this better by running the download in a thread, but since we'll see how in the next chapter, you should consider this problem a preview.

In closing, a few final notes. First of all, I should point out that the scripts in this chapter use Tkinter tricks we've seen before and won't go into here in the interest of space; be sure to see the GUI chapters in this book for implementation hints.

Keep in mind, too, that these interfaces all just add a GUI on top of the existing script to reuse its code; any command-line tool can be easily GUI-ified in this way to make them more appealing and user-friendly. In the next chapter, for example, we'll meet a more useful client-side Tkinter user interface for reading and sending email over sockets (PyMailGui), which largely just adds a GUI to mail-processing tools. Generally speaking, GUIs can often be added as almost an afterthought to a program. Although the degree of user-interface and core logic separation can vary per program, keeping the two distinct makes it easier to focus on each.

And finally, now that I've shown you how to build user interfaces on top of this chapter's `getfile`, I should also say that they aren't really as useful as they might seem. In particular, `getfile` clients can talk only to machines that are running a `getfile` server. In the next chapter, we'll discover another way to download files -- FTP -- which also runs on sockets, but provides a higher-level interface, and is available as a standard service on many machines on the Net. We don't generally need to start up a custom server to transfer files over FTP, the way we do with `getfile`. In fact, the user-interface scripts in this chapter could be easily changed to fetch the desired file with Python's FTP tools, instead of the `getfile` module. But rather than spilling all the beans here, I'll just say "read on."

Using Serial Ports on Windows

Sockets, the main subject of this chapter, are the programmer's interface to network connections in Python scripts. As we've seen, they let us write scripts that converse with computers arbitrarily located on a network, and they form the backbone of the Internet and the Web.

If you're looking for a more low-level way to communicate with devices in general, though, you may also be interested in the topic of Python's serial port interfaces. This isn't quite related to Internet scripting and applies only on Windows machines, but it's similar enough in spirit and is discussed often enough on the Net to merit a quick look here.

Serial ports are known as "COM" ports on Windows (not to be confused with the COM object model), and are identified as "COM1", "COM2", and so on. By using interfaces to these ports, scripts may engage in low-level communication with things like mice, modems, and a wide variety of serial devices. Serial port interfaces are also used to communicate with devices connected over infrared ports (e.g., hand-held computers and remote modems). There are often other higher-level ways to access such devices (e.g., the PyRite package for ceasing Palm Pilot databases, or RAS for using modems), but serial port interfaces let scripts tap into raw data streams and implement device protocols of their own.

There are at least three ways to send and receive data over serial ports in Python scripts -- a public domain C extension package known as Serial, the proprietary MSComm COM server object interface published by Microsoft, and the low-level CreateFile file API call exported by the Python Windows extensions package, available via links at <http://www.python.org>.

Unfortunately, there's no time to cover any of these in detail in this text. For more information, the O'Reilly book Python Programming on Win32 includes an entire section dedicated to serial port communication topics. Also be sure to use the search tools at Python's web site for up-to-date details on this front.

Chapter 10. Network Scripting

[Section 10.1. "Tune in, Log on, and Drop out"](#)

[Section 10.2. Plumbing the Internet](#)

[Section 10.3. Socket Programming](#)

[Section 10.4. Handling Multiple Clients](#)

[Section 10.5. A Simple Python File Server](#)

11.1 "Socket to Me!"

The previous chapter introduced Internet fundamentals and explored sockets -- the underlying communications mechanism over which bytes flow on the Net. In this chapter, we climb the encapsulation hierarchy one level, and shift our focus to Python tools that support the client-side interfaces of common Internet protocols.

We talked about the Internet's higher-level protocols in the abstract at the start of the last chapter, and you should probably review that material if you skipped over it the first time around. In short, protocols define the structure of the conversations that take place to accomplish most of the Internet tasks we're all familiar with -- reading email, transferring files by FTP, fetching web pages, and so on.

At the most basic level, all these protocol dialogs happen over sockets using fixed and standard message structures and ports, so in some sense this chapter builds upon the last. But as we'll see, Python's protocol modules hide most of the underlying details -- scripts generally need deal only with simple objects and methods, and Python automates the socket and messaging logic required by the protocol.

In this chapter, we'll concentrate on the FTP and email protocol modules in Python, and peek at a few others along the way (NNTP news, HTTP web pages, and so on). All of the tools employed in examples here are in the standard Python library and come with the Python system. All of the examples here are also designed to run on the client side of a network connection -- these scripts connect to an already-running server to request interaction and can be run from a simple PC. In the next chapter, we'll move on to explore scripts designed to be run on the server side instead. For now, let's focus on the client.


```
Version = '1.5' # version to download
tarname = 'python%s.tar.gz' % Version # remote/local file name

print 'Connecting...'
localfile = open(tarname, 'wb') # where to store download
connection = FTP('ftp.python.org') # connect to ftp site
connection.login() # default is anonymous login
connection.cwd('pub/python/src') # xfer 1k at a time to localfile

print 'Downloading...'
connection.retrbinary('RETR ' + tarname, localfile.write, 1024)
connection.quit()
localfile.close()

print 'Unpacking...'
os.system('gzip -d ' + tarname) # decompress
os.system('tar -xvf ' + tarname[:-3]) # strip .gz

print 'Building...'
os.chdir('Python-' + Version) # build Python itself
os.system('./configure') # assumes unix-style make
os.system('make')
os.system('make test')
print 'Done: see Python-%s/python.' % Version
```

Most of the FTP protocol details are encapsulated by the Python `ftplib` module imported here. This script uses some of the simplest interfaces in `ftplib` (we'll see others in a moment), but the are representative of the module in general:

```
connection = FTP('ftp.python.org') # connect to ftp site
```

To open a connection to a remote (or local) FTP server, create an instance of the `ftplib.FTP` object, passing in the name (domain or IP-style) of the machine you wish to connect to. Assuming this call doesn't throw an exception, the resulting FTP object exports methods that correspond to the usual FTP operations. In fact, Python scripts act much like typical FTP client programs -- just replace commands you would normally type or select with method calls:

```
connection.login() # default is anonymous login
connection.cwd('pub/python/src') # xfer 1k at a time to localfile
```

Once connected, we log in, and go to the remote directory we want to fetch a file from. The `login` method allows us to pass in additional optional arguments to specify a username and password; by default it performs anonymous FTP:

```
connection.retrbinary('RETR ' + tarname, localfile.write, 1024)
connection.quit()
```

Once we're in the target directory, we simply call the `retrbinary` method to download the target server file in binary mode. The `retrbinary` call will take awhile to complete, since it must download a big file. It gets three arguments:

- An FTP command string -- here, a string `RETR filename`, which is the standard format for FTP retrievals.
- A function or method to which Python passes each chunk of the downloaded file's bytes -- here, the `write` method of a newly created and opened local file.
- A size for those chunks of bytes -- here, 1024 bytes are downloaded at a time, but the default is reasonable if this argument is omitted.

Because this script creates a local file named `localfile`, of the same name as the remote file being fetched, and passes its `write` method to the FTP retrieval method, the remote file's content will automatically appear in a local, client-side file after the download is finished. By the way, notice that this file is opened in "wb" binary output mode; if this script is run on Windows, we want to avoid automatically expanding and `\n` bytes into `\r\n` byte sequences (that happens automatically on Windows when writing files opened in "w" text mode).

Finally, we call the FTP `quit` method to break the connection with the server and manually `close` the local file to force it to be complete before it is further processed by the shell command spawned by `os.system` (it's not impossible that parts of the file are still held in buffers before the `close` call):

```
connection.quit()
localfile.close()
```

And that's all there is to it; all the FTP, socket, and networking details are hidden behind the `ftplib` interface module. Here is this script in action on a Linux machine, with a couple thousand output lines cut in the interest of brevity:

```
[lutz@starship test]$ python getpython.py
Connecting...
Downloading...
Unpacking...
Python-1.5/
Python-1.5/Doc/
Python-1.5/Doc/ref/
Python-1.5/Doc/ref/.cvsignore
Python-1.5/Doc/ref/fixps.py
...
...lots of tar lines deleted...
...
Python-1.5/Tools/webchecker/webchecker.py
Python-1.5/Tools/webchecker/websucker.py
Building...
creating cache ./config.cache
checking MACHDEP... linux2
checking CCC...
checking for --without-gcc... no
checking for gcc... gcc
...
...lots of build lines deleted...
...
Done: see Python-1.5/python.

[lutz@starship test]$ cd Python-1.5/
[lutz@starship Python-1.5]$ ./python
Python 1.5 (#1, Jul 12 2000, 12:35:52) [GCC egcs-2.91.66 19990314/Li on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> print 'The Larch!'
The Larch!
```

Such a script could be automatically executed at regular intervals (e.g., by a Unix cron job) to update a local Python install with a fresh build. But the thing to notice here is that this otherwise typical Python script fetches information from an arbitrarily remote FTP site and machine. Given an Internet link, any information published by an FTP server on the Net can be fetched by and incorporated into Python scripts using interfaces such as these.

11.2.1.1 Using `urllib` to FTP files

In fact, FTP is just one way to transfer information across the Net, and there are more general tools in the Python library to accomplish the prior script's download. Perhaps the most straightforward is the Python `urllib` module: given an Internet address string -- a URL, or Universal Resource Locator -- this module opens a connection to the specified server and returns a file-like object ready to be read with normal file object method calls (e.g., `read`, `readlines`).

We can use such a higher-level interface to download anything with an address on the Web -- files published by FTP sites (using URLs that start with "ftp://"), web pages and outputs of scripts that live on remote servers (using "http://" URLs), local files (using "file://" URLs), Gopher server data, and more. For instance, the script in [Example 11-2](#) does the same as the one in [Example 11-1](#), but it uses the general `urllib` module to fetch the source distribution file, instead of the protocol-specific `ftplib`.

Example 11-2. PP2E\Internet\Ftp\getpython-urllib.py

```
#!/usr/local/bin/python
#####
# A Python script to download and build Python's source code
# use higher-level urllib instead of ftplib to fetch file
# urllib supports ftp, http, and gopher protocols, and local files
# urllib also allows downloads of html pages, images, text, etc.;
# see also Python html/xml parsers for web pages fetched by urllib;
#####

import os
import urllib                # socket-based web tools
Version = '1.5'              # version to download
tarname = 'python%s.tar.gz' % Version # remote/local file name

remoteaddr = 'ftp://ftp.python.org/pub/python/src/' + tarname
print 'Downloading', remoteaddr

# this works too:
# urllib.urlretrieve(remoteaddr, tarname)

remotefile = urllib.urlopen(remoteaddr) # returns input file-like object
localfile = open(tarname, 'wb')        # where to store data locally
localfile.write(remotefile.read())
localfile.close()
remotefile.close()

# the rest is the same
execfile('buildPython.py')
```

Don't sweat the details of the URL string used here; we'll talk much more about URLs in the next chapter. We'll also use `urllib` again in this and later chapters to fetch web pages, format generated URL strings, and get the output of remote scripts on the Web.^[1] Technically speaking `urllib` supports a variety of Internet protocols (HTTP, FTP, Gopher, and local files), is only used for reading remote objects (not writing or uploading them), and retrievals must generally be run in threads if blocking is a concern. But the basic interface shown in this script is straightforward. The call:

[1] For more `urllib` download examples, see the section on HTTP in this chapter. In bigger terms, tools like `urllib.urlopen` allow scripts to both download remote files and invoke programs that are located on a remote server machine. In [Chapter 12](#), we'll also see that `urllib` includes tools for formatting (escaping) URL strings for

safe transmission.

```
remotefile = urllib.urlopen(remoteaddr)      # returns input file-like object
```

contacts the server named in the `remoteaddr` URL string and returns a file-like object connected to its download stream (an FTP-based socket). Calling this file's `read` method pulls down the file's contents, which are written to a local client-side file. An even simpler interface:

```
urllib.urlretrieve(remoteaddr, tarname)
```

also does the work of opening a local file and writing the downloaded bytes into it -- things we do manually in the script as coded. This comes in handy if we mean to download a file, but is less useful if we want to process its data immediately.

Either way, the end result is the same: the desired server file shows up on the client machine. The remainder of the script -- unpacking and building -- is identical to the original version, so it's been moved to a reusable Python file run with the `execfile` built-in (recall that `execfile` runs a file as though its code were pasted into the place where the `execfile` appears). The script is shown in [Example 11-3](#).

Example 11-3. PP2E\Internet\Ftp\buildPython.py

```
#!/usr/local/bin/python
#####
# A Python script to build Python from its source code.
# Run me in directory where Python source distribution lives.
#####

import os
Version = '1.5'                # version to build
tarname = 'python%s.tar.gz' % Version  # remote/local file name

print 'Unpacking...'
os.system('gzip -d ' + tarname)      # decompress file
os.system('tar -xvf ' + tarname[:-3]) # untar without '.gz'

print 'Building...'
os.chdir('Python-' + Version)        # build Python itself
os.system('./configure')              # assumes unix-style make
os.system('make')
os.system('make test')
print 'Done: see Python-%s/python.' % Version
```

The output this time is almost identical to the output of [Example 11-1](#), so I'll show only a few portions (the `gzip` message appears if you don't delete a tar file left by a run in the past):

```
[lutz@starship test]$ python getpython-urllib.py
Downloading ftp://ftp.python.org/pub/python/src/python1.5.tar.gz
Unpacking...
gzip: python1.5.tar already exists; do you wish to overwrite (y or n)? y
...tar lines...
Building...
...build lines...
Done: see Python-1.5/python.

[lutz@starship test]$ python buildPython.py
Unpacking...
...tar and build lines...
```

In fact, although the original script is all top-level code that runs immediately and accomplishes only one task, there really are two potentially reusable activities within it: fetching a file and building Python from source. By splitting each part off into a module of its own, we can reuse its program logic in other contexts, which naturally leads us to the topic in the next section.

11.2.2 FTP get and put Utilities

Almost invariably, when I present the `ftplib` interfaces in Python classes, students ask why programmers need to supply the RETR string in the retrieval method. It's a good question -- the RETR string is the name of the download command in the FTP protocol, but `ftplib` is supposed to encapsulate that protocol. As we'll see in a moment, we have to supply an arguably odd STOR string for uploads as well. It's boilerplate code that you accept on faith once you see it, but that begs the question. You could always email Guido a proposed `ftplib` patch, but that's not really a good answer for beginning Python students.^[2]

[2] This is one point in the class where I also usually threaten to write Guido's home phone number on the whiteboard. But that's generally an empty promise made just for comic effect. If you do want to discuss Python language issues, Guido's email address, as well as contact points for other Python core developers, are readily available on the Net. As someone who's gotten anonymous Python-related calls at home, I never do give out phone numbers (and dialing 1-800-Hi-Guido is only funny the first time).

A better answer is that there is no law against extending the standard library modules with higher-level interfaces of our own -- with just a few lines of reusable code, we can make the FTP interface look any way we want in Python. For instance, we could, once and for all, write utility modules that wrap the `ftplib` interfaces to hide the RETR string. If we place these utility modules in a directory on PYTHONPATH, they become just as accessible as `ftplib` itself, automatically reusable in any Python script we write in the future. Besides removing the RETR string requirement, a wrapper module could also make assumptions that simplify FTP operations into single function calls.

For instance, given a module that encapsulates and simplifies `ftplib`, our Python fetch-and-build script could be further reduced to the script shown in [Example 11-4](#) -- essentially just a function call and file execution.

Example 11-4. PP2E\Internet\Ftp\getpython-modular.py

```
#!/usr/local/bin/python
#####
# A Python script to download and build Python's source code.
# Uses getfile.py, a utility module which encapsulates ftp step.
#####

import getfile
Version = '1.5' # version to download
tarname = 'python%s.tar.gz' % Version # remote/local file name

# fetch with utility
getfile.getfile(tarname, 'ftp.python.org', 'pub/python/src')

# rest is the same
execfile('buildPython.py')
```

Besides having a line count that is much more impressive to marketeers, the meat of this script has been split off into files for reuse elsewhere. If you ever need to download a file again, simply import an existing function rather than copying code with cut-and-paste editing. Changes in download operations would need to be made in only one file, not everywhere we've copied boilerplate code; `getfile.getfile` could even be changed to use `urllib` instead of `ftplib` without effecting any of its clients. It's good engineering.

11.2.2.1 Download utility

So just how would we go about writing such an FTP interface wrapper (he asks, knowingly)? Given the `ftplib` library module, wrapping downloads of a particular file in a particular directory is straightforward. Connected FTP objects support two download methods:

- The `retrbinary` method downloads the requested file in binary mode, sending its bytes in chunks to a supplied function, without line-feed mapping. Typically, the supplied function is a write method of an open local file object, such that the bytes are placed in the local file on the client.
- The `retrlines` method downloads the requested file in ASCII text mode, sending each line of text to a supplied function with all end-of-line characters stripped. Typically, the supplied function adds a `\n` newline (mapped appropriately for the client machine), and writes the line to a local file.

We will meet the `retrlines` method in a later example; the `getfile` utility module in [Example 11-5](#) transfers in binary mode always with `retrbinary`. That is, files are downloaded exactly as they were on the server, byte for byte, with the server's line-feed conventions in text files. You may need to convert line-feeds after downloads if they look odd in your text editor -- see the converter tools in [Chapter 5](#), for pointers.

Example 11-5. PP2E\Internet\Ftp\getfile.py

```
#!/usr/local/bin/python
#####
# Fetch an arbitrary file by ftp. Anonymous
# ftp unless you pass a user=(name, pswd) tuple.
# Gets the Monty Python theme song by default.
#####

from ftplib import FTP          # socket-based ftp tools
from os.path import exists     # file existence test

file = 'sousa.au'              # default file coordinates
site = 'ftp.python.org'       # monty python theme song
dir = 'pub/python/misc'

def getfile(file=file, site=site, dir=dir, user=(), verbose=1, force=0):
    """
    fetch a file by ftp from a site/directory
    anonymous or real login, binary transfer
    """
    if exists(file) and not force:
        if verbose: print file, 'already fetched'
    else:
        if verbose: print 'Downloading', file
        local = open(file, 'wb')          # local file of same name
```

```
try:
    remote = FTP(site)                # connect to ftp site
    apply(remote.login, user)         # anonymous=() or (name, pswd)
    remote.cwd(dir)
    remote.retrbinary('RETR ' + file, local.write, 1024)
    remote.quit()
finally:
    local.close()                    # close file no matter what
    if verbose: print 'Download done.' # caller handles exceptions

if __name__ == '__main__': getfile() # anonymous python.org login
```

This module is mostly just a repackaging of the FTP code we used to fetch the Python source distribution earlier, to make it simpler and reusable. Because it is a callable function, the exported `getfile.getfile` here tries to be as robust and generally useful as possible, but even a function this small implies some design decisions. Here are a few usage notes:

FTP mode

The `getfile` function in this script runs in anonymous FTP mode by default, but a two-item tuple containing a username and password string may be passed to the `user` argument to log in to the remote server in non-anonymous mode. To use anonymous FTP, either don't pass the `user` argument or pass it an empty tuple, `()`. The FTP object `login` method allows two optional arguments to denote a username and password, and the `apply` call in [Example 11-5](#) sends it whatever argument tuple you pass to `user`.

Processing modes

If passed, the last two arguments (`verbose`, `force`) allow us to turn off status messages printed to the `stdout` stream (perhaps undesirable in a GUI context) and force downloads to happen even if the file already exists locally (the download overwrites the existing local file).

Exception protocol

The caller is expected to handle exceptions; this function wraps downloads in a `try/finally` statement to guarantee that the local output file is closed, but lets exceptions propagate. If used in a GUI or run from a thread, for instance, exceptions may require special handling unknown in this file.

Self-test

If run standalone, this file downloads a `sousa.au` audio file from <http://www.python.org> as a self-test, but the function will normally be passed FTP filenames, site names, and directory names as well.

File mode

This script is careful to open the local output file in "wb" binary mode to suppress end-line mapping, in case it is run on Windows. As we learned in [Chapter 2](#), it's not impossible that true binary data files may have bytes whose value is equal to a `\n` line-feed character; opening in "w" text mode instead would make these bytes be automatically expanded to a `\r\n` two-byte sequence when written locally on Windows. This is only an issue for portability to Windows (mode "w" works elsewhere). Again, see [Chapter 5](#) for line-feed converter tools.

Directory model

This function currently uses the same filename to identify both the remote file and the local file where the download should be stored. As such, it should be run in the directory where you want the file to show up; use `os.chdir` to move to directories if needed. (We could instead assume filename is the local file's name, and strip the local directory with `os.path.split` to get the remote name, or accept two distinct filename arguments -- local and remote.)

Notice also that, despite its name, this module is very different than the `getfile.py` script we studied at the end of the sockets material in the previous chapter. The socket-based `getfile` implemented client and server-side logic to download a server file to a client machine over raw sockets.

This new `getfile` here is a client-side tool only. Instead of raw sockets, it uses the simpler FTP protocol to request a file from a server; all socket-level details are hidden in the `ftplib` module's implementation of the FTP client protocol. Furthermore, the server here is a perpetually running program on the server machine, which listens for and responds to FTP requests on a socket, on the dedicated FTP port (number 21). The net functional effect is that this script requires an FTP server to be running on the machine where the desired file lives, but such a server is much more likely to be available.

11.2.2.2 Upload utility

While we're at it, let's write a script to upload a single file by FTP to a remote machine. The upload interfaces in the FTP module are symmetric with the download interfaces. Given a connected FTP object:

- Its `storbinary` method can be used to upload bytes from an open local file object.
- Its `storlines` method can be used to upload text in ASCII mode from an open local file object.

Unlike the download interfaces, both of these methods are passed a file object as a whole, not a file object method (or other function). We will meet the `storlines` method in a later example. The utility module in [Example 11-6](#) uses `storbinary` such that the file whose name is passed in is always uploaded verbatim -- in binary mode, without line-feed translations for the target machine's conventions. If this script uploads a text file, it will arrive exactly as stored on the machine it came from, client line-feed markers and all.

Example 11-6. PP2E\Internet\Ftp\putfile.py

```
#!/usr/local/bin/python
#####
# Store an arbitrary file by ftp.  Anonymous
# ftp unless you pass a user=(name, pswd) tuple.
#####

import ftplib                                # socket-based ftp tools

file = 'sousa.au'                             # default file coordinates
site = 'starship.python.net'                 # monty python theme song
```

```
dir = 'upload'

def putfile(file=file, site=site, dir=dir, user=(), verbose=1):
    """
    store a file by ftp to a site/directory
    anonymous or real login, binary transfer
    """
    if verbose: print 'Uploading', file
    local = open(file, 'rb') # local file of same name
    remote = ftplib.FTP(site) # connect to ftp site
    apply(remote.login, user) # anonymous or real login
    remote.cwd(dir)
    remote.storbinary('STOR ' + file, local, 1024)
    remote.quit()
    local.close()
    if verbose: print 'Upload done.'

if __name__ == '__main__':
    import sys, getpass
    pswd = getpass.getpass(site + ' pswd?') # filename on cmdline
    putfile(file=sys.argv[1], user=('lutz', pswd)) # non-anonymous login
```

Notice that for portability, the local file is opened in "rb" binary mode this time to suppress automatic line-feed character conversions in case this is run on Windows; if this is binary information, we don't want any bytes that happen to have the value of the `\r` carriage-return character to mysteriously go away during the transfer.

Also observe that the standard Python `getpass.getpass` is used to ask for an FTP password in standalone mode. Like the `raw_input` built-in function, this call prompts for and reads a line of text from the console user; unlike `raw_input`, `getpass` does not echo typed characters on the screen at all (in fact, on Windows it uses the low-level direct keyboard interface we met in the stream redirection section of [Chapter 2](#)). This comes in handy for protecting things like passwords from potentially prying eyes.

Like the download utility, this script uploads a local copy of an audio file by default as a self-test but you will normally pass in real remote filename, site name, and directory name strings. Also like the download utility, you may pass a `(username, password)` tuple to the `user` argument to trigger non-anonymous FTP mode (anonymous FTP is the default).

11.2.2.3 Playing the Monty Python theme song

Wake up -- it's time for a bit of fun. Let's make use of these scripts to transfer and play the Monty Python theme song audio file maintained at Python's web site. First off, let's write a module that downloads and plays the sample file, as shown in [Example 11-7](#).

Example 11-7. PP2E\Internet\Ftp\sousa.py

```
#!/usr/local/bin/python
#####
# Usage: % sousa.py
# Fetch and play the Monty Python theme song.
# This may not work on your system as is: it
# requires a machine with ftp access, and uses
# audio filters on Unix and your .au player on
# Windows. Configure playfile.py as needed.
#####

import os, sys
from PP2E.Internet.Ftp.getfile import getfile
```

```
from PP2E.Internet.Ftp.playfile import playfile
sample = 'sousa.au'

getfile(sample)      # fetch audio file by ftp
playfile(sample)    # send it to audio player
```

This script will run on any machine with Python, an Internet link, and a recognizable audio player; it works on my Windows laptop with a dialup Internet connection (if I could insert an audio file hyperlink here to show what it sounds like, I would):

```
C:\...\PP2E\Internet\Ftp>python sousa.py
Downloading sousa.au
Download done.
```

```
C:\...\PP2E\Internet\Ftp>python sousa.py
sousa.au already fetched
```

The `getfile` and `putfile` modules can be used to move the sample file around, too. Both can either be imported by clients that wish to use their functions, or run as top-level programs to trigger self-tests. Let's run these scripts from a command line and the interactive prompt to see how they work. When run standalone, parameters are passed in the command line, and the default file settings are used:

```
C:\...\PP2E\Internet\Ftp>python putfile.py sousa.au
starship.python.net pswd?
Uploading sousa.au
Upload done.
```

When imported, parameters are passed explicitly to functions:

```
C:\...\PP2E\Internet\Ftp>python
>>> from getfile import getfile
>>> getfile(file='sousa.au', site='starship.python.net', dir='upload',
...         user=('lutz', '****'))
Downloading sousa.au
Download done.
>>> from playfile import playfile
>>> playfile('sousa.au')
```

I've left one piece out of the puzzle: all that's left is to write a module that attempts to play an audio file portably (see [Example 11-8](#)). Alas, this is the least straightforward task because audio players vary per platform. On Windows, the following module uses the DOS `start` command to launch whatever you have registered to play audio files (exactly as if you had double-clicked on the file's icon in a file explorer); on the Windows 98 side of my Sony notebook machine, this DOS command line:

```
C:\...\PP2E\Internet\Ftp>python playfile.py sousa.au
```

pops up a media bar playing the sample. On Unix, it attempts to pass the audio file to a command-line player program, if one has been added to the `unixfilter` table -- tweak this for your system (`cat` 'ing audio files to `/dev/audio` works on some Unix systems, too). On other platforms, you'll need to do a bit more; there has been some work towards portable audio interfaces in Python, but it's notoriously platform-specific. Web browsers generally know how to play audio files, so passing the filename in a URL to a browser located via the `LaunchBrowser.py` script we met in [Chapter 4](#), is perhaps a portable solution here as well (see that chapter for interface details).

Example 11-8. PP2E\Internet\Ftp\playfile.py

```
#!/usr/local/bin/python
#####
# Try to play an arbitrary audio file.
# This may not work on your system as is; it
# uses audio filters on Unix, and filename
# associations on Windows via the start command
# line (i.e., whatever you have on your machine
# to run *.au files--an audio player, or perhaps
# a web browser); configure me as needed. We
# could instead launch a web browser here, with
# LaunchBrowser.py. See also: Lib/audiodev.py.
#####

import os, sys
sample = 'sousa.au' # default audio file

unixhelpmsg = """
Sorry: can't find an audio filter for your system!
Add an entry for your system to the "unixfilter"
dictionary in playfile.py, or play the file manually.
"""

unixfilter = {'sunos5': '/usr/bin/audioplay',
              'linux2': '<unknown>',
              'sunos4': '/usr/demo/SOUND/play'}

def playfile(sample=sample):
    """
    play an audio file: use name associations
    on windows, filter command-lines elsewhere
    """
    if sys.platform[:3] == 'win':
        os.system('start ' + sample) # runs your audio player
    else:
        if not (unixfilter.has_key(sys.platform) and
                os.path.exists(unixfilter[sys.platform])):
            print unixhelpmsg
        else:
            theme = open(sample, 'r')
            audio = os.popen(unixfilter[sys.platform], 'w') # spawn shell tool
            audio.write(theme.read()) # send to its stdi

if __name__ == '__main__': playfile()
```

11.2.2.4 Adding user interfaces

If you read the last chapter, you'll recall that it concluded with a quick look at scripts that added user interface to a socket-based `getfile` script -- one that transferred files over a proprietary socket dialog, instead of FTP. At the end of that presentation, I mentioned that FTP is a much more generally useful way to move files around, because FTP servers are so widely available on the Net. For illustration purposes, [Example 11-9](#) shows a simple mutation of the last chapter's user interface, implemented as a new subclass of the last chapter's general form builder.

Example 11-9. P2E\Internet\Ftp\getfilegui.py

```
#####
# launch ftp getfile function with a reusable form gui class;
# uses os.chdir to goto target local dir (getfile currently
# assumes that filename has no local directory path prefix);
# runs getfile.getfile in thread to allow more than one to be
```



```
# running at once and avoid blocking gui during downloads;
# this differs from socket-based getfilegui, but reuses Form;
# supports both user and anonymous ftp as currently coded;
# caveats: the password field is not displayed as stars here,
# errors are printed to the console instead of shown in the
# gui (threads can't touch the gui on Windows), this isn't
# 100% thread safe (there is a slight delay between os.chdir
# here and opening the local output file in getfile) and we
# could display both a save-as popup for picking the local dir,
# and a remote directory listings for picking the file to get;
#####

from Tkinter import Tk, mainloop
from tkMessageBox import showinfo
import getfile, os, sys, thread # ftp getfile here, not socket
from PP2E.Internet.Sockets.form import Form # reuse form tool in socket dir

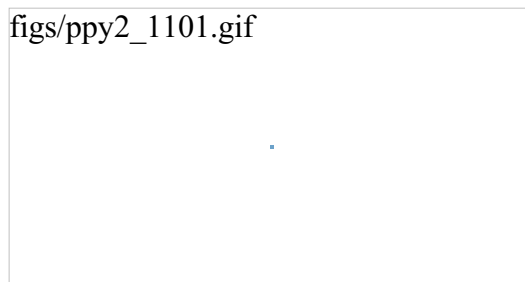
class FtpForm(Form):
    def __init__(self):
        root = Tk()
        root.title(self.title)
        labels = ['Server Name', 'Remote Dir', 'File Name',
                 'Local Dir', 'User Name?', 'Password?']
        Form.__init__(self, labels, root)
        self.mutex = thread.allocate_lock()
        self.threads = 0
    def transfer(self, filename, servername, remotedir, userinfo):
        try:
            self.do_transfer(filename, servername, remotedir, userinfo)
            print '%s of "%s" successful' % (self.mode, filename)
        except:
            print '%s of "%s" has failed:' % (self.mode, filename),
            print sys.exc_info()[0], sys.exc_info()[1]
            self.mutex.acquire()
            self.threads = self.threads - 1
            self.mutex.release()
    def onSubmit(self):
        Form.onSubmit(self)
        localdir = self.content['Local Dir'].get()
        remotedir = self.content['Remote Dir'].get()
        servername = self.content['Server Name'].get()
        filename = self.content['File Name'].get()
        username = self.content['User Name?'].get()
        password = self.content['Password?'].get()
        userinfo = ()
        if username and password:
            userinfo = (username, password)
        if localdir:
            os.chdir(localdir)
            self.mutex.acquire()
            self.threads = self.threads + 1
            self.mutex.release()
            ftpargs = (filename, servername, remotedir, userinfo)
            thread.start_new_thread(self.transfer, ftpargs)
            showinfo(self.title, '%s of "%s" started' % (self.mode, filename))
    def onCancel(self):
        if self.threads == 0:
            Tk().quit()
        else:
            showinfo(self.title,
                    'Cannot exit: %d threads running' % self.threads)

class FtpGetfileForm(FtpForm):
    title = 'FtpGetfileGui'
    mode = 'Download'
    def do_transfer(self, filename, servername, remotedir, userinfo):
        getfile.getfile(filename, servername, remotedir, userinfo, 0, 1)
```

```
if __name__ == '__main__':  
    FtpGetfileForm()  
    mainloop()
```

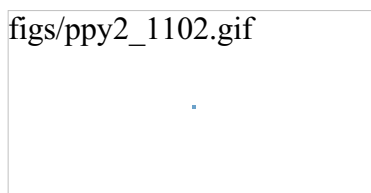
If you flip back to the end of the previous chapter, you'll find that this version is similar in structure to its counterpart there; in fact, it has the same name (and is distinct only because it lives in a different directory). The class here, though, knows how to use the FTP-based `getfile` module from earlier in this chapter, instead of the socket-based `getfile` module we met a chapter ago. When run, this version also implements more input fields, as we see in [Figure 11-1](#).

Figure 11-1. FTP getfile input form



Notice that a full file path is entered for the local directory here. Otherwise, the script assumes the current working directory, which changes after each download and can vary depending on where the GUI is launched (e.g., the current directory differs when this script is run by the `PyDemos` program at the top of the examples tree). When we click this GUI's Submit button (or press the Enter key), this script simply passes the form's input field values as arguments to the `getfile.getfile` FTP utility function shown earlier in this section. It also posts a pop-up to tell us the download has begun ([Figure 11-2](#)).

Figure 11-2. FTP getfile info pop-up



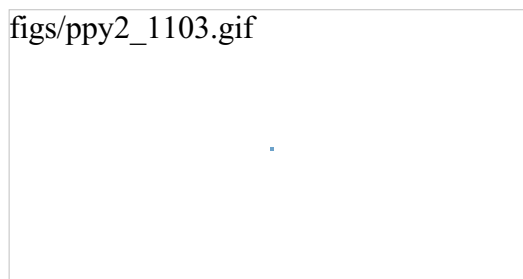
As currently coded, further download status messages from this point on show up in the console window; here are the messages for a successful download, as well as one that failed when I mistyped my password (no, it's not really "xxxxxx"):

```
User Name?      =>      lutz  
Server Name    =>      starship.python.net  
Local Dir      =>      c:\temp  
Password?     =>      xxxxxx  
File Name      =>      index.html  
Remote Dir     =>      public_html/home  
Download of "index.html" successful  
  
User Name?      =>      lutz  
Server Name    =>      starship.python.net  
Local Dir      =>      c:\temp  
Password?     =>      xxxxxx  
File Name      =>      index.html
```

```
Remote Dir      =>      public_html/home  
Download of "index.html" has failed: ftplib.error_perm 530 Login incorrect.
```

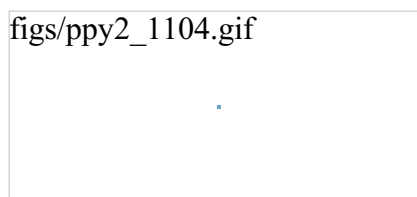
Given a username and password, the downloader logs into the specified account. To do anonymous FTP instead, leave the username and password fields blank. Let's start an anonymous FTP connection to fetch the Python source distribution; [Figure 11-3](#) shows the filled-out form.

Figure 11-3. FTP getfile input form, anonymous FTP

A rectangular input field containing the text "figs/ppy2_1103.gif". The text is in a monospaced font and is positioned at the top left of the field. The rest of the field is empty.

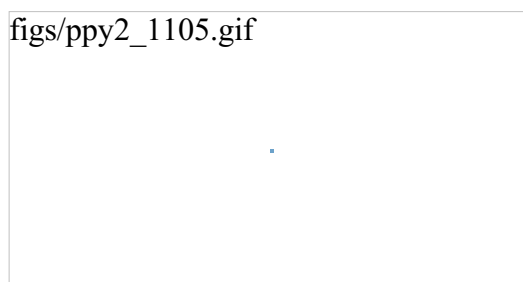
Pressing Submit on this form starts a download running in the background as before; we get the pop-up shown in [Figure 11-4](#) to verify the startup.

Figure 11-4. FTP getfile info pop-up

A rectangular input field containing the text "figs/ppy2_1104.gif". The text is in a monospaced font and is positioned at the top left of the field. The rest of the field is empty.

Now, to illustrate the threading capabilities of this GUI, let's start another download while this one is in progress. The GUI stays active while downloads are under way, so we simply change the input fields and press Submit again, as done in [Figure 11-5](#).

Figure 11-5. FTP getfile input form, second thread

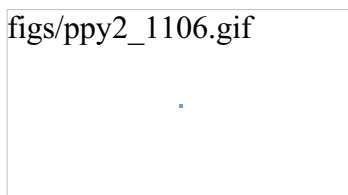
A rectangular input field containing the text "figs/ppy2_1105.gif". The text is in a monospaced font and is positioned at the top left of the field. The rest of the field is empty.

This second download starts in parallel with the one attached to ftp.python.org, because each download is run in a thread, and more than one Internet connection can be active at once. In fact the GUI itself stays active during downloads only because downloads are run in threads; if they were not, even screen redraws wouldn't happen until a download finished.

We discussed threads in [Chapter 3](#), but this script illustrates some practical thread concerns:

- This program takes care to not do anything GUI-related in a download thread. At least in the current release on Windows, only the thread that makes GUIs can process them (a Windows-only rule that has nothing to do with Python or Tkinter).
- To avoid killing spawned download threads on some platforms, the GUI must also be careful to not exit while any downloads are in progress. It keeps track of the number of in-progress threads, and just displays the pop-up in [Figure 11-6](#) if we try to kill the GUI while both of these downloads are in progress by pressing the Cancel button.

Figure 11-6. FTP getfile busy pop-up



We'll see ways to work around the no-GUI rule for threads when we explore the `PyMailGui` example near the end of this chapter. To be portable, though, we can't really close the GUI until the active-thread count falls to zero. Here is the sort of output that appears in the console window for these two downloads:

```
C:\...\PP2E\Internet\Ftp>python getfilegui.py
User Name?      =>
Server Name     =>      ftp.python.org
Local Dir       =>      c:\temp
Password?      =>
File Name       =>      python1.5.tar.gz
Remote Dir      =>      pub/python/src

User Name?      =>      lutz
Server Name     =>      starship.python.net
Local Dir       =>      c:\temp
Password?      =>      xxxxxx
File Name       =>      about-pp.html
Remote Dir      =>      public_html/home
Download of "about-pp.html" successful
Download of "python1.5.tar.gz" successful
```

This all isn't much more useful than a command-line-based tool, of course, but it can be easily modified by changing its Python code, and it provides enough of a GUI to qualify as a simple, first-cut FTP user interface. Moreover, because this GUI runs downloads in Python threads, more than one can be run at the same time from this GUI without having to start or restart a different FTP client tool.

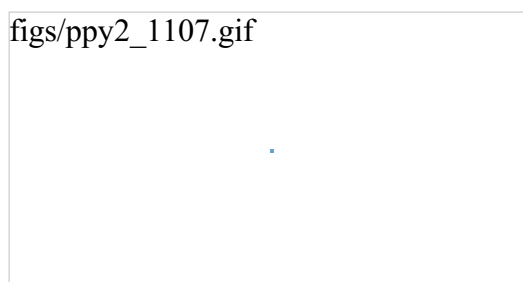
While we're in a GUI mood, let's add a simple interface to the `putfile` utility, too. The script in [Example 11-10](#) creates a dialog that starts uploads in threads. It's almost the same as the `getfile` GUI we just wrote, so there's nothing new to say. In fact, because get and put operations are so similar from an interface perspective, most of the get form's logic was deliberately factored out into a single generic class (`FtpForm`) such that changes need only be made in a single place. That is, the put GUI here is mostly just a reuse of the get GUI, with distinct output labels and transfer method. It's in a file by itself to make it easy to launch as a standalone program.

Example 11-10. PP2E\Internet\Ftp\putfilegui.py

```
#####  
# launch ftp putfile function with a reusable form gui class;  
# see getfilegui for notes: most of the same caveats apply;  
# the get and put forms have been factored into a single  
# class such that changes need only be made in one place;  
#####  
  
from Tkinter import mainloop  
import putfile, getfilegui  
  
class FtpPutfileForm(getfilegui.FtpForm):  
    title = 'FtpPutfileGui'  
    mode = 'Upload'  
    def do_transfer(self, filename, servername, remotedir, userinfo):  
        putfile.putfile(filename, servername, remotedir, userinfo, 0)  
  
if __name__ == '__main__':  
    FtpPutfileForm()  
    mainloop()
```

Running this script looks much like running the download GUI, because it's almost entirely the same code at work. Let's upload a couple of files from the client machine to the *starship* server; [Figure 11-7](#) shows the state of the GUI while starting one.

Figure 11-7. FTP putfile input form



And here is the console window output we get when uploading two files in parallel; here again, uploads run in threads, so if we start a new upload before one in progress is finished, they overlap in time:

```
User Name?      =>      lutz  
Server Name    =>      starship.python.net  
Local Dir      =>      c:\stuff\website\public_html  
Password?     =>      xxxxxx  
File Name      =>      about-pp2e.html  
Remote Dir     =>      public_html  
  
User Name?     =>      lutz  
Server Name    =>      starship.python.net  
Local Dir      =>      c:\stuff\website\public_html  
Password?     =>      xxxxxx  
File Name      =>      about-ppr2e.html  
Remote Dir     =>      public_html  
Upload of "about-pp2e.html" successful  
Upload of "about-ppr2e.html" successful
```

Finally, we can bundle up both GUIs in a single launcher script that knows how to start the get and put interfaces, regardless of which directory we are in when the script is started, and independent of the platform on which it runs. [Example 11-11](#) shows this process.

Example 11-11. PP2E\Internet\Ftp\PyFtpGui.pyw

```
#####  
# spawn ftp get and put guis no matter what dir I'm run from;  
# os.getcwd is not necessarily the place this script lives;  
# could also hard-code a path from $PP2EHOME, or guessLocation;  
# could also do this but need the DOS popup for status messages:  
# from PP2E.launchmodes import PortableLauncher  
# PortableLauncher('getfilegui', '%s/getfilegui.py' % mydir)()  
#####  
  
import os, sys  
from PP2E.Launcher import findFirst  
mydir = os.path.split(findFirst(os.curdir, 'PyFtpGui.pyw'))[0]  
  
if sys.platform[:3] == 'win':  
    os.system('start %s/getfilegui.py' % mydir)  
    os.system('start %s/putfilegui.py' % mydir)  
else:  
    os.system('python %s/getfilegui.py &' % mydir)  
    os.system('python %s/putfilegui.py &' % mydir)
```

When this script is started, both the `get` and `put` GUIs appear as distinct, independently running programs; alternatively, we might attach both forms to a single interface. We could get much fancier than these two interfaces, of course. For instance, we could pop up local file selection dialogs, and we could display widgets that give status of downloads and uploads in progress. We could even list files available at the remote site in a selectable list box by requesting remote directory listings over the FTP connection. To learn how to add features like that, though, we need to move on to the next section.

11.2.3 Downloading Web Sites (Mirrors)

Once upon a time, Telnet was all I needed. My web site lived at an Internet Service Provider (ISP) that provided general and free Telnet access for all its customers. It was a simple time. All of my site's files lived only in one place -- at my account directory on my ISP's server machine. To make changes to web pages, I simply started a Telnet session connected to my ISP's machine and edited my web pages there online. Moreover, because Telnet sessions can be run from almost any machine with an Internet link, I was able to tweak my web pages everywhere -- from my PC, from machines I had access to on the training road, from archaic machines I played with when I was bored at my day job, and so on. Life was good.

But times have changed. Due to a security breach, my ISP made a blanket decision to revoke Telnet access from all of their customers (except, of course, those who elected to pay a substantial premium to retain it). Seemingly, we weren't even supposed to have known about Telnet in the first place. As a replacement, the ISP mandated that all Telnet-inclined users should begin maintaining web page files locally on their own machines, and upload them by FTP after every change.

That's nowhere near as nice as editing files kept in a single place from almost any computer on the planet, of course, and this triggered plenty of complaints and cancellations among the technically savvy. Unfortunately, the technically savvy is a financially insignificant subset; more to the point, my web page's address had by this time been published in multiple books sold around the world, so changing ISPs would have been no less painful than changing update procedures.

After the shouting, it dawned on me that Python could help here: by writing Python scripts to automate the upload and download tasks associated with maintaining my web site on my PC, I could at least get back some of the mobility and ease of use that I'd lost. Because Python FTP scripts will work on any machine with sockets, I could run them both on my PC and on nearly any other computer where Python was installed. Furthermore, the same scripts used to transfer page files to and from my PC could be used to copy ("mirror") my site to another web server as a backup copy, should my ISP experience an outage (trust me -- it happens).

The following two scripts were born of all of the above frustrations. The first, `mirrorflat.py`, automatically downloads (i.e., copies) by FTP all the files in a directory at a remote site, to a directory on the local machine. I keep the main copy of my web site files on my PC these days, but really use this script in two ways:

- To download my web site to client machines where I want to make edits, I fetch the contents of my `public_html` web directory of my account on my ISP's machine.
- To mirror my site to my account on the `starship.python.net` server, I run this script periodically from a Telnet session on the `starship` machine (as I wrote this, `starship` still clung to the radical notion that users are intelligent enough to run Telnet).

More generally, this script (shown in [Example 11-12](#)) will download a directory full of files to any machine with Python and sockets, from any machine running an FTP server.

Example 11-12. `PP2E\Internet\Ftp\mirrorflat.py`

```
#!/bin/env python
#####
# use ftp to copy (download) all files from a remote site
# and directory to a directory on the local machine; e.g.,
# run me periodically to mirror a flat ftp site;
#####

import os, sys, ftplib
from getpass import getpass

remotesite = 'home.rmi.net'
remotedir  = 'public_html'
remoteuser = 'lutz'
remotepass = getpass('Please enter password for %s: ' % remotesite)
localdir   = (len(sys.argv) > 1 and sys.argv[1]) or '.'
if sys.platform[:3] == 'win': raw_input() # clear stream
cleanall   = raw_input('Clean local directory first? ')[:1] in ['y', 'Y']

print 'connecting...'
connection = ftplib.FTP(remotesite)           # connect to ftp site
connection.login(remoteuser, remotepass)     # login as user/password
connection.cwd(remotedir)                    # cd to directory to copy

if cleanall:
    for localname in os.listdir(localdir):    # try to delete all locals
        try:                                 # first to remove old files
            print 'deleting local', localname
            os.remove(os.path.join(localdir, localname))
        except:
            print 'cannot delete local', localname

count = 0                                     # download all remote files
remotefiles = connection.nlst()              # nlst() gives files list
```

```

# dir() gives full details
for remotename in remotefiles:
    localname = os.path.join(localdir, remotename)
    print 'copying', remotename, 'to', localname
    if remotename[-4:] == 'html' or remotename[-3:] == 'txt':
        # use ascii mode xfer
        localfile = open(localname, 'w')
        callback = lambda line, file=localfile: file.write(line + '\n')
        connection.retrlines('RETR ' + remotename, callback)
    else:
        # use binary mode xfer
        localfile = open(localname, 'wb')
        connection.retrbinary('RETR ' + remotename, localfile.write)
    localfile.close()
    count = count+1

connection.quit()
print 'Done:', count, 'files downloaded.'
```

There is not a whole lot new to speak of in this script, compared to other FTP examples we've seen thus far. We open a connection with the remote FTP server, log in with a username and password for the desired account (this script never uses anonymous FTP), and go to the desired remote directory. New here, though, are loops to iterate over all the files in local and remote directories, text-based retrievals, and file deletions:

Deleting all local files

This script has a `cleanall` option, enabled by interactive prompt. If selected, the script first deletes all the files in the local directory before downloading, to make sure there are no extra files there that aren't also on the server (there may be junk here from a prior download). To delete local files, the script calls `os.listdir` to get a list of filenames in the directory, and `os.remove` to delete each; see [Chapter 2](#) earlier in this book (or the Python library manual) for more details if you've forgotten what these calls do.

Notice the use of `os.path.join` to concatenate a directory path and filename according to the host platform's conventions; `os.listdir` returns filenames without their directory paths, and this script is not necessarily run in the local directory where downloads will be placed. The local directory defaults to the current directory ("."), but can be set differently with a command-line argument to the script.

Fetching all remote files

To grab all the files in a remote directory, we first need a list of their names. The FTP object's `nlst` method is the remote equivalent of `os.listdir`: `nlst` returns a list of the string names of all files in the current remote directory. Once we have this list, we simply step through it in a loop, running FTP retrieval commands for each filename in turn (more on this in a minute).

The `nlst` method is, more or less, like requesting a directory listing with an `ls` command in typical interactive FTP programs, but Python automatically splits up the listing's text into a list of filenames. We can pass it a remote directory to be listed; by default it lists the current server directory. A related FTP method, `dir`, returns the list of line strings produced by an FTP `LIST` command; its result is like typing a `dir` command in an FTP session, and its lines contain complete file information, unlike `nlst`. If you need to know more about all the remote files, parse the result of a `dir` method call.

Text-based retrievals

To keep line-feeds in sync with the machines that my web files live on, this script distinguishes between binary and text files. It uses a simple heuristic to do so: filenames ending in *.html* or *.txt* are assumed to be ASCII text data (HTML web pages and simple text files), and all others are assumed to be binary files (e.g., GIF and JPEG images, audio files, tar archives). This simple rule won't work for every web site, but it does the trick at mine.

Binary files are pulled down with the `retrbinary` method we met earlier and a local open mode of "wb" to suppress line-feed byte mapping (this script may be run on Windows or Unix-like platforms). We don't use a chunk size third argument here, though -- it defaults to a reasonable 8K if omitted.

For ASCII text files, the script instead uses the `retrlines` method, passing in a function to be called for each line in the text file downloaded. The text line handler function mostly just writes the line to a local file. But notice that the handler function created by the `lambda` here also adds an `\n` newline character to the end of the line it is passed. Python's `retrlines` method strips all line-feed characters from lines to side-step platform differences. By adding an `\n`, the script is sure to add the proper line-feed marker character sequence for the local platform on which this script runs (`\n` or `\r\n`). For this automapping of the `\n` in the script to work, of course, we must also open text output files in "w" text mode, not "wb" -- the mapping from `\n` to `\r\n` on Windows happens when data is written to the file.

All of this is simpler in action than in words. Here is the command I use to download my entire web site from my ISP server account to my Windows 98 laptop PC, in a single step:

```
C:\Stuff\Website\public_html>python %X%\internet\ftp\mirrorflat.py
Please enter password for home.rmi.net:
Clean local directory first?
connecting...
copying UPDATES to .\UPDATES
copying PythonPowered.gif to .\PythonPowered.gif
copying Pywin.gif to .\Pywin.gif
copying PythonPoweredAnim.gif to .\PythonPoweredAnim.gif
copying PythonPoweredSmall.gif to .\PythonPoweredSmall.gif
copying about-hop1.html to .\about-hop1.html
copying about-lp.html to .\about-lp.html
...
...lines deleted...
...
copying training.html to .\training.html
copying trainingCD.GIF to .\trainingCD.GIF
copying uk-1.jpg to .\uk-1.jpg
copying uk-2.jpg to .\uk-2.jpg
copying uk-3.jpg to .\uk-3.jpg
copying whatsnew.html to .\whatsnew.html
copying whatsold.html to .\whatsold.html
copying xlate-lp.html to .\xlate-lp.html
copying uploadflat.py to .\uploadflat.py
copying ora-lp-france.gif to .\ora-lp-france.gif
Done: 130 files downloaded.
```

This can take awhile to complete (it's bound by network speed constraints), but it is much more accurate and easy than downloading files by hand. The script simply iterates over all the remote files returned by the `nlst` method, and downloads each with the FTP protocol (i.e., over sockets).

in turn. It uses text transfer mode for names that imply obviously text data, and binary mode for others.

With the script running this way, I make sure the initial assignments in it reflect the machines involved, and then run the script from the local directory where I want the site copy to be stored. Because the download directory is usually not where the script lives, I need to give Python the full path to the script file (`%X%` evaluates a shell variable containing the top-level path to book examples on my machine). When run on the starship server in a Telnet session window, the execution and script directory paths are different, but the script works the same way.

If you elect to delete local files in the download directory, you may also see a batch of "deleting local..." messages scroll by on the screen before any "copying..." lines appear:

```
...
deleting local uploadflat.py
deleting local whatsnew.html
deleting local whatsold.html
deleting local xlate-lp.html
deleting local old-book.html
deleting local about-pp2e.html
deleting local about-ppr2e.html
deleting local old-book2.html
deleting local mirrorflat.py
...
copying about-pp-japan.html to ./about-pp-japan.html
copying about-pp.html to ./about-pp.html
copying about-ppr-germany.html to ./about-ppr-germany.html
copying about-ppr-japan.html to ./about-ppr-japan.html
copying about-ppr-toc.html to ./about-ppr-toc.html
...
```

By the way, if you botch the input of the remote site password, a Python exception is raised; I sometimes need to run again (and type slower):

```
C:\Stuff\Website\public_html>python %X%\internet\ftp\mirrorflat.py
Please enter password for home.rmi.net:
Clean local directory first?
connecting...
Traceback (innermost last):
  File "C:\PP2ndEd\examples\PP2E\internet\ftp\mirrorflat.py", line 20, in ?
    connection.login(remoteuser, remotepass) # login as user/pass..
  File "C:\Program Files\Python\Lib\ftplib.py", line 316, in login
    if resp[0] == '3': resp = self.sendcmd('PASS ' + passwd)
  File "C:\Program Files\Python\Lib\ftplib.py", line 228, in sendcmd
    return self.getresp()
  File "C:\Program Files\Python\Lib\ftplib.py", line 201, in getresp
    raise error_perm, resp
ftplib.error_perm: 530 Login incorrect.
```

It's worth noting that this script is at least partially configured by assignments near the top of the file. In addition, the password and deletion options are given by interactive inputs, and one command-line argument is allowed -- the local directory name to store the downloaded files (it defaults to ".", the directory where the script is run). Command-line arguments could be employed to universally configure all the other download parameters and options, too; but because of Python's simplicity and lack of compile/link steps, changing settings in the text of Python scripts is usually just as easy as typing words on a command line.

	Windows input note : If you study the previous code closely, you'll notice that an extra <code>raw_input</code> call is made on Windows only, after the <code>getpass</code>
--	--

password input call and before the `cleanall` option setting is input. This is a workaround for what seems like a bug in Python 1.5.2 for Windows.

Oddly, the Windows port sometimes doesn't synchronize command-line input and output streams as expected. Here, this seems to be due to a `getpass` bug or constraint -- because `getpass` uses the low-level `msvcrt` keyboard interface module we met in [Chapter 2](#), it appears to not mix well with the `stdin` stream buffering used by `raw_input`, and botches the input stream in the process. The extra `raw_input` clears the input stream (`sys.stdin.flush` doesn't help).

In fact, without the superfluous `raw_input` for Windows, this script prompts for `cleanall` option input, but never stops to let you type a reply! This effectively disables `cleanall` altogether. To force distinct input and output lines and correct `raw_input` behavior, some scripts in this book run extra `print` statements or `raw_input` calls to sync up streams before further user interaction. There may be other fixes, and this may be improved in future releases; try this script without the extra `raw_input` to see if this has been repaired in your Python.

11.2.4 Uploading Web Sites

Uploading a full directory is symmetric to downloading: it's mostly a matter of swapping the local and remote machines and operations in the program we just met. The script in [Example 11-13](#) uses FTP to copy all files in a directory on the local machine on which it runs, up to a directory on a remote machine.

I really use this script, too, most often to upload all of the files maintained on my laptop PC to my ISP account in one fell swoop. I also sometimes use it to copy my site from my PC to its starship mirror machine, or from the mirror machine back to my ISP. Because this script runs on any computer with Python and sockets, it happily transfers a directory from any machine on the Net to any machine running an FTP server. Simply change the initial setting in this module as appropriate for the transfer you have in mind.

Example 11-13. PP2E\Internet\Ftp\uploadflat.py

```
#!/bin/env python
#####
# use ftp to upload all files from a local dir to a remote site/directory;
# e.g., run me to copy a web/ftp site's files from your PC to your ISP;
# assumes a flat directory upload: uploadall.py does nested directories.
# to go to my ISP, I change setting to 'home.rmi.net', and 'public_html'.
#####

import os, sys, ftplib, getpass

remotesite = 'starship.python.net'           # upload to starship site
remotedir  = 'public_html/home'            # from win laptop or other
remoteuser = 'lutz'

remotepass = getpass.getpass('Please enter password for %s: ' % remotesite)
localdir   = (len(sys.argv) > 1 and sys.argv[1]) or '.'
if sys.platform[:3] == 'win': raw_input()   # clear stream
cleanall   = raw_input('Clean remote directory first? ')[1] in ['y', 'Y']
```

```
print 'connecting...'  
connection = ftplib.FTP(remotesite)           # connect to ftp site  
connection.login(remotepass, remotepass)     # login as user/password  
connection.cwd(remotedir)                   # cd to directory to copy  
  
if cleanall:  
    for remotename in connection.nlst():     # try to delete all remotes  
        try:                               # first to remove old files  
            print 'deleting remote', remotename  
            connection.delete(remotename)  
        except:  
            print 'cannot delete remote', remotename  
  
count = 0  
localfiles = os.listdir(localdir)          # upload all local files  
                                              # listdir() strips dir path  
  
for localname in localfiles:  
    localpath = os.path.join(localdir, localname)  
    print 'uploading', localpath, 'to', localname  
    if localname[-4:] == 'html' or localname[-3:] == 'txt':  
        # use ascii mode xfer  
        localfile = open(localpath, 'r')  
        connection.storlines('STOR ' + localname, localfile)  
    else:  
        # use binary mode xfer  
        localfile = open(localpath, 'rb')  
        connection.storbinary('STOR ' + localname, localfile, 1024)  
    localfile.close()  
    count = count+1  
  
connection.quit()  
print 'Done:', count, 'files uploaded.'
```

Like the mirror download script, the program here illustrates a handful of new FTP interfaces and a set of FTP scripting techniques:

Deleting all remote files

Just like the mirror script, the upload begins by asking if we want to delete all the files in the remote target directory before copying any files there. This `cleanall` option is useful: we've deleted files in the local copy of the directory in the client -- the deleted files would remain on the server-side copy unless we delete all files there first. To implement the remote cleanup, this script simply gets a listing of all the files in the remote directory with the FTP `nlst` method, and deletes each in turn with the FTP `delete` method. Assuming we have delete permission, the directory will be emptied (file permissions depend on the account we logged into when connecting to the server). We've already moved to the target remote directory when deletions occur, so no directory paths must be prepended to filenames here.

Storing all local files

To apply the upload operation to each file in the local directory, we get a list of local filenames with the standard `os.listdir` call, and take care to prepend the local source directory path to each filename with the `os.path.join` call. Recall that `os.listdir` returns filenames without directory paths, and the source directory may not be the same as the script's execution directory if passed on the command line.

Text-based uploads

This script may be run on both Windows and Unix-like clients, so we need to handle text files specially. Like the mirror download, this script picks text or binary transfer modes by inspecting each filename's extension -- HTML and text files are moved in FTP text mode. We've already met the `storbinary` FTP object method used to upload files in binary mode -- an exact, byte-for-byte copy appears at the remote site.

Text mode transfers work almost identically: the `storlines` method accepts an FTP command string and a local file (or file-like) object opened in text mode, and simply copies each line in the local file to a same-named file on the remote machine. As usual, if we run this script on Windows, opening the input file in "r" text mode means that DOS-style `\r\n` end-of-line sequences are mapped to the `\n` character as lines are read. When the script is run on Unix and Linux, lines end in a single `\n` already, so no such mapping occurs. The net effect is that data is read portably, with `\n` characters to represent end-of-line. For binary files, we open in "rb" mode to suppress such automatic mapping everywhere (we don't want bytes that happen to have the same value as `\r` to magically disappear when read on Windows).^[3]

[3] Technically, Python's `storlines` method automatically sends all lines to the server with `\r\n` line-feed sequences, no matter what it receives from the local file's `readline` method (`\n` or `\r\n`). Because of that, the most important distinctions for uploads are to use the "rb" for binary mode and the `storlines` method for text. Consult module `ftplib.py` in the Python source library directory for more details.

As for the mirror download script, this program simply iterates over all files to be transferred (files in the local directory listing this time), and transfers each in turn -- in either text or binary mode, depending on the files' names. Here is the command I use to upload my entire web site from my laptop Windows 98 PC to the remote Unix server at my ISP, in a single step:

```
C:\Stuff\Website\public_html>python %X%\Internet\Ftp\uploadflat.py
Please enter password for starship.python.net:
Clean remote directory first?
connecting...
uploading .\LJsuppcover.jpg to LJsuppcover.jpg
uploading .\PythonPowered.gif to PythonPowered.gif
uploading .\PythonPoweredAnim.gif to PythonPoweredAnim.gif
uploading .\PythonPoweredSmall.gif to PythonPoweredSmall.gif
uploading .\Pywin.gif to Pywin.gif
uploading .\UPDATES to UPDATES
uploading .\about-hop1.html to about-hop1.html
uploading .\about-lp.html to about-lp.html
uploading .\about-pp-japan.html to about-pp-japan.html
...
...lines deleted...
...
uploading .\trainingCD.GIF to trainingCD.GIF
uploading .\uk-1.jpg to uk-1.jpg
uploading .\uk-2.jpg to uk-2.jpg
uploading .\uk-3.jpg to uk-3.jpg
uploading .\uploadflat.py to uploadflat.py
uploading .\whatsnew.html to whatsnew.html
uploading .\whatsold.html to whatsold.html
uploading .\xlate-lp.html to xlate-lp.html
Done: 131 files uploaded.
```

Like the mirror example, I usually run this command from the local directory where my web file are kept, and I pass Python the full path to the script. When I run this on the starship Linux

server, it works the same, but the paths to the script and my web files directory differ. If you elect to clean the remote directory before uploading, you'll get a bunch of "deleting remote..." messages before the "uploading..." lines here, too:

```
...
deleting remote uk-3.jpg
deleting remote whatsnew.html
deleting remote whatsold.html
deleting remote xlate-lp.html
deleting remote uploadflat.py
deleting remote ora-lp-france.gif
deleting remote LJsuppcover.jpg
deleting remote sonyz505js.gif
deleting remote pic14.html
...
```

11.2.5 Uploads with Subdirectories

Perhaps the biggest limitation of the web site download and upload scripts we just met are that they assume the site directory is flat (hence their names) -- i.e., both transfer simple files only, and neither handles nested subdirectories within the web directory to be transferred.

For my purposes, that's a reasonable constraint. I avoid nested subdirectories to keep things simple, and I store my home web site as a simple directory of files. For other sites (including one I keep at the starship machine), site transfer scripts are easier to use if they also automatically transfer subdirectories along the way.

It turns out that supporting directories is fairly simple -- we need to add only a bit of recursion and remote directory creation calls. The upload script in [Example 11-14](#) extends the one we just saw, to handle uploading all subdirectories nested within the transferred directory. Furthermore, it recursively transfers subdirectories within subdirectories -- the entire directory tree contained within the top-level transfer directory is uploaded to the target directory at the remote server.

Example 11-14. PP2E\Internet\Ftp\uploadall.py

```
#!/bin/env python
#####
# use ftp to upload all files from a local dir to a remote site/directory;
# this version supports uploading nested subdirectories too, but not the
# cleanall option (that requires parsing ftp listings to detect remote
# dirs, etc.); to upload subdirectories, uses os.path.isdir(path) to see
# if a local file is really a directory, FTP().mkd(path) to make the dir
# on the remote machine (wrapped in a try in case it already exists there),
# and recursion to upload all files/dirs inside the nested subdirectory.
# see also: uploadall-2.py, which doesn't assume the topremotedir exists.
#####

import os, sys, ftplib
from getpass import getpass

remotesite = 'home.rmi.net'          # upload from pc or starship to rmi.net
topremotedir = 'public_html'
remoteuser = 'lutz'
remotepass = getpass('Please enter password for %s: ' % remotesite)
toplocaldir = (len(sys.argv) > 1 and sys.argv[1]) or '.'

print 'connecting...'
connection = ftplib.FTP(remotesite)          # connect to ftp site
connection.login(remoteuser, remotepass)     # login as user/password
```

```
connection.cwd(topremotedir)                                # cd to directory to copy to
                                                           # assumes topremotedir exists

def uploadDir(localdir):
    global fcount, dcount
    localfiles = os.listdir(localdir)
    for localname in localfiles:
        localpath = os.path.join(localdir, localname)
        print 'uploading', localpath, 'to', localname
        if os.path.isdir(localpath):
            # recur into subdirs
            try:
                connection.mkd(localname)
                print localname, 'directory created'
            except:
                print localname, 'directory not created'
            connection.cwd(localname)
            uploadDir(localpath)
            connection.cwd('.')
            dcount = dcount+1
        else:
            if localname[-4:] == 'html' or localname[-3:] == 'txt':
                # use ascii mode xfer
                localfile = open(localpath, 'r')
                connection.storlines('STOR ' + localname, localfile)
            else:
                # use binary mode xfer
                localfile = open(localpath, 'rb')
                connection.storbinary('STOR ' + localname, localfile, 1024)
            localfile.close()
            fcount = fcount+1

fcount = dcount = 0
uploadDir(toplocaldir)
connection.quit()
print 'Done:', fcount, 'files and', dcount, 'directories uploaded.'
```

Like the flat upload script, this one can be run on any machine with Python and sockets and upload to any machine running an FTP server; I run it both on my laptop PC and on starship by Telnet to upload sites to my ISP.

In the interest of space, I'll leave studying this variant in more depth as a suggested exercise. Two quick pointers, though:

- The crux of the matter here is the `os.path.isdir` test near the top; if this test detects a directory in the current local directory, we create a same-named directory on the remote machine with `connection.mkd` and descend into it with `connection.cwd`, and recur into the subdirectory on the local machine. Like all FTP object methods, `mkd` and `cwd` methods issue FTP commands to the remote server. When we exit a local subdirectory, we run a remote `cwd('.')` to climb to the remote parent directory and continue. The rest of the script is roughly the same as the original.
- Note that this script handles only directory tree uploads; recursive uploads are generally more useful than recursive downloads, if you maintain your web sites on your local PC and upload to a server periodically, as I do. If you also want to download (mirror) a web site that has subdirectories, see the mirror scripts in the Python source distribution's Tools directory (currently, at file location `Tools/scripts/ftpmirror.py`). It's not much extra work, but requires parsing the output of a remote listing command to detect remote directories, and that is just complicated enough for me to omit here. For the same reason, the recursive upload script shown here doesn't support the remote directory cleanup option of the original -- such a feature would require parsing remote listings as well.

For more context, also see the *uploadall-2.py* version of this script in the examples distribution; it's similar, but coded so as not to assume that the top-level remote directory already exists.

I l@ve RuBoard

11.3 Processing Internet Email

Some of the other most common higher-level Internet protocols have to do with reading and sending email messages: POP and IMAP for fetching email from servers,^[4] SMTP for sending new messages, and other formalisms such as `rfc822` for specifying email message contents and format. You don't normally need to know about such acronyms when using common email tools but internally, programs like Microsoft Outlook talk to POP and SMTP servers to do your bidding.

[4] IMAP, or Internet Message Access Protocol, was designed as an alternative to POP, but is not as widely used today, and so is not presented in this text. See the Python library manual for IMAP support details.

Like FTP, email ultimately consists of formatted commands and byte streams shipped over sockets and ports (port 110 for POP; 25 for SMTP). But also like FTP, Python has standard modules to simplify all aspects of email processing. In this section, we explore the POP and SMTP interfaces for fetching and sending email at servers, and the `rfc822` interfaces for parsing information out of email header lines; other email interfaces in Python are analogous and are documented in the Python library reference manual.

11.3.1 POP: Reading Email

I used to be an old-fashioned guy. I admit it: up until recently, I preferred to check my email by telnetting to my ISP and using a simple command-line email interface. Of course, that's not ideal for mail with attachments, pictures, and the like, but its portability is staggering -- because Telnet runs on almost any machine with a network link, I was able to check my mail quickly and easily from anywhere on the planet. Given that I make my living traveling around the world teaching Python classes, this wild accessibility was a big win.

If you've already read the web site mirror scripts sections earlier in this chapter, you've already heard my tale of ISP woe, so I won't repeat it here. Suffice it to say that times have changed on this front too: when my ISP took away Telnet access, they also took away my email access.^[5] Luckily, Python came to the rescue here, too -- by writing email access scripts in Python, I can still read and send email from any machine in the world that has Python and an Internet connection. Python can be as portable a solution as Telnet.

[5] In the process of losing Telnet, my email account and web site were taken down for weeks on end, and I lost forever a backlog of thousands of messages saved over the course of a year. Such outages can be especially bad if your income is largely driven by email and web contacts, but that's a story for another night, boys and girls.

Moreover, I can still use these scripts as an alternative to tools suggested by the ISP, such as Microsoft Outlook. Besides not being a big fan of delegating control to commercial products of large companies, tools like Outlook generally download mail to your PC and delete it from the mail server as soon as you access it. This keeps your email box small (and your ISP happy), but isn't exactly friendly to traveling Python salespeople -- once accessed, you cannot re-access a prior email from any machine except the one where it was initially downloaded to. If you need to see an old email and don't have your PC handy, you're out of luck.

The next two scripts represent one solution to these portability and single-machine constraints (we'll see others in this and later chapters). The first, `popmail.py`, is a simple mail reader tool, which downloads and prints the contents of each email in an email account. This script is admittedly primitive, but it lets you read your email on any machine with Python and sockets; moreover, it leaves your email intact on the server. The second, `smtpmail.py`, is a one-shot script for writing and sending a new email message.

11.3.1.1 Mail configuration module

Before we get to either of the two scripts, though, let's first take a look at a common module they both import and use. The module in [Example 11-15](#) is used to configure email parameters appropriately for a particular user. It's simply a collection of assignments used by all the mail programs that appear in this book; isolating these configuration settings in this single module makes it easy to configure the book's email programs for a particular user.

If you want to use any of this book's email programs to do mail processing of your own, be sure to change its assignments to reflect your servers, account usernames, and so on (as shown, they refer to my email accounts). Not all of this module's settings are used by the next two scripts; we'll come back to this module at later examples to explain some of the settings here.

Example 11-15. PP2E\Internet\Email\mailconfig.py

```
#####
# email scripts get their server names and other email config
# options from this module: change me to reflect your machine
# names, sig, etc.; could get some from the command line too;
#####

#-----
# SMTP email server machine name (send)
#-----

smtpservername = 'smtp.rmi.net'          # or starship.python.net, 'localhost'

#-----
# POP3 email server machine, user (retrieve)
#-----

popservername  = 'pop.rmi.net'          # or starship.python.net, 'localhost'
popusername    = 'lutz'                 # password fetched or asked when run

#-----
# local file where pymail saves pop mail
# PyMailGui instead asks with a popup dialog
#-----

savemailfile   = r'c:\stuff\etc\savemail.txt'      # use dialog in PyMailGui

#-----
# PyMailGui: optional name of local one-line text file with your
# pop password; if empty or file cannot be read, pswd requested
# when run; pswd is not encrypted so leave this empty on shared
# machines; PyMailCgi and pymail always ask for pswd when run.
#-----

poppasswdfile  = r'c:\stuff\etc\pymailgui.txt'     # set to '' to be asked
```

```
#-----  
# personal information used by PyMailGui to fill in forms;  
# sig -- can be a triple-quoted block, ignored if empty string;  
# addr -- used for initial value of "From" field if not empty,  
# else tries to guess From for replies, with varying success;  
#-----  
  
myaddress = 'lutz@rmi.net'  
mysignature = '--Mark Lutz (http://rmi.net/~lutz) [PyMailGui 1.0]'
```

11.3.1.2 POP mail reader module

On to reading email in Python: the script in [Example 11-16](#) employs Python's standard `poplib` module, an implementation of the client-side interface to POP -- the Post Office Protocol. POP is just a well-defined way to fetch email from servers over sockets. This script connects to a POP server to implement a simple yet portable email download and display tool.

Example 11-16. PP2E\Internet\Email\popmail.py

```
#!/usr/local/bin/python  
#####  
# use the Python POP3 mail interface module to view  
# your pop email account messages; this is just a  
# simple listing--see pymail.py for a client with  
# more user interaction features, and smtpmail.py  
# for a script which sends mail; pop is used to  
# retrieve mail, and runs on a socket using port  
# number 110 on the server machine, but Python's  
# poplib hides all protocol details; to send mail,  
# use the smtplib module (or os.popen('mail...')).  
# see also: unix mailfile reader in App framework.  
#####  
  
import poplib, getpass, sys, mailconfig  
  
mailserver = mailconfig.popservername # ex: 'pop.rmi.net'  
mailuser = mailconfig.popusername # ex: 'lutz'  
mailpasswd = getpass.getpass('Password for %s?' % mailserver)  
  
print 'Connecting...'  
server = poplib.POP3(mailserver)  
server.user(mailuser) # connect, login to mail server  
server.pass_(mailpasswd) # pass is a reserved word  
  
try:  
    print server.getwelcome() # print returned greeting message  
    msgCount, msgBytes = server.stat()  
    print 'There are', msgCount, 'mail messages in', msgBytes, 'bytes'  
    print server.list()  
    print '-'*80  
    if sys.platform[:3] == 'win': raw_input() # windows getpass is odd  
    raw_input('[Press Enter key]')  
  
    for i in range(msgCount):  
        hdr, message, octets = server.retr(i+1) # octets is byte count  
        for line in message: print line # retrieve, print all mail  
        print '-'*80 # mail box locked till quit  
        if i < msgCount - 1:  
            raw_input('[Press Enter key]')  
finally:  
    server.quit() # make sure we unlock mbox  
    # else locked till timeout  
print 'Bye.'
```

Though primitive, this script illustrates the basics of reading email in Python. To establish a connection to an email server, we start by making an instance of the `poplib.POP3` object, passing in the email server machine's name:

```
server = poplib.POP3(mailserver)
```

If this call doesn't raise an exception, we're connected (by socket) to the POP server listening for requests on POP port number 110 at the machine where our email account lives. The next thing we need to do before fetching messages is tell the server our username and password; notice that the password method is called `pass_` -- without the trailing underscore, `pass` would name a reserved word and trigger a syntax error:

```
server.user(mailuser)           # connect, login to mail server
server.pass_(mailpasswd)        # pass is a reserved word
```

To keep things simple and relatively secure, this script always asks for the account password interactively; the `getpass` module we met in the FTP section of this chapter is used to input but not display a password string typed by the user.

Once we've told the server our username and password, we're free to fetch mailbox information with the `stat` method (number messages, total bytes among all messages), and fetch a particular message with the `retr` method (pass the message number; they start at 1):

```
msgCount, msgBytes = server.stat()
hdr, message, octets = server.retr(i+1) # octets is byte count
```

When we're done, we close the email server connection by calling the POP object's `quit` method

```
server.quit() # else locked till timeout
```

Notice that this call appears inside the `finally` clause of a `try` statement that wraps the bulk of the script. To minimize complications associated with changes, POP servers lock your email box between the time you first connect and the time you close your connection (or until an arbitrarily long system-defined time-out expires). Because the POP `quit` method also unlocks the mailbox, it's crucial that we do this before exiting, whether an exception is raised during email processing or not. By wrapping the action in a `try/finally` statement, we guarantee that the script calls `quit` on exit to unlock the mailbox to make it accessible to other processes (e.g., delivery of incoming email).

Here is the `popmail` script in action, displaying two messages in my account's mailbox on machine `pop.rmi.net` -- the domain name of the mail server machine at `rmi.net`, configured in module `mailconfig`:

```
C:\...\PP2E\Internet\Email>python popmail.py
Password for pop.rmi.net?
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <4860000073ed6c39@chevalier>
There are 2 mail messages in 1386 bytes
('+OK 2 messages (1386 octets)', ['1 744', '2 642'], 14)
-----
```

```
[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:13:33 2000)
X-From_: lumber.jack@TheLarch.com Wed Jul 12 16:10:28 2000
```

```
Return-Path: <lumber.jack@TheLarch.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA21434
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:10:27 -0600 (MDT)
From: lumber.jack@TheLarch.com
Message-Id: <200007122210.QAA21434@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:03:59 2000
Subject: I'm a Lumberjack, and I'm okay
X-Mailer: PyMailGui Version 1.0 (Python)
```

I cut down trees, I skip and jump,
I like to press wild flowers...

```
[Press Enter key]
Received: by chevalier (mbox lutz)
        (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:13:54 2000)
X-From_: lutz@rmi.net Wed Jul 12 16:12:42 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
        by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA24093
        for <lutz@rmi.net>; Wed, 12 Jul 2000 16:12:37 -0600 (MDT)
Message-Id: <200007122212.QAA24093@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:06:12 2000
Subject: testing
X-Mailer: PyMailGui Version 1.0 (Python)
```

Testing Python mail tools.

Bye.

This interface is about as simple as it could be -- after connecting to the server, it prints the complete raw text of one message at a time, pausing between each until you type the enter key. The `raw_input` built-in is called to wait for the key press between message displays.^[6] The pause keeps messages from scrolling off the screen too fast; to make them visually distinct, emails are also separated by lines of dashes. We could make the display more fancy (e.g., we'll pick out parts of messages in later examples with the `rfc822` module), but here we simply display the whole message that was sent.

[6] An extra `raw_input` is inserted on Windows only, in order to clear the stream damage of the `getpass` call; see the note about this issue in the FTP section of this chapter.

If you look closely at these mails' text, you may notice that they were actually sent by another program called PyMailGui (a program we'll meet near the end of this chapter). The "X-Mailer" header line, if present, typically identifies the sending program. In fact, there are a variety of extra header lines that can be sent in a message's text. The "Received:" headers, for example, trace the machines that a message passed through on its way to the target mailbox. Because `popmail` prints the entire raw text of a message, you see all headers here, but you may see only a few by default in end-user-oriented mail GUIs such as Outlook.

Before we move on, I should also point out that this script never deletes mail from the server. Mail is simply retrieved and printed and will be shown again the next time you run the script (barring deletion in another tool). To really remove mail permanently, we need to call other methods (e.g., `server.delete(msgnum)`) but such a capability is best deferred until we develop more interactive mail tools.

11.3.2 SMTP: Sending Email

There is a proverb in hackerdom that states that every useful computer program eventually grows complex enough to send email. Whether such somewhat ancient wisdom rings true or not in practice, the ability to automatically initiate email from within a program is a powerful tool.

For instance, test systems can automatically email failure reports, user interface programs can ship purchase orders to suppliers by email, and so on. Moreover, a portable Python mail script could be used to send messages from any computer in the world with Python and an Internet connection. Freedom from dependence on mail programs like Outlook is an attractive feature if you happen to make your living traveling around teaching Python on all sorts of computers.

Luckily, sending email from within a Python script is just as easy as reading it. In fact, there are at least four ways to do so:

Calling `os.popen` to launch a command-line mail program

On some systems, you can send email from a script with a call of the form:

```
os.popen('mail -s "xxx" a@b.c', 'w').write(text)
```

As we've seen earlier in the book, the `popen` tool runs the command-line string passed to its first argument, and returns a file-like object connected to it. If we use an open mode of "w", we are connected to the command's standard input stream -- here, we write the text of the new mail message to the standard Unix `mail` command-line program. The net effect is as if we had run `mail` interactively, but it happens inside a running Python script.

Running the `sendmail` program

The open source `sendmail` program offers another way to initiate mail from a program. Assuming it is installed and configured on your system, you can launch it using Python tools like the `os.popen` call of the previous paragraph.

Using the standard `smtplib` Python module

Python's standard library comes with support for the client-side interface to SMTP -- the Simple Mail Transfer Protocol -- a higher-level Internet standard for sending mail over sockets. Like the `poplib` module we met in the previous section, `smtplib` hides all the socket and protocol details, and can be used to send mail on any machine with Python and a socket-based Internet link.

Fetching and using third party packages and tools

Other tools in the open source library provide higher-level mail handling packages for Python (accessible from <http://www.python.org>). Most build upon one of the prior three

techniques.

Of these four options, `smtplib` is by far the most portable and powerful. Using `popen` to spawn a mail program usually works on Unix-like platforms only, not on Windows (it assumes a command-line mail program). And although the `sendmail` program is powerful, it is also somewhat Unix-biased, complex, and may not be installed even on all Unix-like machines.

By contrast, the `smtplib` module works on any machine that has Python and an Internet link, including Unix, Linux, and Windows. Moreover, SMTP affords us much control over the formatting and routing of email. Since it is arguably the best option for sending mail from a Python script, let's explore a simple mailing program that illustrates its interfaces. The Python script shown in [Example 11-17](#) is intended to be used from an interactive command line; it reads a new mail message from the user and sends the new mail by SMTP using Python's `smtplib` module.

Example 11-17. PP2E\Internet\Email\smtpmail.py

```
#!/usr/local/bin/python
#####
# use the Python SMTP mail interface module to send
# email messages; this is just a simple one-shot
# send script--see pymail, PyMailGui, and PyMailCgi
# for clients with more user interaction features,
# and popmail.py for a script which retrieves mail;
#####

import smtplib, string, sys, time, mailconfig
mailserver = mailconfig.smtpservername # ex: starship.python.net

From = string.strip(raw_input('From? ')) # ex: lutz@rmi.net
To = string.strip(raw_input('To? ')) # ex: python-list@python.org
To = string.split(To, ';') # allow a list of recipients
Subj = string.strip(raw_input('Subj? '))

# prepend standard headers
date = time.ctime(time.time())
text = ('From: %s\nTo: %s\nDate: %s\nSubject: %s\n'
        % (From, string.join(To, ';'), date, Subj))

print 'Type message text, end with line=(ctrl + D or Z)'
while 1:
    line = sys.stdin.readline()
    if not line:
        break # exit on ctrl-d/z
    # if line[:4] == 'From':
    #     line = '>' + line # servers escape for us
    text = text + line

if sys.platform[:3] == 'win': print
print 'Connecting...'
server = smtplib.SMTP(mailserver) # connect, no login step
failed = server.sendmail(From, To, text)
server.quit()
if failed: # smtplib may raise exceptions
    print 'Failed recipients:', failed # too, but let them pass here
else:
    print 'No errors.'
print 'Bye.'
```

Most of this script is user interface -- it inputs the sender's address ("From"), one or more recipient addresses ("To", separated by ";" if more than one), and a subject line. The sending data is picked up from Python's standard `time` module, standard header lines are formatted, and the `while` loop reads message lines until the user types the end-of-file character (Ctrl-Z on Windows; Ctrl-D on Linux).

The rest of the script is where all the SMTP magic occurs: to send a mail by SMTP, simply run these two sorts of calls:

```
server = smtplib.SMTP(mailserver)
```

Make an instance of the SMTP object, passing in the name of the SMTP server that will dispatch the message first. If this doesn't throw an exception, you're connected to the SMTP server via a socket when the call returns.

```
failed = server.sendmail(From, To, text)
```

Call the SMTP object's `sendmail` method, passing in the sender address, one or more recipient addresses, and the text of the message itself with as many standard mail header lines as you care to provide.

When you're done, call the object's `quit` method to disconnect from the server. Notice that, on failure, the `sendmail` method may either raise an exception or return a list of the recipient addresses that failed; the script handles the latter case but lets exceptions kill the script with a Python error message.

11.3.2.1 Sending messages

Okay -- let's ship a few messages across the world. The `smtpmail` script is a one-shot tool: each run allows you to send a single new mail message. Like most of the client-side tools in this chapter, it can be run from any computer with Python and an Internet link. Here it is running on Windows 98:

```
C:\...\PP2E\Internet\Email>python smtpmail.py
From? Eric.the.Half.a.Bee@semibee.com
To? lutz@rmi.net
Subj? A B C D E F G
Type message text, end with line=(ctrl + D or Z)
Fiddle de dum, Fiddle de dee,
Eric the half a bee.

Connecting...
No errors.
Bye.
```

This mail is sent to my address (lutz@rmi.net), so it ultimately shows up in my mailbox at my ISP, but only after being routed through an arbitrary number of machines on the Net, and across arbitrarily distant network links. It's complex at the bottom, but usually, the Internet "just works."

Notice the "From" address, though -- it's completely fictitious (as far as I know, at least). It turns out that we can usually provide any "From" address we like because SMTP doesn't check its validity (only its general format is checked). Furthermore, unlike POP, there is no notion of a username or password in SMTP, so the sender is more difficult to determine. We need only pass email to any machine with a server listening on the SMTP port, and don't need an account on the machine. Here, `Eric.the.Half.a.Bee@semibee.com` works fine as the sender; `Marketing.Geek.From.Hell@spam.com` would work just as well.

I'm going to tell you something now for instructional purposes only: it turns out that this behavior is the basis of all those annoying junk emails that show up in your mailbox without a real sender's address.^[7] Salesmen infected with e-millionaire mania will email advertising to all addresses on a list without providing a real "From" address, to cover their tracks.

[7] Such junk mail is usually referred to as spam, a reference to a Monty Python skit where people trying to order breakfast at a restaurant were repeatedly drowned out by a group of Vikings singing an increasingly loud chorus of "spam, spam, spam,..." (no, really). While spam can be used in many ways, this usage differs both from its appearance in this book's examples, and its much-lauded role as a food product.

Normally, of course, you should use the same "To" address in the message and the SMTP call, and provide your real email address as the "From" value (that's the only way people will be able to reply to your message). Moreover, apart from teasing your significant other, sending phony addresses is just plain bad Internet citizenship. Let's run the script again to ship off another mail with more politically correct coordinates:

```
C:\...\PP2E\Internet\Email>python smtpmail.py
From? lutz@rmi.net
To?   lutz@rmi.net
Subj? testing smtpmail
Type message text, end with line=(ctrl + D or Z)
Lovely Spam! Wonderful Spam!
Connecting...
No errors.
Bye.
```

At this point, we could run whatever email tool we normally use to access our mailbox to verify the results of these two send operations; the two new emails should show up in our mailbox regardless of which mail client is used to view them. Since we've already written a Python script for reading mail, though, let's put it to use as a verification tool -- running the `popmail` script from the last section reveals our two new messages at the end of the mail list:

```
C:\...\PP2E\Internet\Email>python popmail.py
Password for pop.rmi.net?
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <c4050000b6ee6c39@chevalier>
There are 6 mail messages in 10941 bytes
('+OK 6 messages (10941 octets)', ['1 744', '2 642', '3 4456', '4 697', '5 3791', '6 611'], 44)
-----
...
...lines omitted...
```

...
[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:19:20 2000)
X-From_ : Eric.the.Half.a.Bee@semibee.com Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
 by chevalier.rmi.net (8.9.3/8.9.3) with ESMTTP id QAA28647
 for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:19:51 2000)
X-From_ : lutz@rmi.net Wed Jul 12 16:17:58 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
 by chevalier.rmi.net (8.9.3/8.9.3) with ESMTTP id QAA00415
 for <lutz@rmi.net>; Wed, 12 Jul 2000 16:17:57 -0600 (MDT)
Message-Id: <200007122217.QAA00415@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:10:55 2000
Subject: testing smtpmail

Lovely Spam! Wonderful Spam!

Bye.

11.3.2.2 More ways to abuse the Net

The first mail here was the one we sent with a fictitious address; the second was the more legitimate message. Like "From" addresses, header lines are a bit arbitrary under SMTP, too. `smtpmail` automatically adds "From:" and "To:" header lines in the message's text with the same addresses as passed to the SMTP interface, but only as a polite convention. Sometimes, though, you can't tell who a mail was sent to either -- to obscure the target audience, spammers also may play games with "Bcc" blind copies or the contents of headers in the message's text.

For example, if we change `smtpmail` to not automatically generate a "To:" header line with the same address(es) sent to the SMTP interface call, we can manually type a "To:" header that differs from the address we're really sending to:

```
C:\...\PP2E\Internet\Email>python smtpmail-noTo.py
From? Eric.the.Half.a.Bee@semibee.com
To? lutz@starship.python.net
Subj? a b c d e f g
Type message text, end with line=(ctrl + D or Z)
To: nobody.in.particular@marketing.com
```

```
Fiddle de dum, Fiddle de dee,  
Eric the half a bee.  
Connecting...  
No errors.  
Bye.
```

In some ways, the "From" and "To" addresses in send method calls and message header lines are similar to addresses on envelopes and letters in envelopes. The former is used for routing, but the latter is what the reader sees. Here, I gave the "To" address as my mailbox on the starship.python.net server, but gave a fictitious name in the manually typed "To:" header line; the first address is where it really goes. A command-line mail tool running on starship by Telnet reveals two bogus mails sent -- one with a bad "From:", and the one with an additionally bad "To:" that we just sent:

```
[lutz@starship lutz]$ mail  
Mail version 8.1 6/6/93. Type ? for help.  
"/home/crew/lutz/Mailbox": 22 messages 12 new 22 unread  
...more...  
>N 21 Eric.the.Half.a.Bee@ Thu Jul 13 20:22 20/789 "A B C D E F G"  
N 22 Eric.the.Half.a.Bee@ Thu Jul 13 20:26 19/766 "a b c d e f g"
```

```
& 21  
Message 21:  
From Eric.the.Half.a.Bee@semibee.com Thu Jul 13 20:21:18 2000  
Delivered-To: lutz@starship.python.net  
From: Eric.the.Half.a.Bee@semibee.com  
To: lutz@starship.python.net  
Date: Thu Jul 13 14:15:55 2000  
Subject: A B C D E F G
```

```
Fiddle de dum, Fiddle de dee,  
Eric the half a bee.
```

```
& 22  
Message 22:  
From Eric.the.Half.a.Bee@semibee.com Thu Jul 13 20:26:34 2000  
Delivered-To: lutz@starship.python.net  
From: Eric.the.Half.a.Bee@semibee.com  
Date: Thu Jul 13 14:20:22 2000  
Subject: a b c d e f g  
To: nobody.in.particular@marketing.com
```

```
Fiddle de dum, Fiddle de dee,  
Eric the half a bee.
```

If your mail tool picks out the "To:" line, such mails look odd when viewed. For instance, here's another sent to my rmi.net mailbox:

```
C:\...\PP2E\Internet\Email>python smtpmail-noTo.py  
From? Arthur@knights.com  
To? lutz@rmi.net  
Subj? Killer bunnies  
Type message text, end with line=(ctrl + D or Z)  
To: you@home.com  
Run away! Run away! ...  
Connecting...  
No errors.  
Bye.
```

When it shows up in my mailbox on rmi.net, it's difficult to tell much about its origin or destination in either Outlook or a Python-coded mail tool we'll meet near the end of this chapter (see [Figure 11-8](#) and [Figure 11-9](#)). And its raw text will only show the machines it has been

routed through.

Figure 11-8. Bogus mail in Outlook

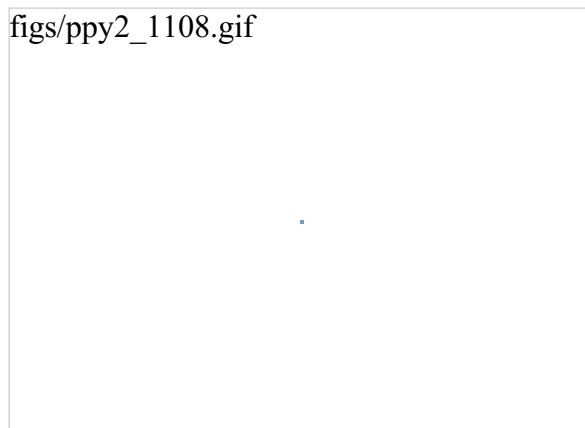


Figure 11-9. Bogus mail in a Python mail tool (PyMailGui)



Once again, though -- don't do this unless you have good reason. I'm showing it for header-line illustration purposes (e.g., in a later section, we'll add an "X-mailer:" header line to identify the sending program). Furthermore, to stop a criminal, you sometimes need to think like one -- you can't do much about spam mail unless you understand how it is generated. To write an automatic spam filter that deletes incoming junk mail, for instance, you need to know the telltale signs to look for in a message's text. And "To" address juggling may be useful in the context of legitimate mailing lists.

But really, sending email with bogus "From:" and "To:" lines is equivalent to making anonymous phone calls. Most mailers won't even let you change the "From" line, and don't distinguish between the "To" address and header line, but SMTP is wide open in this regard. Be good out there; okay?

11.3.2.3 Back to the big Internet picture

So where are we at in the Internet abstraction model now? Because mail is transferred over sockets (remember sockets?), they are at the root of all of this email fetching and sending. All email read and written ultimately consists of formatted bytes shipped over sockets between computers on the Net. As we've seen, though, the POP and SMTP interfaces in Python hide all the details. Moreover, the scripts we've begun writing even hide the Python interfaces and provide higher-level interactive tools.

Both `popmail` and `smtpmail` provide portable email tools, but aren't quite what we'd expect in terms of usability these days. In the next section, we'll use what we've seen thus far to implement a more interactive mail tool. At the end of this email section, we'll also code a Tk email GUI, and then we'll go on to build a web-based interface in a later chapter. All of these tools, though, vary primarily in terms of user interface only; each ultimately employs the mail modules we've met here to transfer mail message text over the Internet with sockets.

11.3.3 A Command-Line Email Client

Now, let's put together what we've learned about fetching and sending email in a simple but functional command-line email tool. The script in [Example 11-18](#) implements an interactive email session -- users may type commands to read, send, and delete email messages.

Example 11-18. PP2E\Internet\Email\pymail.py

```
#!/usr/local/bin/python
#####
# A simple command-line email interface client in
# Python; uses Python POP3 mail interface module to
# view pop email account messages; uses rfc822 and
# StringIO modules to extract mail message headers;
#####

import poplib, rfc822, string, StringIO

def connect(servername, user, passwd):
    print 'Connecting...'
    server = poplib.POP3(servername)
    server.user(user)                # connect, login to mail server
    server.pass_(passwd)             # pass is a reserved word
    print server.getwelcome()        # print returned greeting message
    return server

def loadmessages(servername, user, passwd, loadfrom=1):
    server = connect(servername, user, passwd)
    try:
        print server.list()
        (msgCount, msgBytes) = server.stat()
        print 'There are', msgCount, 'mail messages in', msgBytes, 'bytes'
        print 'Retrieving:',
        msgList = []
        for i in range(loadfrom, msgCount+1):
            print i,                    # empty if low >= high
            (hdr, message, octets) = server.retr(i) # fetch mail now
            msgList.append(string.join(message, '\n')) # save text on list
            # leave mail on server
        print
    finally:
        server.quit()                  # unlock the mail box
    assert len(msgList) == (msgCount - loadfrom) + 1 # msg nums start at 1
```

```
return msgList

def deletemessages(servername, user, passwd, toDelete, verify=1):
    print 'To be deleted:', toDelete
    if verify and raw_input('Delete?')[0] not in ['y', 'Y']:
        print 'Delete cancelled.'
    else:
        server = connect(servername, user, passwd)
        try:
            print 'Deleting messages from server.'
            for msgnum in toDelete:
                server.dele(msgnum) # reconnect to delete mail
                                     # mbox locked until quit()
        finally:
            server.quit()

def showindex(msgList):
    count = 0
    for msg in msgList:
        # strip, show some mail headers
        strfile = StringIO.StringIO(msg) # make string look like a file
        msghdrs = rfc822.Message(strfile) # parse mail headers into a dict
        count = count + 1
        print '%d:\t%d bytes' % (count, len(msg))
        for hdr in ('From', 'Date', 'Subject'):
            try:
                print '\t%s=>%s' % (hdr, msghdrs[hdr])
            except KeyError:
                print '\t%s=>(unknown)' % hdr
            #print '\n\t%s=>%s' % (hdr, msghdrs.get(hdr, '(unknown)'))
        if count % 5 == 0:
            raw_input('[Press Enter key]') # pause after each 5

def showmessage(i, msgList):
    if 1 <= i <= len(msgList):
        print '-'*80
        print msgList[i-1] # this prints entire mail--hdrs+text
        print '-'*80 # to get text only, call file.read()
    else:
        # after rfc822.Message reads hdr lines
        print 'Bad message number'

def savemessage(i, mailfile, msgList):
    if 1 <= i <= len(msgList):
        open(mailfile, 'a').write('\n' + msgList[i-1] + '-'*80 + '\n')
    else:
        print 'Bad message number'

def msgnum(command):
    try:
        return string.atoi(string.split(command)[1])
    except:
        return -1 # assume this is bad

helptext = """
Available commands:
i      - index display
l n?   - list all messages (or just message n)
d n?   - mark all messages for deletion (or just message n)
s n?   - save all messages to a file (or just message n)
m      - compose and send a new mail message
q      - quit pmail
?      - display this help text
"""

def interact(msgList, mailfile):
    showindex(msgList)
    toDelete = []
    while 1:
        try:
```

```
        command = raw_input('[Pymail] Action? (i, l, d, s, m, q, ?) ')
except EOFError:
    command = 'q'

# quit
if not command or command == 'q':
    break

# index
elif command[0] == 'i':
    showindex(msgList)

# list
elif command[0] == 'l':
    if len(command) == 1:
        for i in range(1, len(msgList)+1):
            showmessage(i, msgList)
    else:
        showmessage(msgnum(command), msgList)

# save
elif command[0] == 's':
    if len(command) == 1:
        for i in range(1, len(msgList)+1):
            savemessage(i, mailfile, msgList)
    else:
        savemessage(msgnum(command), mailfile, msgList)

# delete
elif command[0] == 'd':
    if len(command) == 1:
        toDelete = range(1, len(msgList)+1)      # delete all later
    else:
        delnum = msgnum(command)
        if (1 <= delnum <= len(msgList)) and (delnum not in toDelete):
            toDelete.append(delnum)
        else:
            print 'Bad message number'

# mail
elif command[0] == 'm':
    # send a new mail via smtp
    try:
        execfile('smtpmail.py', {})           # reuse existing script
    except:
        print 'Error - mail not sent'        # don't die if script dies

elif command[0] == '?':
    print helptext
else:
    print 'What? -- type "?" for commands help'
return toDelete

if __name__ == '__main__':
    import sys, getpass, mailconfig
    mailserver = mailconfig.popservername      # ex: 'starship.python.net'
    mailuser = mailconfig.popusername         # ex: 'lutz'
    mailfile = mailconfig.savemailfile       # ex: r'c:\stuff\savemail'
    mailpswd = getpass.getpass('Password for %s?' % mailserver)

    if sys.platform[:3] == 'win': raw_input() # clear stream
    print '[Pymail email client]'
    msgList = loadmessages(mailserver, mailuser, mailpswd) # load all
    toDelete = interact(msgList, mailfile)
    if toDelete: deletemessages(mailserver, mailuser, mailpswd, toDelete)
    print 'Bye.'
```

There isn't much new here -- just a combination of user-interface logic and tools we've already met, plus a handful of new tricks:

Loads

This client loads all email from the server into an in-memory Python list only once, on startup; you must exit and restart to reload newly arrived email.

Saves

On demand, `pymail` saves the raw text of a selected message into a local file, whose name you place in the `mailconfig` module.

Deletions

We finally support on-request deletion of mail from the server here: in `pymail`, mails are selected for deletion by number, but are still only physically removed from your server on exit, and then only if you verify the operation. By deleting only on exit, we avoid changing mail message numbers during a session -- under POP, deleting a mail not at the end of the list decrements the number assigned to all mails following the one deleted. Since mail is cached in memory by `pymail`, future operations on the numbered messages in memory may be applied to the wrong mail if deletions were done immediately.^[8]

[8] More on POP message numbers when we study PyMailGui later in this chapter. Interestingly, the list of message numbers to be deleted need not be sorted; they remain valid for the duration of the connection.

Parsing messages

Pymail still displays the entire raw text of a message on listing commands, but the mail index listing only displays selected headers parsed out of each message. Python's `rfc822` module is used to extract headers from a message: the call `rfc822.Message(strfile)` returns an object with dictionary interfaces for fetching the value of a message header by name string (e.g., index the object on string "From" to get the value of the "From" header line).

Although unused here, anything not consumed from `strfile` after a `Message` call is the body of the message, and can be had by calling `strfile.read`. `Message` reads the message headers portion only. Notice that `strfile` is really an instance of the standard `StringIO.StringIO` object. This object wraps the message's raw text (a simple string) in a file-like interface; `rfc822.Message` expects a file interface, but doesn't care if the object is a true file or not. Once again, interfaces are what we code to in Python, not specific types. Module `StringIO` is useful anytime you need to make a string look like a file.

By now, I expect that you know enough Python to read this script for a deeper look, so rather than saying more about its design here, let's jump into an interactive `pymail` session to see how it works.

Does Anybody Really Know What Time It Is?
--

Minor caveat: the simple date format used in the `smtpmail` program (and others in this book) doesn't quite follow the SMTP date formatting standard. Most servers don't care, and will let any sort of date text appear in date header lines. In fact, I've never seen a mail fail due to date formats.

If you want to be more in line with the standard, though, you could format the date header with code like this (adopted from standard module `urllib`, and parseable with standard tools such as the `rfc822` module and the `time.strptime` call):

```
import time
gmt = time.gmtime(time.time())
fmt = '%a, %d %b %Y %H:%M:%S GMT'
str = time.strftime(fmt, gmt)
hdr = 'Date: ' + str
print hdr
```

The `hdr` variable looks like this when this code is run:

```
Date: Fri, 02 Jun 2000 16:40:41 GMT
```

instead of the date format currently used by the `smtpmail` program:

```
>>> import time
>>> time.ctime(time.time())
'Fri Jun 02 10:23:51 2000'
```

The `time.strftime` call allows arbitrary date and time formatting (`time.ctime` is just one standard format), but we will leave rooting out the workings of all these calls as a suggested exercise for the reader; consult the `time` module's library manual entry.

We'll also leave placing such code in a reusable file to the more modular among you. Time and date formatting rules are necessary, but aren't pretty.

11.3.3.1 Running the `pymail` command-line client

Let's start up `pymail` to read and delete email at our mail server and send new messages. `Pymail` runs on any machine with Python and sockets, fetches mail from any email server with a POP interface on which you have an account, and sends mail via the SMTP server you've named in the `mailconfig` module.

Here it is in action running on my Windows 98 laptop machine; its operation is identical on other machines. First, we start the script, supply a POP password (remember, SMTP servers require no password), and wait for the `pymail` email list index to appear:

```
C:\...\PP2E\Internet\Email>python pymail.py
Password for pop.rmi.net?

[Pymail email client]
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <870f000002f56c39@chevalier>
('+OK 5 messages (7150 octets)', ['1 744', '2 642', '3 4456', '4 697', '5 611']
36)
There are 5 mail messages in 7150 bytes
Retrieving: 1 2 3 4 5
There are 5 mail messages in 7150 bytes
Retrieving: 1 2 3 4 5
1:      676 bytes
      From=>lumber.jack@TheLarch.com
      Date=>Wed Jul 12 16:03:59 2000
```

```
2:      Subject=>I'm a Lumberjack, and I'm okay
      587 bytes
      From=>lutz@rmi.net
      Date=>Wed Jul 12 16:06:12 2000
      Subject=>testing
3:      4307 bytes
      From=>"Mark Hammond" <MarkH@ActiveState.com>
      Date=>Wed, 12 Jul 2000 18:11:58 -0400
      Subject=>[Python-Dev] Python .NET (was Preventing 1.5 extensions...
4:      623 bytes
      From=>Eric.the.Half.a.Bee@semibee.com
      Date=>Wed Jul 12 16:09:21 2000
      Subject=>A B C D E F G
5:      557 bytes
      From=>lutz@rmi.net
      Date=>Wed Jul 12 16:10:55 2000
      Subject=>testing smtpmail
```

[Press Enter key]

[Pymail] Action? (i, l, d, s, m, q, ?) **1 5**

```
Received: by chevalier (mbox lutz)
      (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: lutz@rmi.net Wed Jul 12 16:17:58 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
      by chevalier.rmi.net (8.9.3/8.9.3) with ESMTF id QAA00415
      for <lutz@rmi.net>; Wed, 12 Jul 2000 16:17:57 -0600 (MDT)
Message-Id: <200007122217.QAA00415@chevalier.rmi.net>
From: lutz@rmi.net
To: lutz@rmi.net
Date: Wed Jul 12 16:10:55 2000
Subject: testing smtpmail
```

Lovely Spam! Wonderful Spam!

[Pymail] Action? (i, l, d, s, m, q, ?) **1 4**

```
Received: by chevalier (mbox lutz)
      (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: Eric.the.Half.a.Bee@semibee.com Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
      by chevalier.rmi.net (8.9.3/8.9.3) with ESMTF id QAA28647
      for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G
```

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

Once `pymail` downloads your email to a Python list on the local client machine, you type command letters to process it. The "l" command lists (prints) the contents of a given mail number; here, we used it to list the two emails we wrote with the `smtpmail` script in the last section.

Pymail also lets us get command help, delete messages (deletions actually occur at the server on exit from the program), and save messages away in a local text file whose name is listed in the `mailconfig` module we saw earlier:

```
[Pymail] Action? (i, l, d, s, m, q, ?) ?  
  
Available commands:  
i      - index display  
l n?   - list all messages (or just message n)  
d n?   - mark all messages for deletion (or just message n)  
s n?   - save all messages to a file (or just message n)  
m      - compose and send a new mail message  
q      - quit pymail  
?      - display this help text  
  
[Pymail] Action? (i, l, d, s, m, q, ?) d 1  
[Pymail] Action? (i, l, d, s, m, q, ?) s 4
```

Now, let's pick the "m" mail compose option -- `pymail` simply executes the `smtpmail` script we wrote in the prior section and resumes its command loop (why reinvent the wheel?). Because the script sends by SMTP, you can use arbitrary "From" addresses here; but again, you generally shouldn't do that (unless, of course, you're trying to come up with interesting examples for a book).

The `smtpmail` script is run with the built-in `execfile` function; if you look at `pymail`'s code closely, you'll notice that it passes an empty dictionary to serve as the script's namespace to prevent its names from clashing with names in `pymail` code. `execfile` is a handy way to reuse existing code written as a top-level script, and thus is not really importable. Technically speaking, code in the file `smtplib.py` would run when imported, but only on the first import (later imports would simply return the loaded module object). Other scripts that check the `__name__` attribute for `__main__` won't generally run when imported at all:

```
[Pymail] Action? (i, l, d, s, m, q, ?) m  
From? Cardinal@nice.red.suits.com  
To? lutz@rmi.net  
Subj? Among our weapons are these:  
Type message text, end with line=(ctrl + D or Z)  
Nobody Expects the Spanish Inquisition!  
Connecting...  
No errors.  
Bye.  
[Pymail] Action? (i, l, d, s, m, q, ?) q  
To be deleted: [1]  
Delete?y  
Connecting...  
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <8e2e0000aff66c39@chevalier>  
Deleting messages from server.  
Bye.
```

As mentioned, deletions really happen only on exit; when we quit `pymail` with the "q" command it tells us which messages are queued for deletion, and verifies the request. If verified, `pymail` finally contacts the mail server again and issues POP calls to delete the selected mail messages.

Because `pymail` downloads mail from your server into a local Python list only once at startup, though, we need to start `pymail` again to re-fetch mail from the server if we want to see the result of the mail we sent and the deletion we made. Here, our new mail shows up as number 5, and the original mail assigned number 1 is gone:

```
C:\...\PP2E\Internet\Email>python pymail.py
Password for pop.rmi.net?
```

```
[Pymail email client]
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <40310000d5f66c39@chevalier>
...
There are 5 mail messages in 7090 bytes
Retrieving: 1 2 3 4 5
1:      587 bytes
   From=>lutz@rmi.net
   Date=>Wed Jul 12 16:06:12 2000
   Subject=>testing
2:      4307 bytes
   From=>"Mark Hammond" <MarkH@ActiveState.com>
   Date=>Wed, 12 Jul 2000 18:11:58 -0400
   Subject=>[Python-Dev] Python .NET (was Preventing 1.5 extensions...
3:      623 bytes
   From=>Eric.the.Half.a.Bee@semibee.com
   Date=>Wed Jul 12 16:09:21 2000
   Subject=>A B C D E F G
4:      557 bytes
   From=>lutz@rmi.net
   Date=>Wed Jul 12 16:10:55 2000
   Subject=>testing smtpmail
5:      615 bytes
   From=>Cardinal@nice.red.suits.com
   Date=>Wed Jul 12 16:44:58 2000
   Subject=>Among our weapons are these:
[Press Enter key]
[Pymail] Action? (i, l, d, s, m, q, ?) 1 5
```

```
-----
Received: by chevalier (mbox lutz)
   (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:53:24 2000)
X-From_: Cardinal@nice.red.suits.com Wed Jul 12 16:51:53 2000
Return-Path: <Cardinal@nice.red.suits.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
   by chevalier.rmi.net (8.9.3/8.9.3) with ESMTP id QAA11127
   for <lutz@rmi.net>; Wed, 12 Jul 2000 16:51:52 -0600 (MDT)
From: Cardinal@nice.red.suits.com
Message-Id: <200007122251.QAA11127@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:44:58 2000
Subject: Among our weapons are these:
```

Nobody Expects the Spanish Inquisition!

```
-----
[Pymail] Action? (i, l, d, s, m, q, ?) q
Bye.
```

Finally, here is the mail save file, containing the one message we asked to be saved in the prior session; it's simply the raw text of saved emails, with separator lines. This is both human- and machine-readable -- in principle, another script could load saved mail from this file into a Python list, by calling the `string.split` function on the file's text with the separator line as a delimiter:

```
C:\...\PP2E\Internet\Email>type c:\stuff\etc\savemail.txt
```

```
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Wed Jul 12 16:45:38 2000)
X-From_: Eric.the.Half.a.Bee@semibee.com Wed Jul 12 16:16:31 2000
Return-Path: <Eric.the.Half.a.Bee@semibee.com>
Received: from VAIO (dial-218.101.denco.rmi.net [166.93.218.101])
 by chevalier.rmi.net (8.9.3/8.9.3) with ESMTTP id QAA28647
 for <lutz@rmi.net>; Wed, 12 Jul 2000 16:16:30 -0600 (MDT)
From: Eric.the.Half.a.Bee@semibee.com
Message-Id: <200007122216.QAA28647@chevalier.rmi.net>
To: lutz@rmi.net
Date: Wed Jul 12 16:09:21 2000
Subject: A B C D E F G
```

Fiddle de dum, Fiddle de dee,
Eric the half a bee.

11.3.4 Decoding Mail Message Attachments

In the last section, we learned how to parse out email message headers and bodies with the `rfc822` and `StringIO` modules. This isn't quite enough for some messages, though. In this section, I will introduce tools that go further, to handle complex information in the bodies of email messages.

One of the drawbacks of stubbornly clinging to a Telnet command-line email interface is that people sometimes send email with all sorts of attached information -- pictures, MS Word files, uuencoded tar files, base64-encoded documents, HTML pages, and even executable scripts that can trash your computer if opened.^[9] Not all attachments are crucial, of course, but email isn't always just ASCII text these days.

[9] I should explain this one: I'm referring to email viruses that appeared in 2000. The short story behind most of them is that Microsoft Outlook sported a "feature" that allowed email attachments to embed and contain executable scripts, and allowed these scripts to gain access to critical computer components when open and run. Furthermore, Outlook had another feature that automatically ran such attached scripts when an email was inspected, whether the attachment was manually opened or not. I'll leave the full weight of such a security hole for you to ponder, but I want to add that if you use Python's attachment tools in any of the mail programs in this book, please do not execute attached programs under any circumstance, unless you also run them with Python's restricted execution mode presented in [Chapter 15](#).

Before I overcame my Telnet habits, I needed a way to extract and process all those attachments from a command line (I tried the alternative of simply ignoring all attachments completely, but that works only for a while). Luckily, Python's library tools make handling attachments and common encodings easy and portable. For simplicity, all of the following scripts work on the raw text of a saved email message (or parts of such), but they could just as easily be incorporated into the email programs in this book to extract email components automatically.

11.3.4.1 Decoding base64 data

Let's start with something simple. Mail messages and attachments are frequently sent in an encoding format such as uu or base64; binary data files in particular must be encoded in a textua format for transit using one of these encoding schemes. On the receiving end, such encoded data must first be decoded before it can be viewed, opened, or otherwise used. The Python program in [Example 11-19](#) knows how to perform base64 decoding on data stored in a file.

Example 11-19. PP2E\Internet\Email\decode64.py

```
#!/usr/bin/env python
#####
# Decode mail attachments sent in base64 form.
# This version assumes that the base64 encoded
# data has been extracted into a separate file.
# It doesn't understand mime headers or parts.
# uudecoding is similar (uu.decode(iname)),
# as is binhex decoding (binhex.hexbin(iname)).
# You can also do this with module mimetools:
# mimetools.decode(input, output, 'base64').
#####

import sys, base64

iname = 'part.txt'
oname = 'part.doc'

if len(sys.argv) > 1:
    iname, oname = sys.argv[1:]    # % python prog [iname oname]?

input = open(iname, 'r')
output = open(oname, 'wb')        # need wb on windows for docs
base64.decode(input, output)     # this does most of the work
print 'done'
```

There's not much to look at here, because all the low-level translation work happens in the Python `base64` module; we simply call its `decode` method with open input and output files. Other transmission encoding schemes are supported by different Python modules -- `uu` for uuencoding, `binhex` for binhex format, and so on. All of these export interfaces that are analogous to `base64`, and are as easy to use; `uu` and `binhex` use the output filename in the data (see the library manual for details).

At a slightly higher level of generality, the `mimetools` module exports a `decode` method, which supports all encoding schemes. The desired decoding is given by a passed-in argument, but the net result is the same, as shown in [Example 11-20](#).

Example 11-20. PP2E\Internet\Email\decode64_b.py

```
#!/usr/bin/env python
#####
# Decode mail attachments sent in base64 form.
# This version tests the mimetools module.
#####

import sys, mimetools

iname = 'part.txt'
oname = 'part.doc'

if len(sys.argv) > 1:
    iname, oname = sys.argv[1:]    # % python prog [iname oname]?
```

```
input = open(iname, 'r')
output = open(oname, 'wb')
mimertools.decode(input, output, 'base64') # or 'uuencode', etc.
print 'done'
```

To use either of these scripts, you must first extract the base64-encoded data into a text file. Save a mail message in a text file using your favorite email tool, then edit the file to save only the base64-encoded portion with your favorite text editor. Finally, pass the data file to the script, along with a name for the output file where the decoded data will be saved. Here are the base64 decoders at work on a saved data file; the generated output file turns out to be the same as the one saved for an attachment in MS Outlook earlier:

```
C:\Stuff\Mark\etc\jobs\test>python ..\decode64.py t4.64 t4.doc
done
```

```
C:\Stuff\Mark\etc\jobs\test>fc /B cand.agr10.22.doc t4.doc
Comparing files cand.agr10.22.doc and t4.doc
FC: no differences encountered
```

```
C:\Stuff\Mark\etc\jobs\test>python ..\decode64_b.py t4.64 t4.doc
done
```

```
C:\Stuff\Mark\etc\jobs\test>fc /B cand.agr10.22.doc t4.doc
Comparing files cand.agr10.22.doc and t4.doc
FC: no differences encountered
```

11.3.4.2 Extracting and decoding all parts of a message

The decoding procedure in the previous section is very manual and error-prone; moreover, it handles only one type of encoding (base64), and decodes only a single component of an email message. With a little extra logic, we can improve on this dramatically with the Python `mhlib` module's multipart message-decoding tools. For instance, the script in [Example 11-21](#) knows how to extract, decode, and save every component in an email message in one step.

Example 11-21. PP2E\Internet\Email\decodeAll.py

```
#!/usr/bin/env python
#####
# Decode all mail attachments sent in encoded form:
# base64, uu, etc. To use, copy entire mail message
# to mailfile and run:
#   % python ..\decodeAll.py mailfile
# which makes one or more mailfile.part* outputs.
#####

import sys, mhlib
from types import *
iname = 'mailmessage.txt'

if len(sys.argv) == 3:
    iname, oname = sys.argv[1:] # % python prog [iname [oname]]?
elif len(sys.argv) == 2:
    iname = sys.argv[1]
    oname = iname + '.part'

def writeparts(part, oname):
    global partnum
    content = part.getbody() # decoded content or list
```

```
if type(content) == ListType:                # multiparts: recur for each
    for subpart in content:
        writeparts(subpart, oname)
else:                                        # else single decoded part
    assert type(content) == StringType      # use filename if in headers
    print; print part.getparamnames()      # else make one with counter
    fmode = 'wb'
    fname = part.getparam('name')
    if not fname:
        fmode = 'w'
        fname = oname + str(partnum)
        if part.gettype() == 'text/plain':
            fname = fname + '.txt'
        elif part.gettype() == 'text/html':
            fname = fname + '.html'
    output = open(fname, fmode)             # mode must be 'wb' on windows
    print 'writing:', output.name           # for word doc files, not 'w'
    output.write(content)
    partnum = partnum + 1

partnum = 0
input = open(iname, 'r')                    # open mail file
message = mllib.Message('.', 0, input)      # folder, number args ignored
writeparts(message, oname)
print 'done: wrote %s parts' % partnum
```

Because `mllib` recognizes message components, this script processes an entire mail message; there is no need to edit the message to extract components manually. Moreover, the components of an `mllib.Message` object represent the already-decoded parts of the mail message -- any necessary uu, base64, and other decoding steps have already been automatically applied to the mail components by the time we fetch them from the object. `mllib` is smart enough to determine and perform decoding automatically; it supports all common encoding schemes at once, not just particular format such as base64.

To use this script, save the raw text of an email message in a local file (using whatever mail tool you like), and pass the file's name on the script's command line. Here the script is extracting and decoding the components of two saved mail message files, *t4.eml* and *t5.eml*:

```
C:\Stuff\Mark\etc\jobs\test>python ..\decodeall.py t4.eml
```

```
['charset']
writing: t4.eml.part0.txt
```

```
['charset']
writing: t4.eml.part1.html
```

```
['name']
writing: cand.agr10.22.doc
done: wrote 3 parts
```

```
C:\Stuff\Mark\etc\jobs\test>python ..\decodeall.py t5.eml
```

```
['charset']
writing: t5.eml.part0.txt
```

```
['name']
writing: US West Letter.doc
done: wrote 2 parts
```


The end result of decoding a message is a set of one or more local files containing the decoded contents of each part of the message. Because the resulting local files are the crux of this script's purpose, it must assign meaningful names to files it creates. The following naming rules are applied by the script:

1. If a component has an associated "name" parameter in the message, the script stores the component's bytes in a local file of that name. This generally reuses the file's original name on the machine where the mail originated.
2. Otherwise, the script generates a unique filename for the component by adding a "partN" suffix to the original mail file's name, and trying to guess a file extension based on the component's file type given in the message.

For instance, the message saved away as *t4.eml* consists of the message body, an alternative HTML encoding of the message body, and an attached Word doc file. When decoding *t4.eml*:

- The first two message components have no "name" parameter, so the script generates names based on the filename and component types -- *t4.eml.part0.txt* and *t4.eml.part1.htm* -- plain text and HTML code, respectively. On most machines, clicking on the HTML output file should open it in a web browser for formatted viewing.
- The last attachment was given an explicit name when attached -- *cand.agr10.22.doc* -- so it is used as the output file's name directly. Notice that this was an attached MS Word doc file when sent; assuming all went well in transit, double-clicking on the third output file generated by this script should open it in Word.

There are additional tools in the Python library for decoding data fetched over the Net, but we'll defer to the library manual for further details. Again, using this decoding script still involves some manual intervention -- users must save the mail file and type a command to split off its parts into distinct files -- but it's sufficient for handling multipart mail, and it works portably on any machine with Python. Moreover, the decoding interfaces it demonstrates can be adopted in a more automatic fashion by interactive mail clients.

For instance, the decoded text of a message component could be automatically passed to handler programs (e.g., browsers, text editors, Word) when selected, rather than written to local files. It could also be saved in and automatically opened from local temporary files (on Windows, running a simple DOS *start* command with `os.system` would open the temporary file). In fact, popular email tools like Outlook use such schemes to support opening attachments. Python-coded email user interfaces could do so, too -- which is a hint about where this chapter is headed next.

11.4 The PyMailGui Email Client

As a finale for this chapter's email tools coverage, this section presents PyMailGui -- a Python/Tkinter program that implements a client-side email processing user interface. It is presented both as an instance of Python Internet scripting and as an example that ties together other tools we've already seen, such as threads and Tkinter GUIs.

Like the `pymail` program we wrote earlier, PyMailGui runs entirely on your local computer. Your email is fetched from and sent to remote mail servers over sockets, but the program and its user interface run locally. Because of that, PyMailGui is called an email client : it employs Python's client-side tools to talk to mail servers from the local machine. In fact, in some ways, PyMailGui builds on top of `pymail` to add a GUI. Unlike `pymail`, though, PyMailGui is a fairly full-featured user interface: email operations are performed with point-and-click operations.

11.4.1 Why PyMailGui?

Like many examples presented in this text, PyMailGui is also a practical, useful program. In fact I run it on all kinds of machines to check my email while traveling around the world teaching Python classes (it's another workaround for Telnet-challenged ISPs). Although PyMailGui won't put Microsoft Outlook out of business anytime soon, I like it for two reasons:

It's portable

PyMailGui runs on any machine with sockets and a Python with Tkinter installed. Because email is transferred with the Python libraries, any Internet connection will do. Moreover, because the user interface is coded with the Tkinter extension, PyMailGui should work, unchanged, on Windows, the X Windows system (Unix, Linux), and the Macintosh.

Microsoft Outlook is a more feature-rich package, but it has to be run on Windows, and more specifically, on a single Windows machine. Because it generally deletes email from server as it is downloaded and stores it on the client, you cannot run Outlook on multiple machines without spreading your email across all those machines. By contrast, PyMailGui saves and deletes email only on request, and so it is a bit more friendly to people who check their email in an ad-hoc fashion on arbitrary computers.

It's scriptable

PyMailGui can become anything you want it to be, because it is fully programmable. In fact, this is the real killer feature of PyMailGui and of open source software like Python in general -- because you have full access to PyMailGui's source code, you are in complete control of where it evolves from here. You have nowhere near as much control over commercial, closed products like Outlook; you generally get whatever a large company decided you need, along with whatever bugs that company might have introduced.

As a Python script, PyMailGui is a much more flexible tool. For instance, I can change its layout, disable features, and add completely new functionality quickly, by changing its Python source code. Don't like the mail list display? Change a few lines of code to customize it. Want to save and delete your mail automatically as it is loaded? Add some more code and buttons. Tired of seeing junk mail? Add a few lines of text-processing code to the load function to filter spam. These are just a few examples. The point is that because PyMailGui is written in a high-level, easy-to-maintain scripting language, such customizations are relatively simple, and might even be a lot of fun. ^[10]

[10] Example: I added code to pull the POP password from a local file instead of a pop-up in about 10 minutes, and less than 10 lines of code. Of course, I'm familiar with the code, but the wait time for new features in Outlook would be noticeably longer.

It's also worth mentioning that PyMailGui achieves this portability and scriptability, and implements a full-featured email interface along the way, in roughly 500 lines of program code. It doesn't have as many bells and whistles as commercial products, but the fact that it gets as close as it does in so few lines of code is a testament to the power of both the Python language and its libraries.

11.4.2 Running PyMailGui

Of course, to script PyMailGui on your own, you'll need to be able to run it. PyMailGui only requires a computer with some sort of Internet connectivity (a PC with a dialup account and modem will do) and an installed Python with the Tkinter extension enabled. The Windows port of Python has this capability, so Windows PC users should be able to run this program immediately by clicking its icon (the Windows port self-installer is on this book's CD (see <http://examples.oreilly.com/python2>) and also at <http://www.python.org>). You'll also want to change the file `mailconfig.py` in the email examples directory to reflect your account's parameters; more on this as we interact with the system.

11.4.3 Presentation Strategy

PyMailGui is easily one of the longest programs in this book (its main script is some 500 lines long, counting blank lines and comments), but it doesn't introduce many library interfaces that we haven't already seen in this book. For instance:

- The PyMailGui interface is built with Python's Tkinter, using the familiar listboxes, buttons, and text widgets we met earlier.
- Python's `rfc822` email header parser module is applied to pull out headers and text of messages.
- Python's POP and SMTP library modules are used to fetch, send, and delete mail over sockets.
- Python threads, if installed in your Python interpreter, are put to work to avoid blocking during long-running mail operations (loads, sends, deletions).

We're also going to reuse the `TextEditor` object we wrote in [Chapter 9](#), to view and compose

messages, the simple `pymail` module's tools we wrote earlier in this chapter to load and delete mail from the server, and the `mailconfig` module of this chapter to fetch email parameters. PyMailGui is largely an exercise in combining existing tools.

On the other hand, because this program is so long, we won't exhaustively document all of its code. Instead, we'll begin by describing how PyMailGui works from an end-user's perspective. After that, we'll list the system's new source code modules without any additional comments, for further study.

Like most longer case studies in this book, this section assumes that you already know enough Python to make sense of the code on your own. If you've been reading this book linearly, you should also know enough about Tkinter, threads, and mail interfaces to understand the library tools applied here. If you get stuck, you may wish to brush-up on the presentation of these topics earlier in the book.

Open Source Software and Camaros

An analogy might help underscore the importance of PyMailGui's scriptability. There are still a few of us who remember a time when it was completely normal for car owners to work on and repair their own automobiles. I still fondly remember huddling with friends under the hood of a 1970 Camaro in my youth, tweaking and customizing its engine. With a little work, we could make it as fast, flashy, and loud as we liked. Moreover, a breakdown in one of those older cars wasn't necessarily the end of the world. There was at least some chance that I could get the car going again on my own.

That's not quite true today. With the introduction of electronic controls and diabolically cramped engine compartments, car owners are usually better off taking their cars back to the dealer or other repair professional for all but the simplest kinds of changes. By and large, cars are no longer user-maintainable products. And if I have a breakdown in my shiny new Jeep, I'm probably going to be completely stuck until an authorized repairperson can get around to towing and fixing my ride.

I like to think of the closed and open software models in the same terms. When I use Microsoft Outlook, I'm stuck both with the feature set that a large company dictates, as well as any bugs that it may harbor. But with a programmable tool like PyMailGui, I can still get under the hood. I can add features, customize the system, and work my way out of any lurking bugs. And I can do so long before the next Outlook patch or release.

At the end of the day, open source software is about freedom. Users, not an arbitrarily far-removed company, have the final say. Not everyone wants to work on his or her own car, of course. On the other hand, software tends to fail much more often than cars, and Python scripting is considerably less greasy than auto mechanics.

11.4.4 Interacting with PyMailGui

To make this case study easier to understand, let's begin by seeing what PyMailGui actually does -- its user interaction and email processing functionality -- before jumping into the Python code that implements that behavior. As you read this part, feel free to jump ahead to the code listings that appear after the screen shots, but be sure to read this section, too; this is where I will explain all the subtleties of PyMailGui's design. After this section, you are invited to study the system's Python source code listings on your own for a better and more complete explanation than can be crafted in English.

11.4.4.1 Getting started

PyMailGui is a Python/Tkinter program, run by executing its top-level script file, `PyMailGui.py`. Like other Python programs, PyMailGui can be started from the system command line, by clicking on its filename icon in a file explorer interface, or by pressing its button in the PyDemo or PyGadgets launcher bars. However it is started, the first window PyMailGui presents is shown in [Figure 11-10](#).

Figure 11-10. PyMailGui main window start

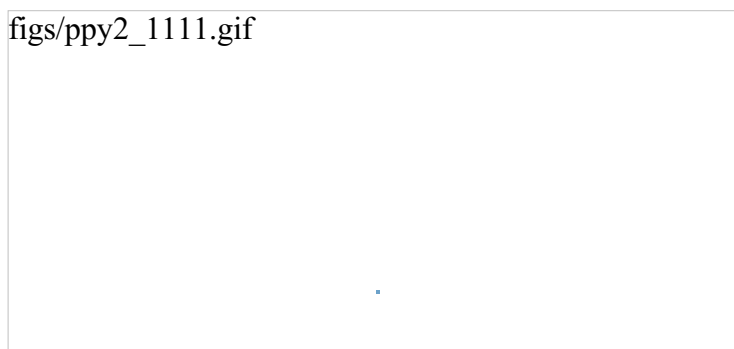


This is the PyMailGui main window -- every operation starts here. It consists of:

- A help button (the light blue bar at the top)
- A clickable email list area for fetched emails (the middle white section)
- A button bar at the bottom for processing messages selected in the list area

In normal operation, users load their email, select an email from the list area by clicking on it, and press a button at the bottom to process it. No mail messages are shown initially; we need to first load them, as we'll see in a moment. Before we do, though, let's press the blue help bar at the top to see what sort of help is available; [Figure 11-11](#) shows the help window pop-up that appears.

Figure 11-11. PyMailGui help pop-up





The main part of this window is simply a block of text in a scrolled-text widget, along with two buttons at the bottom. The entire help text is coded as a single triple-quoted string in the Python program. We could get more fancy and spawn a web browser to view HTML-formatted help, but simple text does the job here.^[11] The Cancel button makes this nonmodal (i.e., nonblocking) window go away; more interestingly, the Source button pops up a viewer window with the source code of PyMailGui's main script, as shown in [Figure 11-12](#).

[11] Actually, the help display started life even less fancy: it originally displayed help text in a standard information box pop-up, generated by the Tkinter `showinfo` call used earlier in the book. This worked fine on Windows (at least with a small amount of help text), but failed on Linux because of a default line-length limit in information pop-up boxes -- lines were broken so badly as to be illegible. The moral: if you're going to use `showinfo` and care about Linux, be sure to make your lines short and your text strings small.

Figure 11-12. PyMailGui source code viewer window



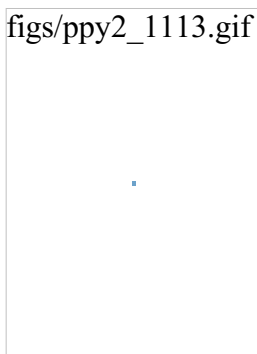
Not every program shows you its source code, but PyMailGui follows Python's open source motif. Politics aside, the main point of interest here is that this source viewer window is the same as PyMailGui's email viewer window. All the information here comes from PyMailGui internally; but this same window is used to display and edit mail shipped across the Net, so let's look at its format here:

- The top portion consists of a Cancel button to remove this nonmodal window, along with a section for displaying email header lines -- "From:", "To:", and so on.
- The bulk of this window is just another reuse of the `TextEditor` class object we wrote earlier in the book for the PyEdit program -- PyMailGui simply attaches an instance of `TextEditor` to every view and compose window, to get a full-featured text editor component for free. In fact, everything but the Cancel button and header lines on this window are implemented by `TextEditor`, not PyMailGui.

For instance, if we pick the Tools menu of the text portion of this window, and select its Info entry, we get the standard PyEdit `TextEditor` object's file text statistics box -- the exact same pop-up we'd get in the standalone PyEdit text editor, and the PyView image view programs we wrote in [Chapter 9](#) (see [Figure 11-13](#)).

In fact, this is the third reuse of `TextEditor` in this book: PyEdit, PyView, and now PyMailGui all present the same text editing interface to users, simply because they all use the same `TextEditor` object. For purposes of showing source code, we could also simply spawn the PyEdit program with the source file's name as a command-line argument (see PyEdit earlier in the text for more details). PyMailGui attaches an instance instead.

Figure 11-13. PyMailGui attached TextEditor info box



To display email, PyMailGui inserts its text into an attached `TextEditor` object; to compose mail, PyMailGui presents a `TextEditor` and later fetches all its text out to ship over the Net. Besides the obvious simplification here, this code reuse also makes it easy to pick up improvements and fixes -- any changes in the `TextEditor` object are automatically inherited by PyMailGui, PyView, and PyEdit.

11.4.4.2 Loading mail

Now, let's go back to the PyMailGui main window, and click the Load button to retrieve incoming email over the POP protocol. Like `pymail`, PyMailGui's load function gets account parameters from the `mailconfig` module listed in [Example 11-15](#), so be sure to change this file to reflect your email account parameters (i.e., server names and usernames) if you wish to use PyMailGui to read your own email.

The account password parameter merits a few extra words. In PyMailGui, it may come from one of two places:

Local file

If you put the name of a local file containing the password in the `mailconfig` module, PyMailGui loads the password from that file as needed.

Pop-up dialog

If you don't put a password filename in `mailconfig` (or PyMailGui can't load it from the file for whatever reason), PyMailGui will instead ask you for your password any time it is needed.

[Figure 11-1](#) shows the password input prompt you get if you haven't stored your password in a local file. Note that the password you type is not shown -- a `show='*'` option for the `Entry` field used in this pop-up tells Tkinter to echo typed characters as stars (this option is similar in spirit to both the `getpass` console input module we met earlier in this chapter, and an `HTML type=password` option we'll meet in a later chapter). Once entered, the password lives only in memory on your machine; PyMailGui itself doesn't store it anywhere in a permanent way.

Also notice that the local file password option requires you to store your password unencrypted in a file on the local client computer. This is convenient (you don't need to retype a password every time you check email), but not generally a good idea on a machine you share with others; leave this setting blank in `mailconfig` if you prefer to always enter your password in a pop-up.

Figure 11-14. PyMailGui password input dialog

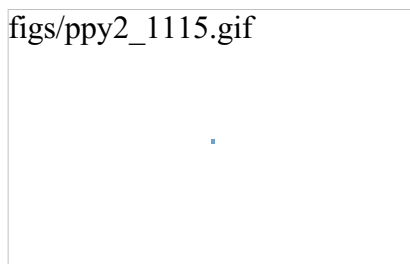


Once PyMailGui fetches your mail parameters and somehow obtains your password, it will next attempt to pull down all your incoming email from your POP server. PyMailGui reuses the load-mail tools in the `pymail` module listed in [Example 11-18](#), which in turn uses Python's standard `poplib` module to retrieve your email.

11.4.4.3 Threading long-running email transfers

Ultimately, though, the load function must download your email over a socket. If you get as much email as I do, this can take awhile. Rather than blocking the GUI while the load is in progress, PyMailGui spawns a thread to do the mail download operation in parallel with the rest of the program. The main program continues responding to window events (e.g., redrawing the display after another window has been moved over it) while your email is being downloaded. To let you know that a download is in progress in the background, PyMailGui pops up the wait dialog box shown in [Figure 11-15](#).

Figure 11-15. PyMailGui load mail wait box (thread running)



This dialog grabs focus and thus effectively disables the rest of the GUI's buttons while a download is in progress. It stays up for the duration of the download, and goes away automatically when the download is complete. Similar wait pop-ups appear during other long-running socket operations (email send and delete operations), but the GUI itself stays alive because the operations run in a thread.

On systems without threads, PyMailGui instead goes into a blocked state during such long-running operations (it stubs out the thread spawn operation to perform a simple function call). Because the GUI is essentially dead without threads, covering and uncovering the GUI during a mail load on such platforms will erase or otherwise distort its contents.^[12] Threads are enabled by default on most platforms that Python runs on (including Windows), so you probably won't see such oddness on your machine.

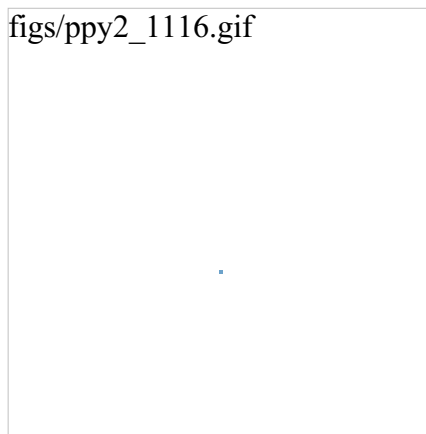
[12] If you want to see how this works, change PyMailGui's code such that the `fakeThread` class near the top of file `PyMailGui.py` is always defined (by default, it is created only if the import of the `thread` module fails), and try covering and uncovering the main window during a load, send, or delete operation. The window won't be redrawn because a single-threaded PyMailGui is busy talking over a socket.

One note here: as mentioned when we met the FTP GUIs earlier in this chapter, on MS Windows, only the thread that creates windows can process them. Because of that, PyMailGui takes care to not do anything related to the user interface within threads that load, send, or delete email. Instead, the main program continues responding to user interface events and updates, and watches for a global "I'm finished" flag to be set by the email transfer threads. Recall that threads share global (i.e., module) memory; since there is at most only two threads active in PyMailGui at once -- the main program and an email transfer thread -- a single global flag is all the cross-thread communication mechanism we need.

11.4.4.4 Load server interface

Because the load operation is really a socket operation, PyMailGui will automatically connect to your email server using whatever connectivity exists on the machine on which it is run. For instance, if you connect to the Net over a modem, and you're not already connected, Windows automatically pops up the standard connection dialog; [Figure 11-16](#) shows the one I get on my laptop. If PyMailGui runs on a machine with a dedicated Internet link, it uses that instead.

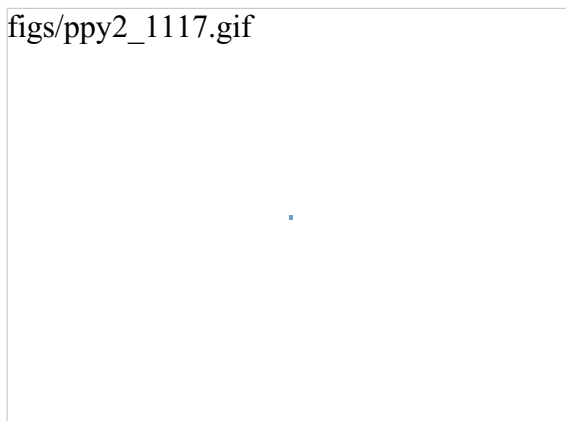
Figure 11-16. PyMailGui connection dialog (Windows)





After PyMailGui finishes loading your email, it populates the main window's list box with all of the messages on your email server, and scrolls to the most recently received. Figure 11-17 shows what the main windows looks like on my machine.

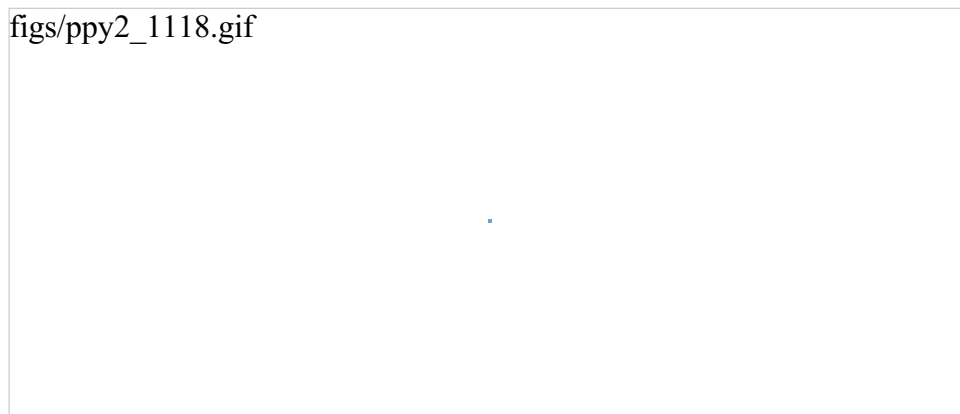
Figure 11-17. PyMailGui main window after load



Technically, the Load button fetches all your mail the first time it is pressed, but fetches only newly arrived email on later presses. PyMailGui keeps track of the last email loaded, and requests only higher email numbers on later loads. Already-loaded mail is kept in memory, in a Python list, to avoid the cost of downloading it again. Like the simple `pymail` command-line interface shown earlier, PyMailGui does not delete email from your server when it is loaded; if you really want to not see an email on a later load, you must explicitly delete it (more on this later).

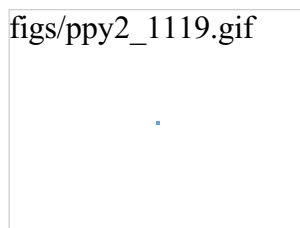
Like most GUIs in this book, the main window can be resized; [Figure 11-18](#) shows what it looks like when stretched to reveal more email details. Entries in the main list show just enough to give the user an idea of what the message contains -- each entry gives the concatenation of portions of the message's "Subject:", "From:", and "Date:" header lines, separated by | characters, and prefixed with the message's POP number (e.g., there are 91 emails in this list). The columns don't always line up neatly (some headers are shorter than others), but it's enough to hint at the message's contents.

Figure 11-18. PyMailGui main window resized



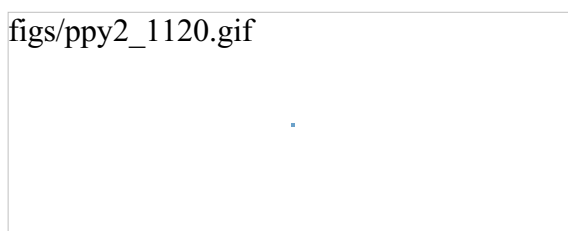
As we've seen, much magic happens when downloading email -- the client (the machine on which PyMailGui runs) must connect to the server (your email account machine) over a socket, and transfer bytes over arbitrary Internet links. If things go wrong, PyMailGui pops up standard error dialog boxes to let you know what happened. For example, if PyMailGui cannot establish a connection at all, you'll get a window like that shown in [Figure 11-19](#).

Figure 11-19. PyMailGui connection error box



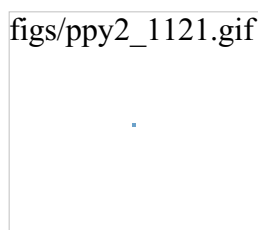
The details displayed here are just the Python exception type and exception data. If you typed an incorrect username or password for your account (in the `mailconfig` module or in the password pop-up), you'll see the message in [Figure 11-20](#).

Figure 11-20. PyMailGui invalid password error box



This box shows the exception raised by the Python `poplib` module. If PyMailGui cannot contact your server (e.g., it's down, or you listed its name wrong in `mailconfig`), you'll get the pop-up shown in [Figure 11-21](#).

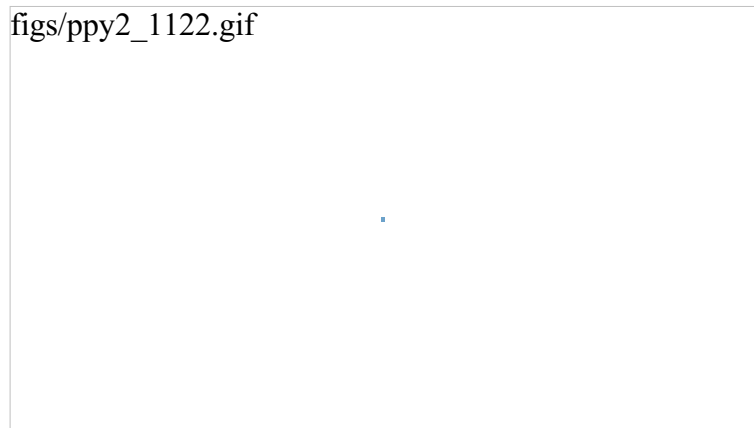
Figure 11-21. PyMailGui invalid or down server error box



11.4.4.5 Sending email

Once we've loaded email, we can process our messages with buttons on the main window. We can, however, send new emails at any time, even before a load. Pressing the Write button on the main window generates a mail composition window; one has been captured in [Figure 11-22](#).

Figure 11-22. PyMailGui write mail compose window



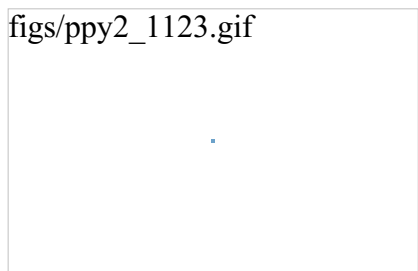
figs/ppy2_1122.gif

This window is just like the help source code viewer we saw a moment ago -- it has fields for entering header line details, and an attached `TextEditor` object for writing the body of the new email. For write operations, PyMailGui automatically fills the "From" line and inserts a signature text line (" -- Mark..."), from your `mailconfig` module settings. You can change these to any text you like, but the defaults are filled in automatically from your `mailconfig`.

There is also a new "Send" button here: when pressed, the text you typed into the the body of this window is mailed to the addresses you typed into the "To" and "Cc" lines, using Python's `smtplib` module. PyMailGui adds the header fields you type as mail header lines in the sent message. To send to more than one address, separate them with a ";" in the "To" and "Cc" lines (we'll see an example of this in a moment). In this mail, I fill in the "To" header with my own email address, to send the message to myself for illustration purposes.

As we've seen, `smtplib` ultimately sends bytes to a server over a socket. Since this can be a long running operation, PyMailGui delegates this operation to a spawned thread, too. While the send thread runs, the wait window in [Figure 11-23](#) appears, and the entire GUI stays alive; redraw and move events are handled in the main program thread, while the send thread talks to the SMTP server.

Figure 11-23. PyMailGui send mail wait box (thread running)

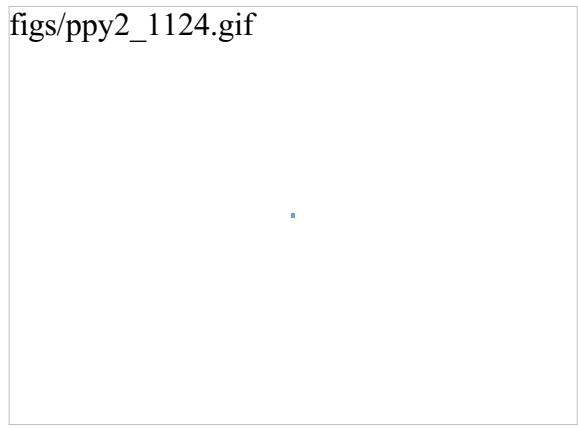


figs/ppy2_1123.gif

You'll get an error pop-up if Python cannot send a message to any of the target recipients, for any reason. If you don't get an error pop-up, everything worked correctly, and your mail will show up in the recipients' mailboxes on their email servers. Since I sent the message above to myself, it shows up in mine the next time I press the main window's Load button, as we see in [Figure 11-24](#).

Figure 11-24. PyMailGui main window after sends, load

figs/ppy2_1124.gif



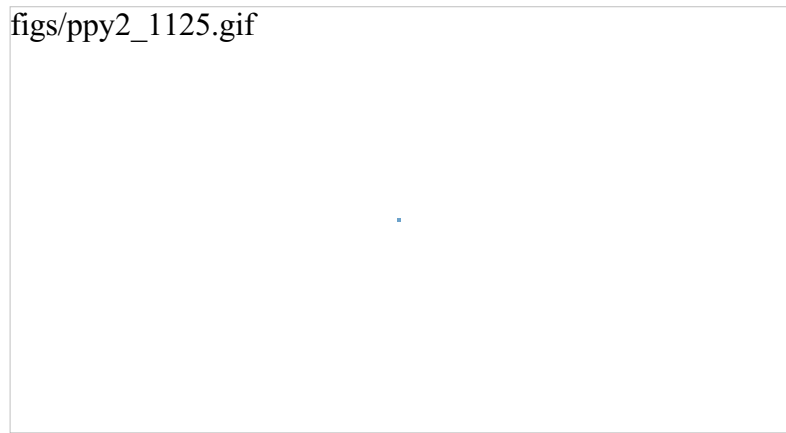
If you look back to the last main window shot, you'll notice that there are only two new emails now -- numbers 92 (from Python-Help) and 93 (the one I just wrote); PyMailGui is smart enough to download only the two new messages, and tack them onto the end of the loaded email list.

11.4.4.6 Viewing email

Now, let's view the mail message that was sent and received. PyMailGui lets us view email in formatted or raw modes. First, highlight (single-click) the mail you want to see in the main window, and press the View button. A formatted mail viewer window like that shown in [Figure 11-25](#) appears.

Figure 11-25. PyMailGui view incoming mail window

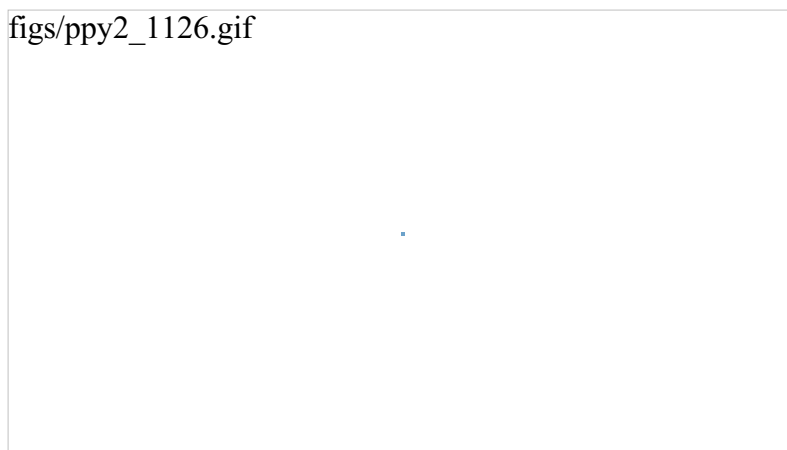
figs/ppy2_1125.gif



This is the exact same window we saw displaying source code earlier, only now all the information is filled in by extracting bits of the selected email message. Python's `rfc822` module is used to parse out header lines from the raw text of the email message; their text is placed into the fields in the top right of the window. After headers are parsed, the message's body text is left behind (in a `StringIO` file-like string wrapper), and is read and stuffed into a new `TextEditor` object for display (the white part in the middle of the window).

Besides the nicely formatted view window, PyMailGui also lets us see the raw text of a mail message. Double-click on a message's entry in the main window's list to bring up a simple unformatted display of the mail's text. The raw version of the mail I sent to myself is shown in [Figure 11-26](#).

Figure 11-26. PyMailGui raw mail text view window



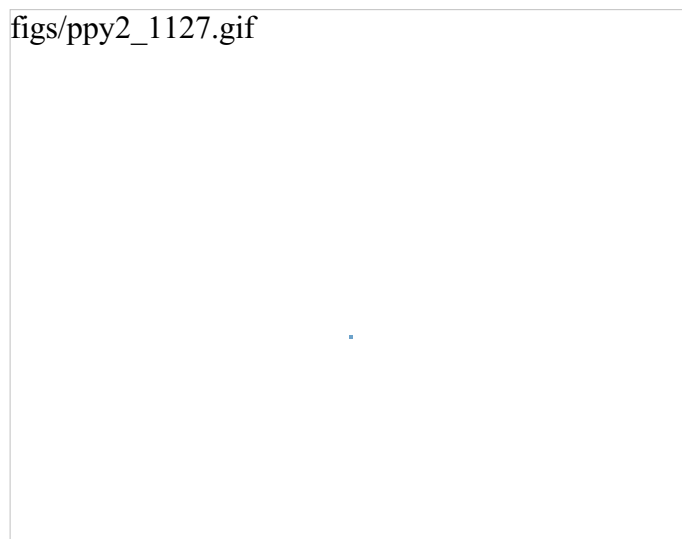
This raw text display can be useful to see special mail headers not shown in the formatted view. For instance, the optional "X-Mailer:" header in the raw text display identifies the program that transmitted a message; PyMailGui adds it automatically, along with standard headers like "From:" and "To:". Other headers are added as the mail is transmitted: the "Received:" headers name machines that the message was routed through on its way to our email server.

And really, the raw text form is all there is to an email message -- it's what is transferred from machine to machine when mail is sent. The nicely formatted display simply parses out pieces of the mail's raw text with standard Python tools and places them in associated fields of the display of [Figure 11-25](#).

11.4.4.7 Email replies and forwards

Besides allowing reading and writing email, PyMailGui also lets users forward and reply to incoming email sent from others. To reply to an email, select its entry in the main window's list and click the Reply button. If I reply to the mail I just sent to myself (arguably narcissistic, but demonstrative), the mail composition window shown in [Figure 11-27](#) appears.

Figure 11-27. PyMailGui reply compose window





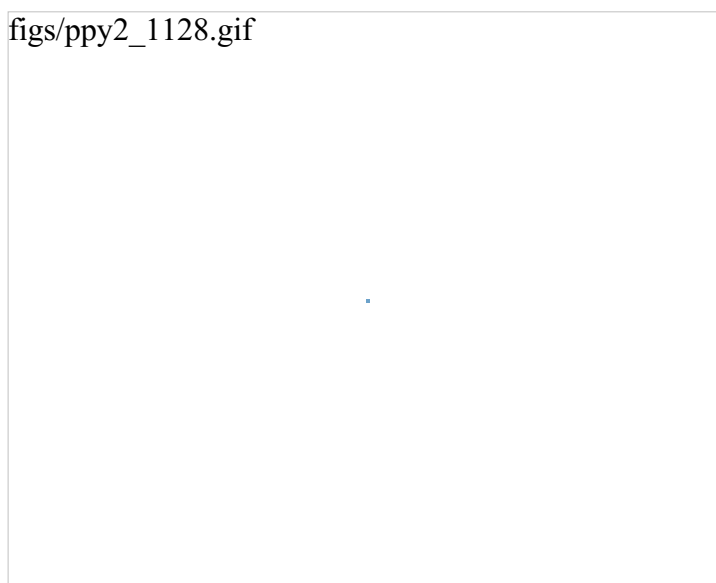
This window is identical in format to the one we saw for the "Write" operation, except that PyMailGui fills in some parts automatically:

- The "From" line is set to your email address in your `mailconfig` module.
- The "To" line is initialized to the original message's "From" address (we're replying to the original sender, after all). See the sidebar "More on Reply Addresses" for additional detail on the target address.
- The "Subject" line is set to the original message's subject line prepended with a "Re:", the standard follow-up subject line form.
- The body of the reply is initialized with the signature line in `mailconfig`, along with the original mail message's text. The original message text is quoted with > characters and prepended with a few header lines extracted from the original message to give some context.

Luckily, all of this is much easier than it may sound. Python's standard `rfc822` module is used to extract all the original message's header lines, and a single `string.replace` call does the work of adding the > quotes to the original message body. I simply type what I wish to say in reply (the initial paragraph in the mail's text area), and press the Send button to route the reply message to the mailbox on my mail server again. Physically sending the reply works the same as sending a brand new message -- the mail is routed to your SMTP server in a spawned send mail thread, and the send mail wait pop-up appears.

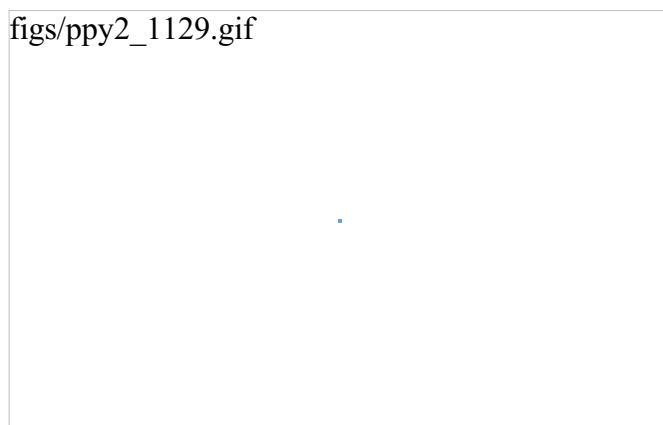
Forwarding a message is similar to replying: select the message in the main window, press the "Fwd" button, and fill in the fields and text area of the popped-up composition window. [Figure 11-28](#) shows the window created to forward the mail we originally wrote and received.

Figure 11-28. PyMailGui forward compose window



Much like replies, "From" is filled from `mailconfig`, the original text is automatically quoted in the message body again, and the subject line is preset to the original message's subject prepended with the string "Fwd:". I have to fill in the "To" line manually, though, because this is not a direct reply (it doesn't necessarily go back to the original sender). Notice that I'm forwarding this message to two different addresses; multiple recipient addresses are separated with a ";" character in "To" and "Cc" header fields. The Send button in this window fires the forward message off to all addresses listed in "To" and "Cc".

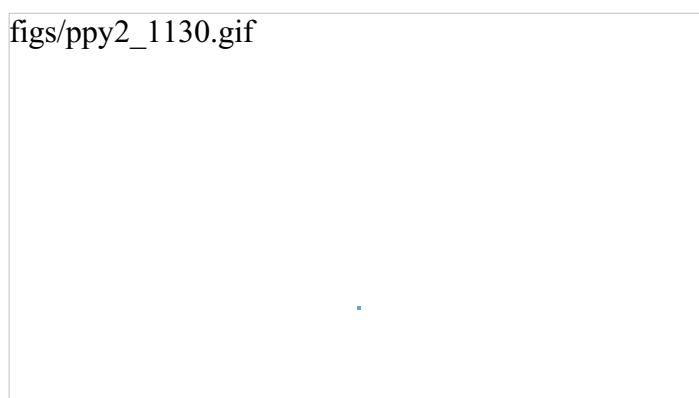
Figure 11-29. PyMailGui mail list after sends and load

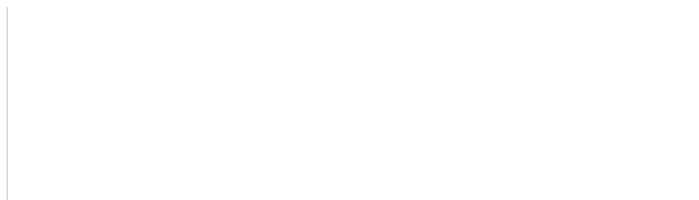


Okay, I've now written a new message, and replied to and forwarded it. The reply and forward were sent to my email address, too; if we press the main window's Load button again, the reply and forward messages should show up in the main window's list. In [Figure 11-29](#), they appear as messages 94 and 95.

Keep in mind that PyMailGui runs on the local computer, but the messages you see in the main window's list actually live in a mailbox on your email server machine. Every time we press Load PyMailGui downloads but does not delete newly arrived email from the server to your computer. The three messages we just wrote (93-95) will also appear in any other email program you use on your account (e.g., in Outlook). PyMailGui does not delete messages as they are downloaded, but simply stores them in your computer's memory for processing. If we now select message 95 and press View, we see the forward message we sent, as in [Figure 11-30](#). Really, this message went from my machine to a remote email server, and was downloaded from there into a Python list from which it is displayed.

Figure 11-30. PyMailGui view forwarded mail





[Figure 11-31](#) shows what the forward message's raw text looks like; again, double-click on a main window's entry to display this form. The formatted display in [Figure 11-30](#) simply extracts bits and pieces out of the text shown in the raw display form.

Figure 11-31. PyMailGui view forwarded mail, raw



11.4.4.8 Saving and deleting email

So far, we've covered everything except two of the main window's processing buttons and the All checkbox. PyMailGui lets us save mail messages in local text files, and delete messages from the server permanently, such that we won't see them the next time we access our account. Moreover we can save and delete a single mail at a time, or all mails displayed in the main windows list:

- To save one email, select it in the main window's list and press the Save button.
- To save all the emails in the list in one step, click the All checkbox at the bottom right corner of the main window and then press Save.

More on Reply Addresses

A subtle thing: technically, the "To" address in replies is made from whatever we get back from a standard library call of the form `hdrs.getaddr('From')` -- an `rfc822` module interface that parses and formats the original message sender's address automatically -- plus quotes added in a few rare cases.

Refer to function `onReplyMail` in the code listings. This library call returns a pair (full name, email address) parsed from the mail's "From:" header line. For instance, if a mail's first "From:" header contains the string:

```
'joe@spam.net (Joe Blow)'
```

then a call `hdrs.getaddr('From')` will yield the pair `('Joe Blow', 'joe@spam.net')`, with an empty name string if none exists in the original sender address string. If the header contains:

```
'Joe Blow <joe@spam.net>'
```

instead, the call yields the exact same result tuple.

Unfortunately, though, the Python 1.5.2 `rfc822` module had a bug that makes this call not always correct: the `getaddr` function yields bogus results if a full name part of the address contains a comma (e.g., "Blow, Joe"). This bug may be fixed in Python 2.0, but to work around it for earlier releases, PyMailGui puts the name part in explicit quotes if it contains a comma, before stuffing it into the target full-name `<email-address>` address used in the "To:" line of replies. For example, here are four typical "From" addresses and the reply "To" address PyMailGui generates for each (after the `=>`):

```
joe@spam.net => <joe@spam.net>  
Joe Blow <joe@spam.net> => Joe Blow <joe@spam.net>  
joe@spam.net (Joe Blow) => Joe Blow <joe@spam.net>  
"Blow, Joe" <joe@spam.net> => "Blow, Joe" <joe@spam.net>
```

Without the added quotes around the name in the last of these, the comma would confuse my SMTP server into seeing two recipients -- `Blow@rmi.net` and `Joe <joe@spam.net >` (the first incorrectly gets my ISP's domain name added because it is assumed to be a local user). The added quotes won't hurt if the bug is removed in later releases.

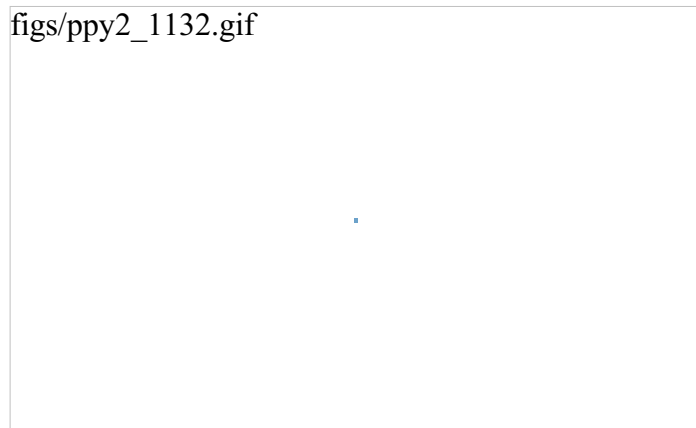
A less complex alternative solution (and one we'll use in a program called PyMailCgi later in this book) is to simply use the original "From" address exactly as the reply's "To". A library call of the form `hdrs.get('From')` would return the sender's address verbatim, quotes and all, without trying to parse out its components at all.

As coded, the PyMailGui reply address scheme works on every message I've ever replied to, but may need to be tweaked for some unique address formats or future Python releases. I've tested and used this program a lot, but much can happen on the Net, despite mail address standards. Officially speaking, any remaining bugs you find in it are really suggested exercises in disguise (at least I didn't say they were "features").

Delete operations are kicked off the same way, but press the Del button instead. In typical operation, I eventually delete email I'm not interested in, and save and delete emails that are important. Save operations write the raw text of one or more emails to a local text file you pick in the pop-up dialog shown in [Figure 11-32](#).

Figure 11-32. PyMailGui save mail dialog

figs/ppy2_1132.gif

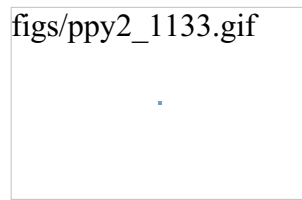


Technically, saves always append raw message text to the chosen file; the file is opened in 'a' mode, which creates the file if needed, and writes at its end. The save operation is also smart enough to remember the last directory you selected; the file dialog begins navigation there the next time you press Save.

Delete operations can also be applied to one or all messages. Unlike other operations, though, delete requests are simply queued up for later action. Messages are actually deleted from your mail server only as PyMailGui is exiting. For instance, if we've selected some messages for deletion and press the main window's Quit button, a standard verification dialog appears ([Figure 11-33](#)).

Figure 11-33. PyMailGui quit verification

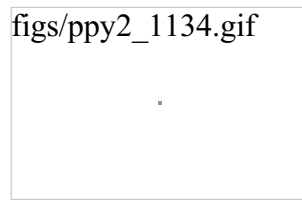
figs/ppy2_1133.gif



If we then verify the quit request, a second dialog appears ([Figure 11-34](#)), asking us to verify deletion of the queued up messages. If we press No here, no deletes happen, and PyMailGui silently exits. If we select Yes, PyMailGui spawns one last thread to send deletion requests to the email server for all the emails selected for deletion during the session. Another wait-state pop-up appears while the delete thread is running; when that thread is finished, PyMailGui exits as well.

Figure 11-34. PyMailGui delete verification on quit

figs/ppy2_1134.gif



By default and design, no mail is ever removed: you will see the same messages the next time PyMailGui runs. It deletes mail from your server only when you ask it to, deletes messages only on exit, and then only if verified in the last pop-up shown (this is your last chance to prevent permanent mail removal).

11.4.4.9 POP message numbers

This may seem a roundabout way to delete mail, but it accommodates a property of the POP interface. POP assigns each message a sequential number, starting from 1, and these numbers are passed to the server to fetch and delete messages. It's okay if new mail arrives while we're displaying the result of a prior download -- the new mail is assigned higher numbers, beyond what is displayed on the client. But if we delete a message in the middle of a mailbox, the numbers of all messages after the one deleted change (they are decremented by one). That means that some message numbers may be no longer valid if deletions are made while viewing previously loaded email (deleting by some number N may really delete message N+1!).


PyMailGui could adjust all the displayed numbers to work around this. To keep things simple, though, it postpones deletions instead. Notice that if you run multiple instances of PyMailGui at once, you shouldn't delete in one and then another because message numbers may become confused. You also may not be happy with the results of running something like Outlook at the same time as a PyMailGui session, but the net effect of such a combination depends on how another mail client handles deletions. In principle, PyMailGui could be extended to prevent other instances from running at the same time, but we leave that as an exercise.

11.4.4.10 Windows and status messages

Before we close this section, I want to point out that PyMailGui is really meant to be a multiple-window interface -- something not made obvious by the earlier screen shots. For example, [Figure 11-35](#) shows PyMailGui with a main list box, help, and three mail view windows. All these windows are nonmodal; that is, they are all active and independent, and do not block other windows from being selected. This interface all looks slightly different on Linux, but has the same functionality.

Figure 11-35. PyMailGui multiple windows and text editors

figs/ppy2_1135.gif



In general, you can have any number of mail view or compose windows up at once, and cut and paste between them. This matters, because PyMailGui must take care to make sure that each window has a distinct text editor object. If the text editor object was a global, or used globals internally, you'd likely see the same text in each window (and the send operations might wind up sending text from another window). To avoid this, PyMailGui creates and attaches a new `TextEditor` instance to each view and compose window it creates, and associates the new editor with the Send button's callback handler to make sure we get the right text.

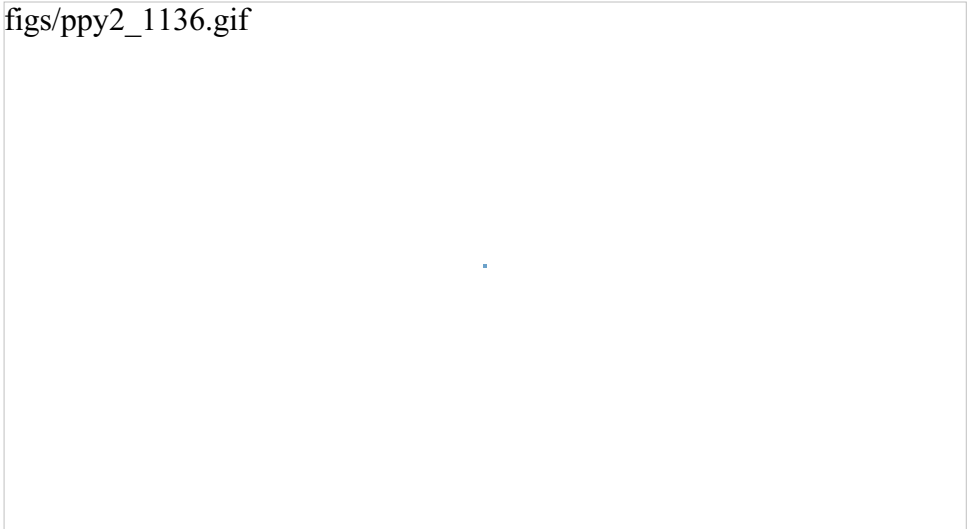
Finally, PyMailGui prints a variety of status messages as it runs, but you see them only if you launch the program from the system command line (e.g., a DOS box on Windows or an xterm on Linux), or by double-clicking on its filename icon (its main script is a `.py`, not a `.pyw`). On Windows, you won't see these messages when it is started from another program, such as the PyDemos or PyGadgets launcher bar GUIs. These status messages print server information, show mail loading status, and trace the load, store, and delete threads that are spawned along the way. If you want PyMailGui to be more verbose, launch it from a command line and watch:

```
C:\...\PP2E\Internet\Email>python PyMailGui.py
load start
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready <594100005a655e39@chevalier>
('+OK 5 messages (8470 octets)', ['1 709', '2 1136', '3 998', '4 2679',
'5 2948'], 38)
There are 5 mail messages in 8470 bytes
Retrieving: 1 2 3 4 5
load exit
thread exit caught
send start
Connecting to mail... ['<lutz@rmi.net>']
send exit
thread exit caught
```

You can also double-click on the `PyMailGui.py` filename in your file explorer GUI and monitor the popped-up DOS console box on Windows; [Figure 11-36](#) captures this window in action.

Figure 11-36. PyMailGui console status message window

figs/ppy2_1136.gif



PyMailGui status messages display the mail currently being downloaded (i.e., the "Retrieving:" lines are appended with a new mail number as each message is downloaded), and so give a more informative download status indicator than the wait pop-up window.

11.4.5 Implementing PyMailGui

Last but not least, we get to the code. There are really only two new modules here: one where the help text is stored and another that implements the system.

In fact, PyMailGui gets a lot of mileage out of reusing modules we wrote earlier and won't repeat here: `pymail` for mail load and delete operations, `mailconfig` for mail parameters, the GUI section's `TextEditor` for displaying and editing mail message text, and so on. In addition, standard Python modules used here such as `poplib`, `smtpplib`, and `rfc822` hide most of the details of pushing bytes around the Net and extracting message components. `Tkinter` implements GUI components in a portable fashion.

11.4.5.1 Help text module

The net effect of all this reuse is that PyMailGui implements a fairly feature-rich email program in roughly 500 lines of code, plus one support module. [Example 11-22](#) shows the support module -- used only to define the help text string, to avoid cluttering the main script file.

Example 11-22. PP2E\Internet\Email\PyMailGuiHelp.py

```
#####
# PyMailGui help text string, in this separate module only to avoid
# distracting from executable code. As coded, we throw up this text
# in a simple scrollable text box; in the future, we might instead
# use an HTML file opened under a web browser (e.g., run a "netscape
# help.html" or DOS "start help.html" command using os.system call
#####

# must be narrow for Linux info box popups;
# now uses scrolledtext with buttons instead;

helptext = """
PyMail, version 1.0
February, 2000
Programming Python, 2nd Edition
O'Reilly & Associates

Click main window buttons to process email:
- Load:\t fetch all (or newly arrived) POP mail from server
- View:\t display selected message nicely formatted
- Save:\t write selected (or all) emails to a chosen file
- Del:\t mark selected (or all) email to be deleted on exit
- Write:\t compose a new email message, send it by SMTP
- Reply:\t compose a reply to selected email, send it by SMTP
- Fwd:\t compose a forward of selected email, send by SMTP
- Quit:\t exit PyMail, delete any marked emails from server

Click an email in the main window's listbox to select it.
Click the "All" checkbox to apply Save or Del buttons to
all retrieved emails in the list. Double-click on an email
in the main window's listbox to view the mail's raw text,
including mail headers not shown by the View button.
```

Mail is removed from POP servers on exit only, and only mails marked for deletion with the Del button are removed, if and only if you verify the deletion in a confirmation popup.

Change the mailconfig.py module file on your own machine to reflect your email server names, user name, email address, and optional mail signature line added to all composed mails. Miscellaneous hints:

- Passwords are requested if needed, and not stored by PyMail.
- You may put your password in a file named in mailconfig.py.
- Use ';' between multiple addresses in "To" and "Cc" headers.
- Reply and Fwd automatically quote the original mail text.
- Save pops up a dialog for selecting a file to hold saved mails.
- Load only fetches newly-arrived email after the first load.

This client-side program currently requires Python and Tkinter. It uses Python threads, if installed, to avoid blocking the GUI. Sending and loading email requires an Internet connection.

```
if __name__ == '__main__':
    print helptext                # to stdout if run alone
    raw_input('Press Enter key')  # pause in DOS console popups
```

11.4.5.2 Main module

And finally, here is the main PyMailGui script -- the file run to start the system (see [Example 11.23](#)). I've already told what it does and why, so studying this listing's code and its comments for a deeper look is left as a suggested exercise. Python is so close to pseudocode already, that additional narrative here would probably be redundant.

Although I use this example on a daily basis as is, it is also prime for extension. For instance:

- Deleted messages could be marked as such graphically.
- Email attachments could be displayed, parsed out, decoded, and opened automatically when clicked using the Python multipart email extraction tools we met earlier in this chapter.
- Download status could be made more informative during mail load operations by updating a progress bar after each fetch (e.g., by periodically reconfiguring the size of a rectangle drawn on a popped-up canvas).
- Hyperlink URLs within messages could be highlighted visually and made to spawn a web browser automatically when clicked, by using the launcher tools we met in the GUI and system tools parts of this book.
- Because Internet newsgroup posts are similar in structure to emails (header lines plus body text: see the `ntplib` example in the next section), this script could in principle be extended to display both email messages and news articles. Classifying such a possible mutation as clever generalization or diabolical hack is left as an exercise in itself.
- This script still uses the nonstandard but usually harmless sending date format discussed in an earlier sidebar in this chapter; it would be trivial to import a conforming date format function from the `pymail` module.

- PyMailGui displays a wait dialog box during mail transfers that effectively disables the rest of the GUI. This is by design, to minimize timing complexity. In principle, though, the system could allow mail operation threads to overlap in time (e.g., allow the user to send new messages while a download is in progress). Since each transfer runs on a socket of its own, PyMailGui need not block other operations during transfers. This might be implemented with periodic Tkinter `after` events that check the status of transfers in progress. See the PyFtpGui scripts earlier in this chapter for an example of overlapping transfer threads.

And so on; because this software is open source, it is also necessarily open-ended. Suggested exercises in this category are delegated to your imagination.

Example 11-23. PP2E\Internet\Email\PyMailGui.py

```
#####
# PyMailGui 1.0 - A Python/Tkinter email client.
# Adds a Tkinter-based GUI interface to the pymail
# script's functionality. Works for POP/SMTP based
# email accounts using sockets on the machine on
# which this script is run. Uses threads if
# installed to run loads, sends, and deletes with
# no blocking; threads are standard on Windows.
# GUI updates done in main thread only (Windows).
# Reuses and attaches TextEditor class object.
# Run from command-line to see status messages.
# See use notes in help text in PyMailGuiHelp.py.
# To do: support attachments, shade deletions.
#####

# get services
import pymail, mailconfig
import rfc822, StringIO, string, sys
from Tkinter import *
from tkFileDialog import asksaveasfilename, SaveAs
from tkMessageBox import showinfo, showerror, askyesno
from PP2E.Gui.TextEditor.textEditor import TextEditorComponentMinimal

# run if no threads
try:
    import thread
except ImportError:
    class fakeThread:
        def start_new_thread(self, func, args):
            apply(func, args)
    thread = fakeThread()

# init global/module vars
msgList = [] # list of retrieved emails text
toDelete = [] # msgnums to be deleted on exit
listBox = None # main window's scrolled msg list
rootWin = None # the main window of this program
allModeVar = None # for All mode checkbox value
threadExitVar = 0 # used to signal child thread exit
debugme = 0 # enable extra status messages

mailserver = mailconfig.popservername # where to read pop email from
mailuser = mailconfig.popusername # smtp server in mailconfig too
mailpswd = None # pop passwd via file or popup here
#mailfile = mailconfig.savemailfile # from a file select dialog here
```



```
def fillIndex(msgList):
    # fill all of main listbox
    listBox.delete(0, END)
    count = 1
    for msg in msgList:
        hdrs = rfc822.Message(StringIO.StringIO(msg))
        msginfo = '%02d' % count
        for key in ('Subject', 'From', 'Date'):
            if hdrs.has_key(key): msginfo = msginfo + ' | ' + hdrs[key][:30]
        listBox.insert(END, msginfo)
        count = count+1
    listBox.see(END)          # show most recent mail=last line

def selectedMsg():
    # get msg selected in main listbox
    # print listBox.curselection()
    if listBox.curselection() == ():
        return 0                # empty tuple:no selection
    else:
        return eval(listBox.curselection()[0]) + 1  # else zero-based index
                                                # in a 1-item tuple of str

def waitForThreadExit(win):
    import time
    global threadExitVar        # in main thread, watch shared global var
    delay = 0.0                # 0.0=no sleep needed on Win98 (but hogs cpu)
    while not threadExitVar:
        win.update()           # dispatch any new GUI events during wait
        time.sleep(delay)      # if needed, sleep so other thread can run
    threadExitVar = 0          # at most one child thread active at once

def busyInfoBoxWait(message):
    # popup wait message box, wait for a thread exit
    # main gui event thread stays alive during wait
    # as coded returns only after thread has finished
    # popup.wait_variable(threadExitVar) may work too

    popup = Toplevel()
    popup.title('PyMail Wait')
    popup.protocol('WM_DELETE_WINDOW', lambda:0)        # ignore deletes
    label = Label(popup, text=message+'...')
    label.config(height=10, width=40, cursor='watch')  # busy cursor
    label.pack()
    popup.focus_set()                                  # grab application
    popup.grab_set()                                   # wait for thread exit
    waitForThreadExit(popup)                           # gui alive during wait
    print 'thread exit caught'
    popup.destroy()

def loadMailThread():
    # load mail while main thread handles gui events
    global msgList, errInfo, threadExitVar
    print 'load start'
    errInfo = ''
    try:
        nextnum = len(msgList) + 1
        newmail = py mail.loadmessages(mailserver, mailuser, mailpswd, nextnum)
        msgList = msgList + newmail
    except:
        exc_type, exc_value = sys.exc_info()[:2]        # thread exc
        errInfo = '\n' + str(exc_type) + '\n' + str(exc_value)
    print 'load exit'
```

```
threadExitVar = 1    # signal main thread

def onLoadMail():
    # load all (or new) pop email
    getpassword()
    thread.start_new_thread(loadMailThread, ())
    busyInfoBoxWait('Retrieving mail')
    if errInfo:
        global mailpswd                # zap pswd so can reinput
        mailpswd = None
        showerror('PyMail', 'Error loading mail\n' + errInfo)
    fillIndex(msgList)

def onViewRawMail():
    # view selected message - raw mail text with header lines
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = msgList[msgnum-1]        # put in ScrolledText
        from ScrolledText import ScrolledText
        window = Toplevel()
        window.title('PyMail raw message viewer #' + str(msgnum))
        browser = ScrolledText(window)
        browser.insert('0.0', text)
        browser.pack(expand=YES, fill=BOTH)

def onViewFormatMail():
    # view selected message - popup formatted display
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        mailtext = msgList[msgnum-1]    # put in a TextEditor form
        textfile = StringIO.StringIO(mailtext)
        headers = rfc822.Message(textfile) # strips header lines
        bodytext = textfile.read()      # rest is message body
        editmail('View #%d' % msgnum,
                 headers.get('From', '?'),
                 headers.get('To', '?'),
                 headers.get('Subject', '?'),
                 bodytext,
                 headers.get('Cc', '?'))

# use objects that retain prior directory for the next
# select, instead of simple asksaveasfilename() dialog

saveOneDialog = saveAllDialog = None

def myasksaveasfilename_one():
    global saveOneDialog
    if not saveOneDialog:
        saveOneDialog = SaveAs(title='PyMail Save File')
    return saveOneDialog.show()

def myasksaveasfilename_all():
    global saveAllDialog
    if not saveAllDialog:
        saveAllDialog = SaveAs(title='PyMail Save All File')
```

```
return saveAllDialog.show()

def onSaveMail():
    # save selected message in file
    if allModeVar.get():
        mailfile = myasksaveasfilename_all()
        if mailfile:
            try:
                # maybe this should be a thread
                for i in range(1, len(msgList)+1):
                    py mail.savemessage(i, mailfile, msgList)
            except:
                showerror('PyMail', 'Error during save')
        else:
            msgnum = selectedMsg()
            if not (1 <= msgnum <= len(msgList)):
                showerror('PyMail', 'No message selected')
            else:
                mailfile = myasksaveasfilename_one()
                if mailfile:
                    try:
                        py mail.savemessage(msgnum, mailfile, msgList)
                    except:
                        showerror('PyMail', 'Error during save')

def onDeleteMail():
    # mark selected message for deletion on exit
    global toDelete
    if allModeVar.get():
        toDelete = range(1, len(msgList)+1)
    else:
        msgnum = selectedMsg()
        if not (1 <= msgnum <= len(msgList)):
            showerror('PyMail', 'No message selected')
        elif msgnum not in toDelete:
            toDelete.append(msgnum) # fails if in list twice

def sendMailThread(From, To, Cc, Subj, text):
    # send mail while main thread handles gui events
    global errInfo, threadExitVar
    import smtplib, time
    from mailconfig import smtpservername
    print 'send start'

    date = time.ctime(time.time())
    Cchdr = (Cc and 'Cc: %s\n' % Cc) or ''
    hdrs = ('From: %s\nTo: %s\n%sDate: %s\nSubject: %s\n'
            % (From, To, Cchdr, date, Subj))
    hdrs = hdrs + 'X-Mailer: PyMailGui Version 1.0 (Python)\n'

    Ccs = (Cc and string.split(Cc, ';')) or [] # some servers reject ['']
    Tos = string.split(To, ';') + Ccs # cc: hdr line, and To list
    Tos = map(string.strip, Tos) # some addrs can have ','s
    print 'Connecting to mail...', Tos # strip spaces around addrs

    errInfo = ''
    failed = {} # smtplib may raise except
    try: # or return failed Tos dict
        server = smtplib.SMTP(smtpservername)
        failed = server.sendmail(From, Tos, hdrs + text)
        server.quit()
    except:
        exc_type, exc_value = sys.exc_info()[2] # thread exc
        excinfo = '\n' + str(exc_type) + '\n' + str(exc_value)
```

```
        errInfo = 'Error sending mail\n' + excinfo
    else:
        if failed: errInfo = 'Failed recipients:\n' + str(failed)

    print 'send exit'
    threadExitVar = 1                                # signal main thread

def sendMail(From, To, Cc, Subj, text):
    # send completed email
    thread.start_new_thread(sendMailThread, (From, To, Cc, Subj, text))
    busyInfoBoxWait('Sending mail')
    if errInfo:
        showerror('PyMail', errInfo)

def onWriteReplyFwdSend(window, editor, hdrs):
    # mail edit window send button press
    From, To, Cc, Subj = hdrs
    sendtext = editor.getAllText()
    sendMail(From.get(), To.get(), Cc.get(), Subj.get(), sendtext)
    if not errInfo:
        window.destroy()    # else keep to retry or save

def editmail(mode, From, To='', Subj='', origtext='', Cc=''):
    # create a new mail edit/view window
    win = Toplevel()
    win.title('PyMail - '+ mode)
    win.iconname('PyMail')
    viewOnly = (mode[:4] == 'View')

    # header entry fields
    frm = Frame(win); frm.pack( side=TOP,    fill=X)
    lfrm = Frame(frm); lfrm.pack(side=LEFT,  expand=NO,  fill=BOTH)
    mfrm = Frame(frm); mfrm.pack(side=LEFT,  expand=NO,  fill=NONE)
    rfrm = Frame(frm); rfrm.pack(side=RIGHT, expand=YES,  fill=BOTH)
    hdrs = []
    for (label, start) in [('From:', From),
                          ('To:', To),           # order matters on send
                          ('Cc:', Cc),
                          ('Subj:', Subj)]:
        lab = Label(mfrm, text=label, justify=LEFT)
        ent = Entry(rfrm)
        lab.pack(side=TOP, expand=YES, fill=X)
        ent.pack(side=TOP, expand=YES, fill=X)
        ent.insert('0', start)
        hdrs.append(ent)

    # send, cancel buttons (need new editor)
    editor = TextEditorComponentMinimal(win)
    sendit = (lambda w=win, e=editor, h=hdrs: onWriteReplyFwdSend(w, e, h))

    for (label, callback) in [('Cancel', win.destroy), ('Send', sendit)]:
        if not (viewOnly and label == 'Send'):
            b = Button(lfrm, text=label, command=callback)
            b.config(bg='beige', relief=RIDGE, bd=2)
            b.pack(side=TOP, expand=YES, fill=BOTH)

    # body text editor: pack last=clip first
    editor.pack(side=BOTTOM)                                # may be multiple editors
    if (not viewOnly) and mailconfig.mysignature:          # add auto signature text?
        origtext = ('\n%s\n' % mailconfig.mysignature) + origtext
```

```
editor.setAllText(origtext)

def onWriteMail():
    # compose new email
    editmail('Write', From=mailconfig.myaddress)

def quoteorigtext(msgnum):
    origtext = msgList[msgnum-1]
    textfile = StringIO.StringIO(origtext)
    headers = rfc822.Message(textfile)          # strips header lines
    bodytext = textfile.read()                 # rest is message body
    quoted = '\n-----Original Message-----\n'
    for hdr in ('From', 'To', 'Subject', 'Date'):
        quoted = quoted + ( '%s: %s\n' % (hdr, headers.get(hdr, '?')) )
    quoted = quoted + '\n' + bodytext
    quoted = '\n' + string.replace(quoted, '\n', '\n> ')
    return quoted

def onReplyMail():
    # reply to selected email
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = quoteorigtext(msgnum)
        hdrs = rfc822.Message(StringIO.StringIO(msgList[msgnum-1]))
        toname, toaddr = hdrs.getaddr('From')
        if toname and ',' in toname: toname = '"%s"' % toname
        To = '%s <%s>' % (toname, toaddr)
        From = mailconfig.myaddress or ('%s <%s>' % hdrs.getaddr('To'))
        Subj = 'Re: ' + hdrs.get('Subject', '(no subject)')
        editmail('Reply', From, To, Subj, text)

def onFwdMail():
    # forward selected email
    msgnum = selectedMsg()
    if not (1 <= msgnum <= len(msgList)):
        showerror('PyMail', 'No message selected')
    else:
        text = quoteorigtext(msgnum)
        hdrs = rfc822.Message(StringIO.StringIO(msgList[msgnum-1]))
        From = mailconfig.myaddress or ('%s <%s>' % hdrs.getaddr('To'))
        Subj = 'Fwd: ' + hdrs.get('Subject', '(no subject)')
        editmail('Forward', From, '', Subj, text)

def deleteMailThread(toDelete):
    # delete mail while main thread handles gui events
    global errInfo, threadExitVar
    print 'delete start'
    try:
        pymail.deletemessages(mailserver, mailuser, mailpswd, toDelete, 0)
    except:
        exc_type, exc_value = sys.exc_info()[1:]
        errInfo = '\n' + str(exc_type) + '\n' + str(exc_value)
    else:
        errInfo = ''
    print 'delete exit'
    threadExitVar = 1 # signal main thread

def onQuitMail():
    # exit mail tool, delete now
```

```
if askyesno('PyMail', 'Verify Quit?'):
    if toDelete and askyesno('PyMail', 'Really Delete Mail?'):
        getpassword()
        thread.start_new_thread(deleteMailThread, (toDelete,))
        busyInfoBoxWait('Deleting mail')
        if errInfo:
            showerror('PyMail', 'Error while deleting:\n' + errInfo)
        else:
            showinfo('PyMail', 'Mail deleted from server')
    rootWin.quit()

def askpassword(prompt, app='PyMail'):    # getpass.getpass uses stdin, not GUI
    win = Toplevel()                    # tkSimpleDialog.askstring echos input
    win.title(app + ' Prompt')
    Label(win, text=prompt).pack(side=LEFT)
    entvar = StringVar()
    ent = Entry(win, textvariable=entvar, show='*')
    ent.pack(side=RIGHT, expand=YES, fill=X)
    ent.bind('<Return>', lambda event, savewin=win: savewin.destroy())
    ent.focus_set(); win.grab_set(); win.wait_window()
    win.update()
    return entvar.get()                # ent widget is now gone

def getpassword():
    # unless known, set global pop password
    # from client-side file or popup dialog
    global mailpswd
    if mailpswd:
        return
    else:
        try:
            localfile = open(mailconfig.poppasswdfile)
            mailpswd = localfile.readline()[:-1]
            if debugme: print 'local file password', repr(mailpswd)
        except:
            prompt = 'Password for %s on %s?' % (mailuser, mailserver)
            mailpswd = askpassword(prompt)
            if debugme: print 'user input password', repr(mailpswd)

def decorate(rootWin):
    # window manager stuff for main window
    rootWin.title('PyMail 1.0')
    rootWin.iconname('PyMail')
    rootWin.protocol('WM_DELETE_WINDOW', onQuitMail)

def makemainwindow(parent=None):
    # make the main window
    global rootWin, listBox, allModeVar
    if parent:
        rootWin = Frame(parent)          # attach to a parent
        rootWin.pack(expand=YES, fill=BOTH)
    else:
        rootWin = Tk()                  # assume I'm standalone
        decorate(rootWin)

# add main buttons at bottom
frame1 = Frame(rootWin)
frame1.pack(side=BOTTOM, fill=X)
allModeVar = IntVar()
Checkbutton(frame1, text="All", variable=allModeVar).pack(side=RIGHT)
actions = [ ('Load', onLoadMail), ('View', onViewFormatMail),
            ('Save', onSaveMail), ('Del', onDeleteMail),
            ('Write', onWriteMail), ('Reply', onReplyMail),
```

```
        ('Fwd', onFwdMail), ('Quit', onQuitMail) ]
for (title, callback) in actions:
    Button(frame1, text=title, command=callback).pack(side=LEFT, fill=X)

# add main listbox and scrollbar
frame2 = Frame(rootWin)
vscroll = Scrollbar(frame2)
fontsz = (sys.platform[:3] == 'win' and 8) or 10
listBox = Listbox(frame2, bg='white', font=('courier', fontsz))

# crosslink listbox and scrollbar
vscroll.config(command=listBox.yview, relief=SUNKEN)
listBox.config(yscrollcommand=vscroll.set, relief=SUNKEN, selectmode=SINGLE)
listBox.bind('<Double-1>', lambda event: onViewRawMail())
frame2.pack(side=TOP, expand=YES, fill=BOTH)
vscroll.pack(side=RIGHT, fill=BOTH)
listBox.pack(side=LEFT, expand=YES, fill=BOTH)
return rootWin

# load text block string
from PyMailGuiHelp import helptext

def showhelp(helptext=helptext, appname='PyMail'): # show helptext in
    from ScrolledText import ScrolledText        # a non-modal dialog
    new = Toplevel()                              # make new popup window
    bar = Frame(new)                              # pack first=clip last
    bar.pack(side=BOTTOM, fill=X)
    code = Button(bar, bg='beige', text="Source", command=showsourc)
    quit = Button(bar, bg='beige', text="Cancel", command=new.destroy)
    code.pack(pady=1, side=LEFT)
    quit.pack(pady=1, side=LEFT)
    text = ScrolledText(new)                     # add Text + scrollbar
    text.config(font='system', width=70)         # too big for showinfo
    text.config(bg='steelblue', fg='white')      # erase on btn or return
    text.insert('0.0', helptext)
    text.pack(expand=YES, fill=BOTH)
    new.title(appname + " Help")
    new.bind("<Return>", (lambda event, new=new: new.destroy()))

def showsourc():
    # tricky, but open
    try:
        source = open('PyMailGui.py').read()    # like web getfile.cgi
                                                # in cwd or below it?
    except:
        try:
            import os                            # or use find.find(f)[0],
            from PP2E.Launcher import findFirst # $PP2EHOME, guessLocation
            here = os.curdir                     # or spawn pyedit with arg
            source = open(findFirst(here, 'PyMailGui.py')).read()
        except:
            source = 'Sorry - cannot find my source file'
    subject = 'Main script [see also: PyMailGuiHelp, pmail, mailconfig]'
    editmail('View Source Code', 'PyMailGui', 'User', subject, source)

def container():
    # use attachment to add help button
    # this is a bit easier with classes
    root = Tk()
    title = Button(root, text='PyMail - a Python/Tkinter email client')
    title.config(bg='steelblue', fg='white', relief=RIDGE)
    title.config(command=showhelp)
    title.pack(fill=X)
```

```
    decorate(root)
    return root

if __name__ == '__main__':
    # run stand-alone or attach
    rootWin = makemainwindow(container()) # or makemainwindow()
    rootWin.mainloop()
```

I l@ve RuBoard

11.5 Other Client-Side Tools

So far in this chapter, we have focused on Python's FTP and email processing tools and have met a handful of client-side scripting modules along the way: `ftplib`, `poplib`, `smtpplib`, `mhlib`, `mimetools`, `urllib`, `rfc822`, and so on. This set is representative of Python's library tools for transferring and processing information over the Internet, but it's not at all complete. A more or less comprehensive list of Python's Internet-related modules appears at the start of the previous chapter. Among other things, Python also includes client-side support libraries for Internet news, Telnet, HTTP, and other standard protocols.

11.5.1 NNTP: Accessing Newsgroups

Python's `nntplib` module supports the client-side interface to NNTP -- the Network News Transfer Protocol -- which is used for reading and posting articles to Usenet newsgroups in the Internet. Like other protocols, NNTP runs on top of sockets and merely defines a standard message protocol; like other modules, `nntplib` hides most of the protocol details and presents an object-based interface to Python scripts.

We won't get into protocol details here, but in brief, NNTP servers store a range of articles on the server machine, usually in a flat-file database. If you have the domain or IP name of a server machine that runs an NNTP server program listening on the NNTP port, you can write scripts that fetch or post articles from any machine that has Python and an Internet connection. For instance, the script in [Example 11-24](#) by default fetches and displays the last 10 articles from Python's Internet news group, `comp.lang.python`, from the `news.rmi.net` NNTP server at my ISP.

Example 11-24. PP2E\Internet\Other\readnews.py

```
#####
# fetch and print usenet newsgroup postings
# from comp.lang.python via the nntplib module
# which really runs on top of sockets; nntplib
# also supports posting new messages, etc.;
# note: posts not deleted after they are read;
#####

listonly = 0
showhdrs = ['From', 'Subject', 'Date', 'Newsgroups', 'Lines']
try:
    import sys
    servername, groupname, showcount = sys.argv[1:]
    showcount = int(showcount)
except:
    servername = 'news.rmi.net'
    groupname = 'comp.lang.python'          # cmd line args or defaults
    showcount = 10                          # show last showcount posts

# connect to nntp server
print 'Connecting to', servername, 'for', groupname
from nntplib import NNTP
connection = NNTP(servername)
(reply, count, first, last, name) = connection.group(groupname)
print '%s has %s articles: %s-%s' % (name, count, first, last)

# get request headers only
```

```
fetchfrom = str(int(last) - (showcount-1))
(reply, subjects) = connection.xhdr('subject', (fetchfrom + '-' + last))

# show headers, get message hdr+body
for (id, subj) in subjects:                                # [-showcount:] if fetch all hdrs
    print 'Article %s [%s]' % (id, subj)
    if not listonly and raw_input('=> Display?') in ['y', 'Y']:
        reply, num, tid, list = connection.head(id)
        for line in list:
            for prefix in showhdrs:
                if line[:len(prefix)] == prefix:
                    print line[:80]; break
            if raw_input('=> Show body?') in ['y', 'Y']:
                reply, num, tid, list = connection.body(id)
                for line in list:
                    print line[:80]
        print
print connection.quit()
```

As for FTP and email tools, the script creates an NNTP object and calls its methods to fetch newsgroup information and articles' header and body text. The `xhdr` method, for example, loads selected headers from a range of messages. When run, this program connects to the server and displays each article's subject line, pausing to ask whether it should fetch and show the article's header information lines (headers listed in variable `showhdrs` only) and body text:

```
C:\...\PP2E\Internet\Other>python readnews.py
Connecting to news.rmi.net for comp.lang.python
comp.lang.python has 3376 articles: 30054-33447
Article 33438 [Embedding? file_input and eval_input]
=> Display?

Article 33439 [Embedding? file_input and eval_input]
=> Display?y
From: James Spears <jimsp@ichips.intel.com>
Newsgroups: comp.lang.python
Subject: Embedding? file_input and eval_input
Date: Fri, 11 Aug 2000 10:55:39 -0700
Lines: 34
=> Show body?

Article 33440 [Embedding? file_input and eval_input]
=> Display?

Article 33441 [Embedding? file_input and eval_input]
=> Display?

Article 33442 [Embedding? file_input and eval_input]
=> Display?

Article 33443 [Re: PYHTONPATH]
=> Display?y
Subject: Re: PYHTONPATH
Lines: 13
From: sp00fd <sp00fdNOspSPAM@yahoo.com.invalid>
Newsgroups: comp.lang.python
Date: Fri, 11 Aug 2000 11:06:23 -0700
=> Show body?y
```

Is this not what you were looking for?

```
Add to cgi script:
import sys
sys.path.insert(0, "/path/to/dir")
import yourmodule
```

```
-----
Got questions? Get answers over the phone at Keen.com.
Up to 100 minutes free!
http://www.keen.com
```

```
Article 33444 [Loading new code...]
=> Display?
```

```
Article 33445 [Re: PYHTONPATH]
=> Display?
```

```
Article 33446 [Re: Compile snags on AIX & IRIX]
=> Display?
```

```
Article 33447 [RE: string.replace() can't replace newline characters???]
=> Display?
```

205 GoodBye

We can also pass this script an explicit server name, newsgroup, and display count on the command line to apply it in different ways. Here is this Python script checking the last few messages in Perl and Linux newsgroups:

```
C:\...\PP2E\Internet\Other>python readnews.py news.rmi.net comp.lang.perl.misc
Connecting to news.rmi.net for comp.lang.perl.misc
comp.lang.perl.misc has 5839 articles: 75543-81512
Article 81508 [Re: Simple Argument Passing Question]
=> Display?
```

```
Article 81509 [Re: How to Access a hash value?]
=> Display?
```

```
Article 81510 [Re: London =?iso-8859-1?Q?=A330-35K?= Perl Programmers Required]
=> Display?
```

```
Article 81511 [Re: ODBC question]
=> Display?
```

```
Article 81512 [Re: ODBC question]
=> Display?
```

205 GoodBye

```
C:\...\PP2E\Internet\Other>python readnews.py news.rmi.net comp.os.linux 4
Connecting to news.rmi.net for comp.os.linux
comp.os.linux has 526 articles: 9015-9606
Article 9603 [Re: Simple question about CD-Writing for Linux]
=> Display?
```

```
Article 9604 [Re: How to start the ftp?]
=> Display?
```

```
Article 9605 [Re: large file support]
=> Display?
```

```
Article 9606 [Re: large file support]
=> Display?y
```

```
From: andy@physast.uga.edu (Andreas Schweitzer)
Newsgroups: comp.os.linux.questions,comp.os.linux.admin,comp.os.linux
Subject: Re: large file support
Date: 11 Aug 2000 18:32:12 GMT
Lines: 19
=> Show body?n
```

205 GoodBye

With a little more work, we could turn this script into a full-blown news interface. For instance, new articles could be posted from within a Python script with code of this form (assuming the local file already contains proper NNTP header lines):

```
# to post, say this (but only if you really want to post!)
connection = NNTP(servername)
localfile = open('filename')      # file has proper headers
connection.post(localfile)        # send text to newsgroup
connection.quit()
```

We might also add a Tkinter-based GUI frontend to this script to make it more usable, but we'll leave such an extension on the suggested exercise heap (see also the PyMailGui interface's suggested extensions in the previous section).

11.5.2 HTTP: Accessing Web Sites

Python's standard library (that is, modules that are installed with the interpreter) also includes client-side support for HTTP -- the Hypertext Transfer Protocol -- a message structure and port standard used to transfer information on the World Wide Web. In short, this is the protocol that your web browser (e.g., Internet Explorer, Netscape) uses to fetch web pages and run applications on remote servers as you surf the Net. At the bottom, it's just bytes sent over port 80

To really understand HTTP-style transfers, you need to know some of the server-side scripting topics covered in the next three chapters (e.g., script invocations and Internet address schemes), so this section may be less useful to readers with no such background. Luckily, though, the basic HTTP interfaces in Python are simple enough for a cursory understanding even at this point in the book, so let's take a brief look here.

Python's standard `httplib` module automates much of the protocol defined by HTTP and allows scripts to fetch web pages much like web browsers. For instance, the script in [Example 11-25](#) can be used to grab any file from any server machine running an HTTP web server program. As usual, the file (and descriptive header lines) is ultimately transferred over a standard socket port, but most of the complexity is hidden by the `httplib` module.

Example 11-25. PP2E\Internet\Other\http-getfile.py

```
#####
# fetch a file from an http (web) server over sockets via httplib;
# the filename param may have a full directory path, and may name a cgi
# script with query parameters on the end to invoke a remote program;
# fetched file data or remote program output could be saved to a local
# file to mimic ftp, or parsed with string.find or the htmllib module;
#####

import sys, httplib
showlines = 6
try:
    servername, filename = sys.argv[1:]          # cmdline args?
```

```
except:
    servername, filename = 'starship.python.net', '/index.html'

print servername, filename
server = httplib.HTTP(servername)           # connect to http site/server
server.putrequest('GET', filename)          # send request and headers
server.putheader('Accept', 'text/html')     # POST requests work here too
server.endheaders()                         # as do cgi script file names

errcode, errmsh, replyheader = server.getreply() # read reply info headers
if errcode != 200:                           # 200 means success
    print 'Error sending request', errcode
else:
    file = server.getfile()                  # file obj for data received
    data = file.readlines()
    file.close()                             # show lines with eoln at end
    for line in data[:showlines]: print line, # to save, write data to file
```

Desired server names and filenames can be passed on the command line to override hardcoded defaults in the script. You need to also know something of the HTTP protocol to make the most sense of this code, but it's fairly straightforward to decipher. When run on the client, this script makes a HTTP object to connect to the server, sends it a GET request along with acceptable reply types, and then reads the server's reply. Much like raw email message text, the HTTP server's reply usually begins with a set of descriptive header lines, followed by the contents of the requested file. The HTTP object's `getfile` method gives us a file object from which we can read the downloaded data.

Let's fetch a few files with this script. Like all Python client-side scripts, this one works on any machine with Python and an Internet connection (here it runs on a Windows client). Assuming that all goes well, the first few lines of the downloaded file are printed; in a more realistic application, the text we fetch would probably be saved to a local file, parsed with Python's `htmlplib` module, and so on. Without arguments, the script simply fetches the HTML index page at <http://starship.python.org>:

```
C:\...\PP2E\Internet\Other>python http-getfile.py
starship.python.net /index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel
```

But we can also list a server and file to be fetched on the command line, if we want to be more specific. In the following code, we use the script to fetch files from two different web sites by listing their names on the command lines (I've added line breaks to make these lines fit in this book). Notice that the filename argument can include an arbitrary remote directory path to the desired file, as in the last fetch here:

```
C:\...\PP2E\Internet\Other>python http-getfile.py
www.python.org /index.html
www.python.org /index.html
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED. DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:

C:\...\PP2E\Internet\Other>python http-getfile.py www.python.org /index
```

```
www.python.org /index
Error sending request 404

C:\...\PP2E\Internet\Other>python http-getfile.py starship.python.net
                               /~lutz/index.html

starship.python.net /~lutz/index.html
<HTML>
<HEAD><TITLE>Mark Lutz's Starship page</TITLE></HEAD>
<BODY>

<H1>Greetings</H1>
```

Also notice the second attempt in this code: if the request fails, the script receives and displays a HTTP error code from the server (we forgot the .html on the filename). With the raw HTTP interfaces, we need to be precise about what we want.

Technically, the string we call `filename` in the script can refer to either a simple static web page file, or a server-side program that generates HTML as its output. Those server-side programs are usually called CGI scripts -- the topic of the next three chapters. For now, keep in mind that when `filename` refers to a script, this program can be used to invoke another program that resides on a remote server machine. In that case, we can also specify parameters (called a query string) to be passed to the remote program after a `?`. Here, for instance, we pass a `language=Python` parameter to a CGI script we will meet in the next chapter:

```
C:\...\PP2E\Internet\Other>python http-getfile.py starship.python.net
                               /~lutz/Basics/languages.cgi?language=Python

starship.python.net /~lutz/Basics/languages.cgi?language=Python
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Python</H3><P><PRE>
  print 'Hello World'
</PRE></P><BR>
<HR>
```

This book has much more to say about HTML, CGI scripts, and the meaning of an HTTP GET request (one way to format information sent to a HTTP server) later, so we'll skip additional details here. Suffice it to say, though, that we could use the HTTP interfaces to write our own web browsers and build scripts that use web sites as though they were subroutines. By sending parameters to remote programs and parsing their results, web sites can take on the role of simple in-process functions (albeit, much more slowly and indirectly).

11.5.2.1 urllib revisited

The `httplib` module we just met provides low-level control for HTTP clients. When dealing with items available on the Web, though, it's often easier to code downloads with Python's standard `urllib` module introduced in the FTP section of this chapter. Since this module is another way to talk HTTP, let's expand on its interfaces here.

Recall that given a URL, `urllib` either downloads the requested object over the Net to a local file, or gives us a file-like object from which we can read the requested object's contents. Because of that, the script in [Example 11-26](#) does the same work as the `httplib` script we just wrote, but requires noticeably less typing.

Example 11-26. PP2E\Internet\Other\http-getfile-urllib1.py

```
#####
# fetch a file from an http (web) server over sockets via urllib;
# urllib supports http, ftp, files, etc. via url address strings;
# for http, the url can name a file or trigger a remote cgi script;
# see also the urllib example in the ftp section, and the cgi
# script invocation in a later chapter; files can be fetched over
# the net with Python in many ways that vary in complexity and
# server requirements: sockets, ftp, http, urllib, cgi outputs;
# caveat: should run urllib.quote on filename--see later chapters;
#####

import sys, urllib
showlines = 6
try:
    servername, filename = sys.argv[1:]          # cmdline args?
except:
    servername, filename = 'starship.python.net', '/index.html'

remoteaddr = 'http://%s%s' % (servername, filename) # can name a cgi script to
print remoteaddr
remotefile = urllib.urlopen(remoteaddr)          # returns input file objec
remotedata = remotefile.readlines()             # read data directly here
remotefile.close()
for line in remotedata[:showlines]: print line,
```

Almost all HTTP transfer details are hidden behind the `urllib` interface here. This version works about the same as the `httpplib` version we wrote first, but builds and submits an Internet URL address to get its work done (the constructed URL is printed as the script's first output line). As we saw in the FTP section of this chapter, the `urllib` `urlopen` function returns a file-like object from which we can read the remote data. But because the constructed URLs begin with "http://" here, the `urllib` module automatically employs the lower-level HTTP interfaces to download the requested file, not FTP:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py
http://starship.python.net/index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel

C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py www.python.org /index
http://www.python.org/index
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED. DO NOT EDIT. -->
<!-- Fri Mar 3 10:28:30 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:

C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py starship.python.net
/~lutz/index.html
http://starship.python.net/~lutz/index.html
<HTML>
<HEAD><TITLE>Mark Lutz's Starship page</TITLE></HEAD>
<BODY>

<H1>Greetings</H1>

C:\...\PP2E\Internet\Other>python http-getfile-urllib1.py starship.python.net
/~lutz/Basics/languages.cgi?language=Java
```

```
http://starship.python.net/~lutz/Basics/languages.cgi?language=Java
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Java</H3><P><PRE>
  System.out.println("Hello World");
</PRE></P><BR>
<HR>
```

As before, the filename argument can name a simple file or a program invocation with optional parameters at the end. If you read this output carefully, you'll notice that this script still works if you leave the .html off the end of a filename (in the second command line); unlike the raw HTTP version, the URL-based interface is smart enough to do the right thing.

11.5.2.2 Other urllib interfaces

One last mutation: the following `urllib` downloader script uses the slightly higher-level `urlretrieve` interface in that module to automatically save the downloaded file or script output to a local file on the client machine. This interface is handy if we really mean to store the fetched data (e.g., to mimic the FTP protocol). If we plan on processing the downloaded data immediately, though, this form may be less convenient than the version we just met: we need to open and read the saved file. Moreover, we need to provide extra protocol for specifying or extracting a local filename, as in [Example 11-27](#).

Example 11-27. PP2E\Internet\Other\http-getfile-urllib2.py

```
#####
# fetch a file from an http (web) server over sockets via urlllib;
# this version uses an interface that saves the fetched data to a
# local file; the local file name is either passed in as a cmdline
# arg or stripped from the url with urlparse: the filename argument
# may have a directory path at the front and query parmams at end,
# so os.path.split is not enough (only splits off directory path);
# caveat: should run urllib.quote on filename--see later chapters;
#####

import sys, os, urllib, urlparse
showlines = 6
try:
    servername, filename = sys.argv[1:3]          # first 2 cmdline args?
except:
    servername, filename = 'starship.python.net', '/index.html'

remoteaddr = 'http://%s%s' % (servername, filename) # any address on the net
if len(sys.argv) == 4:                               # get result file name
    localname = sys.argv[3]
else:
    (scheme, server, path, parms, query, frag) = urlparse.urlparse(remoteaddr)
    localname = os.path.split(path)[1]

print remoteaddr, localname
urllib.urlretrieve(remoteaddr, localname)           # can be file or script
remotedata = open(localname).readlines()           # saved to local file
for line in remotedata[:showlines]: print line,
```

Let's run this last variant from a command line. Its basic operation is the same as the last two versions: like the prior one, it builds a URL, and like both of the last two, we can list an explicit target server and file path on the command line:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py
```



```
http://starship.python.net/index.html index.html
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen">
  <TITLE>Starship Python</TITLE>
  <SCRIPT language="JavaScript">
<!-- // mask from the infidel

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py
                               www.python.org /index.html
http://www.python.org/index.html index.html
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:
```

Because this version uses an `urllib` interface that automatically saves the downloaded data in a local file, it's more directly like FTP downloads in spirit. But this script must also somehow come up with a local filename for storing the data. You can either let the script strip and use the base filename from the constructed URL, or explicitly pass a local filename as a last command-line argument. In the prior run, for instance, the downloaded web page is stored in local file `index.html` -- the base filename stripped from the URL (the script prints the URL and local filename as its first output line). In the next run, the local filename is passed explicitly as `python org-index.html`:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py www.python.org
                               /index.html python-org-index.html
http://www.python.org/index.html python-org-index.html
<HTML>
<!-- THIS PAGE IS AUTOMATICALLY GENERATED.  DO NOT EDIT. -->
<!-- Wed Aug 23 17:29:24 2000 -->
<!-- USING HT2HTML 1.1 -->
<!-- SEE http://www.python.org/~bwarsaw/software/pyware.html -->
<!-- User-specified headers:

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                               /~lutz/home/index.html
http://starship.python.net/~lutz/home/index.html index.html
<HTML>

<HEAD>
<TITLE>Mark Lutz's Home Page</TITLE>
</HEAD>

C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
                               /~lutz/home/about-pp.html
http://starship.python.net/~lutz/home/about-pp.html about-pp.html
<HTML>

<HEAD>
<TITLE>About "Programming Python"</TITLE>
</HEAD>
```

Below is a listing showing this third version being used to trigger a remote program. As before, if you don't give the local filename explicitly, the script strips the base filename out of the filename argument. That's not always easy or appropriate for program invocations -- the filename can contain both a remote directory path at the front, and query parameters at the end for a remote program invocation.

Given a script invocation URL and no explicit output filename, the script extracts the base filename in the middle by using first the standard `urlparse` module to pull out the file path, and then `os.path.split` to strip off the directory path. However, the resulting filename is a remote script's name, and may or may not be an appropriate place to store the data locally. In the first run below, for example, the script's output goes in a local file called `languages.cgi`, the script name in the middle of the URL; in the second, we name the output `CxxSyntax.html` explicitly instead to suppress filename extraction:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
/~lutz/Basics/languages.cgi?language=Perl
http://starship.python.net/~lutz/Basics/languages.cgi?language=Perl
languages.cgi

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Perl</H3><P><PRE>
  print "Hello World\n";
</PRE></P><BR>
<HR>
```

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
/~lutz/Basics/languages.cgi?language=C++ CxxSyntax.html
http://starship.python.net/~lutz/Basics/languages.cgi?language=C++
CxxSyntax.html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>C </H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

The remote script returns a not-found message when passed "C++" in the last command here. It turns out that "+" is a special character in URL strings (meaning a space), and to be robust, both of the `urllib` scripts we've just written should really run the `filename` string through something called `urllib.quote`, a tool that escapes special characters for transmission. We will talk about this in depth in the next chapter, so consider this all a preview for now. But to make this invocation work, we need to use special sequences in the constructed URL; here's how to do it by hand:

```
C:\...\PP2E\Internet\Other>python http-getfile-urllib2.py starship.python.net
/~lutz/Basics/languages.cgi?language=C%2b%2b CxxSyntax.html
http://starship.python.net/~lutz/Basics/languages.cgi?language=C%2b%2b
CxxSyntax.html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>C++</H3><P><PRE>
  cout &lt;&lt; "Hello World" &lt;&lt; endl;
</PRE></P><BR>
<HR>
```

The odd "%2b" strings in this command line are not entirely magical: the escaping required for URLs can be seen by running standard Python tools manually (this is what these scripts should do automatically to handle all possible cases well):

```
C:\...\PP2E\Internet\Other>python
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import urllib
>>> urllib.quote('C++')
'C%2b%2b'
```

Again, don't work too hard at understanding these last few commands; we will revisit URLs and URL escapes in the next chapter, while exploring server-side scripting in Python. I will also explain there why the C++ result came back with other oddities like `<<` -- HTML escapes for `<<`.

11.5.3 Other Client-Side Scripting Options

In this chapter, we've focused on client-side interfaces to standard protocols that run over sockets, but client-side programming can take other forms, too. For instance, in [Chapter 15](#) we'll also see that Python code can be embedded inside the HTML code that defines a web page, with the Windows Active Scripting extension. When Internet Explorer downloads such a web page file from a web server, the embedded Python scripts are actually executed on the client machine, with an object API that gives access to the browser's context. Code in HTML is downloaded over a socket initially, but its execution is not bound up with a socket-based protocol.

In [Chapter 15](#), we'll also meet client-side options such as the JPython (a.k.a. "Jython") system, a compiler that supports Python-coded Java *applets* -- general-purpose programs downloaded from a server and run locally on the client when accessed or referenced by a URL. We'll also peek at Python tools for processing *XML* -- structured text that may become a common language of client/server dialogs in the future.

In deference to time and space, though, we won't go into further details on these and other client side tools here. If you are interested in using Python to script clients, you should take a few minutes to become familiar with the list of Internet tools documented in the Python library reference manual. All work on similar principles, but have slightly distinct interfaces.

In the next chapter, we'll hop the fence to the other side of the Internet world and explore scripts that run on server machines. Such programs give rise to the grander notion of applications that live entirely on the Web and are launched by web browsers. As we take this leap in structure, keep in mind that the tools we met in this and the previous chapter are often sufficient to implement all the distributed processing that many applications require, and they can work in harmony with scripts that run on a server. To completely understand the web world view, though, we need to explore the server realm, too.

Chapter 11. Client-Side Scripting

[Section 11.1. "Socket to Me!"](#)

[Section 11.2. Transferring Files over the Net](#)

[Section 11.3. Processing Internet Email](#)

[Section 11.4. The PyMailGui Email Client](#)

[Section 11.5. Other Client-Side Tools](#)

12.1 "Oh What a Tangled Web We Weave"

This chapter is the third part of our look at Python Internet programming. In the last two chapters, we explored sockets and basic client-side programming interfaces such as FTP and email. In this chapter, our main focus will be on writing server-side scripts in Python -- a type of program usually referred to as CGI scripts. Server-side scripting and its derivatives are at the heart of much of what happens on the Web these days.

As we'll see, Python makes an ideal language for writing scripts to implement and customize web sites, due to both its ease of use and its library support. In the following two chapters, we will use the basics we learn in this chapter to implement full-blown web sites. After that, we will wrap up with a chapter that looks at other Internet-related topics and technologies. Here, our goal is to understand the fundamentals of server-side scripting, before exploring systems that build upon that basic model.

A House upon the Sand

As you read the next three chapters of this book, please keep in mind that they are intended only as an introduction to server-side scripting with Python. The webmaster domain is large and complex, changes continuously, and often prescribes many ways to accomplish a given goal -- some of which can vary from browser to browser and server to server. For instance, the password encryption scheme of the next chapter may be unnecessary under certain scenarios, and special HTML tags may sometimes obviate some work we'll do here.

Given such a large and shifting knowledge base, this part of the book does not even pretend to be a complete look at the server-side scripting domain. To become truly proficient in this area, you should study other texts for additional webmaster-y details and tricks (e.g., O'Reilly's *HTML & XHTML: The Definitive Guide*). Here, you will meet Python's CGI toolset and learn enough to start writing substantial web sites of your own in Python. But you should not take this text as the final word on the subject.

12.2 What's a Server-Side CGI Script?

Simply put, CGI scripts implement much of the interaction you typically experience on the Web. They are a standard and widely used mechanism for programming web site interaction. There are other ways to add interactive behavior to web sites with Python, including client-side solutions (e.g., JPython applets and Active Scripting), as well as server-side technologies, which build upon the basic CGI model (e.g., Active Server Pages and Zope), and we will discuss these briefly at the end of [Chapter 15](#), too. But by and large, CGI server-side scripts are used to program much of the activity on the Web.

12.2.1 The Script Behind the Curtain

Formally speaking, CGI scripts are programs that run on a server machine and adhere to the Common Gateway Interface -- a model for browser/server communications, from which CGI scripts take their name. Perhaps a more useful way to understand CGI, though, is in terms of the interaction it implies.

Most people take this interaction for granted when browsing the Web and pressing buttons in web pages, but there is a lot going on behind the scenes of every transaction on the Web. From the perspective of a user, it's a fairly familiar and simple process:

1. **Submission.** When you visit a web site to purchase a product or submit information online, you generally fill in a form in your web browser, press a button to submit your information, and begin waiting for a reply.
2. **Response.** Assuming all is well with both your Internet connection and the computer you are contacting, you eventually get a reply in the form of a new web page. It may be a simple acknowledgement (e.g., "Thanks for your order") or a new form that must be filled out and submitted again.

And, believe it or not, that simple model is what makes most of the Web hum. But internally, it's a bit more complex. In fact, there is a subtle client/server socket-based architecture at work -- your web browser running on your computer is the client, and the computer you contact over the Web is the server. Let's examine the interaction scenario again, with all the gory details that users usually never see.

Submission

When you fill out a form page in a web browser and press a submission button, behind the scenes your web browser sends your information across the Internet to the server machine specified as its receiver. The server machine is usually a remote computer that lives somewhere else in both cyberspace and reality. It is named in the URL you access (the Internet address string that appears at the top of your browser). The target server and file can be named in a URL you type explicitly, but more typically they are specified in the HTML that defines the submission page itself -- either in a hyperlink, or in the "action" tag of a form's HTML. However the server is specified, the browser running on your computer ultimately sends your information to the server as bytes over a socket, using techniques we saw in the last two chapters. On the server machine, a program called an HTTP server runs *perpetually*, listening on a socket for incoming data

from browsers, usually on port number 80.

Processing

When your information shows up at the server machine, the HTTP server program notices it first and decides how to handle the request. If the requested URL names a simple web page (e.g., a URL ending in `.html`), the HTTP server opens the named HTML file on the server machine and sends its text back to the browser over a socket. On the client, the browser reads the HTML and uses it to construct the next page you see. But if the URL requested by the browser names an executable program instead (e.g., a URL ending in `.cgi`), the HTTP server starts the named program on the server machine to process the request and redirects the incoming browser data to the spawned program's `stdin` input stream and environment variables. That program is usually a CGI script -- a program run on the remote server machine somewhere in cyberspace, not on your computer. The CGI script is responsible for handling the request from this point on; it may store your information in a database, charge your credit card, and so on.

Response

Ultimately, the CGI script prints HTML to generate a new response page in your browser. When a CGI script is started, the HTTP server takes care to connect the script's `stdout` standard output stream to a socket that the browser is listening to. Because of this, HTML code printed by the CGI script is sent over the Internet, back to your browser, to produce a new page. The HTML printed back by the CGI script works just as if it had been stored and read in from an HTML file; it can define a simple response page or a brand new form coded to collect additional information.

In other words, CGI scripts are something like callback handlers for requests generated by web browsers that require a program to be run dynamically; they are automatically run on the server machine in response to actions in a browser. Although CGI scripts ultimately receive and send standard structured messages over sockets, CGI is more like a higher-level procedural convention for sending and receiving information between a browser and a server.

12.2.2 Writing CGI Scripts in Python

If all of the above sounds complicated, relax -- Python, as well as the resident HTTP server, automates most of the tricky bits. CGI scripts are written as fairly autonomous programs, and they assume that startup tasks have already been accomplished. The HTTP web server program, not the CGI script, implements the server-side of the HTTP protocol itself. Moreover, Python's library modules automatically dissect information sent up from the browser and give it to the CGI script in an easily digested form. The upshot is that CGI scripts may focus on application details like processing input data and producing a result page.

As mentioned earlier, in the context of CGI scripts, the `stdin` and `stdout` streams are automatically tied to sockets connected to the browser. In addition, the HTTP server passes some browser information to the CGI script in the form of shell environment variables. To CGI programmers, that means:

- Input data sent from the browser to the server shows up as a stream of bytes in the `stdin` input stream, along with shell environment variables.

- Output is sent back from the server to the client by simply printing properly formatted HTML to the `stdout` output stream.

The most complex parts of this scheme include parsing all the input information sent up from the browser and formatting information in the reply sent back. Happily, Python's standard library largely automates both tasks:

Input

With the Python `cgi` module, inputs typed into a web browser form or appended to a URL string show up as values in a dictionary-like object in Python CGI scripts. Python parses the data itself and gives us an object with one `key:value` pair per input sent by the browser that is fully independent of transmission style (form or URL).

Output

The `cgi` module also has tools for automatically escaping strings so that they are legal to use in HTML (e.g., replacing embedded `<`, `>`, and `&` characters with HTML escape codes). Module `urllib` provides other tools for formatting text inserted into generated URL strings (e.g., adding `%XX` and `+` escapes).

We'll study both of these interfaces in detail later in this chapter. For now, keep in mind that although any language can be used to write CGI scripts, Python's standard modules and language attributes make it a snap.

Less happily, CGI scripts are also intimately tied to the syntax of HTML, since they must generate it to create a reply page. In fact, it can be said that Python CGI scripts embed HTML, which is an entirely distinct language in its own right. As we'll also see, the fact that CGI scripts create a user interface by printing HTML syntax means that we have to take special care with the text we insert into a web page's code (e.g., escaping HTML operators). Worse, CGI scripts require at least a cursory knowledge of HTML forms, since that is where the inputs and target script's address are typically specified. This book won't teach HTML in-depth; if you find yourself puzzled by some of the arcane syntax of the HTML generated by scripts here, you should glance at an HTML introduction, such as O'Reilly's *HTML and XHTML: The Definitive Guide*.

12.2.3 Running Server-Side Examples

Like GUIs, web-based systems are highly interactive, and the best way to get a feel for some of these examples is to test-drive them live. Before we get into some code, it's worth noting that all you need to run the examples in the next few chapters is a web browser. That is, all the Web examples we will see here can be run from any web browser on any machine, whether you've installed Python on that machine or not. Simply type this URL at the top:^[1]

[1] Given that this edition may not be updated for many years, it's not impossible that the server name in this address `starship.python.net` might change over time. If this address fails, check the book updates at <http://rmi.net/~lutz/about-pp.html> to see if a new examples site address has been posted. The rest of the main page's URL will likely be unchanged. Note, though, that some examples hardcode the *starship* host server name in URLs; these will be fixed on the new server if

moved, but not on your book CD. Run script *fixsitename.py* later in this chapter to change site names automatically.

`http://starship.python.net/~lutz/PyInternetDemos.html`

That address loads a launcher page with links to all the example files installed on a server machine whose domain name is `starship.python.net` (a machine dedicated to Python developers). The launcher page itself appears as shown in [Figure 12-1](#), running under Internet Explorer. It looks similar in other browsers. Each major example has a link on this page, which runs when clicked.

Figure 12-1. The PyInternetDemos launcher page



The launcher page, and all the HTML files in this chapter, can also be loaded locally, from the book's example distribution directory on your machine. They can even be opened directly off the book's CD (view CD-ROM content online at <http://examples.oreilly.com/python2>) and may be opened by buttons on the top-level book demo launchers. However, the CGI scripts ultimately invoked by some of the example links must be run on a server, and thus require a live Internet connection. If you browse root pages locally on your machine, your browser will either display the scripts' source code or tell you when you need to connect to the Web to run a CGI script. On Windows, a connection dialog will likely pop up automatically, if needed.

12.2.3.1 Changing server-side examples

Of course, running scripts in your browser isn't quite the same as writing scripts on your own. If you do decide to change these CGI programs or write new ones from scratch, you must be able to access web server machines:

- To change server-side scripts, you need an account on a web server machine with an installed version of Python. A basic account on such a server is often enough. Then edit scripts on your machine and upload to the server by FTP.
- To type explicit command lines on a server machine or edit scripts on the server directly, you will need to also have shell access on the web server. Such access lets you telnet to that machine to get a command-line prompt.

Unlike the last chapter's examples, Python server-side scripts require both Python and a server. That is, you'll need access to a web server machine that supports CGI scripts in general and that either already has an installed Python interpreter or lets you install one of your own. Some Internet Service Providers (ISPs) are more supportive than others on this front, but there are many options here, both commercial and free (more on this later).

Once you've located a server to host your scripts, you may modify and upload the CGI source code file from this book's CD to your own server and site by FTP. If you do, you may also want to run two Python command-line scripts on your server after uploading: *fixcgi.py* and *fixsitename.py*, both presented later in this chapter. The former sets CGI script permissions, and the latter replaces any *starship* server name references in example links and forms with your own server's name. We'll study additional installation details later in this chapter, and explore a few custom server options at the end of [Chapter 15](#).

12.2.3.2 Viewing server-side examples and output

The source code of examples in this part of the book is listed in the text and included on the book's CD (see <http://examples.oreilly.com/python2>). In all cases, if you wish to view the source code of an HTML file, or the HTML generated by a Python CGI script, you can also simply select your browser's View Source menu option while the corresponding web page is displayed.

Keep in mind, though, that your browser's View Source option lets you see the output of a server-side script after it has run, but not the source code of the script itself. There is no automatic way to view the Python source code of the CGI scripts themselves, short of finding them in this book or its CD.

To address this issue, later in this chapter we'll also write a CGI-based program called `getfile`, which allows the source code of any file on this book's web site (HTML, CGI script, etc.) to be downloaded and viewed. Simply type the desired file's name into a web page form referenced by the `getfile.html` link on the Internet demos launcher page, or add it to the end of an explicitly typed URL as a parameter like this:

```
http://.../getfile.cgi?filename=somefile.cgi
```

In response, the server will ship back the text of the named file to your browser. This process requires explicit interface steps, though, and much more knowledge than we've gained thus far, so see ahead for details.

12.3 Climbing the CGI Learning Curve

Okay, it's time to get into concrete programming details. This section introduces CGI coding one step at a time -- from simple, noninteractive scripts to larger programs that utilize all the common web page user input devices (what we called "widgets" in the Tkinter GUI chapters of [Part II](#)). We'll move slowly at first, to learn all the basics; the next two chapters will use the ideas presented here to build up larger and more realistic web site examples. For now, let's work through a simple CGI tutorial, with just enough HTML thrown in to write basic server-side scripts.

12.3.1 A First Web Page

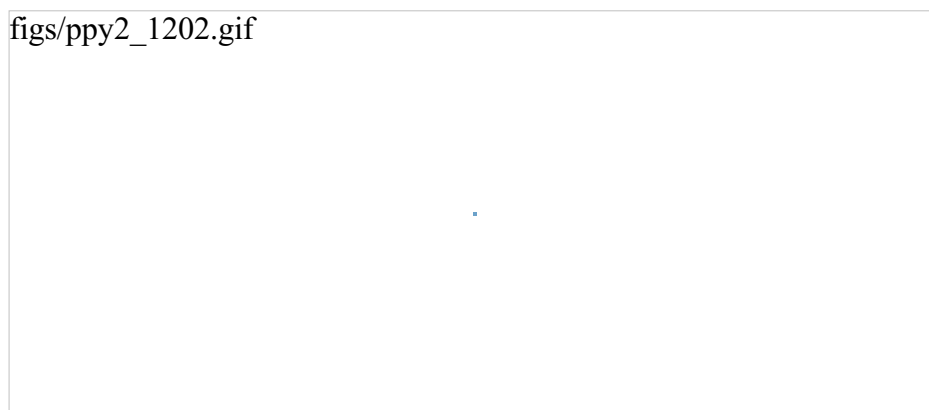
As mentioned, CGI scripts are intimately bound up with HTML, so let's start with a simple HTML page. The file *test0.html*, shown in [Example 12-1](#), defines a bona fide, fully functional web page -- a text file containing HTML code, which specifies the structure and contents of a simple web page.

Example 12-1. PP2E\Internet\Cgi-Web\Basics\test0.html

```
<HTML><BODY>
<TITLE>HTML 101</TITLE>
<H1>A First HTML page</H1>
<P>Hello, HTML World!</P>
</BODY></HTML>
```

If you point your favorite web browser to the Internet address of this file (or to its local path on your own machine), you should see a page like that shown in [Figure 12-2](#). This figure shows the Internet Explorer browser at work; other browsers render the page similarly.

Figure 12-2. A simple web page from an HTML file



To truly understand how this little file does its work, you need to know something about permission rules, HTML syntax, and Internet addresses. Let's take a quick first look at each of these topics before we move on to larger examples.

12.3.1.1 HTML file permission constraints

First of all, if you want to install this code on a different machine, it's usually necessary to grant web page files and their directories world-readable permission. That's because they are loaded by arbitrary people over the Web (actually, by someone named "nobody", who we'll introduce in a moment). An appropriate *chmod* command can be used to change permissions on Unix-like machines. For instance, a `chmod 755 filename` shell command usually suffices; it makes *filename* readable and executable by everyone, and writable by you only.^[2] These directory and file permission details are typical, but they can vary from server to server. Be sure to find out about the local server's conventions if you upload this file to your site.

[2] These are not necessarily magic numbers. On Unix machines, mode 755 is a bit mask. The first 7 simply means that you (the file's owner) can read, write, and execute the file (7 in binary is 111 -- each bit enables an access mode). The two 5s (binary 101) say that everyone else (your group and others) can read and execute (but not write) the file. See your system's manpage on the *chmod* command for more details.

12.3.1.2 HTML basics

I promised that I wouldn't teach much HTML in this book, but you need to know enough to make sense of examples. In short, HTML is a descriptive markup language, based on tags -- items enclosed in `<>` pairs. Some tags stand alone (e.g., `<HR>` specifies a horizontal rule). Others appear in begin/end pairs where the end tag includes an extra slash.

For instance, to specify the text of a level-1 header line, we write HTML code of the form `<H1>text</H1>`; the text between the tags shows up on the web page. Some tags also allow us to specify options. For example, a tag pair like `text` specifies a hyperlink: pressing the link's text in the page directs the browser to access the Internet address (URL) listed in the `href` option.

It's important to keep in mind that HTML is used only to describe pages: your web browser reads it and translates its description to a web page with headers, paragraphs, links, and the like. Notably absent is both layout information -- the browser is responsible for arranging components on the page -- and syntax for programming logic -- there are no "if" statements, loops, and so on. There is also no Python code in this file anywhere to be found; raw HTML is strictly for defining pages, not for coding programs or specifying all user-interface details.

HTML's lack of user interface control and programmability is both a strength and a weakness. It's well-suited to describing pages and simple user interfaces at a high level. The browser, not you, handles physically laying out the page on your screen. On the other hand, HTML does not directly support full-blown GUIs and requires us to introduce CGI scripts (and other technologies) to web sites, in order to add dynamic programmability to otherwise static HTML.

12.3.1.3 Internet addresses (URLs)

Once you write an HTML file, you need to put it some place where the outside world can find it. Like all HTML files, *test0.html* must be stored in a directory on the server machine, from which the resident web server program allows browsers to fetch pages. On the server where this example lives, the page's file must be stored in or below the *public_html* directory of my personal home directory -- that is, somewhere in the directory tree rooted at */home/lutz/public_html*. For

this section, examples live in a *Basics* subdirectory, so the complete Unix pathname of this file on the server is:

```
/home/lutz/public_html/Basics/test0.html
```

This path is different than its *PP2E\Internet\Cgi-Web\Basics* location on the book's CD (<http://examples.oreilly.com/python2>), as given in the example file listing's title. When you reference this file on the client, though, you must specify its Internet address, sometimes called a URL, instead. To load the remote page, type the following text in your browser's address field (or click the example root page's [test0.html](#) hyperlink, which refers to same address):

```
http://starship.python.net/~lutz/Basics/test0.html
```

This string is a URL composed of multiple parts:

Protocol name: http

The protocol part of this URL tells the browser to communicate with the HTTP server program on the server machine, using the HTTP message protocol. URLs used in browser can also name different protocols -- for example, `ftp://` to reference a file managed by the FTP protocol and server, `telnet` to start a Telnet client session, and so on.

Server machine name: starship.python.net

A URL also names the target server machine following the protocol type. Here, we list the domain name of the server machine where the examples are installed; the machine name listed is used to open a socket to talk to the server. For HTTP, the socket is usually connected to port number 80.

File path: ~lutz/Basics/test0.html

Finally, the URL gives the path to the desired file on the remote machine. The HTTP web server automatically translates the URL's file path to the file's true Unix pathname: on my server, `~lutz` is automatically translated to the `public_html` directory in my home directory. URLs typically map to such files, but can reference other sorts of items as well.

Parameters (used in later examples)

URLs may also be followed by additional input parameters for CGI programs. When used they are introduced by a `?` and separated by `&` characters; for instance, a string of the form `?name=bob&job=hacker` at the end of a URL passes parameters named `name` and `job` to the CGI script named earlier in the URL. These values are sometimes called URL query string parameters and are treated the same as form inputs. More on both forms and parameters in a moment.

For completeness, you should also know that URLs can contain additional information (e.g., the server name part can specify a port number following a `:`), but we'll ignore these extra formatting rules here. If you're interested in more details, you might start by reading the `urlparse` module's entry in Python's library manual, as well as its source code in the Python standard library. You might also notice that a URL you type to access a page looks a bit different after the page is fetched (spaces become `+` characters, `%s` are added, etc.). This is simply because browsers must also generally follow URL escaping (i.e., translation) conventions, which we'll explore later in this chapter.

12.3.1.4 Using minimal URLs

Because browsers remember the prior page's Internet address, URLs embedded in HTML files can often omit the protocol and server names, as well as the file's directory path. If missing, the browser simply uses these components' values from the last page's address. This minimal syntax works both for URLs embedded in hyperlinks and form actions (we'll meet forms later in this chapter). For example, within a page that was fetched from directory *dirpath* on server *www.server.com*, minimal hyperlinks and form actions such as:

```
<A HREF="more.html">  
<FORM ACTION="next.cgi" ...>
```

are treated exactly as if we had specified a complete URL with explicit server and path components, like the following:

```
<A HREF="http://www.server.com/dirpath/more.html">  
<FORM ACTION="http://www.server.com/dirpath/next.cgi" ...>
```

The first minimal URL refers to file *more.html* on the same server and in the same directory that the page containing this hyperlink was fetched from; it is expanded to a complete URL within the browser. URLs can also employ Unix-style relative path syntax in the file path component. For instance, a hyperlink tag like `` names a GIF file on the server machine and parent directory of the file that contains this link's URL.

Why all the fuss about shorter URLs? Besides extending the life of your keyboard and eyesight, the main advantage of such minimal URLs is that they don't need to be changed if you ever move your pages to a new directory or server -- the server and path are inferred when the page is used, not hardcoded into its HTML. The flipside of this can be fairly painful: examples that do include explicit site and pathnames in URLs embedded within HTML code cannot be copied to other servers without source code changes. Scripts can help here, but editing source code can be error-prone.^[3]

[3] To make this process easier, the *fixsitename.py* script presented in the next section largely automates the necessary changes by performing global search-and-replace operations and directory walks. A few book examples do use complete URLs, so be sure to run this script after copying examples to a new site.

The downside of minimal URLs is that they don't trigger automatic Internet connection when followed. This becomes apparent only when you load pages from local files on your computer. For example, we can generally open HTML pages without connecting to the Internet at all, by pointing a web browser to a page's file that lives on the local machine (e.g., by clicking on its file icon). When browsing a page locally like this, following a fully specified URL makes the browser automatically connect to the Internet to fetch the referenced page or script. Minimal URLs, though, are opened on the local machine again; usually, the browser simply displays the referenced page or script's source code.

The net effect is that minimal URLs are more portable, but tend to work better when running all pages live on the Internet. To make it easier to work with the examples in this book, they will often omit the server and path components in URLs they contain. In this book, to derive a page or script's true URL from a minimal URL, imagine that the string:

`http://starship.python.net/~lutz/subdir`

appears before the filename given by the URL. Your browser will, even if you don't.

12.3.2 A First CGI Script

The HTML file we just saw is just that -- an HTML file, not a CGI script. When referenced by a browser, the remote web server simply sends back the file's text to produce a new page in the browser. To illustrate the nature of CGI scripts, let's recode the example as a Python CGI program, as shown in [Example 12-2](#).

Example 12-2. PP2E\Internet\Cgi-Web\Basics\test0.cgi

```
#!/usr/bin/python
#####
# runs on the server, prints html to create a new page;
# executable permissions, stored in ~lutz/public_html,
# url=http://starship.python.net/~lutz/Basics/test0.cgi
#####

print "Content-type: text/html\n"
print "<TITLE>CGI 101</TITLE>"
print "<H1>A First CGI script</H1>"
print "<P>Hello, CGI World!</P>"
```

This file, *test0.cgi*, makes the same sort of page if you point your browser at it (simply replace [.html](#) with [.cgi](#) in the URL). But it's a very different kind of animal -- it's an executable program that is run on the server in response to your access request. It's also a completely legal Python program, in which the page's HTML is printed dynamically, rather than being precoded in a static file. In fact, there is little that is CGI-specific about this Python program at all; if run from the system command line, it simply prints HTML rather than generating a browser page:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python test0.cgi
Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A First CGI script</H1>
<P>Hello, CGI World!</P>
```

When run by the HTTP server program on a web server machine, however, the standard output stream is tied to a socket read by the browser on the client machine. In this context, all the output is sent across the Internet to your browser. As such, it must be formatted per the browser's expectations. In particular, when the script's output reaches your browser, the first printed line is interpreted as a header, describing the text that follows. There can be more than one header line in the printed response, but there must always be a blank line between the headers and the start of the HTML code (or other data).

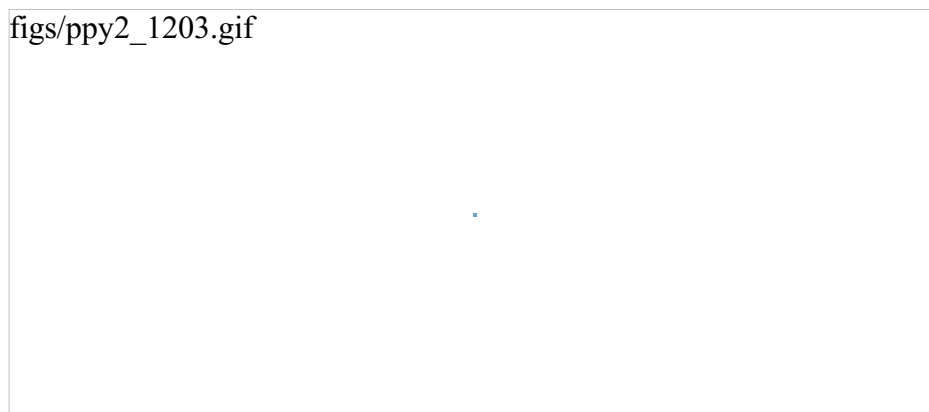
In this script, the first header line tells the browser that the rest of the transmission is HTML text ([text/html](#)), and the newline character (`\n`) at the end of the first `print` statement generates one more line-feed than the `print` statement itself. The rest of this program's output is standard HTML and is used by the browser to generate a web page on a client, exactly as if the HTML lived in a static HTML file on the server.^[4]

[4] Notice that the script does not generate the enclosing `<HEAD>` and `<BODY>` tags in

the static HTML file of the prior section. Strictly speaking, it should -- HTML without such tags is invalid. But all commonly used browsers simply ignore the omission.

CGI scripts are accessed just like HTML files: you either type the full URL of this script into your browser's address field, or click on the *test0.cgi* link line in the examples root page (which follows a minimal hyperlink that resolves to the script's full URL). [Figure 12-3](#) shows the result page generated if you point your browser at this script to make it go.

Figure 12-3. A simple web page from a CGI script



12.3.2.1 Installing CGI scripts

Like HTML files, CGI scripts are simple text files that you can either create on your local machine and upload to the server by FTP, or write with a text editor running directly on the server machine (perhaps using a telnet client). However, because CGI scripts are run as programs, they have some unique installation requirements that differ from simple HTML files. In particular, they usually must be stored and named specially, and they must be configured as programs that are executable by arbitrary users. Depending on your needs, CGI scripts may also need help finding imported modules and may need to be converted to the server platform's text file format after being uploaded. Let's look at each install constraint in more depth:

Directory and filename conventions

First of all, CGI scripts need to be placed in a directory that your web server recognizes as a program directory, and they need to be given a name that your server recognizes as a CGI script. On the server where these examples reside, CGI scripts can be stored in each user's *public_html* directory just like HTML files, but must have a filename ending in a *.cgi* suffix, not *.py*. Some servers allow *.py* filename suffixes too, and may recognize other program directories (*cgi-bin* is common), but this varies widely, too, and can sometimes be configured per server or user.

Execution conventions

Because they must be executed by the web server on behalf of arbitrary users on the Web, CGI script files also need to be given executable file permissions to mark them as programs, and they must be made executable by others. Again, a shell command `chmod 0755 filename` does the trick on most servers. CGI scripts also generally need the special `#!` line at the top, to identify the Python interpreter that runs the file's code. The text after

the `#!` in the first line simply gives the directory path to the Python executable on your server machine. See [Chapter 2](#), for more details on this special first line, and be sure to check your server's conventions for more details on non-Unix platforms.

One subtlety worth noting. As we saw earlier in the book, the special first line in executable text files can normally contain either a hardcoded path to the Python interpreter (e.g., `#!/usr/bin/python`) or an invocation of the `env` program (e.g., `#!/usr/bin/env python`), which deduces where Python lives from environment variable settings (i.e., your `$PATH`). The `env` trick is less useful in CGI scripts, though, because their environment settings are those of user "nobody" (not your own), as explained in the next paragraph.

Module search path configuration (optional)

HTTP servers generally run CGI scripts with username "nobody" for security reasons (this limits the user's access to the server machine). That's why files you publish on the Web must have special permission settings that make them accessible to other users. It also means that CGI scripts can't rely on the Python module search path to be configured in any particular way. As we've seen, the module path is normally initialized from the user's `PYTHONPATH` setting plus defaults. But because CGI scripts are run by user "nobody", `PYTHONPATH` may be arbitrary when a CGI script runs.

Before you puzzle over this too hard, you should know that this is often not a concern in practice. Because Python usually searches the current directory for imported modules by default, this is not an issue if all of your scripts and any modules and packages they use are stored in your web directory (which is the installation structure on the book's site). But if the module lives elsewhere, you may need to tweak the `sys.path` list in your scripts to adjust the search path manually before imports (e.g., with `sys.path.append(dirname)` calls, index assignments, and so on).

End-of-line conventions (optional)

Finally, on some Unix (and Linux) servers, you might also have to make sure that your script text files follow the Unix end-of-line convention (`\n`), not DOS (`\r\n`). This isn't an issue if you edit and debug right on the server (or on another Unix machine) or FTP files one by one in text mode. But if you edit and upload your scripts from a PC to a Unix server in a tar file (or in FTP binary mode), you may need to convert end-of-lines after the upload. For instance, the server that was used to develop this text returns a default error page for scripts whose end-of-lines are in DOS format (see later in this chapter for a converter script).

This installation process may sound a bit complex at first glance, but it's not bad once you've worked through it on your own: it's only a concern at install time and can usually be automated to some extent with Python scripts run on the server. To summarize, most Python CGI scripts are text files of Python code, which:

- Are named according to your web server's conventions (e.g., [file.cgi](#))
- Are stored in a directory recognized by your web server (e.g., `cgi-bin/`)
- Are given executable file permissions (e.g., `chmod 755 file.cgi`)
- Usually have the special `#!/pythonpath` line at the top (but not `env`)

- Configure `sys.path` only if needed to see modules in other directories
- Use Unix end-of-line conventions, only if your server rejects DOS format
- Print headers and HTML to generate a response page in the browser, if any
- Use the `cgi` module to parse incoming form data, if any (more about forms later in this chapter)

Even if you must use a server machine configured by someone else, most of the machine's conventions should be easy to root out. For instance, on some servers you can rename this example to `test0.py` and it will continue to be run when accessed. On others, you might instead see the file's source code in a popped-up text editor when you access it. Try a `.cgi` suffix if the text is displayed rather than executed. CGI directory conventions can vary, too, but try the directory where you normally store HTML files first. As usual, you should consult the conventions for any machine that you plan to copy these example files to.

12.3.2.2 Automating installation steps

But wait -- why do things the hard way? Before you start installing scripts by hand, remember that Python programs can usually do much of your work for you. It's easy to write Python scripts that automate some of the CGI installation steps using the operating systems tools that we met earlier in the book.

For instance, while developing the examples in this chapter, I did all editing on my PC (it's generally more dependable than a telnet client). To install, I put all the examples in a `tar` file, which is uploaded to the Linux server by FTP in a single step. Unfortunately, my server expects CGI scripts to have Unix (not DOS) end-of-line markers; unpacking the `tar` file did not convert end-of-lines or retain executable permission settings. But rather than tracking down all the web CGI scripts and fixing them by hand, I simply run the Python script in [Example 12-3](#) from within a Unix `find` command after each upload.

Example 12-3. PP2E\Internet\Cgi-Web\fixcgi.py

```
#####  
# run fom a unix find command to automate some cgi script install steps;  
# example: find . -name "*.cgi" -print -exec python fixcgi.py \{} \;  
# which converts all cgi scripts to unix line-feed format (needed on  
# starship) and gives all cgi files executable mode, else won't be run;  
# do also: chmod 777 PyErrata/DbaseFiles/*, vi Extern/Email/mailconfig*;  
# related: fixsitename.py, PyTools/fixeoln*.py, System/Filetools  
#####  
  
# after: ungzip, untar, cp -r Cgi-Web/* ~/public_html  
  
import sys, string, os  
fname = sys.argv[1]  
old = open(fname, 'rb').read( )  
new = string.replace(old, '\r\n', '\n')  
open(fname, 'wb').write(new)  
if fname[-3:] == 'cgi': os.chmod(fname, 0755) # note octal int: rwx,sgo
```

This script is kicked off at the top of the `Cgi-Web` directory, using a Unix `bash` shell command to apply it to every CGI file in a directory tree, like this:

```
% find . -name "*.cgi" -print -exec python fixcgi.py \{} \;  
./Basics/languages-src.cgi  
./Basics/getfile.cgi  
./Basics/languages.cgi  
./Basics/languages2.cgi  
./Basics/languages2reply.cgi  
./Basics/putfile.cgi  
...more...
```

Recall from [Chapter 2](#) that there are various ways to walk directory trees and find matching files in pure Python code, including the `find` module, `os.path.walk`, and one we'll use in the next section's script. For instance, a pure Python and more portable alternative could be kicked off like this:

```
C:\...\PP2E\Internet\Cgi-Web>python  
>>> import os  
>>> from PP2E.PyTools.find import find  
>>> for filename in find('*.cgi', '.'):  
...     print filename  
...     stat = os.system('python fixcgi.py ' + filename)  
...  
.\Basics\getfile.cgi  
.\Basics\languages-src.cgi  
.\Basics\languages.cgi  
.\Basics\languages2.cgi  
...more...
```

The Unix `find` command simply does the same, but outside the scope of Python: the command line after `-exec` is run for each matching file found. For more details about the `find` command, see its manpage. Within the Python script, `string.replace` translates to Unix end-of-line markers, and `os.chmod` works just like a shell `chmod` command. There are other ways to translate end-of-lines, too; see [Chapter 5](#).

12.3.2.3 Automating site move edits

Speaking of installation tasks, a common pitfall of web programming is that hardcoded site names embedded in HTML code stop working the minute you relocate the site to a new server. Minimal URLs (just the filename) are more portable, but for various reasons are not always used. Somewhere along the way, I also grew tired of updating URLs in hyperlinks and form actions, and wrote a Python script to do it all for me (see [Example 12-4](#)).

Example 12-4. PP2E\Internet\Cgi-Web\fixsitename.py

```
#!/usr/bin/env python  
#####  
# run this script in Cgi-Web dir after copying book web  
# examples to a new server--automatically changes all starship  
# server references in hyperlinks and form action tags to the  
# new server/site; warns about references that weren't changed  
# (may need manual editing); note that starship references are  
# not usually needed or used--since browsers have memory, server  
# and path can usually be omitted from a URL in the prior page  
# if it lives at the same place (e.g., "file.cgi" is assumed to  
# be in the same server/path as a page that contains this name,  
# with a real url like "http://lastserver/lastpath/file.cgi"),  
# but a handful of URLs are fully specified in book examples;  
# reuses the Visitor class developed in the system chapters,  
# to visit and convert all files at and below current dir;
```

```
#####

import os, string
from PP2E.PyTools.visitor import FileVisitor          # os.path.walk wrapper

listonly = 0
oldsite = 'starship.python.net/~lutz'                # server/rootdir in book
newsite = 'XXXXXX/YYYYYY'                            # change to your site
warnof = ['starship.python', 'lutz']                 # warn if left after fix
fixext = ['.py', '.html', '.cgi']                   # file types to check

class FixStarship(FileVisitor):
    def __init__(self, listonly=0):                    # replace oldsite refs
        FileVisitor.__init__(self, listonly=listonly) # in all web text files
        self.changed, self.warning = [], []          # need diff lists here
    def visitfile(self, fname):                       # or use find.find list
        FileVisitor.visitfile(self, fname)
        if self.listonly:
            return
        if os.path.splitext(fname)[1] in fixext:
            text = open(fname, 'r').read( )
            if string.find(text, oldsite) != -1:
                text = string.replace(text, oldsite, newsite)
                open(fname, 'w').write(text)
                self.changed.append(fname)
            for word in warnof:
                if string.find(text, word) != -1:
                    self.warning.append(fname); break

if __name__ == '__main__':
    # don't run auto if clicked
    go = raw_input('This script changes site in all web files; continue?')
    if go != 'y':
        raw_input('Canceled - hit enter key')
    else:
        walker = FixStarship(listonly)
        walker.run( )
        print 'Visited %d files and %d dirs' % (walker.fcount, walker.dcount)

        def showhistory(label, flist):
            print '\n%s in %d files:' % (label, len(flist))
            for fname in flist:
                print '=>', fname
            showhistory('Made changes', walker.changed)
            showhistory('Saw warnings', walker.warning)

        def edithistory(flist):
            for fname in flist:                            # your editor here
                os.system('vi ' + fname)
            if raw_input('Edit changes?') == 'y': edithistory(walker.changed)
            if raw_input('Edit warnings?') == 'y': edithistory(walker.warning)
```

This is a more complex script that reuses the *visitor.py* module we wrote in [Chapter 5](#) to wrap the `os.path.walk` call. If you read that chapter, this script will make sense. If not, we won't go into many more details here again. Suffice it to say that this program visits all source code files at and below the directory where it is run, globally changing all `starship.python.net/~lutz` appearances to whatever you've assigned to variable `newsite` within the script. On request, it will also launch your editor to view files changed, as well as files that contain potentially suspicious strings. As coded, it launches the Unix `vi` text editor at the end, but you can change this to start whatever editor you like (this is Python, after all):

```
C:\...\PP2E\Internet\Cgi-Web>python fixsitename.py
This script changes site in all web files; continue?y
```

```
. . . .
1 => .\PyInternetDemos.html
2 => .\README.txt
3 => .\fixcgi.py
4 => .\fixsitename.py
5 => .\index.html
6 => .\python_snake_ora.gif
.\Basics ...
7 => .\Basics\mlutz.jpg
8 => .\Basics\languages.html
9 => .\Basics\languages-src.cgi
...more...
146 => .\PyMailCgi\temp\secret.doc.txt
Visited 146 files and 16 dirs
```

```
Made changes in 8 files:
=> .\fixsitename.py
=> .\Basics\languages.cgi
=> .\Basics\test3.html
=> .\Basics\test0.py
=> .\Basics\test0.cgi
=> .\Basics\test5c.html
=> .\PyMailCgi\commonhtml.py
=> .\PyMailCgi\sendurl.py
```

```
Saw warnings in 14 files:
=> .\PyInternetDemos.html
=> .\fixsitename.py
=> .\index.html
=> .\Basics\languages.cgi
...more...
=> .\PyMailCgi\pymailcgi.html
=> .\PyMailCgi\commonhtml.py
=> .\PyMailCgi\sendurl.py
Edit changes?n
Edit warnings?y
```

The net effect is that this script automates part of the site relocation task: running it will update all pages' URLs for the new site name automatically, which is considerably less aggravating than manually hunting down and editing each such reference by hand.

There aren't many hardcoded *starship* site references in web examples in this book (the script found and fixed eight above), but be sure to run this script in the *Cgi-Web* directory from a command line, after copying the book examples to your own site. To use this script for other site moves, simply set both `oldsite` and `newsite` as appropriate. The truly ambitious scriptmaster might even run such a script from within another that first copies a site's contents by FTP (see `ftplib` in the previous chapter).^[5]

[5] As I mentioned at the start of this chapter, there are often multiple ways to accomplish any given webmaster-y task. For instance, the HTML `<BASE>` tag may provide an alternative way to map absolute URLs, and FTPing your web site files to your server individually and in text mode might obviate line-end issues. There are undoubtedly other ways to handle such tasks, too. On the other hand, such alternatives wouldn't be all that useful in a book that illustrates Python coding techniques.

12.3.2.4 Finding Python on the server

One last install pointer: even though Python doesn't have to be installed on any clients in the

context of a server-side web application, it does have to exist on the server machine where your CGI scripts are expected to run. If you are using a web server that you did not configure yourself you must be sure that Python lives on that machine. Moreover, you need to find where it is on that machine so that you can specify its path in the `#!` line at the top of your script.

By now, Python is a pervasive tool, so this generally isn't as big a concern as it once was. As time goes by, it will become even more common to find Python as a standard component of server machines. But if you're not sure if or where Python lives on yours, here are some tips:

- Especially on Unix systems, you should first assume that Python lives in a standard place (e.g., `/usr/local/bin/python`), and see if it works. Chances are that Python already lives on such machines. If you have Telnet access on your server, a Unix `find` command starting at `/usr` may help.
- If your server runs Linux, you're probably set to go. Python ships as a standard part of Linux distributions these days, and many web sites and Internet Service Providers (ISPs) run the Linux operating system; at such sites, Python probably already lives at `/usr/bin/python`.
- In other environments where you cannot control the server machine yourself, it may be harder to obtain access to an already-installed Python. If so, you can relocate your site to a server that does have Python installed, talk your ISP into installing Python on the machine you're trying to use, or install Python on the server machine yourself.

If your ISP is unsympathetic to your need for Python and you are willing to relocate your site to one that is, you can find lists of Python-friendly ISPs by searching <http://www.python.org>. And if you choose to install Python on your server machine yourself, be sure to check out the freeze tool shipped with the Python source distribution (in the *Tools* directory). With freeze, you can create a single executable program file that contains the entire Python interpreter, as well as all the standard library modules. Such a frozen interpreter can be uploaded to your web account by FTP in a single step, and it won't require a full-blown Python installation on the server.

12.3.3 Adding Pictures and Generating Tables

Now let's get back to writing server-side code. As anyone who's ever surfed the Web knows, we pages usually consist of more than simple text. [Example 12-5](#) is a Python CGI script that prints an `` HTML tag in its output to produce a graphic image in the client browser. There's not much Python-specific about this example, but note that just as for simple HTML files, the image file (*ppsmall.gif*) lives on and is downloaded from the server machine when the browser interprets the output of this script.

Example 12-5. PP2E\Internet\Cgi-Web\Basics\test1.cgi

```
#!/usr/bin/python

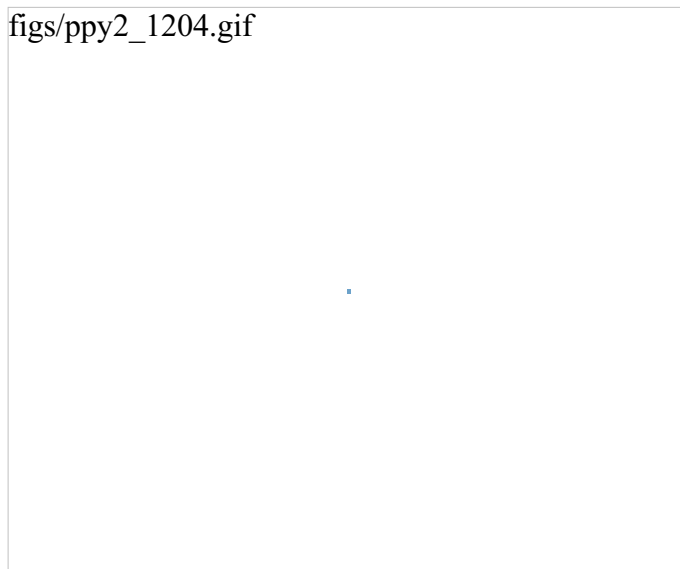
text = """Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A Second CGI script</H1>
<HR>
<P>Hello, CGI World!</P>
<IMG src="ppsmall.gif" BORDER=1 ALT=[image]>
<HR>
"""
```

```
print text
```

Notice the use of the triple-quoted string block here; the entire HTML string is sent to the browser in one fell swoop, with the print statement at the end. If client and server are both functional, a page that looks like [Figure 12-4](#) will be generated when this script is referenced and run.

Figure 12-4. A page with an image generated by test1.cgi



So far, our CGI scripts have been putting out canned HTML that could have just as easily been stored in an HTML file. But because CGI scripts are executable programs, they can also be used to generate HTML on the fly, dynamically -- even, possibly, in response to a particular set of user inputs sent to the script. That's the whole purpose of CGI scripts, after all. Let's start using this to better advantage now, and write a Python script that builds up response HTML programmatically (see [Example 12-6](#)).

Example 12-6. PP2E\Internet\Cgi-Web\Basics\test2.cgi

```
#!/usr/bin/python

print """Content-type: text/html

<TITLE>CGI 101</TITLE>
<H1>A Third CGI script</H1>
<HR>
<P>Hello, CGI World!</P>

<table border=1>
"""

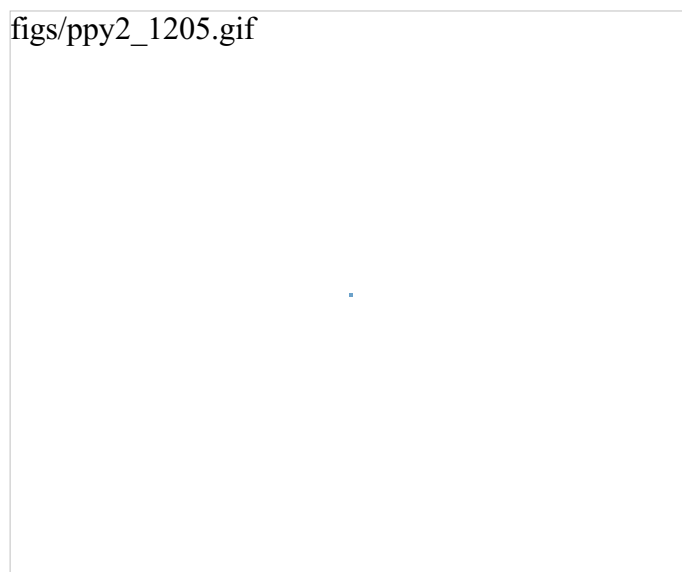
for i in range(5):
    print "<tr>"
    for j in range(4):
        print "<td>%d.%d</td>" % (i, j)
```

```
print "</tr>"

print """
</table>
<HR>
"""
```

Despite all the tags, this really is Python code -- the *test2.cgi* script uses triple-quoted strings to embed blocks of HTML again. But this time, the script also uses nested Python `for` loops to dynamically generate part of the HTML that is sent to the browser. Specifically, it emits HTML to lay out a two-dimensional table in the middle of a page, as shown in [Figure 12-5](#).

Figure 12-5. A page with a table generated by *test2.cgi*



Each row in the table displays a "row.column" pair, as generated by the executing Python script. If you're curious how the generated HTML looks, select your browser's View Source option after you've accessed this page. It's a single HTML page composed of the HTML generated by the first `print` in the script, then the `for` loops, and finally the last `print`. In other words, the concatenation of this script's output is an HTML document with headers.

12.3.3.1 Table tags

This script generates HTML table tags. Again, we're not out to learn HTML here, but we'll take a quick look just so you can make sense of the example. Tables are declared by the text between `<table>` and `</table>` tags in HTML. Typically, a table's text in turn declares the contents of each table row between `<tr>` and `</tr>` tags and each column within a row between `<td>` and `</td>` tags. The loops in our script build up HTML to declare five rows of four columns each, by printing the appropriate tags, with the current row and column number as column values. For instance, here is part of the script's output, defining the first two rows:


```
<table border=1>
<tr>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
</tr>
. . .
</table>
```

Other table tags and options let us specify a row title (`<th>`), layout borders, and so on. We'll see more table syntax put to use to lay out forms in a later section.

12.3.4 Adding User Interaction

CGI scripts are great at generating HTML on the fly like this, but they are also commonly used to implement interaction with a user typing at a web browser. As described earlier in this chapter web interactions usually involve a two-step process and two distinct web pages: you fill out a form page and press submit, and a reply page eventually comes back. In between, a CGI script processes the form input.

12.3.4.1 Submission

That description sounds simple enough, but the process of collecting user inputs requires an understanding of a special HTML tag, `<form>`. Let's look at the implementation of a simple web interaction to see forms at work. First off, we need to define a form page for the user to fill out, as shown in [Example 12-7](#).


Example 12-7. PP2E\Internet\Cgi-Web\Basics\test3.html

```
<html><body>
<title>CGI 101</title>
<H1>A first user interaction: forms</H1>
<hr>
<form method=POST action="http://starship.python.net/~lutz/Basics/test3.cgi">
  <P><B>Enter your name:</B>
  <P><input type=text name=user>
  <P><input type=submit>
</form>
</BODY></HTML>
```

test3.html is a simple HTML file, not a CGI script (though its contents could be printed from a script as well). When this file is accessed, all the text between its `<form>` and `</form>` tags generate the input fields and Submit button shown in [Figure 12-6](#).

Figure 12-6. A simple form page generated by test3.html

figs/ppy2_1206.gif



12.3.4.2 More on form tags

We won't go into all the details behind coding HTML forms, but a few highlights are worth underscoring. Within a form's HTML code:

- The form's `action` option gives the URL of a CGI script that will be invoked to process submitted form data. This is the link from a form to its handler program -- in this case, a program called `test3.cgi` in my web home directory, on a server machine called `starship.python.net`. The `action` option is the moral equivalent to `command` options in Tkinter buttons -- it's where a callback handler (here, a remote handler) is registered to the browser.
- Input controls are specified with nested `<input>` tags. In this example, input tags have two key options. The `type` option accepts values such as `text` for text fields and `submit` for a Submit button (which sends data to the server and is labeled "Submit Query" by default). The `name` option is the hook used to identify the entered value by key, once all the form data reaches the server. For instance, the server-side CGI script we'll see in a moment uses the string `user` as a key to get the data typed into this form's text field. As we'll see in later examples, other input tag options can specify initial values (`value=X`), display-only mode (`readonly`), and so on. Other input `type` option values may transmit hidden data (`type=hidden`), reinitialize fields (`type=reset`), or make multiple-choice buttons (`type=checkbox`).
- Forms also include a `method` option to specify the encoding style to be used to send data over a socket to the target server machine. Here, we use the `post` style, which contacts the server and then ships it a stream of user input data in a separate transmission. An alternative `get` style ships input information to the server in a single transmission step, by adding user inputs to the end of the URL used to invoke the script, usually after a `?` character (more on this soon). With `get`, inputs typically show up on the server in environment variables or as arguments in the command line used to start the script. With `post`, they must be read from standard input and decoded. Luckily, Python's `cgi` module transparently handles either encoding style, so our CGI scripts don't need to know or care which is used.

Notice that the action URL in this example's form spells out the full address for illustration. Because the browser remembers where the enclosing HTML page came from, it works the same with just the script's filename, as shown in [Example 12-8](#).

Example 12-8. PP2E\Internet\Cgi-Web\Basics\test3-minimal.html

```
<html><body>
<title>CGI 101</title>
<H1>A first user interaction: forms</H1>
<hr>
<form method=POST action="test3.cgi">
  <P><B>Enter your name:</B>
  <P><input type=text name=user>
  <P><input type=submit>
</form>
</BODY></HTML>
```

It may help to remember that URLs embedded in form action tags and hyperlinks are directions to the browser first, not the script. The *test3.cgi* script itself doesn't care which URL form is used to trigger it -- minimal or complete. In fact, all parts of a URL through the script filename (and up to URL query parameters) is used in the conversation between browser and HTTP server, before a CGI script is ever spawned. As long as the browser knows which server to contact, the URL will work, but URLs outside of a page (e.g., typed into a browser's address field or sent to Python's `urllib` module) usually must be completely specified, because there is no notion of a prior page.

12.3.4.3 Response

So far, we've created only a static page with an input field. But the Submit button on this page is loaded to work magic. When pressed, it triggers the remote program whose URL is listed in the form's `action` option, and passes this program the input data typed by the user, according to the form's `method` encoding style option. On the server, a Python script is started to handle the form's input data while the user waits for a reply on the client, as shown in [Example 12-9](#).

Example 12-9. PP2E\Internet\Cgi-Web\Basics\test3.cgi

```
#!/usr/bin/python
#####
# runs on the server, reads form input, prints html;
# url=http://server-name/root-dir/Basics/test3.cgi
#####

import cgi
form = cgi.FieldStorage( )          # parse form data
print "Content-type: text/html"    # plus blank line

html = """
<TITLE>test3.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<P>%s</P>
<HR>"""

if not form.has_key('user'):
    print html % "Who are you?"
else:
    print html % ("Hello, %s." % form['user'].value)
```

As before, this Python CGI script prints HTML to generate a response page in the client's browser. But this script does a bit more: it also uses the standard `cgi` module to parse the input data entered by the user on the prior web page (see [Figure 12-6](#)). Luckily, this is all automatic in

Python: a call to the `cgi` module's `FieldStorage` class automatically does all the work of extracting form data from the input stream and environment variables, regardless of how that data was passed -- in a `post` style stream or in `get` style parameters appended to the URL. Inputs sent in both styles look the same to Python scripts.

Scripts should call `cgi.FieldStorage` only once and before accessing any field values. When called, we get back an object that looks like a dictionary -- user input fields from the form (or URL) show up as values of keys in this object. For example, in the script, `form['user']` is an object whose `value` attribute is a string containing the text typed into the form's text field. If you flip back to the form page's HTML, you'll notice that the input field's `name` option was `user` -- the name in the form's HTML has become a key we use to fetch the input's value from a dictionary. The object returned by `FieldStorage` supports other dictionary operations, too -- for instance, the `has_key` method may be used to check if a field is present in the input data.

Before exiting, this script prints HTML to produce a result page that echoes back what the user typed into the form. Two string-formatting expressions (%) are used to insert the input text into a reply string, and the reply string into the triple-quoted HTML string block. The body of the script's output looks like this:

```
<TITLE>test3.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<P>Hello, King Arthur.</P>
<HR>
```

In a browser, the output is rendered into a page like the one in [Figure 12-7](#).

Figure 12-7. test3.cgi result for parameters in a form



12.3.4.4 Passing parameters in URLs

Notice that the URL address of the script that generated this page shows up at the top of the browser. We didn't type this URL itself -- it came from the `action` tag of the prior page's `form` HTML. However, there is nothing stopping us from typing the script's URL explicitly in our browser's address field to invoke the script, just as we did for our earlier CGI script and HTML file examples.

But there's a catch here: where does the input field's value come from if there is no form page? That is, if we type the CGI script's URL ourselves, how does the input field get filled in? Earlier when we talked about URL formats, I mentioned that the `get` encoding scheme tacks input parameters onto the end of URLs. When we type script addresses explicitly, we can also append input values on the end of URLs, where they serve the same purpose as `<input>` fields in forms.

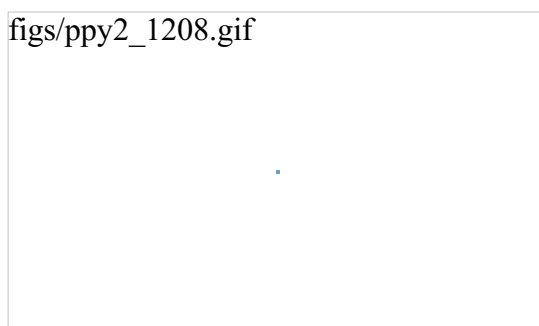
Moreover, the Python `cgi` module makes URL and form inputs look identical to scripts.

For instance, we can skip filling out the input form page completely, and directly invoke our `test3.cgi` script by visiting a URL of the form:

```
http://starship.python.net/~lutz/Basics/test3.cgi?user=Brian
```

In this URL, a value for the input named `user` is specified explicitly, as if the user had filled out the input page. When called this way, the only constraint is that the parameter name `user` must match the name expected by the script (and hardcoded in the form's HTML). We use just one parameter here, but in general, URL parameters are typically introduced with a `?` and followed by one or more `name=value` assignments, separated by `&` characters if there is more than one. [Figure 12-8](#) shows the response page we get after typing a URL with explicit inputs.

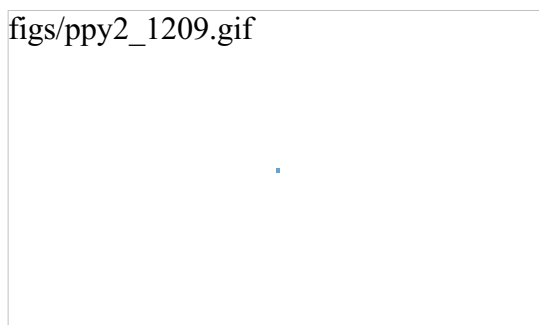
Figure 12-8. test3.cgi result for parameters in a URL



In general, any CGI script can be invoked either by filling out and submitting a form page or by passing inputs at the end of a URL. When CGI scripts are invoked with explicit input parameters this way, it's difficult to not see their similarity to functions, albeit ones that live remotely on the Net. Passing data to scripts in URLs is similar to keyword arguments in Python functions, both operationally and syntactically. In fact, in [Chapter 15](#) we will meet a system called Zope that makes the relationship between URLs and Python function calls even more literal (URLs become more direct function calls).

Incidentally, if you clear out the name input field in the form input page (i.e., make it empty) and press submit, the `user` name field becomes empty. More accurately, the browser may not send this field along with the form data at all, even though it is listed in the form layout HTML. The CGI script detects such a missing field with the dictionary `has_key` method and produces the page captured in [Figure 12-9](#) in response.

Figure 12-9. An empty name field produces an error page



In general, CGI scripts must check to see if any inputs are missing, partly because they might not be typed by a user in the form, but also because there may be no form at all -- input fields might not be tacked on to the end of an explicitly typed URL. For instance, if we type the script's URL without any parameters at all (i.e., omit the text ? and beyond), we get this same error response page. Since we can invoke any CGI through a form or URL, scripts must anticipate both scenarios.

12.3.5 Using Tables to Lay Out Forms

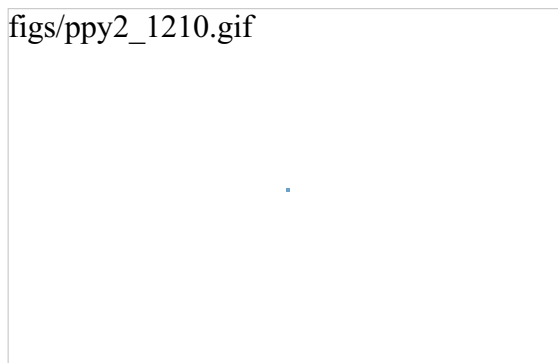
Now let's move on to something a bit more realistic. In most CGI applications, input pages are composed of multiple fields. When there is more than one, input labels and fields are typically laid out in a table, to give the form a well-structured appearance. The HTML file in [Example 12-10](#) defines a form with two input fields.

Example 12-10. PP2E\Internet\Cgi-Web\Basics\test4.html

```
<html><body>
<title>CGI 101</title>
<H1>A second user interaction: tables</H1>
<hr>
<form method=POST action="test4.cgi">
  <table>
    <TR>
      <TH align=right>Enter your name:
      <TD><input type=text name=user>
    <TR>
      <TH align=right>Enter your age:
      <TD><input type=text name=age>
    <TR>
      <TD colspan=2 align=center>
        <input type=submit value="Send">
    </table>
  </form>
</body></html>
```

The `<TH>` tag defines a column like `<TD>`, but also tags it as a header column, which generally means it is rendered in a bold font. By placing the input fields and labels in a table like this, we get an input page like that shown in [Figure 12-10](#). Labels and inputs are automatically lined up vertically in columns much as they were by the Tkinter GUI geometry managers we met earlier in this book.

Figure 12-10. A form laid out with table tags



When this form's Submit button (labeled "Send" by the page's HTML) is pressed, it causes the script in [Example 12-11](#) to be executed on the server machine, with the inputs typed by the user.

Example 12-11. PP2E\Internet\Cgi-Web\Basics\test4.cgi

```
#!/usr/bin/python
#####
# runs on the server, reads form input, prints html;
# url http://server-name/root-dir/Basics/test4.cgi
#####

import cgi, sys
sys.stderr = sys.stdout          # errors to browser
form = cgi.FieldStorage( )       # parse form data
print "Content-type: text/html\n" # plus blank line

# class dummy:
#     def __init__(self, s): self.value = s
# form = {'user': dummy('bob'), 'age':dummy('10')}

html = """
<TITLE>test4.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>%s</H4>
<H4>%s</H4>
<H4>%s</H4>
<HR>"""

if not form.has_key('user'):
    line1 = "Who are you?"
else:
    line1 = "Hello, %s." % form['user'].value

line2 = "You're talking to a %s server." % sys.platform

line3 = ""
if form.has_key('age'):
    try:
        line3 = "Your age squared is %d!" % (int(form['age'].value) ** 2)
    except:
        line3 = "Sorry, I can't compute %s ** 2." % form['age'].value

print html % (line1, line2, line3)
```

The table layout comes from the HTML file, not this Python CGI script. In fact, this script doesn't do much new -- it uses string formatting to plug input values into the response page's HTML triple-quoted template string as before, this time with one line per input field. There are, however, a few new tricks here worth noting, especially regarding CGI script debugging and security. We'll talk about them in the next two sections.

12.3.5.1 Converting strings in CGI scripts

Just for fun, the script echoes back the name of the server platform by fetching `sys.platform` along with the square of the `age` input field. Notice that the `age` input's value must be converted to an integer with the built-in `int` function; in the CGI world, all inputs arrive as strings. We could also convert to an integer with the built-in `string.atoi` or `eval` function. Conversion (and other) errors are trapped gracefully in a `try` statement to yield an error line, rather than letting our script die.

You should never use `eval` to convert strings that were sent over the Internet like the `age` field in this example, unless you can be absolutely sure that the string is not even potentially malicious code. For instance, if this example were available on the general Internet, it's not impossible that someone could type a value into the `age` field (or append an `age` parameter to the URL) with a value like: `os.system('rm *')`. When passed to `eval`, such a string might delete all the files in your server script directory!

We talk about ways to minimize this risk with Python's restricted execution mode (module `rexec`) in [Chapter 15](#). But by default, strings read off the Net can be very bad things to say in CGI scripting. You should never pass them to dynamic coding tools like `eval` and `exec`, or to tools that run arbitrary shell commands such as `os.popen` and `os.system`, unless you can be sure that they are safe, or unless you enable Python's restricted execution mode in your scripts.

12.3.5.2 Debugging CGI scripts

Errors happen, even in the brave new world of the Internet. Generally speaking, debugging CGI scripts can be much more difficult than debugging programs that run on your local machine. Not only do errors occur on a remote machine, but scripts generally won't run without the context implied by the CGI model. The script in [Example 12-11](#) demonstrates the following two common debugging tricks.

Error message trapping

This script assigns `sys.stderr` to `sys.stdout` so that Python error messages wind up being displayed in the response page in the browser. Normally, Python error messages are written to `stderr`. To route them to the browser, we must make `stderr` reference the same file object as `stdout` (which is connected to the browser in CGI scripts). If we don't do this assignment, Python errors, including program errors in our script, never show up in the browser.

Test case mock-up

The `dummy` class definition, commented out in this final version, was used to debug the script before it was installed on the Net. Besides not seeing `stderr` messages by default, CGI scripts also assume an enclosing context that does not exist if they are tested outside the CGI environment. For instance, if run from the system command line, this script has no form input data. Uncomment this code to test from the system command line. The `dummy` class masquerades as a parsed form field object, and `form` is assigned a dictionary containing two form field objects. The net effect is that `form` will be plug-and-play compatible with the result of a `cgi.FieldStorage` call. As usual in Python, object interfaces (not datatypes) are all we must adhere to.

Here are a few general tips for debugging your server-side CGI scripts:

Run the script from the command line.

It probably won't generate HTML as is, but running it standalone will detect any syntax errors in your code. Recall that a Python command line can run source code files regardless of their extension: e.g., `python somescript.cgi` works fine.

Assign `sys.stderr` to `sys.stdout` as early as possible in your script.

This will make the text of Python error messages and stack dumps appear in your client browser when accessing the script. In fact, short of wading through server logs, this may be the only way to see the text of error messages after your script aborts.

Mock up inputs to simulate the enclosing CGI context.

For instance, define classes that mimic the CGI inputs interface (as done with the `dummy` class in this script), so that you can view the script's output for various test cases by running it from the system command line.^[6] Setting environment variables to mimic form or URL inputs sometimes helps, too (we'll see how later in this chapter).

[6] This technique isn't unique to CGI scripts, by the way. In [Chapter 15](#), we'll meet systems that embed Python code inside HTML. There is no good way to test such code outside the context of the enclosing system, without extracting the embedded Python code (perhaps by using the `htmllib` HTML parser that comes with Python) and running it with a passed-in mock-up of the API that it will eventually use.

Call utilities to display CGI context in the browser.

The CGI module includes utility functions that send a formatted dump of CGI environment variables and input values to the browser (e.g., `cgi.test`, `cgi.print_form`). Sometimes, this is enough to resolve connection problems. We'll use some of these in the mailer case study in the next chapter.

Show exceptions you catch.

If you catch an exception that Python raises, the Python error message won't be printed to `stderr` (that is simply the default behavior). In such cases, it's up to your script to display the exception's name and value in the response page; exception details are available in the built-in `sys` module. We'll use this in a later example, too.

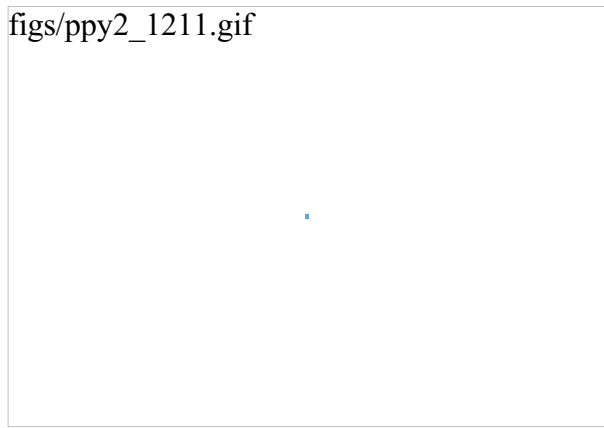
Run it live.

Of course, once your script is at least half working, your best bet is likely to start running live on the server, with real inputs coming from a browser.

When this script is run by submitting the input form page, its output produces the new reply page shown in [Figure 12-11](#).

Figure 12-11. Reply page generated by `test4.cgi`

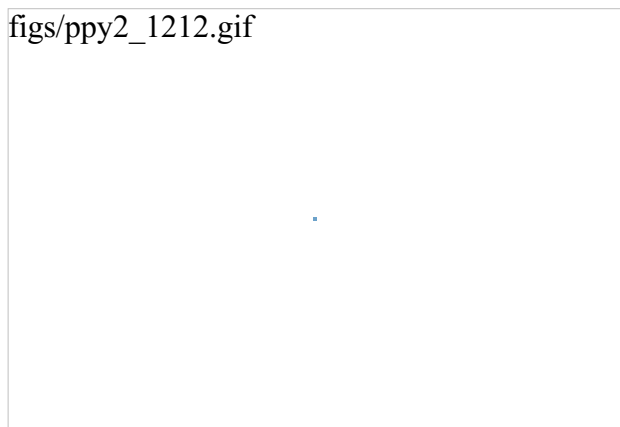
figs/ppy2_1211.gif



As usual, we can pass parameters to this CGI script at the end of a URL, too. [Figure 12-12](#) show the page we get when passing a `user` and `age` explicitly in the URL. Notice that we have two parameters after the `?` this time; we separate them with `&`. Also note that we've specified a blank space in the `user` value with `+`. This is a common URL encoding convention. On the server side, the `+` is automatically replaced with a space again. It's also part of the standard escape rule for URL strings, which we'll revisit later.

Figure 12-12. Reply page generated by test4.cgi for parameters in URL

figs/ppy2_1212.gif



12.3.6 Adding Common Input Devices

So far, we've been typing inputs into text fields. HTML forms support a handful of input control (what we'd call widgets in the traditional GUI world) for collecting user inputs. Let's look at a CGI program that shows all the common input controls at once. As usual, we define both an HTML file to lay out the form page and a Python CGI script to process its inputs and generate a response. The HTML file is presented in [Example 12-12](#).

Example 12-12. PP2E\Internet\Cgi-Web\Basics\test5a.html

```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices</H1>
<HR>
<FORM method=POST action="test5.cgi">
  <H3>Please complete the following form and click Send</H3>
```

```
<P><TABLE>
  <TR>
    <TH align=right>Name:
    <TD><input type=text name=name>
  <TR>
    <TH align=right>Shoe size:
    <TD><table>
      <td><input type=radio name=shoesize value=small>Small
      <td><input type=radio name=shoesize value=medium>Medium
      <td><input type=radio name=shoesize value=large>Large
    </table>
  <TR>
    <TH align=right>Occupation:
    <TD><select name=job>
      <option>Developer
      <option>Manager
      <option>Student
      <option>Evangelist
      <option>Other
    </select>
  <TR>
    <TH align=right>Political affiliations:
    <TD><table>
      <td><input type=checkbox name=language value=Python>Pythonista
      <td><input type=checkbox name=language value=Perl>Perlmonger
      <td><input type=checkbox name=language value=Tcl>Tcler
    </table>
  <TR>
    <TH align=right>Comments:
    <TD><textarea name=comment cols=30 rows=2>Enter text here</textarea>
  <TR>
    <TD colspan=2 align=center>
      <input type=submit value="Send">
</TABLE>
</FORM>
<HR>
</BODY></HTML>
```

When rendered by a browser, the page in [Figure 12-13](#) appears.

Figure 12-13. Form page generated by test5a.html



This page contains a simple text field as before, but it also has radiobuttons, a pull-down selection list, a set of multiple-choice checkbuttons, and a multiple-line text input area. All have name option in the HTML file, which identifies their selected value in the data sent from client to server. When we fill out this form and click the Send submit button, the script in [Example 12-13](#) runs on the server to process all the input data typed or selected in the form.

Example 12-13. PP2E\Internet\Cgi-Web\Basics\test5.cgi

```
#!/usr/bin/python
#####
# runs on the server, reads form input, prints html;
# url=http://server-name/root-dir/Basics/test5.cgi
#####

import cgi, sys, string
form = cgi.FieldStorage( )           # parse form data
print "Content-type: text/html"      # plus blank line

html = """
<TITLE>test5.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is %(name)s</H4>
<H4>You wear rather %(shoesize)s shoes</H4>
<H4>Your current job: %(job)s</H4>
<H4>You program in %(language)s</H4>
<H4>You also said:</H4>
<P>%(comment)s</P>
<HR>"""

data = {}
for field in ['name', 'shoesize', 'job', 'language', 'comment']:
    if not form.has_key(field):
        data[field] = '(unknown)'
    else:
        if type(form[field]) != type([]):
            data[field] = form[field].value
        else:
            values = map(lambda x: x.value, form[field])
            data[field] = string.join(values, ' and ')
print html % data
```

This Python script doesn't do much; it mostly just copies form field information into a dictionary called `data`, so that it can be easily plugged into the triple-quoted response string. A few of its tricks merit explanation:

Field validation

As usual, we need to check all expected fields to see if they really are present in the input data, using the dictionary `has_key` method. Any or all of the input fields may be missing if they weren't entered on the form or appended to an explicit URL.

String formatting

We're using dictionary key references in the format string this time -- recall that `%(name)s` means pull out the value for key `name` in the data dictionary and perform a to-string conversion on its value.

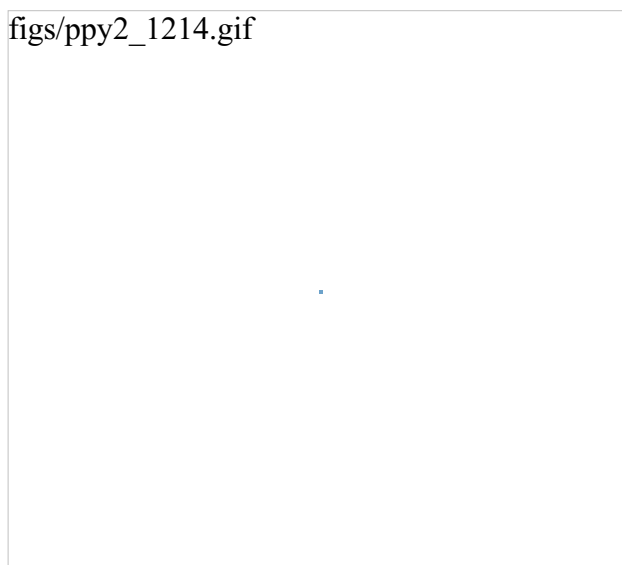
Multiple-choice fields

We're also testing the type of all the expected fields' values to see if they arrive as a list instead of the usual string. Values of multiple-choice input controls, like the `language` choice field in this input page, are returned from `cgi.FieldStorage` as a list of objects with `value` attributes, rather than a simple single object with a `value`. This script copies simple field values to the dictionary verbatim, but uses `map` to collect the value fields of multiple-choice selections, and `string.join` to construct a single string with an `and` inserted between each selection value (e.g., `Python and Tcl`).^[7]

[7] Two forward references are worth noting here. Besides simple strings and lists, later we'll see a third type of form input object, returned for fields that specify file uploads. The script in this example should really also escape the echoed text inserted into the HTML reply to be robust, lest it contain HTML operators. We will discuss escapes in detail later.

When the form page is filled out and submitted, the script creates the response shown in [Figure 12-14](#) -- essentially just a formatted echo of what was sent.

Figure 12-14. Response page created by test5.cgi (1)



12.3.6.1 Changing input layouts

Suppose that you've written a system like this, and your users, clients, and significant other start complaining that the input form is difficult to read. Don't worry. Because the CGI model naturally separates the user interface (the HTML page definition) from the processing logic (the CGI script), it's completely painless to change the form's layout. Simply modify the HTML file; there's no need to change the CGI code at all. For instance, [Example 12-14](#) contains a new definition of the input that uses tables a bit differently to provide a nicer layout with borders.

Example 12-14. PP2E\Internet\Cgi-Web\Basics\test5b.html

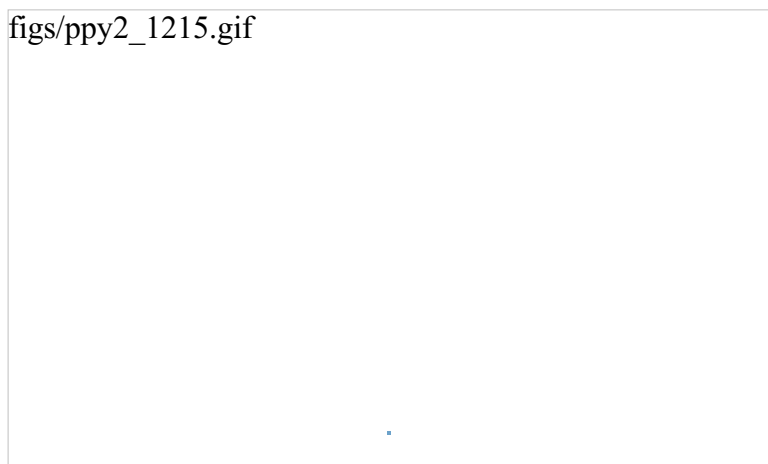
```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: alternative layout</H1>
<P>Use the same test5.cgi server side script, but change the
layout of the form itself. Notice the separation of user interface
and processing logic here; the CGI script is independent of the
HTML used to interact with the user/client.</P><HR>

<FORM method=POST action="test5.cgi">
  <H3>Please complete the following form and click Submit</H3>
  <P><TABLE border cellpadding=3>
    <TR>
      <TH align=right>Name:</TH>
      <TD><input type=text name=name>
    <TR>
      <TH align=right>Shoe size:</TH>
      <TD><input type=radio name=shoesize value=small>Small
        <input type=radio name=shoesize value=medium>Medium
        <input type=radio name=shoesize value=large>Large
    <TR>
      <TH align=right>Occupation:</TH>
      <TD><select name=job>
        <option>Developer
        <option>Manager
        <option>Student
        <option>Evangelist
        <option>Other
      </select>
    <TR>
      <TH align=right>Political affiliations:</TH>
      <TD><P><input type=checkbox name=language value=Python>Pythonista
        <P><input type=checkbox name=language value=Perl>Perlmonger
        <P><input type=checkbox name=language value=Tcl>Tcler
    <TR>
      <TH align=right>Comments:</TH>
      <TD><textarea name=comment cols=30 rows=2>Enter spam here</textarea>
    <TR>
      <TD colspan=2 align=center>
        <input type=submit value="Submit">
        <input type=reset value="Reset">
      </TD>
    </TABLE>
  </FORM>
</BODY></HTML>
```

When we visit this alternative page with a browser, we get the interface shown in [Figure 12-15](#).

Figure 12-15. Form page created by test5b.html

figs/ppy2_1215.gif

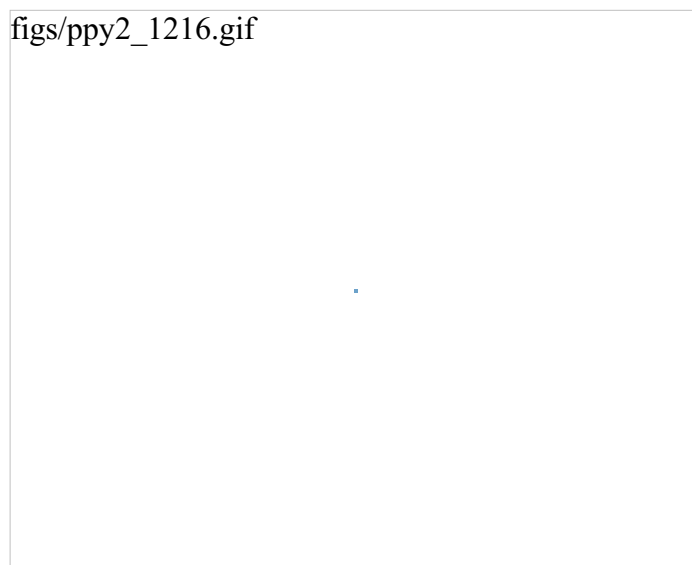




Now, before you go blind trying to detect the differences in this and the prior HTML file, I should note that the HTML differences that produce this page are much less important than the fact that the `action` fields in these two pages' forms reference identical URLs. Pressing this version's Submit button triggers the exact same and totally unchanged Python CGI script again, *test5.cgi* ([Example 12-13](#)).

That is, scripts are completely independent of the layout of the user-interface used to send them information. Changes in the response page require changing the script, of course; but we can change the input page's HTML as much as we like, without impacting the server-side Python code. [Figure 12-16](#) shows the response page produced by the script this time around.

Figure 12-16. Response page created by test5.cgi (2)



12.3.7 Passing Parameters in Hardcoded URLs

Earlier, we passed parameters to CGI scripts by listing them at the end of a URL typed into the browser's address field (after a `?`). But there's nothing sacred about the browser's address field. In particular, there's nothing stopping us from using the same URL syntax in hyperlinks that we hardcode in web page definitions. For example, the web page from [Example 12-15](#) defines three hyperlinks (the text between `<A>` and `` tags), which all trigger our original *test5.cgi* script again, but with three different pre-coded sets of parameters.

Example 12-15. PP2E\Internet\Cgi-Web\Basics\test5c.html

```
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: URL parameters</H1>

<P>This demo invokes the test5.cgi server-side script again,
but hardcodes input data to the end of the script's URL,
within a simple hyperlink (instead of packaging up a form's
inputs). Click your browser's "show page source" button
to view the links associated with each list item below.

<P>This is really more about CGI than Python, but notice that
Python's cgi module handles both this form of input (which is
also produced by GET form actions), as well as POST-ed forms;
they look the same to the Python CGI script. In other words,
cgi module users are independent of the method used to submit
data.

<P>Also notice that URLs with appended input values like this
can be generated as part of the page output by another CGI script,
to direct a next user click to the right place and context; together
with type 'hidden' input fields, they provide one way to
save state between clicks.
</P><HR>

<UL>
<LI><A href="test5.cgi?name=Bob&shoesize=small">Send Bob, small</A>
<LI><A href="test5.cgi?name=Tom&language=Python">Send Tom, Python</A>
<LI><A href=
"http://starship.python.net/~lutz/Basics/test5.cgi?job=Evangelist&comment=spam"
Send Evangelist, spam</A>
</UL>

<HR></BODY></HTML>
```

This static HTML file defines three hyperlinks -- the first two are minimal and the third is fully specified, but all work similarly (again, the target script doesn't care). When we visit this file's URL, we see the page shown in [Figure 12-17](#). It's mostly just a page for launching canned calls to the CGI script.

Figure 12-17. Hyperlinks page created by test5c.html



Clicking on this page's second link creates the response page in [Figure 12-18](#). This link invokes the CGI script, with the `name` parameter set to "Tom" and the `language` parameter set to "Python," simply because those parameters and values are hardcoded in the URL listed in the HTML for the second hyperlink. It's exactly as if we had manually typed the line shown at the top of the browser in [Figure 12-18](#).

Figure 12-18. Response page created by test5.cgi (3)



Notice that lots of fields are missing here; the *test5.cgi* script is smart enough to detect and handle missing fields and generate an `unknown` message in the reply page. It's also worth pointing out that we're reusing the Python CGI script again here. The script itself is completely independent of both the user-interface format of the submission page, as well as the technique used to invoke it (from a submitted form or a hardcoded URL). By separating user interface from processing logic, CGI scripts become reusable software components, at least within the context of the CGI environment.

12.3.7.1 Saving CGI script state information

But the real reason for showing this technique is that we're going to use it extensively in the larger case studies in the next two chapters to implement lists of dynamically generated selections that "know" what to do when clicked. Precoded parameters in URLs are a way to retain state information between pages -- they can be used to direct the action of the next script to be run. As such, hyperlinks with such parameters are sometimes known as "smart links."

Normally, CGI scripts run autonomously, with no knowledge of any other scripts that may have run before. That hasn't mattered in our examples so far, but larger systems are usually composed of multiple user interaction steps and many scripts, and we need a way to keep track of information gathered along the way. Generating hardcoded URLs with parameters is one way for a CGI script to pass data to the next script in the application. When clicked, such URL parameters send pre-programmed selection information back to another server-side handler script.

For example, a site that lets you read your email may present you with a list of viewable email messages, implemented in HTML as a list of hyperlinks generated by another script. Each hyperlink might include the name of the message viewer script, along with parameters identifying the selected message number, email server name, and so on -- as much data as is needed to fetch the message associated with a particular link. A retail site may instead serve up a generated list of product links, each of which triggers a hardcoded hyperlink containing the product number, its price, and so on.

In general, there are a variety of ways to pass or retain state information between CGI script executions:

- Hardcoded URL parameters in dynamically generated hyperlinks and embedded in web pages (as discussed here)
- Hidden form input fields that are attached to form data and embedded in web pages, but not displayed on web pages
- HTTP "cookies" that are stored on the client machine and transferred between client and server in HTTP message headers
- General server-side data stores that include databases, persistent object shelves, flat files, and so on

We'll meet most of these mediums in later examples in this chapter and in the two chapters that follow.

12.4 The Hello World Selector

It's now time for something a bit more useful (well, more entertaining, at least). This section presents a program that displays the basic syntax required by various programming languages to print the string "Hello World", the classic language benchmark. To keep this simple, it assumes the string shows up in the standard output stream, not a GUI or web page. It also gives just the output command itself, not the complete programs. The Python version happens to be a complete program, but we won't hold that against its competitors here.

Structurally, the first cut of this example consists of a main page HTML file, along with a Python-coded CGI script that is invoked by a form in the main HTML page. Because no state or database data is stored between user clicks, this is still a fairly simple example. In fact, the main HTML page implemented by [Example 12-16](#) is really just one big pull-down selection list within a form.

Example 12-16. PP2E\Internet\Cgi-Web\Basics\languages.html

```
<html><body>
<title>Languages</title>
<h1>Hello World selector</h1>

<P>This demo shows how to display a "hello world" message in various
programming languages' syntax. To keep this simple, only the output command
is shown (it takes more code to make a complete program in some of these
languages), and only text-based solutions are given (no GUI or HTML
construction logic is included). This page is a simple HTML file; the one
you see after pressing the button below is generated by a Python CGI script
which runs on the server. Pointers:

<UL>
<LI>To see this page's HTML, use the 'View Source' command in your browser.
<LI>To view the Python CGI script on the server,
  <A HREF="languages-src.cgi">click here</A> or
  <A HREF="getfile.cgi?filename=languages.cgi">here</A>.
<LI>To see an alternative version that generates this page dynamically,
  <A HREF="languages2.cgi">click here</A>.
<LI>For more syntax comparisons, visit
  <A HREF="http://www.ionet.net/~timtroyr/funhouse/beer.html">this site</A>.
</UL></P>

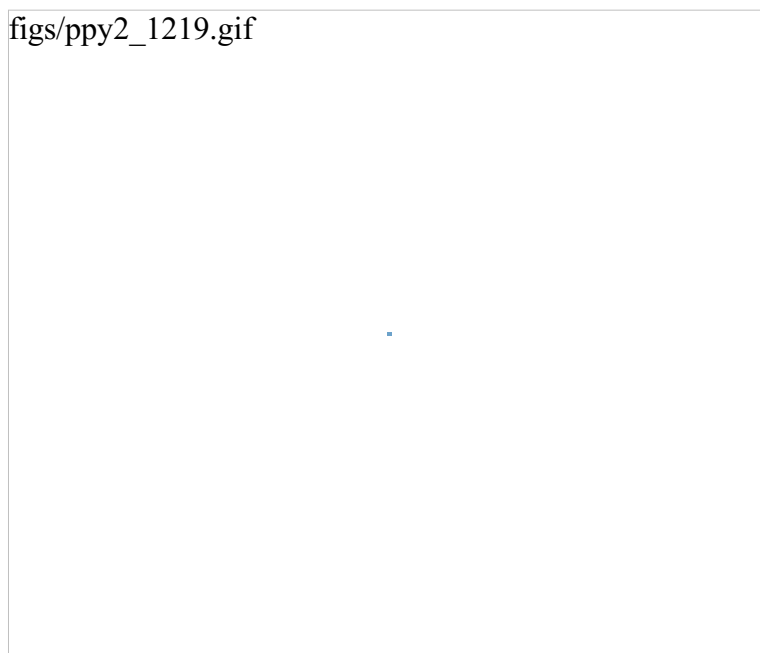
<hr>
<form method=POST action="languages.cgi">
  <P><B>Select a programming language:</B>
  <P><select name=language>
    <option>All
    <option>Python
    <option>Perl
    <option>Tcl
    <option>Scheme
    <option>SmallTalk
    <option>Java
    <option>C
    <option>C++
    <option>Basic
    <option>Fortran
    <option>Pascal
    <option>Other
  </select>
```

```
<P><input type=Submit>
</form>

</body></html>
```

For the moment, let's ignore some of the hyperlinks near the middle of this file; they introduce bigger concepts like file transfers and maintainability that we will explore in the next two sections. When visited with a browser, this HTML file is downloaded to the client and rendered into the new browser page shown in [Figure 12-19](#).

Figure 12-19. The "Hello World" main page



That widget above the Submit button is a pull-down selection list that lets you choose one of the `<option>` tag values in the HTML file. As usual, selecting one of these language names and pressing the Submit button at the bottom (or pressing your Enter key) sends the selected language name to an instance of the server-side CGI script program named in the form's `action` option. [Example 12-17](#) contains the Python script that runs on the server upon submission.

Example 12-17. PP2E\Internet\Cgi-Web\Basics\languages.cgi

```
#!/usr/bin/python
#####
# show hello world syntax for input language name;
# note that it uses r'...' raw strings so that '\n'
# in the table are left intact, and cgi.escape( ) on
# the string so that things like '<<' don't confuse
# browsers--they are translated to valid html code;
# any language name can arrive at this script: e.g.,
# can type "http://starship.python.net/~lutz/Basics
# /languages.cgi?language=Cobol" in any web browser.
# caveats: the languages list appears in both the cgi
# and html files--could import from a single file if
# selection list generated by another cgi script too;
#####
debugme = 0 # 1=test from cmd line
```

```
inputkey = 'language' # input parameter name

hellos = {
    'Python': r" print 'Hello World'           ",
    'Perl': r' print "Hello World\n";         ',
    'Tcl': r' puts "Hello World"              ',
    'Scheme': r' (display "Hello World") (newline) ',
    'SmallTalk': r" 'Hello World' print.      ",
    'Java': r' System.out.println("Hello World"); ',
    'C': r' printf("Hello World\n");         ',
    'C++': r' cout << "Hello World" << endl; ',
    'Basic': r' 10 PRINT "Hello World"        ',
    'Fortran': r" print *, 'Hello World'      ",
    'Pascal': r" WriteLn('Hello World');     "
}

class dummy: # mocked-up input obj
    def __init__(self, str): self.value = str

import cgi, sys
if debugme:
    form = {inputkey: dummy(sys.argv[1])} # name on cmd line
else:
    form = cgi.FieldStorage( ) # parse real inputs

print "Content-type: text/html\n" # adds blank line
print "<TITLE>Languages</TITLE>"
print "<H1>Syntax</H1><HR>"

def showHello(form): # html for one language
    choice = form[inputkey].value
    print "<H3>%s</H3><P><PRE>" % choice
    try:
        print cgi.escape(hellos[choice])
    except KeyError:
        print "Sorry--I don't know that language"
    print "</PRE></P><BR>"

if not form.has_key(inputkey) or form[inputkey].value == 'All':
    for lang in hellos.keys( ):
        mock = {inputkey: dummy(lang)}
        showHello(mock)
else:
    showHello(form)
print '<HR>'
```

And as usual, this script prints HTML code to the standard output stream to produce a response page in the client's browser. There's not much new to speak of in this script, but it employs a few techniques that merit special focus:

Raw strings

Notice the use of raw strings (string constants preceded by an "r" character) in the language syntax dictionary. Recall that raw strings retain \ backslash characters in the string literally, rather than interpreting them as string escape-code introductions. Without them, the \n newline character sequences in some of the language's code snippets would be interpreted by Python as line-feeds, rather than being printed in the HTML reply as \n.

Escaping text embedded in HTML and URLs

This script takes care to format the text of each language's code snippet with the `cgi.escape` utility function. This standard Python utility automatically translates characters that are special in HTML into HTML escape code sequences, such that they are not treated as HTML operators by browsers. Formally, `cgi.escape` translates characters to escape code sequences, according to the standard HTML convention: `<`, `>`, and `&` become `<`, `>`, and `&`. If you pass a second true argument, the double-quote character (`"`) is also translated to `"`.

For example, the `<<` left-shift operator in the C++ entry is translated to `<<` -- a pair of HTML escape codes. Because printing each code snippet effectively embeds it in the HTML response stream, we must escape any special HTML characters it contains. HTML parsers (including Python's standard `htmllib` module) translate escape codes back to the original characters when a page is rendered.

More generally, because CGI is based upon the notion of passing formatted strings across the Net, escaping special characters is a ubiquitous operation. CGI scripts almost always need to escape text generated as part of the reply to be safe. For instance, if we send back arbitrary text input from a user or read from a data source on the server, we usually can't be sure if it will contain HTML characters or not, so we must escape it just in case.

In later examples, we'll also find that characters inserted into URL address strings generated by our scripts may need to be escaped as well. A literal `&` in a URL is special, for example, and must be escaped if it appears embedded in text we insert into a URL. However, URL syntax reserves different special characters than HTML code, and so different escaping conventions and tools must be used. As we'll see later in this chapter, `cgi.escape` implements escape translations in HTML code, but `urllib.quote` (and its relatives) escapes characters in URL strings.

Mocking up form inputs

Here again, form inputs are "mocked up" (simulated), both for debugging and for responding to a request for all languages in the table. If the script's global `debugme` variable is set to a true value, for instance, the script creates a dictionary that is plug-and-play compatible with the result of a `cgi.FieldStorage` call -- its "languages" key references an instance of the dummy mock-up class. This class in turn creates an object that has the same interface as the contents of a `cgi.FieldStorage` result -- it makes an object with a `value` attribute set to a passed-in string.

The net effect is that we can test this script by running it from the system command line: the generated dictionary fools the script into thinking it was invoked by a browser over the Net. Similarly, if the requested language name is "All," the script iterates over all entries in the languages table, making a mocked-up form dictionary for each (as though the user had requested each language in turn). This lets us reuse the existing `showHello` logic to display each language's code in a single page. As always in Python, object interfaces and protocols are what we usually code for, not specific datatypes. The `showHello` function will happily process any object that responds to the syntax `form['language'].value`.^[8]

[8] If you are reading closely, you might notice that this is the second time we've used mock-ups in this chapter (see the earlier `test4.cgi` example). If

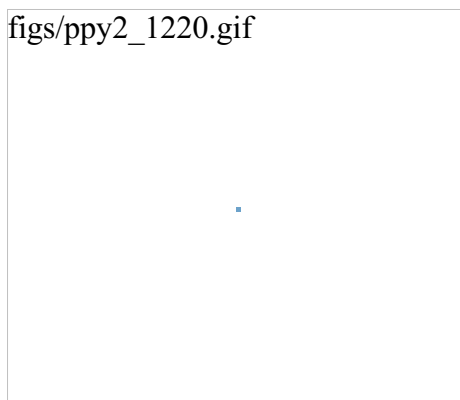
you find this technique generally useful, it would probably make sense to put the `dummy` class, along with a function for populating a form dictionary on demand, into a module so it can be reused. In fact, we will do that in the next section. Even for two-line classes like this, typing the same code the third time around will do much to convince you of the power of code reuse.

Now let's get back to interacting with this program. If we select a particular language, our CGI script generates an HTML reply of the following sort (along with the required content-type header and blank line):

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Scheme</H3><P><PRE>
  (display "Hello World") (newline)
</PRE></P><BR>
<HR>
```

Program code is marked with a `<PRE>` tag to specify preformatted text (the browser won't reformat it like a normal text paragraph). This reply code shows what we get when we pick "Scheme." [Figure 12-20](#) shows the page served up by the script after selecting "Python" in the pull-down selection list.

Figure 12-20. Response page created by `languages.cgi`



Our script also accepts a language name of "All," and interprets it as a request to display the syntax for every language it knows about. For example, here is the HTML that is generated if we set global variable `debugme` to 1 and run from the command line with a single argument, "All." This output is the same as what's printed to the client's browser in response to an "All" request:^[9]

[9] Interestingly, we also get the "All" reply if `debugme` is set to when we run the script from the command line. The `cgi.FieldStorage` call returns an empty dictionary if called outside the CGI environment rather than throwing an exception, so the test for a missing key kicks in. It's likely safer to not rely on this behavior, however.

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python languages.cgi All
```


```
Content-type: text/html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>Perl</H3><P><PRE>
  print "Hello World\n";
</PRE></P><BR>
<H3>SmallTalk</H3><P><PRE>
  'Hello World' print.
</PRE></P><BR>
<H3>Basic</H3><P><PRE>
  10 PRINT "Hello World"
</PRE></P><BR>
<H3>Scheme</H3><P><PRE>
  (display "Hello World") (newline)
</PRE></P><BR>
<H3>Python</H3><P><PRE>
  print 'Hello World'
</PRE></P><BR>
<H3>C++</H3><P><PRE>
  cout &lt;&lt; "Hello World" &lt;&lt; endl;
</PRE></P><BR>
<H3>Pascal</H3><P><PRE>
  WriteLn('Hello World');
</PRE></P><BR>
<H3>Java</H3><P><PRE>
  System.out.println("Hello World");
</PRE></P><BR>
<H3>C</H3><P><PRE>
  printf("Hello World\n");
</PRE></P><BR>
<H3>Tcl</H3><P><PRE>
  puts "Hello World"
</PRE></P><BR>
<H3>Fortran</H3><P><PRE>
  print *, 'Hello World'
</PRE></P><BR>
<HR>
```

Each language is represented here with the same code pattern -- the `showHello` function is called for each table entry, along with a mocked-up form object. Notice the way that C++ code is escaped for embedding inside the HTML stream; this is the `cgi.escape` call's handiwork. When viewed with a browser, the "All" response page is rendered as shown in [Figure 12-21](#).

Figure 12-21. Response page for "all languages" choice

figs/ppy2_1221.gif





12.4.1 Checking for Missing and Invalid Inputs

So far, we've been triggering the CGI script by selecting a language name from the pull-down list in the main HTML page. In this context, we can be fairly sure that the script will receive valid inputs. Notice, though, that there is nothing to prevent a user from passing the requested language name at the end of the CGI script's URL as an explicit parameter, instead of using the HTML page form. For instance, a URL of the form:

```
http://starship.python.net/~lutz/Basics/languages.cgi?language=Python
```

yields the same "Python" response page shown in [Figure 12-20](#).^[10] However, because it's always possible for a user to bypass the HTML file and use an explicit URL, it's also possible that a user could invoke our script with an unknown language name that is not in the HTML file's pull-down list (and so not in our script's table). In fact, the script might be triggered with no language input at all, if someone explicitly types its URL with no parameter at the end.

[10] See the `urllib` module examples in the prior and following chapters for a way to send this URL from a Python script. `urllib` lets programs fetch web pages and invoke remote CGI scripts by building and submitting URL strings like this one, with any required parameters filled in at the end of the string. You could use this module, for instance, to automatically send information to order Python books at an online bookstore from within a Python script, without ever starting a web browser.

To be robust, the script checks for both cases explicitly, as all CGI scripts generally should. For instance, here is the HTML generated in response to a request for the fictitious language "GuiDO":

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>
<H3>GuiDO</H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

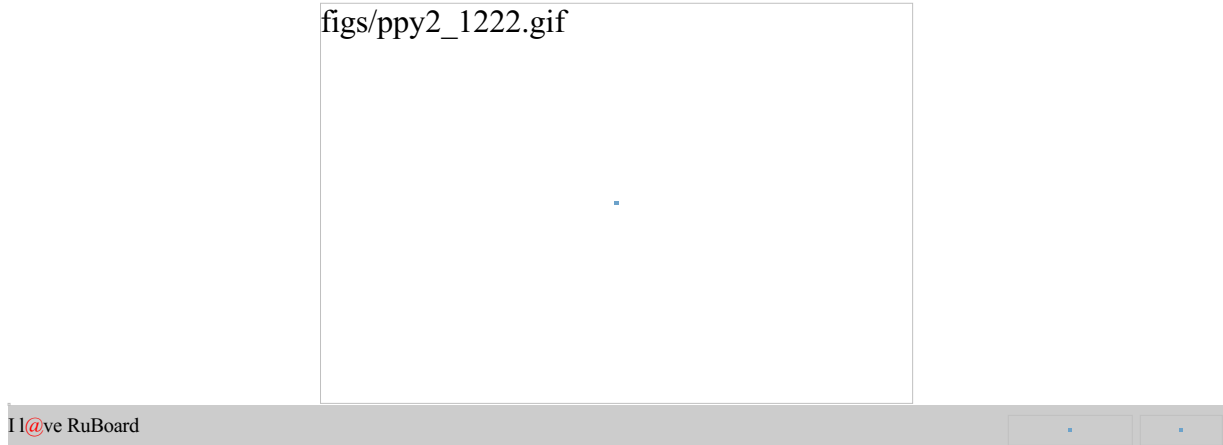
If the script doesn't receive any language name input, it simply defaults to the "All" case. If we didn't detect these cases, chances are that our script would silently die on a Python exception and leave the user with a mostly useless half-complete page or with a default error page (we didn't assign `stderr` to `stdout` here, so no Python error message would be displayed). In pictures, [Figure 12-22](#) shows the page generated if the script is invoked with an explicit URL like this:

```
http://starship.python.net/~lutz/Basics/languages.cgi?language=COBOL
```

To test this error case, the pull-down list includes an "Unknown" name, which produces a similar error page reply. Adding code to the script's table for the COBOL "Hello World" program is left as an exercise for the reader.

Figure 12-22. Response page for unknown language

figs/ppy2_1222.gif



12.5 Coding for Maintainability

Let's step back from coding details for just a moment to gain some design perspective. As we've seen, Python code, by and large, automatically lends itself to systems that are easy to read and maintain; it has a simple syntax that cuts much of the clutter of other tools. On the other hand, coding styles and program design can often impact maintainability as much as syntax. For example, the "Hello World" selector pages earlier in this chapter work as advertised, and were very easy and fast to throw together. But as currently coded, the languages selector suffers from substantial maintainability flaws.

Imagine, for instance, that you actually take me up on that challenge posed at the end of the last section, and you attempt to add another entry for COBOL. If you add COBOL to the CGI script's table, you're only half done: the list of supported languages lives redundantly in two places -- in the HTML for the main page as well as the script's syntax dictionary. Changing one does not change the other. More generally, there are a handful of ways that this program might fail the scrutiny of a rigorous code review:

Selection list

As just mentioned, the list of languages supported by this program lives in two places: the HTML file and the CGI script's table.

Field name

The field name of the input parameter, "language," is hardcoded into both files, as well. You might remember to change it in the other if you change it in one, but you might not.

Form mock ups

We've redundantly coded classes to mock-up form field inputs twice in this chapter already; the "dummy" class here is clearly a mechanism worth reusing.

HTML Code

HTML embedded in and generated by the script is sprinkled throughout the program in `print` statements, making it difficult to implement broad web page layout changes.

This is a short example, of course, but issues of redundancy and reuse become more acute as your scripts grow larger. As a rule of thumb, if you find yourself changing multiple source files to modify a single behavior, or if you notice that you've taken to writing programs by cut-and-paste copying of existing code, then it's probably time to think about more rational program structures. To illustrate coding styles and practices that are more friendly to maintainers, let's rewrite this example to fix all of these weaknesses in a single mutation.

12.5.1 Step 1: Sharing Objects Between Pages

We can remove the first two maintenance problems listed above with a simple transformation; the trick is to generate the main page dynamically, from an executable script, rather than from a precoded HTML file. Within a script, we can import the input field name and selection list value from a common Python module file, shared by the main and reply page generation scripts. Changing the selection list or field name in the common module changes both clients automatically. First, we move shared objects to a common module file, as shown in [Example 12-18](#).

Example 12-18. PP2E\Internet\Cgi-Web\Basics\languages2common.py

```
#####
# common objects shared by main and reply page scripts;
# need change only this file to add a new language.
#####

inputkey = 'language'                                # input parameter name

hellos = {
    'Python':    r" print 'Hello World'                ",
    'Perl':      r' print "Hello World\n";            ',
    'Tcl':       r' puts "Hello World"                ',
    'Scheme':    r' (display "Hello World") (newline) ',
    'SmallTalk': r" 'Hello World' print.              ",
    'Java':      r' System.out.println("Hello World"); ',
    'C':         r' printf("Hello World\n");          ',
    'C++':       r' cout << "Hello World" << endl;    ',
    'Basic':     r' 10 PRINT "Hello World"            ',
    'Fortran':   r" print *, 'Hello World'            ",
    'Pascal':    r" WriteLn('Hello World');          "
}
}
```

Module `languages2common` contains all the data that needs to agree between pages: the field name, as well as the syntax dictionary. The `hellos` syntax dictionary isn't quite HTML code, but its keys list can be used to generate HTML for the selection list on the main page dynamically. Next, in [Example 12-19](#), we recode the main page as an executable script, and populate the response HTML with values imported from the common module file in the previous example.

Example 12-19. PP2E\Internet\Cgi-Web\Basics\languages2.cgi

```
#!/usr/bin/python
#####
# generate html for main page dynamically from an executable
# Python script, not a pre-coded HTML file; this lets us
# import the expected input field name and the selection table
# values from a common Python module file; changes in either
# now only have to be made in one place, the Python module file;
#####

REPLY = """Content-type: text/html

<html><body>
<title>Languages2</title>
<h1>Hello World selector</h1>
<P>Similar to file <a href="languages.html">languages.html</a>, but
this page is dynamically generated by a Python CGI script, using
selection list and input field names imported from a common Python
module on the server. Only the common module must be maintained as
new languages are added, because it is shared with the reply script.

To see the code that generates this page and the reply, click
```

```
<a href="getfile.cgi?filename=languages2.cgi">here</a>,
<a href="getfile.cgi?filename=languages2reply.cgi">here</a>,
<a href="getfile.cgi?filename=languages2common.py">here</a>, and
<a href="getfile.cgi?filename=formMockup.py">here</a>.</P>
<hr>
<form method=POST action="languages2reply.cgi">
  <P><B>Select a programming language:</B>
  <P><select name=%s>
    <option>All
    %s
    <option>Other
  </select>
  <P><input type=Submit>
</form>
</body></html>
"""

import string
from languages2common import hellos, inputkey

options = []
for lang in hellos.keys( ):
    options.append('<option>' + lang)          # wrap table keys in html code
options = string.join(options, '\n\t')
print REPLY % (inputkey, options)           # field name and values from module
```

Here again, ignore the `getfile` hyperlinks in this file for now; we'll learn what they mean in the next section. You should notice, though, that the HTML page definition becomes a printed Python string here (named `REPLY`), with `%s` format targets where we plug in values imported from the `common` module.^[11] It's otherwise similar to the original HTML file's code; when we visit this script's URL, we get a similar page, shown in [Figure 12-23](#). But this time, the page is generated by running a script on the server that populates the pull-down selection list from the keys list of the `common` syntax table.

[11] The HTML code template could be loaded from an external text file, too, but external text files are no more easily changed than Python scripts. In fact, Python scripts are text files, and this is a major feature of the language: it's usually easy to change the Python scripts of an installed system onsite, without re-compile or re-link steps.

Figure 12-23. Alternative main page made by `languages2.cgi`



12.5.2 Step 2: A Reusable Form Mock-up Utility

Moving the languages table and input field name to a module file solves the first two maintenance problems we noted. But if we want to avoid writing a dummy field mock-up class in every CGI script we write, we need to do something more. Again, it's merely a matter of exploiting the Python module's affinity for code reuse: let's move the dummy class to a utility module, as in [Example 12-20](#).

Example 12-20. PP2E\Internet\Cgi-Web\Basics\formMockup.py

```
#####
# Tools for simulating the result of a cgi.FieldStorage( )
# call; useful for testing CGI scripts outside the web
#####

import types

class FieldMockup:                                # mocked-up input object
    def __init__(self, str):
        self.value = str

def formMockup(**kwargs):                         # pass field=value args
    mockup = {}                                  # multi-choice: [value,...
    for (key, value) in kwargs.items( ):         # simple fields have .valu
        if type(value) is not types.ListType:
            mockup[key] = FieldMockup(str(value))
        else:                                    # multi-choice have list
            mockup[key] = []                    # to do: file upload field
            for pick in value:
                mockup[key].append(FieldMockup(pick))
    return mockup

def selftest( ):
    # use this form if fields can be hard-coded
    form = formMockup(name='Bob', job='hacker', food=['Spam', 'eggs', 'ham'])
    print form['name'].value
    print form['job'].value
    for item in form['food']:
        print item.value,
    # use real dict if keys are in variables or computed
    print
    form = {'name':FieldMockup('Brian'), 'age':FieldMockup(38)}
    for key in form.keys( ):
        print form[key].value

if __name__ == '__main__': selftest( )
```

By placing our mock-up class in this module, *formMockup.py*, it automatically becomes a reusable tool, and may be imported by any script we care to write.^[12] For readability, the dummy field simulation class has been renamed `FieldMockup` here. For convenience, we've also added a `formMockup` utility function that builds up an entire form dictionary from passed-in keyword arguments. Assuming you can hardcode the names of the form to be faked, the mock-up can be created in a single call. This module includes a self-test function invoked when the file is run from the command line, which demonstrates how its exports are used. Here is its test output, generated by making and querying two form mock-up objects:

[12] This assumes of course that this module can be found on the Python module

THIS ASSUMES, OF COURSE, THAT THE MODULE CAN BE FOUND ON THE PYTHON MODULE search path when those scripts are run. See the search path discussion earlier in this chapter. Since Python searches the current directory for imported modules by default, this always works without `sys.path` changes if all of our files are in our main web directory.

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python formMockup.py
Bob
hacker
Spam eggs ham
38
Brian
```

Since the mock-up now lives in a module, we can reuse it any time we want to test a CGI script offline. To illustrate, the script in [Example 12-21](#) is a rewrite of the `test5.cgi` example we saw earlier, using the form mock-up utility to simulate field inputs. If we had planned ahead, we could have tested this script like this without even needing to connect to the Net.

Example 12-21. PP2E\Internet\Cgi-Web\Basics\test5_mockup.cgi

```
#!/usr/bin/python
#####
# run test5 logic with formMockup instead of cgi.FieldStorage( )
# to test: python test5_mockup.cgi > temp.html, and open temp.html
#####

from formMockup import formMockup
form = formMockup(name='Bob',
                  shoesize='Small',
                  language=['Python', 'C++', 'HTML'],
                  comment='ni, Ni, NI')

# rest same as original, less form assignment
```

Running this script from a simple command line shows us what the HTML response stream will look like:


```
C:\...\PP2E\Internet\Cgi-Web\Basics>python test5_mockup.cgi
Content-type: text/html

<TITLE>test5.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is Bob</H4>
<H4>You wear rather Small shoes</H4>
<H4>Your current job: (unknown)</H4>
<H4>You program in Python and C++ and HTML</H4>
<H4>You also said:</H4>
<P>ni, Ni, NI</P>
<HR>
```

Running it live yields the page in [Figure 12-24](#). Field inputs here are hardcoded, similar in spirit to the `test5` extension that embedded input parameters at the end of hyperlink URLs. Here, they come from form mock-up objects created in the reply script that cannot be changed without editing the script. Because Python code runs immediately, though, modifying a Python script during the debug cycle goes as quickly as you can type.

Figure 12-24. A response page with simulated inputs

figs/ppy2_1224.gif



12.5.3 Step 3: Putting It All Together -- A New Reply Script

There's one last step on our path to software maintenance nirvana: we still must recode the reply page script itself, to import data that was factored out to the common module and import the reusable form mock-up module's tools. While we're at it, we move code into functions (in case we ever put things in this file that we'd like to import in another script), and all HTML code to triple-quoted string blocks (see [Example 12-22](#)). Changing HTML is generally easier when it has been isolated in single strings like this, rather than being sprinkled throughout a program.

Example 12-22. PP2E\Internet\Cgi-Web\Basics\languages2reply.cgi

```
#!/usr/bin/python
#####
# for easier maintenance, use html template strings, get
# the language table and input key from common module file,
# and get reusable form field mockup utilities module.
#####

import cgi, sys
from formMockup import FieldMockup           # input field simulator
from languages2common import hellos, inputkey # get common table, name
debugme = 0

hdrhtml = """Content-type: text/html\n
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>"""

langhtml = """
<H3>%s</H3><P><PRE>
%s
</PRE></P><BR>"""

def showHello(form):                          # html for one language
    choice = form[inputkey].value             # escape lang name too
    try:
        print langhtml % (cgi.escape(choice),
                           cgi.escape(hellos[choice]))
    except KeyError:
        print langhtml % (cgi.escape(choice),
```



```
                "Sorry--I don't know that language")

def main( ):
    if debugme:
        form = {inputkey: FieldMockup(sys.argv[1])} # name on cmd line
    else:
        form = cgi.FieldStorage( )                # parse real inputs

    print hdrhtml
    if not form.has_key(inputkey) or form[inputkey].value == 'All':
        for lang in hellos.keys( ):
            mock = {inputkey: FieldMockup(lang)}
            showHello(mock)
    else:
        showHello(form)
    print '<HR>'

if __name__ == '__main__': main( )
```

When global `debugme` is set to 1, the script can be tested offline from a simple command line as before:

```
C:\...\PP2E\Internet\Cgi-Web\Basics>python languages2reply.cgi Python
Content-type: text/html

<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>

<H3>Python</H3><P><PRE>
  print 'Hello World'
</PRE></P><BR>
<HR>
```

When run online, we get the same reply pages we saw for the original version of this example (we won't repeat them here again). This transformation changed the program's architecture, not its user interface.

Most of the code changes in this version of the reply script are straightforward. If you test-drive these pages, the only differences you'll find are the URLs at the top of your browser (they're different files, after all), extra blank lines in the generated HTML (ignored by the browser), and potentially different ordering of language names in the main page's pull-down selection list.

This selection list ordering difference arises because this version relies on the order of the Python dictionary's keys list, not on a hardcoded list in an HTML file. Dictionaries, you'll recall, arbitrarily order entries for fast fetches; if you want the selection list to be more predictable, simply sort the keys list before iterating over it using the list `sort` method.

Faking Inputs with Shell Variables

If you know what you're doing, you can sometimes also test CGI scripts from the command line by setting the same environment variables that HTTP servers set, and then launching your script. For example, we can pretend to be a web server by storing input parameters in the `QUERY_STRING` environment variable, using the same syntax we employ at the end of a URL string after the `?`:

```
$ setenv QUERY_STRING "name=Mel&job=trainer,+writer"
$ python test5.cgi
```

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

```
<TITLE>test5.cgi<?TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is Mel</H4>
<H4>You wear rather (unknown) shoes</H4>
<H4>Your current job: trainer, writer</H4>
<H4>You program in (unknown)</H4>
<H4>You also said:</H4>
<P>(unknown)</P>
<HR>
```

Here, we mimic the effects of a GET style form submission or explicit URL. HTTP servers place the query string (parameters) in the shell variable `QUERY_STRING`. Python's `cgi` module finds them there as though they were sent by a browser. POST-style inputs can be simulated with shell variables, too, but it's more complex -- so much so that you're likely best off not learning how. In fact, it may be more robust in general to mock-up inputs with Python objects (e.g., as in *formMockup.py*). But some CGI scripts may have additional environment or testing constraints that merit unique treatment.

12.6 More on HTML and URL Escapes

Perhaps the most subtle change in the last section's rewrite is that, for robustness, this version also calls `cgi.escape` for the language name, not just for the language's code snippet. It's unlikely but not impossible that someone could pass the script a language name with an embedded HTML character. For example, a URL like:

```
http://starship.python.net/~lutz/Basics/languages2reply.cgi?language=a<b
```

embeds a `<` in the language name parameter (the name is `a<b`). When submitted, this version uses `cgi.escape` to properly translate the `<` for use in the reply HTML, according to the standard HTML escape conventions discussed earlier:

```
<TITLE>Languages</TITLE>
<H1>Syntax</H1><HR>

<H3>a&lt;b</H3><P><PRE>
Sorry--I don't know that language
</PRE></P><BR>
<HR>
```

The original version doesn't escape the language name, such that the embedded `<b` is interpreted as an HTML tag (which may make the rest of the page render in bold font!). As you can probably tell by now, text escapes are pervasive in CGI scripting -- even text that you may think is safe must generally be escaped before being inserted into the HTML code in the reply stream.

12.6.1 URL Escape Code Conventions

Notice, though, that while it's wrong to embed an unescaped `<` in the HTML code reply, it's perfectly okay to include it literally in the earlier URL string used to trigger the reply. In fact, HTML and URLs define completely different characters as special. For instance, although `&` must be escaped as `&` inside HTML code, we have to use other escaping schemes to code a literal `&` within a URL string (where it normally separates parameters). To pass a language name like `a&b` to our script, we have to type the following URL:

```
http://starship.python.net/~lutz/Basics/languages2reply.cgi?language=a%26b
```

Here, `%26` represents `&` -- the `&` is replaced with a `%` followed by the hexadecimal value (0x26) of its ASCII code value (38). By URL standard, most nonalphanumeric characters are supposed to be translated to such escape sequences, and spaces are replaced by `+` signs. Technically, this convention is known as the *application/x-www-form-urlencoded* query string format, and it's part of the magic behind those bizarre URLs you often see at the top of your browser as you surf the Web.

12.6.2 Python HTML and URL Escape Tools

If you're like me, you probably don't have the hexadecimal value of the ASCII code for `&` committed to memory. Luckily, Python provides tools that automatically implement URL escapes, just as `cgi.escape` does for HTML escapes. The main thing to keep in mind is that

HTML code and URL strings are written with entirely different syntax, and so they employ distinct escaping conventions. Web users don't generally care, unless they need to type complex URLs explicitly (browsers handle most escape code details internally). But if you write scripts that must generate HTML or URLs, you need to be careful to escape characters that are reserved in either syntax.

Because HTML and URLs have different syntaxes, Python provides two distinct sets of tools for escaping their text. In the standard Python library:

- `cgi.escape` escapes text to be embedded in HTML.
- `urllib.quote` and `quote_plus` escape text to be embedded in URLs.

The `urllib` module also has tools for undoing URL escapes (`unquote`, `unquote_plus`), but HTML escapes are undone during HTML parsing at large (`htmllib`). To illustrate the two escape conventions and tools, let's apply each toolset to a few simple examples.

12.6.3 Escaping HTML Code

As we saw earlier, `cgi.escape` translates code for inclusion within HTML. We normally call this utility from a CGI script, but it's just as easy to explore its behavior interactively:

```
>>> import cgi
>>> cgi.escape('a < b > c & d "spam"', 1)
'a &lt; b &gt; c &amp; d &quot;spam&quot;'
```

```
>>> s = cgi.escape("<2 <b>hello</b>")
>>> s
'1&lt;2 &lt;b&gt;hello&lt;/b&gt;'
```

Python's `cgi` module automatically converts characters that are special in HTML syntax according to the HTML convention. It translates `<`, `>`, `&`, and with an extra true argument, `"`, into escape sequences of the form `&X;`, where the `X` is a mnemonic that denotes the original character. For instance, `<` stands for the "less than" operator (`<`) and `&` denotes a literal ampersand (`&`).

There is no un-escaping tool in the CGI module, because HTML escape code sequences are recognized within the context of an HTML parser, like the one used by your web browser when a page is downloaded. Python comes with a full HTML parser, too, in the form of standard module `htmllib`, which imports and specializes tools in module `sgmlib` (HTML is a kind of SGML syntax). We won't go into details on the HTML parsing tools here (see the library manual for details), but to illustrate how escape codes are eventually undone, here is the SGML module at work reading back the last output above:

```
>>> from sgmlib import TestSGMLParser
>>> p = TestSGMLParser(1)
>>> s
'1&lt;2 &lt;b&gt;hello&lt;/b&gt;'
```

```
>>> for c in s:
...     p.feed(c)
...
>>> p.close( )
data: '<2 <b>hello</b>'
```

12.6.4 Escaping URLs

By contrast, URLs reserve other characters as special and must adhere to different escape conventions. Because of that, we use different Python library tools to escape URLs for transmission. Python's `urllib` module provides two tools that do the translation work for us: `quote`, which implements the standard `%XX` hexadecimal URL escape code sequences for most nonalphanumeric characters, and `quote_plus`, which additionally translates spaces to `+ plus signs`. The `urllib` module also provides functions for unescaping quoted characters in a URL string: `unquote` undoes `%XX` escapes, and `unquote_plus` also changes plus signs back to spaces. Here is the module at work, at the interactive prompt:

```
>>> import urllib
>>> urllib.quote("a & b #! c")
'a%20%26%20b%20%23%21%20c'

>>> urllib.quote_plus("C:\stuff\spam.txt")
'C%3a%5cstuff%5cspam.txt'

>>> x = urllib.quote_plus("a & b #! c")
>>> x
'a+%26+b+%23%21+c'

>>> urllib.unquote_plus(x)
'a & b #! c'
```

URL escape sequences embed the hexadecimal values of non-safe characters following a `%` sign (usually, their ASCII codes). In `urllib`, non-safe characters are usually taken to include everything except letters, digits, a handful of safe special characters (any of `_`, `.`, `-`), and `/` by default). You can also specify a string of safe characters as an extra argument to the `quote` calls to customize the translations; the argument defaults to `/`, but passing an empty string forces `/` to be escaped:

```
>>> urllib.quote_plus("uploads/index.txt")
'uploads/index.txt'

>>> urllib.quote_plus("uploads/index.txt", '')
'uploads%2findex.txt'
```

Note that Python's `cgi` module also translates URL escape sequences back to their original characters and changes `+ signs` to spaces during the process of extracting input information. Internally, `cgi.FieldStorage` automatically calls `urllib.unquote` if needed to parse and unescape parameters passed at the end of URLs (most of the translation happens in `cgi.parse_qs`). The upshot is that CGI scripts get back the original, unescaped URL strings, and don't need to unquote values on their own. As we've seen, CGI scripts don't even need to know that inputs came from a URL at all.

12.6.5 Escaping URLs Embedded in HTML Code

But what do we do for URLs inside HTML? That is, how do we escape when we generate and embed text inside a URL, which is itself embedded inside generated HTML code? Some of our earlier examples used hardcoded URLs with appended input parameters inside `<A HREF>` hyperlink tags; file `languages2.cgi`, for instance, prints HTML that includes a URL:

```
<a href="getfile.cgi?filename=languages2.cgi">
```

Because the URL here is embedded in HTML, it must minimally be escaped according to HTML conventions (e.g., any `<` characters must become `<`), and any spaces should be translated to `+` signs. A `cgi.escape(url)` call, followed by a `string.replace(url, " ", "+")` would take us this far, and would probably suffice for most cases.

That approach is not quite enough in general, though, because HTML escaping conventions are not the same as URL conventions. To robustly escape URLs embedded in HTML code, you should instead call `urllib.quote_plus` on the URL string before adding it to the HTML text. The escaped result also satisfies HTML escape conventions, because `urllib` translates more characters than `cgi.escape`, and the `%` in URL escapes is not special to HTML.

But there is one more wrinkle here: you also have to be careful with `&` characters in URL strings that are embedded in HTML code (e.g., within `<A>` hyperlink tags). Even if parts of the URL string are URL-escaped, when more than one parameter is separated by a `&`, the `&` separator might also have to be escaped as `&`; according to HTML conventions. To see why, consider the following HTML hyperlink tag:

```
<A HREF="file.cgi?name=a&job=b&amp=c&sect=d&lt=e">hello</a>
```

When rendered in most browsers I've tested, this URL link winds up looking incorrectly like this (the "S" character is really a non-ASCII section marker):

```
file.cgi?name=a&job=b&c&S=d<=e
```

The first two parameters are retained as expected (`name=a`, `job=b`), because `name` is not preceded with an `&`, and `&job` is not recognized as a valid HTML character escape code. However, the `&`, `§`, and `<` parts are interpreted as special characters, because they do name valid HTML escape codes. To make this work as expected, the `&` separators should be escaped:

```
<A HREF="file.cgi?name=a&amp;job=b&amp;amp=c&amp;sect=d&amp;lt=e">hello</a>
```

Browsers render this fully escaped link as expected:

```
file.cgi?name=a&job=b&amp=c&sect=d&lt=e
```

The moral of this story is that unless you can be sure that the names of all but the leftmost URL query parameters embedded in HTML are not the same as the name of any HTML character escape code like `amp`, you should generally run the entire URL through `cgi.escape` after escaping its parameter names and values with `urllib.quote_plus`:

```
>>> import cgi
>>> cgi.escape('file.cgi?name=a&job=b&amp=c&sect=d&lt=e')
'file.cgi?name=a&amp;job=b&amp;amp=c&amp;sect=d&amp;lt=e'
```

Having said that, I should add that some examples in this book do not escape `&` URL separators embedded within HTML simply because their URL parameter names are known to not conflict with HTML escapes. This is not, however, the most general solution; when in doubt, escape much and often.

"Always Look on the Bright Side of Life"

Lest these formatting rules sound too clumsy (and send you screaming into the night!), note that the HTML and URL escaping conventions are imposed by the Internet itself, not by Python. (As we've seen, Python has a different mechanism for escaping special characters in string constants with backslashes.) These rules stem from the fact that the Web is based upon the notion of shipping formatted strings around the planet, and they were surely influenced by the tendency of different interest groups to develop very different notations.

You can take heart, though, in the fact that you often don't need to think in such cryptic terms; when you do, Python automates the translation process with library tools. Just keep in mind that any script that generates HTML or URLs dynamically probably needs to call Python's escaping tools to be robust. We'll see both the HTML and URL escape tool sets employed frequently in later examples in this chapter and the next two. In [Chapter 15](#), we'll also meet systems such as Zope that aim to get rid of some of the low-level complexities that CGI scripters face. And as usual in programming, there is no substitute for brains; amazing technologies like the Internet come at a cost in complexity.

12.7 Sending Files to Clients and Servers

It's time to explain a bit of HTML code we've been keeping in the shadows. Did you notice those hyperlinks on the language selector example's main page for showing the CGI script's source code? Normally, we can't see such script source code, because accessing a CGI script makes it execute (we can see only its HTML output, generated to make the new page). The script in [Example 12-23](#), referenced by a hyperlink in the main `language.html` page, works around that by opening the source file and sending its text as part of the HTML response. The text is marked with `<PRE>` as pre-formatted text, and escaped for transmission inside HTML with `cgi.escape`.

Example 12-23. PP2E\Internet\Cgi-Web\Basics\languages-src.cgi

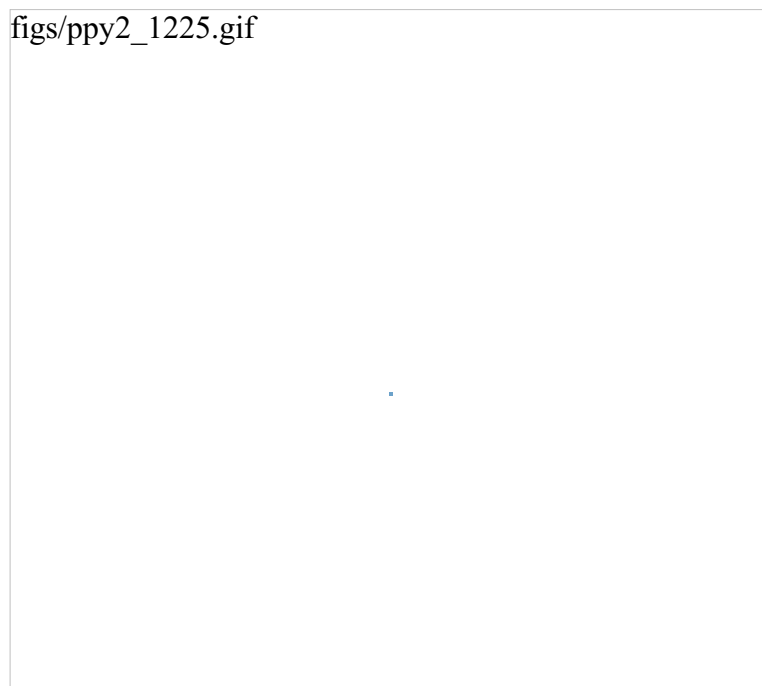
```
#!/usr/bin/python
#####
# Display languages.cgi script code without running it.
#####

import cgi
filename = 'languages.cgi'

print "Content-type: text/html\n"          # wrap up in html
print "<TITLE>Languages</TITLE>"
print "<H1>Source code: '%s'</H1>" % filename
print '<HR><PRE>'
print cgi.escape(open(filename).read( ))
print '</PRE><HR>'
```

When we visit this script on the Web via the hyperlink or a manually typed URL, the script delivers a response to the client that includes the text of the CGI script source file. It appears as in [Figure 12-25](#).

Figure 12-25. Source code viewer page





Note that here, too, it's crucial to format the text of the file with `cgi.escape`, because it is embedded in the HTML code of the reply. If we don't, any characters in the text that mean something in HTML code are interpreted as HTML tags. For example, the C++ `<` operator character within this file's text may yield bizarre results if not properly escaped. The `cgi.escape` utility converts it to the standard sequence `<`; for safe embedding.

12.7.1 Displaying Arbitrary Server Files on the Client

Almost immediately after writing the languages source code viewer script in the previous example, it occurred to me that it wouldn't be much more work, and would be much more useful to write a generic version -- one that could use a passed-in filename to display any file on the site. It's a straightforward mutation on the server side; we merely need to allow a filename to be passed in as an input. The *getfile.cgi* Python script in [Example 12-24](#) implements this generalization. It assumes the filename is either typed into a web page form or appended to the end of the URL as a parameter. Remember that Python's `cgi` module handles both cases transparently, so there is no code in this script that notices any difference.

Example 12-24. PP2E\Internet\Cgi-Web\Basics\getfile.cgi

```
#!/usr/bin/python
#####
# Display any cgi (or other) server-side file without running it.
# The filename can be passed in a URL param or form field; e.g.,
# http://server/~lutz/Basics/getfile.cgi?filename=somefile.cgi.
# Users can cut-and-paste or "View source" to save file locally.
# On IE, running the text/plain version (formatted=0) sometimes
# pops up Notepad, but end-of-lines are not always in DOS format;
# Netscape shows the text correctly in the browser page instead.
# Sending the file in text/html mode works on both browsers--text
# is displayed in the browser response page correctly. We also
# check the filename here to try to avoid showing private files;
# this may or may not prevent access to such files in general.
#####

import cgi, os, sys
formatted = 1 # 1=wrap text in html
privates = ['./PyMailCgi/secret.py'] # don't show these

html = """
<html><title>Getfile response</title>
<h1>Source code for: '%s'</h1>
<hr>
<pre>%s</pre>
<hr></html>"""

def restricted(filename):
    for path in privates:
        if os.path.samefile(path, filename): # unify all paths by os.stat
            return 1 # else returns None=false

try:
    form = cgi.FieldStorage( )
    filename = form['filename'].value # url param or form field
except:
    filename = 'getfile.cgi' # else default filename
```

```
try:
    assert not restricted(filename)                # load unless private
    filetext = open(filename).read( )
except AssertionError:
    filetext = '(File access denied)'
except:
    filetext = '(Error opening file: %s)' % sys.exc_value

if not formatted:
    print "Content-type: text/plain\n"           # send plain text
    print filetext                               # works on NS, not IE
else:
    print "Content-type: text/html\n"           # wrap up in html
    print html % (filename, cgi.escape(filetext))
```

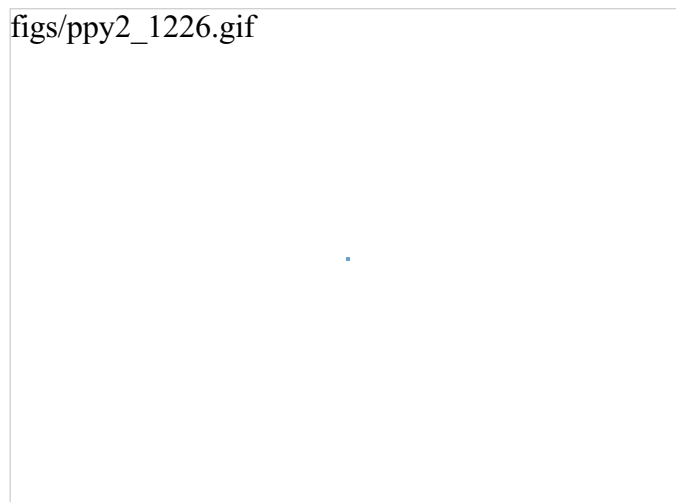
This Python server-side script simply extracts the filename from the parsed CGI inputs object, and reads and prints the text of the file to send it to the client browser. Depending on the `formatted` global variable setting, it either sends the file in plain text mode (using `text/plain` in the response header) or wrapped up in an HTML page definition (`text/html`).

Either mode (and others) works in general under most browsers, but Internet Explorer doesn't handle the plain text mode as gracefully as Netscape -- during testing, it popped up the Notepad text editor to view the downloaded text, but end-of-line characters in Unix format made the file appear as one long line. (Netscape instead displays the text correctly in the body of the response web page itself.) HTML display mode works more portably with current browsers. More on this script's restricted file logic in a moment.

Let's launch this script by typing its URL at the top of a browser, along with a desired filename appended after the script's name. [Figure 12-26](#) shows the page we get by visiting this URL:

<http://starship.python.net/~lutz/Basics/getfile.cgi?filename=languages-src.cgi>

Figure 12-26. Generic source code viewer page



The body of this page shows the text of the server-side file whose name we passed at the end of the URL; once it arrives, we can view its text, cut-and-paste to save it in a file on the client, and so on. In fact, now that we have this generalized source code viewer, we could replace the hyperlink to script `languages-src.cgi` in `language.html`, with a URL of this form:

<http://starship.python.net/~lutz/Basics/getfile.cgi?filename=languages.cgi>

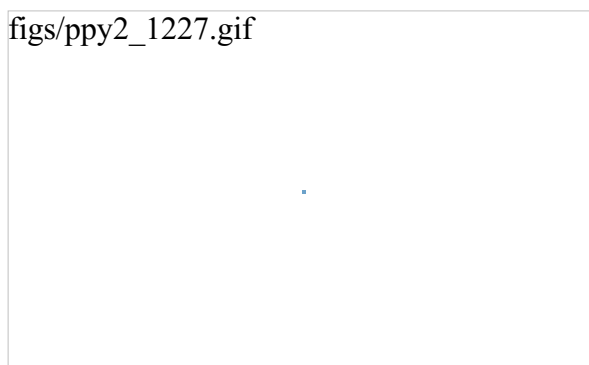
For illustration purposes, the main HTML page in [Example 12-16](#) has links both to the original source code display script, as well as to the previous URL (less the server and directory paths, since the HTML file and `getfile` script live in the same place). Really, URLs like these are direct calls (albeit, across the Web) to our Python script, with filename parameters passed explicitly. As we've seen, parameters passed in URLs are treated the same as field inputs in forms; for convenience, let's also write a simple web page that allows the desired file to be typed directly into a form, as shown in [Example 12-25](#).

Example 12-25. PP2E\Internet\Cgi-Web\Basics\getfile.html

```
<html><title>Getfile: download page</title>
<body>
<form method=get action="getfile.cgi">
  <h1>Type name of server file to be viewed</h1>
  <p><input type=text size=50 name=filename>
  <p><input type=submit value=Download>
</form>
<hr><a href="getfile.cgi?filename=getfile.cgi">View script code</a>
</body></html>
```

[Figure 12-27](#) shows the page we receive when we visit this file's URL. We need to type only the filename in this page, not the full CGI script address.

Figure 12-27. source code viewer selection page



When we press this page's Download button to submit the form, the filename is transmitted to the server, and we get back the same page as before, when the filename was appended to the URL (see [Figure 12-26](#)). In fact, the filename will be appended to the URL here, too; the `get` method in the form's HTML instructs the browser to append the filename to the URL, exactly as if we had done so manually. It shows up at the end of the URL in the response page's address field, even though we really typed it into a form.^[13]

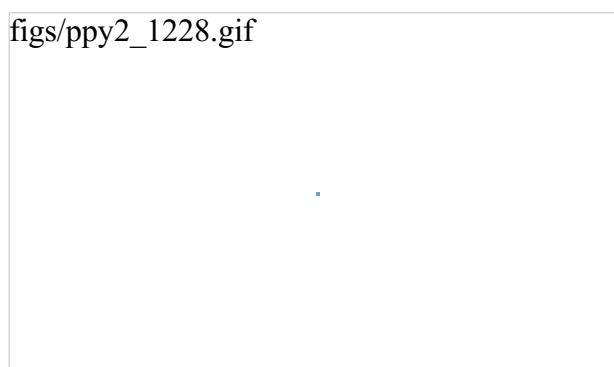
[13] You may notice one difference in the response pages produced by the form and an explicitly typed URL: for the form, the value of the "filename" parameter at the end of the URL in the response may contain URL escape codes for some characters in the file path you typed. Browsers automatically translate some non-ASCII characters into URL escapes (just like `urllib.quote`). URL escapes are discussed earlier in this chapter; we'll see an example of this automatic browser escaping at work in a moment.

12.7.1.1 Handling private files and errors

As long as CGI scripts have permission to open the desired server-side file, this script can be used to view and locally save any file on the server. For instance, [Figure 12-28](#) shows the page we're served after asking for file path `../PyMailCgi/index.html` -- an HTML text file in another application's subdirectory, nested within the parent directory of this script.^[14] Users can specify both relative and absolute paths to reach a file -- any path syntax the server understands will do.

[14] PyMailCgi is described in the next chapter. If you're looking for source files for PyErrata (also in the next chapter), use a path like `../PyErrata/xxx`. In general, the top level of the book's web site corresponds to the top level of the *Internet/Cgi-Web* directory in the examples on the book's CD-ROM (see <http://examples.oreilly.com/python2>); `getfile` runs in subdirectory *Basics*.

Figure 12-28. Viewing files with relative paths



More generally, this script will display any file path for which the user "nobody" (the username under which CGI scripts usually run) has read access. Just about every server-side file used in web applications will, or else they wouldn't be accessible from browsers in the first place. That makes for a flexible tool, but it's also potentially dangerous. What if we don't want users to be able to view some files on the server? For example, in the next chapter, we will implement an encryption module for email account passwords. Allowing users to view that module's source code would make encrypted passwords shipped over the Net much more vulnerable to cracking.

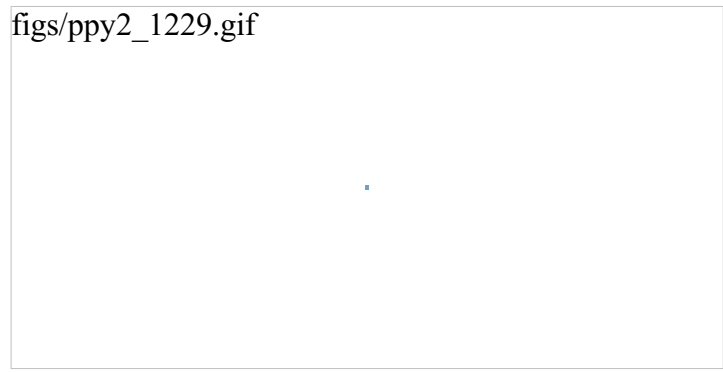
To minimize this potential, the `getfile` script keeps a list, `privates`, of restricted filenames, and uses the `os.path.samefile` built-in to check if a requested filename path points to one of the names on `privates`. The `samefile` call checks to see if the `os.stat` built-in returns the same identifying information for both file paths; because of that, pathnames that look different syntactically but reference the same file are treated as identical. For example, on my server, the following paths to the encryptor module are different strings, but yield a true result from `os.path.samefile`:

```
../PyMailCgi/secret.py  
/home/crew/lutz/public_html/PyMailCgi/secret.py
```

Accessing either path form generates an error page like that in [Figure 12-29](#).

Figure 12-29. Accessing private files


figs/ppy2_1229.gif



Notice that bona fide file errors are handled differently. Permission problems and accesses to nonexistent files, for example, are trapped by a different exception handler clause, and display the exception's message to give additional context. [Figure 12-30](#) shows one such error page.

Figure 12-30. File errors display

figs/ppy2_1230.gif



As a general rule of thumb, file-processing exceptions should always be reported in detail, especially during script debugging. If we catch such exceptions in our scripts, it's up to us to display the details (assigning `sys.stderr` to `sys.stdout` won't help if Python doesn't print an error message). The current exception's type, data, and traceback objects are always available in the `sys` module for manual display.

·

The private files list check does prevent the encryption module from being viewed directly with this script, but it still may or may not be vulnerable to attack by malicious users. This book isn't about security, so I won't go into further details here, except to say that on the Internet, a little paranoia goes a long way. Especially for systems installed on the general Internet (instead of closed intranets), you should assume that the worst-case scenario will eventually happen.

12.7.2 Uploading Client Files to the Server

The `getfile` script lets us view server files on the client, but in some sense, it is a general-purpose file download tool. Although not as direct as fetching a file by FTP or over raw sockets, it serves similar purposes. Users of the script can either cut-and-paste the displayed code right of the web page or use their browser's View Source option to view and cut.

But what about going the other way -- uploading a file from the client machine to the server? As we saw in the last chapter, that is easy enough to accomplish with a client-side script that uses Python's FTP support module. Yet such a solution doesn't really apply in the context of a web browser; we can't usually ask all of our program's clients to start up a Python FTP script in another window to accomplish an upload. Moreover, there is no simple way for the server-side script to request the upload explicitly, unless there happens to be an FTP server running on the client machine (not at all the usual case).

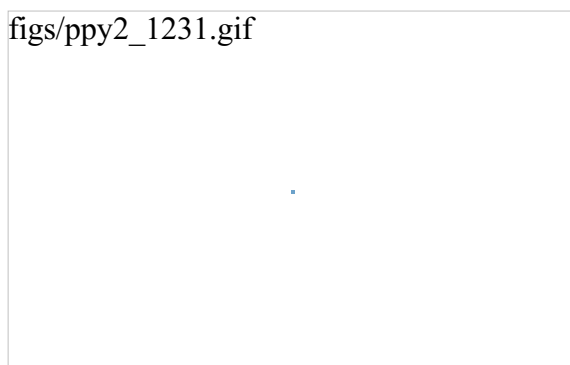
So is there no way to write a web-based program that lets its users upload files to a common server? In fact, there is, though it has more to do with HTML than with Python itself. HTML `<input>` tags also support a `type=file` option, which produces an input field, along with a button that pops up a file-selection dialog. The name of the client-side file to be uploaded can either be typed into the control, or selected with the pop-up dialog. The HTML page file in [Example 12-26](#) defines a page that allows any client-side file to be selected and uploaded to the server-side script named in the form's `action` option.

Example 12-26. PP2E\Internet\Cgi-Web\Basics\putfile.html

```
<html><title>Putfile: upload page</title>
<body>
<form enctype="multipart/form-data"
      method=post
      action="putfile.cgi">
  <h1>Select client file to be uploaded</h1>
  <p><input type=file size=50 name=clientfile>
  <p><input type=submit value=Upload>
</form>
<hr><a href="getfile.cgi?filename=putfile.cgi">View script code</a>
</body></html>
```

One constraint worth noting: forms that use `file` type inputs must also specify a `multipart/form-data` encoding type and the `post` submission method, as shown in this file; `get` style URLs don't work for uploading files. When we visit this page, the page shown in [Figure 12-31](#) is delivered. Pressing its Browse button opens a file-selection dialog, while Upload sends the file.

Figure 12-31. File upload selection page



On the client side, when we press this page's Upload button, the browser opens and reads the selected file, and packages its contents with the rest of the form's input fields (if any). When this information reaches the server, the Python script named in the form `action` tag is run as always, as seen in [Example 12-27](#).

Example 12-27. PP2E\Internet\Cgi-Web\Basics\putfile.cgi

```
#!/usr/bin/python
#####
# extract file uploaded by http from web browser;
# users visit putfile.html to get the upload form
# page, which then triggers this script on server;
# note: this is very powerful, and very dangerous:
# you will usually want to check the filename, etc.
# this will only work if file or dir is writeable;
# a unix 'chmod 777 uploads' command may suffice;
# file path names arrive in client's path format;
#####

import cgi, string, os, sys
import posixpath, dospath, macpath      # for client paths
debugmode = 0                          # 1=print form info
loadtextauto = 0                        # 1=read file at once
uploaddir = './uploads'                 # dir to store files

sys.stderr = sys.stdout                 # show error msgs
form = cgi.FieldStorage( )              # parse form data
print "Content-type: text/html\n"      # with blank line
if debugmode: cgi.print_form(form)     # print form fields

# html templates

html = """
<html><title>Putfile response page</title>
<body>
<h1>Putfile response page</h1>
%s
</html>"""

goodhtml = html % """
<p>Your file, '%s', has been saved on the server as '%s'.
<p>An echo of the file's contents received and saved appears below.
</p><hr>
<p><pre>%s</pre>
</p><hr>
"""

# process form data

def splitpath(origpath):
    for pathmodule in [posixpath, dospath, macpath]:
        basename = pathmodule.split(origpath)[1]
        if basename != origpath:
            return basename
    return origpath

def saveonserver(fileinfo):
    basename = splitpath(fileinfo.filename)
    srvrname = os.path.join(uploaddir, basename)
    if loadtextauto:
        filetext = fileinfo.value
        open(srvrname, 'w').write(filetext)
    else:
        srvrfile = open(srvrname, 'w')
        numlines, filetext = 0, ''
        while 1:
            line = fileinfo.file.readline( )
            if not line: break
            srvrfile.write(line)
            filetext = filetext + line
```

```
        numlines = numlines + 1
        filetext = ('[Lines=%d]\n' % numlines) + filetext
    os.chmod(srvrname, 0666) # make writeable: owned by 'nobody'
    return filetext, srvrname

def main( ):
    if not form.has_key('clientfile'):
        print html % "Error: no file was received"
    elif not form['clientfile'].filename:
        print html % "Error: filename is missing"
    else:
        fileinfo = form['clientfile']
        try:
            filetext, srvrname = saveonserver(fileinfo)
        except:
            errmsg = '<h2>Error</h2><p>%s<p>%s' % (sys.exc_type, sys.exc_value)
            print html % errmsg
        else:
            print goodhtml % (cgi.escape(fileinfo.filename),
                              cgi.escape(srvrname),
                              cgi.escape(filetext))

main( )
```

Within this script, the Python-specific interfaces for handling uploaded files are employed. They aren't much different, really; the file comes into the script as an entry in the parsed form object returned by `cgi.FieldStorage` as usual; its key is `clientfile`, the input control's name in the HTML page's code.

This time, though, the entry has additional attributes for the file's name on the client. Moreover, accessing the `value` attribute of an uploaded file input object will automatically read the file's contents all at once into a string on the server. For very large files, we can instead read line by line (or in chunks of bytes). For illustration purposes, the script implements either scheme: based on the setting of the `loadtextauto` global variable, it either asks for the file contents as a string, or reads it line by line.^[16] In general, the CGI module gives us back objects with the following attributes for file upload controls:

[16] Note that reading line means that this CGI script is biased towards uploading text files, not binary data files. The fact that it also uses a "w" open mode makes it ill suited for binary uploads if run on a Windows server -- `\r` characters might be added to the data when written. See [Chapter 2](#) for details if you've forgotten why.

filename

The name of the file as specified on the client

file

A file object from which the uploaded file's contents can be read

value

The contents of the uploaded file (read from file on demand)

There are additional attributes not used by our script. Files represent a third input field object; as we've also seen, the `value` attribute is a string for simple input fields, and we may receive a list of objects for multiple-selection controls.

For uploads to be saved on the server, CGI scripts (run by user "nobody") must have write access to the enclosing directory if the file doesn't yet exist, or to the file itself if it does. To help isolate uploads, the script stores all uploads in whatever server directory is named in the `uploaddir` global. On my site's Linux server, I had to give this directory a mode of 777 (universal read/write/execute permissions) with `chmod` to make uploads work in general. Your mileage may vary, but be sure to check permissions if this script fails.

The script also calls `os.chmod` to set the permission on the server file such that it can be read and written by everyone. If created anew by an upload, the file's owner will be "nobody," which means anyone out in cyberspace can view and upload the file. On my server, though, the file will also be only writable by user "nobody" by default, which might be inconvenient when it comes time to change that file outside the Web (the degree of pain can vary per operation).

Isolating client-side file uploads by placing them in a single directory on the server helps minimize security risks: existing files can't be overwritten arbitrarily. But it may require you to copy files on the server after they are uploaded, and it still doesn't prevent all security risks -- mischievous clients can still upload huge files, which we would need to trap with additional logic not present in this script as is. Such traps may only be needed in scripts open to the Internet at large.

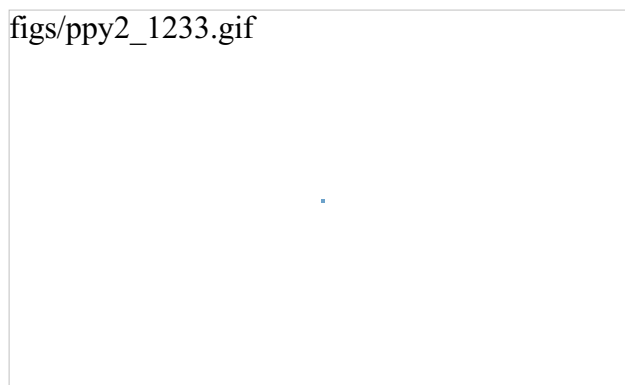
If both client and server do their parts, the CGI script presents us with the response page shown in [Figure 12-32](#), after it has stored the contents of the client file in a new or existing file on the server. For verification, the response gives the client and server file paths, as well as an echo of the uploaded file with a line count (in line-by-line reader mode).

Figure 12-32. Putfile response page



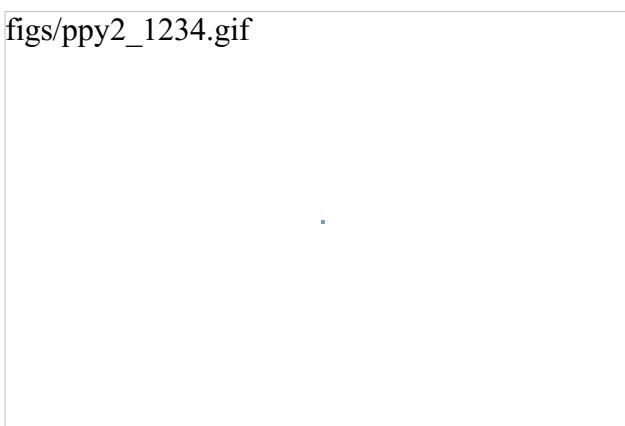
Incidentally, we can also verify the upload with the `getfile` program we wrote in the prior section. Simply access the selection page to type the pathname of the file on the server, as shown in [Figure 12-33](#).

Figure 12-33. Verifying putfile with getfile -- selection



Assuming uploading the file was successful, [Figure 12-34](#) shows the resulting viewer page we will obtain. Since user "nobody" (CGI scripts) was able to write the file, "nobody" should be able to view it as well.

Figure 12-34. Verifying putfile with getfile -- response



Notice the URL in this page's address field -- the browser translated the / character we typed into the selection page to a `%2F` hexadecimal escape code before adding it to the end of the URL as a parameter. We met URL escape codes like this earlier in this chapter. In this case, the browser did the translation for us, but the end result is as if we had manually called one of the `urllib` quoting functions on the file path string.

Technically, the `%2F` escape code here represents the standard URL translation for non-ASCII characters, under the default encoding scheme browsers employ. Spaces are usually translated to `+` characters as well. We can often get away without manually translating most non-ASCII characters when sending paths explicitly (in typed URLs). But as we saw earlier, we sometimes need to be careful to escape characters (e.g., `&`) that have special meaning within URL strings with `urllib` tools.

12.7.2.1 Handling client path formats

In the end, the `putfile.cgi` script stores the uploaded file on the server, within a hardcoded `uploaddir` directory, under the filename at the end of the file's path on the client (i.e., less its client-side directory path). Notice, though, that the `splitpath` function in this script needs to do

extra work to extract the base name of the file on the right. Browsers send up the filename in the directory path format used on the client machine; this path format may not be the same as that used on the server where the CGI script runs.

The standard way to split up paths, `os.path.split`, knows how to extract the base name, but only recognizes path separator characters used on the platform it is running on. That is, if we run this CGI script on a Unix machine, `os.path.split` chops up paths around a `/` separator. If a user uploads from a DOS or Windows machine, however, the separator in the passed filename is `\`, not `/`. Browsers running on a Macintosh may send a path that is more different still.

To handle client paths generically, this script imports platform-specific, path-processing modules from the Python library for each client it wishes to support, and tries to split the path with each until a filename on the right is found. For instance, `posixpath` handles paths sent from Unix-style platforms, and `dospath` recognizes DOS and Windows client paths. We usually don't import these modules directly since `os.path.split` is automatically loaded with the correct one for the underlying platform; but in this case, we need to be specific since the path comes from another machine. Note that we could have instead coded the path splitter logic like this to avoid some split calls:

```
def splitpath(origpath):
    basename = os.path.split(origpath)[1]
    if basename == origpath:
        if '\\\' in origpath:
            basename = string.split(origpath, '\\\')[-1]
        elif '/' in origpath:
            basename = string.split(origpath, '/')[-1]
    return basename
```

get name at end
try server paths
didn't change it?
try dos clients
try unix clients

But this alternative version may fail for some path formats (e.g., DOS paths with a drive but no backslashes). As is, both options waste time if the filename is already a base name (i.e., has no directory paths on the left), but we need to allow for the more complex cases generically.

This upload script works as planned, but a few caveats are worth pointing out before we close the book on this example:

- First, `putfile` doesn't do anything about cross-platform incompatibilities in filenames themselves. For instance, spaces in a filename shipped from a DOS client are not translated to nonspace characters; they will wind up as spaces in the server-side file's name, which may be legal but which are difficult to process in some scenarios.
- Second, the script is also biased towards uploading text files; it opens the output file in text mode (which will convert end-of-line marker codes in the file to the end-of-line convention on the web server machine), and reads input line-by-line (which may fail for binary data).

If you run into any of these limitations, you will have crossed over into the domain of suggested exercises.

12.7.3 More Than One Way to Push Bits Over the Net

Finally, let's discuss some context. We've seen three `getfile` scripts at this point in the book. The one in this chapter is different than the other two we wrote in earlier chapters, but it accomplishes a similar goal:

- This chapter's `getfile` is a server-side CGI script that displays files over the HTTP protocol (on port 80).
- In [Chapter 10](#), we built a client and server-side `getfile` to transfer with raw sockets (on port 50001) and [Chapter 11](#) implemented a client-side `getfile` to ship over FTP (on port 21)

The CGI- and HTTP-based `putfile` script here is also different from the FTP-based `putfile` in the last chapter, but it can be considered an alternative to both socket and FTP uploads. To help underscore the distinctions, [Figure 12-35](#) and [Figure 12-36](#) show the new `putfile` uploading the original socket-based `getfile`.^[17]

[17] Shown here being loaded from a now defunct *Part2* directory -- replace *Part2* with *PP2E* to find its true location, and don't be surprised if a few difference show up in transferred files contents if you run such examples yourself. Like I said, engineers love to change things.

Figure 12-35. A new putfile with the socket-based getfile uploaded



Really, the `getfile` CGI script in this chapter simply displays files only, but can be considered a download tool when augmented with cut-and-paste operations in a web browser. Figures [Figure 12-37](#) and [Figure 12-38](#) show the CGI `getfile` displaying the uploaded socket-based `getfile`.

Figure 12-36. A new putfile with the socket-based getfile

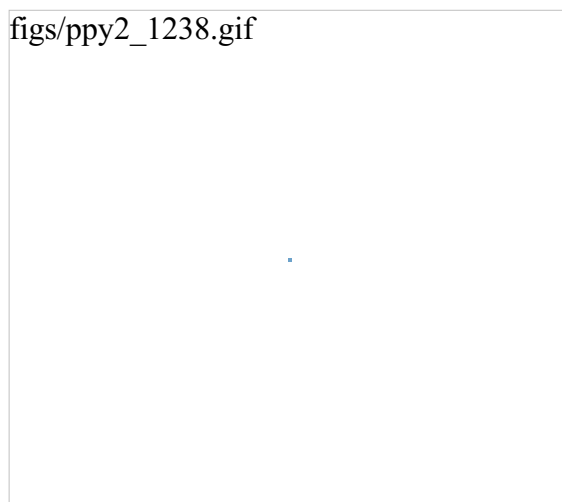




Figure 12-37. A new getfile with the socket-based getfile



Figure 12-38. A new getfile with the socket-based getfile downloaded



The point to notice here is that there are a variety of ways to ship files around the Internet -- sockets, FTP, and HTTP (web pages) can all move files between computers. Technically speaking, we can transfer files with other techniques and protocols, too -- POP email, NNTP news, and so on.

Each technique has unique properties but does similar work in the end: moving bits over the Net. All ultimately run over sockets on a particular port, but protocols like FTP add additional structure to the socket layer, and application models like CGI add both structure and programmability.

Chapter 12. Server-Side Scripting

[Section 12.1. "Oh What a Tangled Web We Weave"](#)

[Section 12.2. What's a Server-Side CGI Script?](#)

[Section 12.3. Climbing the CGI Learning Curve](#)

[Section 12.4. The Hello World Selector](#)

[Section 12.5. Coding for Maintainability](#)

[Section 12.6. More on HTML and URL Escapes](#)

[Section 12.7. Sending Files to Clients and Servers](#)

13.1 "Things to Do When Visiting Chicago"

This chapter is the fourth part of our look at Python Internet programming, and continues the last chapter's discussion. In the prior chapter, we explored the fundamentals of server-side CGI scripting in Python. Armed with that knowledge, in this and the following chapter we move on to two larger case studies that underscore advanced CGI topics:

PyMailCgi

This chapter presents PyMailCgi, a web site for reading and sending email that illustrates security concepts, hidden form fields, URL generation, and more. Because this system is similar in spirit to the PyMailGui program shown in [Chapter 11](#), this example also serves as a comparison of web and non-web applications.

PyErrata

[Chapter 14](#), presents PyErrata, a web site for posting book comments and bugs that introduces database concepts in the CGI domain. This system demonstrates common ways to store data persistently on the server between web transactions, and addresses concurrent update problems inherent in the CGI model.

Both of these case studies are based on CGI scripting, but implement full-blown web sites that do something more useful than the last chapter's examples.

As usual, these chapters split their focus between application-level details and Python programming concepts. Because both of the case studies presented are fairly large, they illustrate system design concepts that are important in actual projects. They also say more about CGI scripts in general. PyMailCgi, for example, introduces the notions of state retention in hidden fields and URLs, as well as security concerns and encryption. PyErrata provides a vehicle for exploring persistent database concepts in the context of web sites.

Neither system here is particularly flashy or feature-rich as web sites go (in fact, the initial cut of PyMailCgi was thrown together during a layover at a Chicago airport). Alas, you will find neither dancing bears nor blinking lights at either of these sites. On the other hand, they were written to serve real purposes, speak more to us about CGI scripting, and hint at just how far Python server-side programs can take us. In [Chapter 15](#), we will explore higher-level systems and tools that build upon ideas we will apply here. For now, let's have some fun with Python on the Web.

13.2 The PyMailCgi Web Site

Near the end of [Chapter 11](#), we built a program called PyMailGui that implemented a complete Python+Tk email client GUI (if you didn't read that section, you may want to take a quick look at it now). Here, we're going to do something of the same, but on the Web: the system presented in this section, PyMailCgi, is a collection of CGI scripts that implement a simple web-based interface for sending and reading email in any browser.

Our goal in studying this system is partly to learn a few more CGI tricks, partly to learn a bit about designing larger Python systems in general, and partly to underscore the trade-offs between systems implemented for the Web (PyMailCgi) and systems written to run locally (PyMailGui). This chapter hints at some of these trade-offs along the way, and returns to explore them in more depth after the presentation of this system.

13.2.1 Implementation Overview

At the top level, PyMailCgi allows users to view incoming email with the POP interface and to send new mail by SMTP. Users also have the option of replying to, forwarding, or deleting an incoming email while viewing it. As implemented, anyone can send email from the PyMailCgi site, but to view your email, you generally have to install PyMailCgi at your own site with your own mail server information (due to security concerns described later).

Viewing and sending email sounds simple enough, but the interaction involved involves a number of distinct web pages, each requiring a CGI script or HTML file of its own. In fact, PyMailCgi is a fairly linear system -- in the most complex user interaction scenario, there are six states (and hence six web pages) from start to finish. Because each page is usually generated by distinct file in the CGI world, that also implies six source files.

To help keep track of how all of PyMailCgi's files fit into the overall system, I wrote the file in [Example 13-1](#) before starting any real programming. It informally sketches the user's flow through the system and the files invoked along the way. You can certainly use more formal notations to describe the flow of control and information through states such as web pages (e.g., dataflow diagrams), but for this simple example this file gets the job done.

Example 13-1. PP2E\Internet\Cgi-Web\PyMailCgi\pageflow.txt

```
file or script                                creates
-----
[pymailcgi.html]                             Root window
=> [onRootViewLink.cgi]                      Pop password window
    => [onViewPswdSubmit.cgi]                 List window (loads all pop mail)
        => [onViewListLink.cgi]              View Window + pick=del|reply|fwd (fetch
            => [onViewSubmit.cgi]            Edit window, or delete+confirm (del)
                => [onSendSubmit.cgi]       Confirmation (sends smtp mail)
                    => back to root

=> [onRootSendLink.cgi]                       Edit Window
    => [onSendSubmit.cgi]                     Confirmation (sends smtp mail)
        => back to root
```



This file simply lists all the source files in the system, using => and indentation to denote the scripts they trigger.

For instance, links on the *pymailcgi.html* root page invoke *onRootViewLink.cgi* and *onRootSendLink.cgi*, both executable scripts. The script *onRootViewLink.cgi* generates a password page, whose Submit button in turn triggers *onViewPswdSubmit.cgi*, and so on. Notice that both the view and send actions can wind up triggering *onSendSubmit.cgi* to send a new mail view operations get there after the user chooses to reply to or forward an incoming mail.

In a system like this, CGI scripts make little sense in isolation, so it's a good idea to keep the overall page flow in mind; refer to this file if you get lost. For additional context, [Figure 13-1](#) shows the overall contents of this site, viewed on Windows with the PyEdit "Open" function.

Figure 13-1. PyMailCgi contents

figs/ppy2_1301.gif



The *temp* directory was used only during development. To install this site, all the files you see here are uploaded to a *PyMailCgi* subdirectory of my *public_html* web directory. Besides the page-flow HTML and CGI script files invoked by user interaction, PyMailCgi uses a handful of utility modules as well:

- *commonhtml.py* is a library of HTML tools.
- *externs.py* isolates access to modules imported from other systems.
- *loadmail.py* encapsulates mailbox fetches.
- *secret.py* implements configurable password encryption.

PyMailCgi also reuses parts of the *pymail.py* and *mailconfig.py* modules we wrote in [Chapter 11](#) on my web server, these are installed in a special directory that is not necessarily the same as their location in the examples distribution (they show up in another server directory, not shown in [Figure 13-1](#)). As usual, PyMailCgi also uses a variety of standard Python library modules: `smtplib`, `poplib`, `rfc822`, `cgi`, `urllib`, `time`, `rotor`, and the like.

Carry-on Software

PyMailCgi works as planned and illustrates more CGI and email concepts, but I want to point out a few caveats up front. The application was initially written during a two-hour layover in Chicago's O'Hare airport (though debugging took a few hours more). I wrote it to meet a specific need -- to be able to read and send email from any web browser while traveling around the world teaching Python classes. I didn't design it to be aesthetically pleasing to others and didn't spend much time focusing on its efficiency.

I also kept this example intentionally simple for this book. For example, PyMailCgi doesn't provide all the features of the PyMailGui program in [Chapter 11](#), and reloads email more than it probably should. In other words, you should consider this system a work in progress; it's not yet software worth selling. On the other hand, it does what it was intended to do, and can be customized by tweaking its Python source code -- something that can't be said of all software sold.

13.2.2 Presentation Overview

PyMailCgi is a challenge to present in a book like this, because most of the "action" is encapsulated in shared utility modules (especially one called *commonhtml.py*); the CGI scripts that implement user interaction don't do much by themselves. This architecture was chosen deliberately, to make scripts simple and implement a common look-and-feel. But it means you must jump between files to understand how the system works.

To make this example easier to digest, we're going to explore its code in two chunks: page script first, and then the utility modules. First, we'll study screen shots of the major web pages served up by the system and the HTML files and top-level Python CGI scripts used to generate them. We begin by following a send mail interaction, and then trace how existing email is processed. Most implementation details will be presented in these sections, but be sure to flip ahead to the utility modules listed later to understand what the scripts are really doing.

I should also point out that this is a fairly complex system, and I won't describe it in exhaustive detail; be sure to read the source code along the way for details not made explicit in the narrative. All of the system's source code appears in this section (and also at <http://examples.oreilly.com/python2>), and we will study the key concepts in this system here. But as usual with case studies in this book, I assume that you can read Python code by now and will consult the example's source code for more details. Because Python's syntax is so close to executable pseudocode, systems are sometimes better described in Python than in English.

13.3 The Root Page

Let's start off by implementing a main page for this example. The file shown in [Example 13-2](#) is primarily used to publish links to the Send and View functions' pages. It is coded as a static HTML file, because there is nothing to generate on the fly here.

Example 13-2. PP2E\Internet\Cgi-Web\PyMailCgi\pymailcgi.html

```
<HTML><BODY>
<TITLE>PyMailCgi Main Page</TITLE>
<H1 align=center>PyMailCgi</H1>
<H2 align=center>A POP/SMTP Email Interface</H2>
<P align=center><I>Version 1.0, April 2000</I></P>

<table><tr><td><hr>
<P>
<A href="http://rmi.net/~lutz/about-pp.html">
<IMG src="../PyErrata/ppsmall.gif" align=left
alt="[Book Cover]" border=1 hspace=10></A>
This site implements a simple web-browser interface to POP/SMTP email
accounts. Anyone can send email with this interface, but for security
reasons, you cannot view email unless you install the scripts with your
own email account information, in your own server account directory.
PyMailCgi is implemented as a number of Python-coded CGI scripts that run on
a server machine (not your local computer), and generate HTML to interact
with the client/browser. See the book <I>Programming Python, 2nd Edition</I>
for more details.</P>

<tr><td><hr>
<h2>Actions</h2>
<P><UL>
<LI><a href="onRootViewLink.cgi">View, Reply, Forward, Delete POP mail</a>
<LI><a href="onRootSendLink.cgi">Send a new email message by SMTP</a>
</UL></P>

<tr><td><hr>
<P>Caveats: PyMailCgi 1.0 was initially written during a 2-hour layover at
Chicago's O'Hare airport. This release is not nearly as fast or complete
as PyMailGui (e.g., each click requires an Internet transaction, there
is no save operation, and email is reloaded often). On the other hand,
PyMailCgi runs on any web browser, whether you have Python (and Tk)
installed on your machine or not.

<P>Also note that if you use these scripts to read your own email, PyMailCgi
does not guarantee security for your account password, so be careful out there.
See the notes in the View action page as well as the book for more information
on security policies. Also see:

<UL>
<li>The <I>PyMailGui</I> program in the Email directory, which
implements a client-side Python+Tk email GUI
<li>The <I>pymail.py</I> program in the Email directory, which
provides a simple command-line email interface
<li>The Python imaplib module which supports the IMAP email protocol
instead of POP
<li>The upcoming openSSL support for secure transactions in the new
Python 1.6 socket module
```

```
</UL></P>
</table><hr>

<A href="http://www.python.org">
<IMG SRC="../../../PyErrata/PythonPoweredSmall.gif" ALIGN=left
ALT="[Python Logo]" border=0 hspace=15></A>
<A href="http://PyInternetDemos.html">More examples</A>
</BODY></HTML>
```

The file *pymailcgi.html* is the system's root page and lives in a *PyMailCgi* subdirectory of my web directory that is dedicated to this application (and helps keep its files separate from other examples). To access this system, point your browser to:

<http://starship.python.net/~lutz/PyMailCgi/pymailcgi.html>

If you do, the server will ship back a page like that shown in [Figure 13-2](#).

Figure 13-2. PyMailCgi main page



Now, before you click on the View link here expecting to read your own email, I should point out that by default, PyMailCgi allows anybody to send email from this page with the Send link (as we learned earlier, there are no passwords in SMTP). It does not, however, allow arbitrary users on the Web to read their email accounts without typing an explicit and unsafe URL or doing a bit of installation and configuration. This is on purpose, and has to do with security constraints; as we'll see later, I wrote the system such that it never associates your email username and password together without encryption.

By default, then, this page is set up to read my (the author's) email account, and requires my POP password to do so. Since you probably can't guess my password (and wouldn't find my email helpful if you could), PyMailCgi is not incredibly useful as installed at this site. To use it to read your email instead, you should install the system's source code on your own server and tweak a mail configuration file that we'll see in a moment. For now, let's proceed by using the system as it is installed on my server, with my POP email account; it works the same way, regardless of which account it accesses.

I l@ve RuBoard

13.4 Sending Mail by SMTP

PyMailCgi supports two main functions (as links on the root page): composing and sending new mail to others, and viewing your incoming mail. The View function leads to pages that let users reply to, forward, and delete existing email. Since the Send function is the simplest, let's start with its pages and scripts first.

13.4.1 The Message Composition Page

The Send function steps users through two pages: one to edit a message and one to confirm delivery. When you click on the Send link on the main page, the script in [Example 13-3](#) runs on the server.

Example 13-3. PP2E\Internet\Cgi-Web\PyMailCgi\onRootSendLink.cgi

```
#!/usr/bin/python
# On 'send' click in main root window

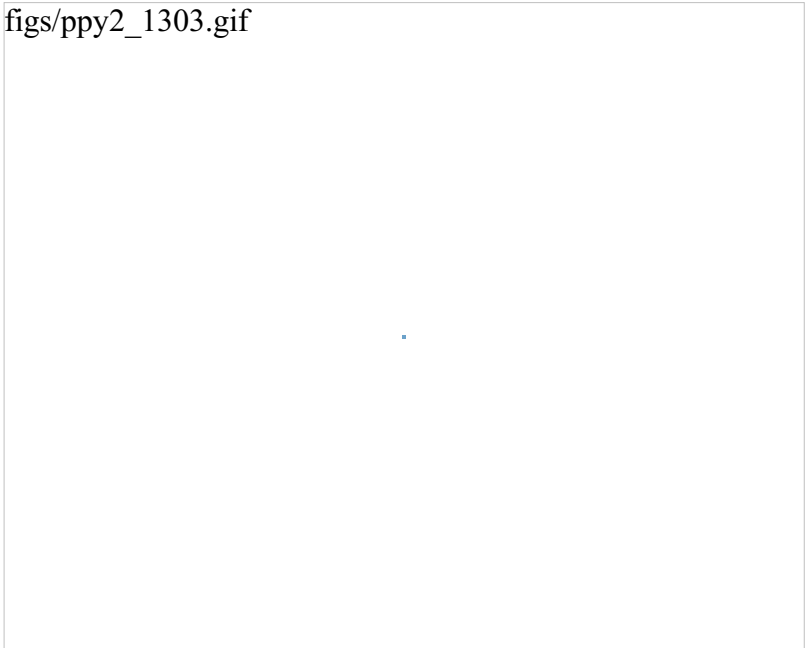
import commonhtml
from externs import mailconfig

commonhtml.editpage(kind='Write', headers={'From': mailconfig.myaddress})
```

No, this file wasn't truncated; there's not much to see in this script, because all the action has been encapsulated in the `commonhtml` and `externs` modules. All that we can tell here is that the script calls something named `editpage` to generate a reply, passing in something called `myaddress` for its "From" header. That's by design -- by hiding details in utility modules, we make top-level scripts like this much easier to read and write. There are no inputs to this script either; when run, it produces a page for composing a new message, as shown in [Figure 13-3](#).

Figure 13-3. PyMailCgi send (write) page

figs/ppy2_1303.gif



13.4.2 Send Mail Script

Much like the Tkinter-based PyMailGui client program we met in [Chapter 11](#), this page provides fields for entering common header values as well as the text of the message itself. The "From" field is prefilled with a string imported from a module called `mailconfig`. As we'll discuss in a moment, that module lives in another directory on the server in this system, but its contents are the same as in the PyMailGui example. When we click the Send button of the edit page, [Example 13-4](#) runs on the server.

Example 13-4. PP2E\Internet\Cgi-Web\PyMailCgi\onSendSubmit.cgi

```
#!/usr/bin/python
# On submit in edit window--finish a write, reply, or forward

import cgi, smtplib, time, string, commonhtml
commonhtml.dumpstatepage(0)
form = cgi.FieldStorage( )           # parse form input data

# server name from module or get-style url
smtpservername = commonhtml.getstandardsmtpfields(form)

# parms assumed to be in form or url here
from commonhtml import getfield      # fetch value attributes
From = getfield(form, 'From')        # empty fields may not be sent
To   = getfield(form, 'To')
Cc   = getfield(form, 'Cc')
Subj = getfield(form, 'Subject')
text = getfield(form, 'text')

# caveat: logic borrowed from PyMailGui
date = time.ctime(time.time( ))
Cchdr = (Cc and 'Cc: %s\n' % Cc) or ''
hdrs = ('From: %s\nTo: %s\n%sDate: %s\nSubject: %s\n'
        % (From, To, Cchdr, date, Subj))
hdrs = hdrs + 'X-Mailer: PyMailCgi Version 1.0 (Python)\n'

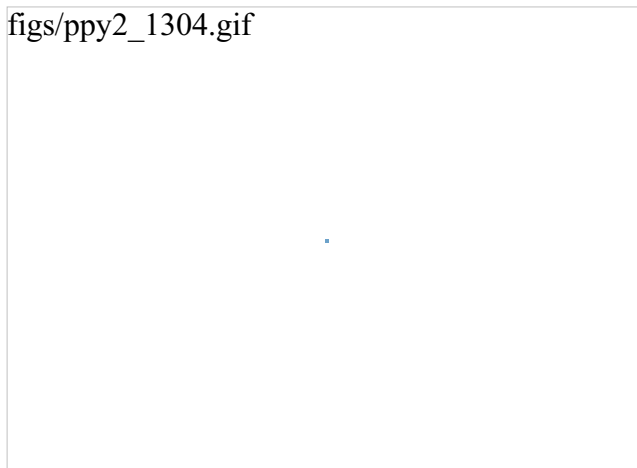
Ccs = (Cc and string.split(Cc, ';')) or [] # some servers reject ['']
Tos = string.split(To, ';') + Ccs         # cc: hdr line, and To list
Tos = map(string.strip, Tos)              # some addrs can have ', 's

try:                                     # smtplib may raise except
    server = smtplib.SMTP(smtpservername) # or return failed Tos dict
    failed = server.sendmail(From, Tos, hdrs + text)
    server.quit( )
except:
    commonhtml.errorpage('Send mail error')
else:
    if failed:
        errInfo = 'Send mail error\nFailed recipients:\n' + str(failed)
        commonhtml.errorpage(errInfo)
    else:
        commonhtml.confirmationpage('Send mail')
```

This script gets mail header and text input information from the edit page's form (or from parameters in an explicit URL), and sends the message off using Python's standard `smtplib` module. We studied `smtplib` in depth in [Chapter 11](#), so I won't say much more about it now. In fact, the send mail code here looks much like that in PyMailGui (despite what I've told you about code reuse; this code would be better made a utility).

A utility in `commonhtml` ultimately fetches the name of the SMTP server to receive the message from either the `mailconfig` module or the script's inputs (in a form field or URL parameter). If all goes well, we're presented with a generated confirmation page, as in [Figure 13-4](#).

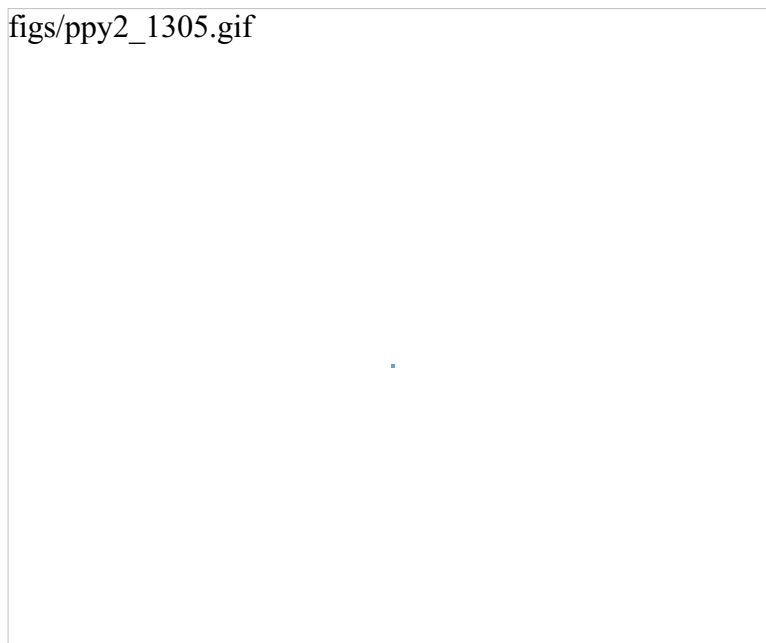
Figure 13-4. PyMailCgi send confirmation page



Notice that there are no usernames or passwords to be found here; as we saw in [Chapter 11](#), SMTP requires only a server that listens on the SMTP port, not a user account or password. As we also saw in that chapter, SMTP send operations that fail either raise a Python exception (e.g., if the server host can't be reached) or return a dictionary of failed recipients.

If there is a problem during mail delivery, we get an error page like the one shown in [Figure 13-5](#). This page reflects a failed recipient -- the `else` clause of the `try` statement we used to wrap the send operation. On an actual exception, the Python error message and extra details would be displayed.

Figure 13-5. PyMailCgi send error page





Before we move on, you should know that this send mail script is also used to deliver reply and forward messages for incoming POP mail. The user interface for those operations is slightly different than for composing new email from scratch, but as in PyMailGui, the submission handler logic is the same code -- they are really just mail send operations.

It's also worth pointing out that the `commonhtml` module encapsulates the generation of both the confirmation and error pages, so that all such pages look the same in PyMailCgi no matter where and when they are produced. Logic that generates the mail edit page in `commonhtml` is reused by the reply and forward actions too (but with different mail headers).

In fact, `commonhtml` makes all pages look similar -- it also provides common page header (top) and footer (bottom) generation functions, which are used everywhere in the system. You may have already noticed that all the pages so far follow the same pattern: they start with a title and horizontal rule, have something unique in the middle, and end with another rule, followed by a Python icon and link at the bottom. This common look-and-feel is the product of `commonhtml`; it generates everything but the middle section for every page in the system (except the root page, a static HTML file).

If you are interested in seeing how this encapsulated logic works right now, flip ahead to [Example 13-14](#). We'll explore its code after we study the rest of the mail site's pages.

13.4.2.1 Using the send mail script outside a browser

I initially wrote the send script to be used only within PyMailCgi, using values typed into the mail edit form. But as we've seen, inputs can be sent in either form fields or URL parameters; because the send mail script checks for inputs in CGI inputs before importing from the `mailconfig` module, it's also possible to call this script outside the edit page to send email. For instance, explicitly typing a URL of this nature into your browser (but all on one line and with no intervening spaces):

```
http://starship.python.net/~lutz/  
    PyMailCgi/onSendSubmit.cgi?site=smtplib.rmi.net&  
        From=lutz@rmi.net&  
        To=lutz@rmi.net&  
        Subject=test+url&  
        text=Hello+Mark;this+is+Mark
```

will indeed send an email message as specified by the input parameters at the end. That URL string is a lot to type into a browser's address field, of course, but might be useful if generated automatically by another script. As we saw in [Chapter 11](#), module `urllib` can then be used to submit such a URL string to the server from within a Python program. [Example 13-5](#) shows one way to do it.

Example 13-5. PP2E\Internet\Cgi-Web\PyMailCgi\sendurl.py

```
#####  
# Send email by building a URL like this from inputs:  
# http://starship.python.net/~lutz/  
#     PyMailCgi/onSendSubmit.cgi?site=smtplib.rmi.net&  
#         From=lutz@rmi.net&  
#         To=lutz@rmi.net&
```

```
#                                     Subject=test+url&
#                                     text=Hello+Mark;this+is+Mark
#####

from urllib import quote_plus, urlopen

url = 'http://starship.python.net/~lutz/PyMailCgi/onSendSubmit.cgi'
url = url + '?site=%s'      % quote_plus(raw_input('Site>'))
url = url + '&From=%s'     % quote_plus(raw_input('From>'))
url = url + '&To=%s'      % quote_plus(raw_input('To >'))
url = url + '&Subject=%s' % quote_plus(raw_input('Subj>'))
url = url + '&text=%s'    % quote_plus(raw_input('text>'))      # or input loop

print 'Reply html:'
print urlopen(url).read( )      # confirmation or error page html
```

Running this script from the system command line is yet another way to send an email message -- this time, by contacting our CGI script on a remote server machine to do all the work. Script *sendurl.py* runs on any machine with Python and sockets, lets us input mail parameters interactively, and invokes another Python script that lives on a remote machine. It prints HTML returned by our CGI script:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python sendurl.py
Site>smtp.rmi.net
From>lutz@rmi.net
To >lutz@rmi.net
Subj>test sendurl.py
text>But sir, it's only wafer-thin...
Reply html:
<html><head><title>PyMailCgi: Confirmation page (PP2E)</title></head>
<body bgcolor="#FFFFFF"><h1>PyMailCgi Confirmation</h1><hr>
<h2>Send mail operation was successful</h2>
<p>Press the link below to return to the main page.</p>
</p><hr><a href="http://www.python.org">
</a>
<a href="pymailcgi.html">Back to root page</a>
</body></html>
```

The HTML reply printed by this script would normally be rendered into a new web page if caught by a browser. Such cryptic output might be less than ideal, but you could easily search the reply string for its components to determine the result (e.g., using `string.find` to look for "successful"), parse out its components with Python's standard `htmllib` module, and so on. The resulting mail message -- viewed, for variety, with [Chapter 11](#)'s PyMailGui program -- shows up in my account as seen in [Figure 13-6](#).

Figure 13-6. *sendurl.py* result



Of course, there are other, less remote ways to send email from a client machine. For instance, the Python `smtpplib` module itself depends only upon the client and POP server connections being operational, whereas this script also depends on the CGI server machine (requests go from client to CGI server to POP server and back). Because our CGI script supports general URLs, though, it can do more than a "mailto:" HTML tag, and can be invoked with `urllib` outside the context of a running web browser. For instance, scripts like *sendurl.py* can be used to invoke and test server-side programs.

13.5 Reading POP Email

So far, we've stepped through the path the system follows to send new mail. Let's now see what happens when we try to view incoming POP mail.

13.5.1 The POP Password Page

If you flip back to the main page in [Figure 13-2](#), you'll see a View link; pressing it triggers the script in [Example 13-6](#) to run on the server:

Example 13-6. PP2E\Internet\Cgi-Web\PyMailCgi\onRootViewLink.cgi

```
#!/usr/bin/python
#####
# on view link click on main/root html page
# this could almost be a html file because there are likely
# no input params yet, but I wanted to use standard header/
# footer functions and display the site/user names which must
# be fetched; On submission, doesn't send the user along with
# password here, and only ever sends both as URL params or
# hidden fields after the password has been encrypted by a
# user-uploadable encryption module; put html in commonhtml?
#####

# page template

pswdhtml = """
<form method=post action=%s/onViewPswdSubmit.cgi>
<p>
Please enter POP account password below, for user "%s" and site "%s".
<p><input name=pswd type=password>
<input type=submit value="Submit"></form></p>

<hr><p><i>Security note</i>: The password you enter above will be transmitted
over the Internet to the server machine, but is not displayed, is never
transmitted in combination with a username unless it is encrypted, and is
never stored anywhere: not on the server (it is only passed along as hidden
fields in subsequent pages), and not on the client (no cookies are generated).
This is still not totally safe; use your browser's back button to back out of
PyMailCgi at any time.</p>
"""

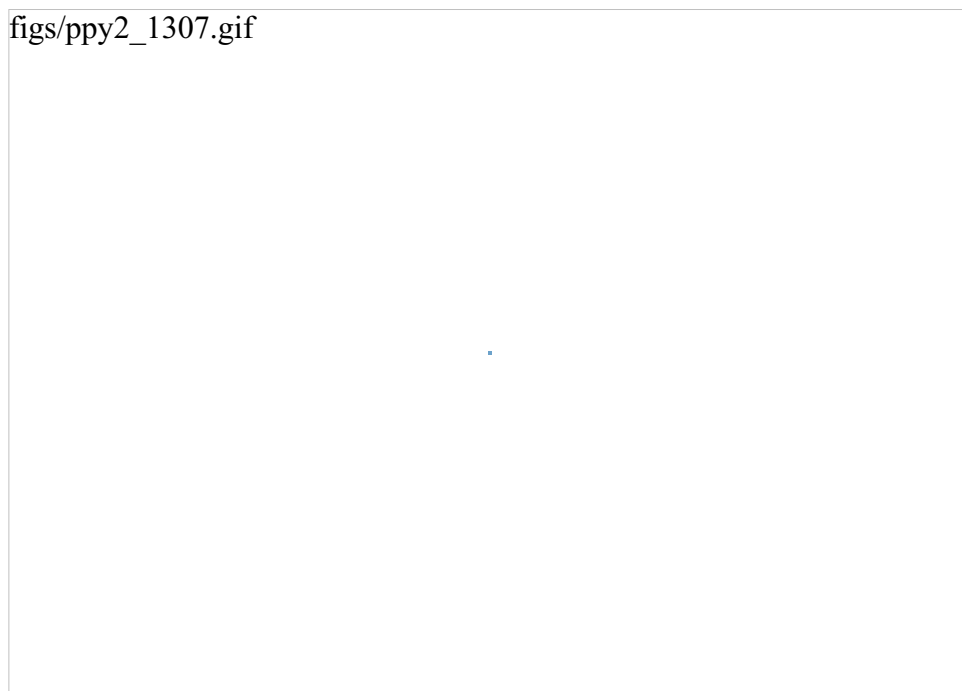
# generate the password input page

import commonhtml                                     # usual parms case:
user, pswd, site = commonhtml.getstandardpopfields({}) # from module here,
commonhtml.pageheader(kind='POP password input')      # from html|url later
print pswdhtml % (commonhtml.urlroot, user, site)
commonhtml.pagefooter( )
```

This script is almost all embedded HTML: the triple-quoted `pswdhtml` string is printed, with strii formatting, in a single step. But because we need to fetch the user and server names to display on the generated page, this is coded as an executable script, not a static HTML file. Module `commonhtml` either loads user and server names from script inputs (e.g., appended to the script's URL), or imports them from the `mailconfig` file; either way, we don't want to hardcode them in this script or its HTML, so an HTML file won't do.

Since this is a script, we can also make use of the `commonhtml` page header and footer routines to render the generated reply page with the common look-and-feel; this is shown in [Figure 13-7](#).

Figure 13-7. PyMailCgi view password login page



At this page, the user is expected to enter the password for the POP email account of the user and server displayed. Notice that the actual password isn't displayed; the input field's HTML specifies `type=password`, which works just like a normal text field, but shows typed input as stars. (See also [Example 11-6](#) for doing this at a console, and [Example 11-23](#) for doing this in a GUI.)

13.5.2 The Mail Selection List Page

After filling out the last page's password field and pressing its Submit button, the password is shipped off to the script shown in [Example 13-7](#).

Example 13-7. PP2E\Internet\Cgi-Web\PyMailCgi\onViewPswdSubmit.cgi

```
#!/usr/bin/python
# On submit in pop password input window--make view list

import cgi, StringIO, rfc822, string
import loadmail, commonhtml
from secret import encode          # user-defined encoder module
MaxHdr = 35                       # max length of email hdrs in list

# only pswd comes from page here, rest usually in module
formdata = cgi.FieldStorage( )
mailuser, mailpswd, mailsite = commonhtml.getstandardpopfields(formdata)

try:
    newmail = loadmail.loadnewmail(mailsite, mailuser, mailpswd)
    mailnum = 1
    maillist = []
    for mail in newmail:
```

```
msginfo = []
hdrs = rfc822.Message(StringIO.StringIO(mail))
for key in ('Subject', 'From', 'Date'):
    msginfo.append(hdrs.get(key, '?')[:MaxHdr])
msginfo = string.join(msginfo, ' | ')
maillist.append((msginfo, commonhtml.urlroot + '/onViewListLink.cgi',
                {'mnum': mailnum,
                 'user': mailuser,           # data params
                 'pswd': encode(mailpswd),   # pass in url
                 'site': mailsite}))        # not inputs

    mailnum = mailnum+1
commonhtml.listpage(maillist, 'mail selection list')
except:
    commonhtml.errorpage('Error loading mail index')
```

This script's main purpose is to generate a selection list page for the user's email account, using the password typed into the prior page (or passed in a URL). As usual with encapsulation, most of the details are hidden in other files:

- `loadmail.loadnewmail` reuses the mail module from [Chapter 11](#) to fetch email with the POP protocol; we need a message count and mail headers here to display an index list.
- `commonhtml.listpage` generates HTML to display a passed-in list of (`text`, `URL`, `parameter-dictionary`) tuples as a list of hyperlinks in the reply page; parameter values show up at the end of URLs in the response.

The `maillist` list built here is used to create the body of the next page -- a clickable email message selection list. Each generated hyperlink in the list page references a constructed URL that contains enough information for the next script to fetch and display a particular email message.

If all goes well, the mail selection list page HTML generated by this script is rendered as in [Figure 13-8](#). If you get as much email as I do, you'll probably need to scroll down to see the end of this page. It looks like [Figure 13-9](#), and follows the common look-and-feel for all PyMailCgi pages, thanks to `commonhtml`.

Figure 13-8. PyMailCgi view selection list page, top

figs/ppy2_1308.gif


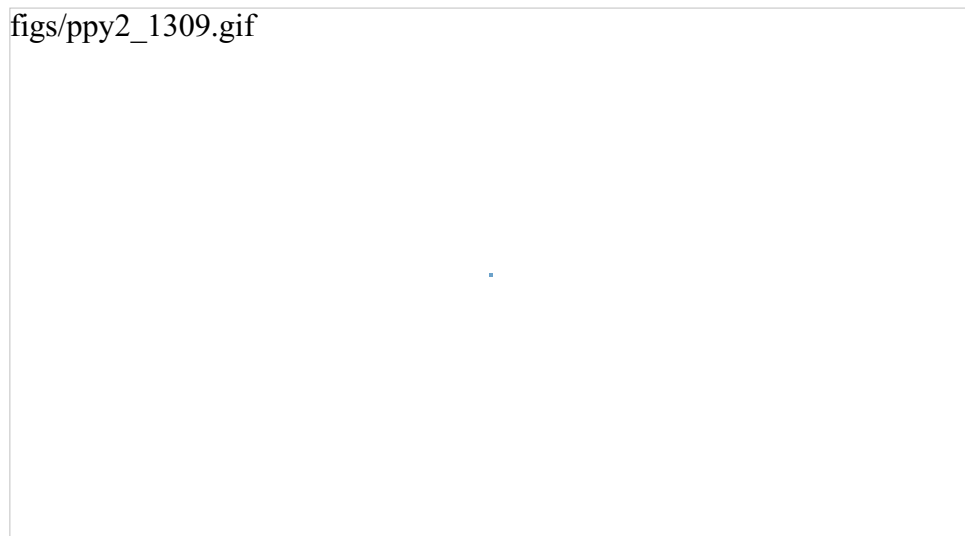
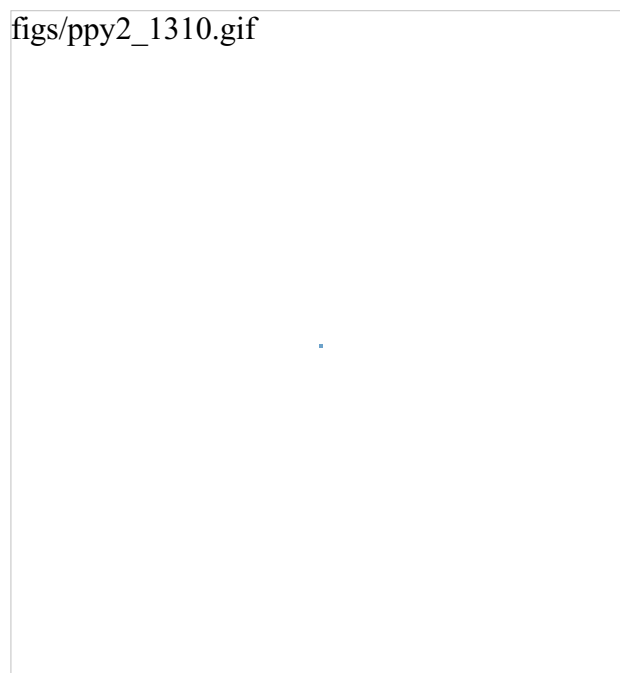


Figure 13-9. PyMailCgi view selection list page, bottom



If the script can't access your email account (e.g., because you typed the wrong password), then `try` statement handler instead produces a commonly formatted error page. [Figure 13-10](#) shows one that gives the Python exception and details as part of the reply after a genuine exception is caught.

Figure 13-10. PyMailCgi login error page

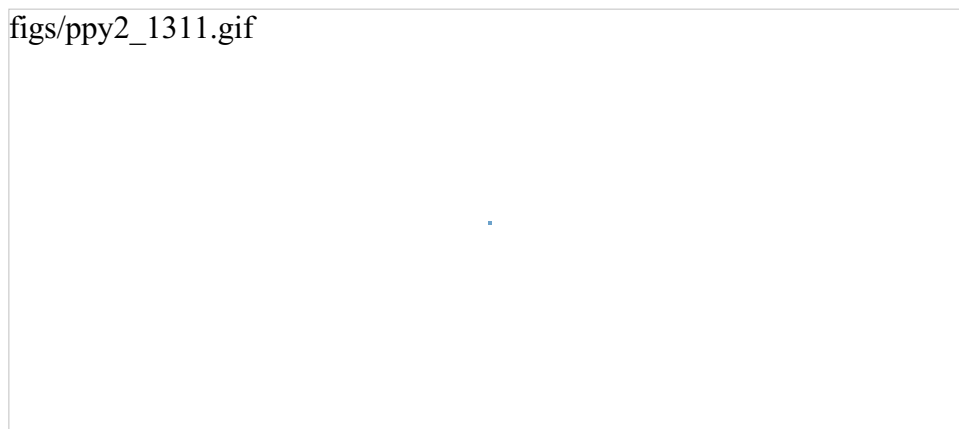


13.5.2.1 Passing state information in URL link parameters

The central mechanism at work in [Example 13-7](#) is the generation of URLs that embed message

numbers and mail account information. Clicking on any of the View links in the selection list triggers another script, which uses information in the link's URL parameters to fetch and display the selected email. As mentioned in the prior chapter, because the list's links are effectively programmed to "know" how to load a particular message, it's not too far-fetched to refer to them as smart links -- URLs that remember what to do next. [Figure 13-11](#) shows part of the HTML generated by this script.

Figure 13-11. PyMailCgi view list, generated HTML



Did you get all that? You may not be able to read generated HTML like this, but your browser can. For the sake of readers afflicted with human parsing limitations, here is what one of those lines looks like, reformatted with line breaks and spaces to make it easier to understand:

```
<tr><th><ahref="http://starship.python.net/~lutz/  
    PyMailCgi/onViewListLink.cgi  
        ?user=lutz&  
        mnum=66&  
        pswd=%8cg%c2P%1e%f3%5b%c5J%1c%f0&  
        site=pop.rmi.net">View</a> 66  
<td>test sendurl.py | lutz@rmi.net | Mon Jun  5 17:51:11 2000
```

PyMailCgi generates fully specified URLs (with server and pathname values imported from a common module). Clicking on the word "View" in the hyperlink rendered from this HTML code triggers the `onViewListLink` script as usual, passing it all the parameters embedded at the end of the URL: POP username, the POP message number of the message associated with this link, and POP password and site information. These values will be available in the object returned by `cgi.FieldStorage` in the next script run. Note that the `mnum` POP message number parameter differs in each link because each opens a different message when clicked, and that the text after `<td>` comes from message headers extracted with the `rfc822` module.

The `commonhtml` module escapes all of the link parameters with the `urllib` module, not `cgi.escape`, because they are part of a URL. This is obvious only in the `pswd` password parameter -- its value has been encrypted, but `urllib` additionally escapes non-safe characters in the encrypted string per URL convention (that's where all those `%xx` come from). It's okay if the encryptor yields odd -- even non-printable -- characters, because URL encoding makes them legible for transmission. When the password reaches the next script, `cgi.FieldStorage` undoes URL escape sequences, leaving the encrypted password string without `%` escapes.

It's instructive to see how `commonhtml` builds up the smart link parameters. Earlier, we learned how to use the `urllib.quote_plus` call to escape a string for inclusion in URLs:


```
>>> import urllib
>>> urllib.quote_plus("There's bugger all down here on Earth")
'There%27s+bugger+all+down+here+on+Earth'
```

Module `commonhtml`, though, calls the higher-level `urllib.urlencode` function, which translates a dictionary of `name:value` pairs into a complete URL parameter string, ready to add after a `?` marker in a URL. For instance, here is `urlencode` in action at the interactive prompt:

```
>>> parmdict = {'user': 'Brian',
...            'pswd': '#!/spam',
...            'text': 'Say no more, squire!'}
>>> urllib.urlencode(parmdict)
'pswd=%23%21/spam&user=Brian&text=Say+no+more,+squire%21'
>>> "%s?%s" % ("http://scriptname.cgi", urllib.urlencode(parmdict))
'http://scriptname.cgi?pswd=%23%21/spam&user=Brian&text=Say+no+more,+squire%21'
```

Internally, `urlencode` passes each name and value in the dictionary to the built-in `str` function (make sure they are strings) and then runs each one through `urllib.quote_plus` as they are added to the result. The CGI script builds up a list of similar dictionaries and passes it to `commonhtml` to be formatted into a selection list page.^[2]


[2] Technically, again, you should generally escape `&` separators in generated URL links like by running the URL through `cgi.escape`, if any parameter's name could be the same as that of an HTML character escape code (e.g., "`&=high`"). See the prior chapter for more details; they aren't escaped here because there are no clashes.

In broader terms, generating URLs with parameters like this is one way to pass state information to the next script (along with databases and hidden form input fields, discussed later). Without such state information, the user would have to re-enter the username, password, and site name on every page they visit along the way. We'll use this technique again in the next case study, to generate links that "know" how to fetch a particular database record.

Incidentally, the list generated by this script is not radically different in functionality from what we built in the `PyMailGui` program of [Chapter 11](#). [Figure 13-12](#) shows this strictly client-side GUI's view on the same email list displayed in [Figures 13-8](#) and [Figure 13-9](#).

Figure 13-12. PyMailGui displaying the same view list

figs/ppy2_1312.gif



However, PyMailGui uses the Tkinter GUI library to build up a user interface instead of sending HTML to a browser. It also runs entirely on the client and downloads mail from the POP server to the client machine over sockets on demand. In contrast, PyMailCgi runs on the server machine and simply displays mail text on the client's browser -- mail is downloaded from the POP server machine to the starship server, where CGI scripts are run. These architecture differences have some important ramifications, which we'll discuss in a few moments.

13.5.2.2 Security protocols

In `onViewPswdSubmit`'s source code ([Example 13-7](#)), notice that password inputs are passed to a `encode` function as they are added to the parameters dictionary, and hence show up encrypted in hyperlink URLs. They are also URL-encoded for transmission (with `%` escapes) and are later decoded and decrypted within other scripts as needed to access the POP account. The password encryption step, `encode`, is at the heart of PyMailCgi's security policy.

Beginning in Python 1.6, the standard socket module will include optional support for OpenSSL, an open source implementation of secure sockets that prevents transmitted data from being intercepted by eavesdroppers on the Net. Unfortunately, this example was developed under Python 1.5.2 and runs on a server whose Python did not have secure socket support built in, so an alternative scheme was devised to minimize the chance that email account information could be stolen off the Net in transit.

Here's how it works. When this script is invoked by the password input page's form, it gets only one input parameter: the password typed into the form. The username is imported from a `mailconfig` module installed on the server instead of transmitted together with the unencrypted password (that would be much too easy for malicious users to intercept).

To pass the POP username and password to the next page as state information, this script adds them to the end of the mail selection list URLs, but only after the password has been encrypted to `secret.encode` -- a function in a module that lives on the server and may vary in every location that PyMailCgi is installed. In fact, PyMailCgi was written to not have to know about the password encryptor at all; because the encoder is a separate module, you can provide any flavor you like. Unless you also publish your encoder module, the encoded password shipped with the username won't be of much help to snoopers.

That upshot is that normally, PyMailGui never sends or receives both user and password values together in a single transaction unless the password is encrypted with an encryptor of your choice. This limits its utility somewhat (since only a single account username can be installed on the server), but the alternative of popping up two pages -- one for password entry and one for user -- is even more unfriendly. In general, if you want to read your mail with the system as coded, you have to install its files on your server, tweak its `mailconfig.py` to reflect your account details, and change its `secret.py` encryptor as desired.

One exception: since any CGI script can be invoked with parameters in an explicit URL instead of form field values, and since `commonhtml` tries to fetch inputs from the form object before importing them from `mailconfig`, it is possible for any person to use this script to check his or her mail without installing and configuring a copy of PyMailCgi. For example, a URL like the following (but without the linebreak used to make it fit here):

```
http://starship.python.net/~lutz/PyMailCgi/  
onViewPswdSubmit.cgi?user=lutz&pswd=asif&site=pop.rmi.net
```

will actually load email into a selection list using whatever user, password, and mail site names are appended. From the selection list, you may then view, reply, forward, and delete email. Notice that at this point in the interaction, the password you send in a URL of this form is not encrypted. Later scripts expect that the password inputs will be sent encrypted, though, which makes it more difficult to use them with explicit URLs (you would need to match the encrypted form produced by the `secret` module on the server). Passwords are encrypted as they are added to links in the reply page's selection list, and remain encrypted in URLs and hidden form fields thereafter.

But please don't use a URL like this, unless you don't care about exposing your email password. Really. Sending both your unencrypted mail user ID and password strings across the Net in a URL like this is extremely unsafe and wide open to snoopers. In fact, it's like giving them a loaded gun -- anyone who intercepts this URL will have complete access to your email account. It is made even more treacherous by the fact that this URL format appears in a book that will be widely distributed all around the world.

If you care about security and want to use PyMailCgi, install it on your own server and configure `mailconfig` and `secret`. That should at least guarantee that your user and password information will never both be transmitted unencrypted in a single transaction. This scheme still is not foolproof, so be careful out there, folks. Without secure sockets, the Internet is a "use at your own risk" medium.

13.5.3 The Message View Page

Back to our page flow. At this point, we are still viewing the message selection list in [Figure 13-13](#). When we click on one of its generated hyperlinks, the smart URL invokes the script in [Example 13-8](#) on the server, sending the selected message number and mail account information (user, password, and site) as parameters on the end of the script's URL.

Example 13-8. PP2E\Internet\Cgi-Web\PyMailCgi\onViewListLink.cgi

```
#!/usr/bin/python
#####
# On user click of message link in main selection list;
# cgi.FieldStorage undoes any urllib escapes in the link's
# input parameters (%xx and '+' for spaces already undone);
#####

import cgi, rfc822, StringIO
import commonhtml, loadmail
from secret import decode
#commonhtml.dumpstatepage(0)

form = cgi.FieldStorage( )
user, pswd, site = commonhtml.getstandardpopfields(form)
try:
    msgnum    = form['mnum'].value                # from url link
    newmail   = loadmail.loadnewmail(site, user, decode(pswd))
    textfile  = StringIO.StringIO(newmail[int(msgnum) - 1]) # don't eval!
    headers   = rfc822.Message(textfile)
    bodytext  = textfile.read( )
    commonhtml.viewpage(msgnum, headers, bodytext, form) # encoded pswd
```

```
except:  
    commonhtml.errorpage('Error loading message')
```

Again, most of the work here happens in the `loadmail` and `commonhtml` modules, which are listed later in this section ([Example 13-12](#) and [Example 13-14](#)). This script adds logic to decode the input password (using the configurable `secret` encryption module) and extract the selected mail headers and text using the `rfc822` and `StringIO` modules, just as we did in [Chapter 11](#).^[3]

[3] Notice that the message number arrives as a string and must be converted to an integer in order to be used to fetch the message. But we're careful not to convert with `eval` here, since this is a string passed over the Net and could have arrived embedded at the end of an arbitrary URL (remember that earlier warning?).

If the message can be loaded and parsed successfully, the result page (shown in [Figure 13-13](#)) allows us to view, but not edit, the mail's text. The function `commonhtml.viewpage` generates a "read-only" HTML option for all the text widgets in this page.

Figure 13-13. PyMailCgi view page



View pages like this have a pull-down action selection list near the bottom; if you want to do more, use this list to pick an action (Reply, Forward, or Delete), and click on the Next button to proceed to the next screen. If you're just in a browsing frame of mind, click the "Back to root page" link at the bottom to return to the main page, or use your browser's Back button to return to the selection list page.

13.5.3.1 Passing state information in HTML hidden input fields

What you don't see on the view page in [Figure 13-13](#) is just as important as what you do. We need to refer to [Example 13-14](#) for details, but there's something new going on here. The original

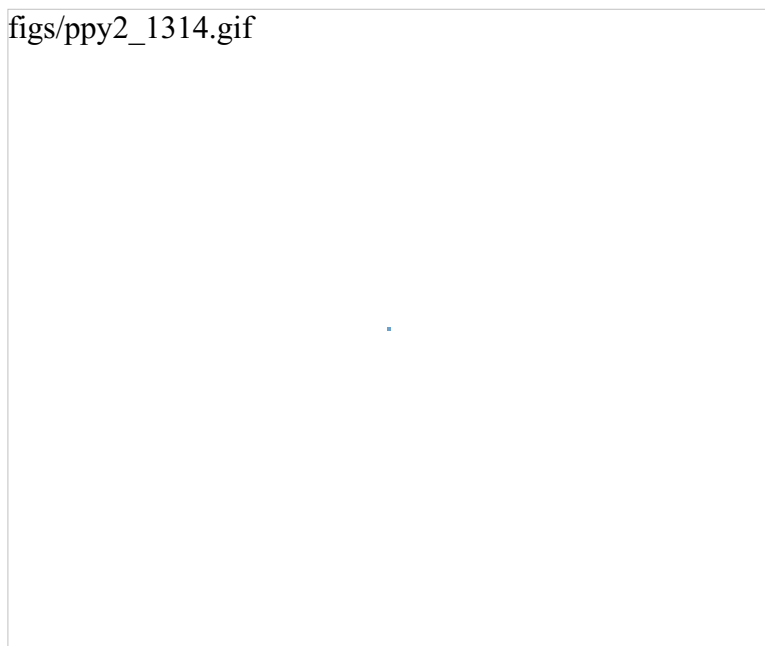
message number, as well as the POP user and (still encrypted) password information sent to this script as part of the smart link's URL, wind up being copied into the HTML used to create this view page, as the values of "hidden" input fields in the form. The hidden field generation code in `commonhtml` looks like this:

```
print '<form method=post action="%s/onViewSubmit.cgi">' % urlroot
print '<input type=hidden name=mnum value="%s">' % msgnum
print '<input type=hidden name=user value="%s">' % user      # from page|url
print '<input type=hidden name=site value="%s">' % site     # for deletes
print '<input type=hidden name=pswd value="%s">' % pswd     # pswd encoded
```

Much like parameters in generated hyperlink URLs, hidden fields in a page's HTML allow us to embed state information inside this web page itself. Unless you view that page's source, you can see this state information, because hidden fields are never displayed. But when this form's Submit button is clicked, hidden field values are automatically transmitted to the next script along with the visible fields on the form.

[Figure 13-14](#) shows the source code generated for a different message's view page; the hidden input fields used to pass selected mail state information are embedded near the top.

Figure 13-14. PyMailCgi view page, generated HTML



figs/ppy2_1314.gif

The net effect is that hidden input fields in HTML, just like parameters at the end of generated URLs, act like temporary storage areas and retain state between pages and user interaction steps. Both are the Web's equivalent to programming language variables. They come in handy any time your application needs to remember something between pages.

Hidden fields are especially useful if you cannot invoke the next script from a generated URL hyperlink with parameters. For instance, the next action in our script is a form submit button (Next), not a hyperlink, so hidden fields are used to pass state. As before, without these hidden fields, users would need to re-enter POP account details somewhere on the view page if they were needed by the next script (in our example, they are required if the next action is Delete).

13.5.3.2 Escaping mail text and passwords in HTML

Notice that everything you see on the message view page in [Figure 13-13](#) is escaped with `cgi.escape`. Header fields and the text of the mail itself might contain characters that are special to HTML and must be translated as usual. For instance, because some mailers allow you to send messages in HTML format, it's possible that an email's text could contain a `</textarea>` tag, which would throw the reply page hopelessly out of sync if not escaped.

One subtlety here: HTML escapes are important only when text is sent to the browser initially (to the CGI script). If that text is later sent out again to another script (e.g., by sending a reply), the text will be back in its original, non-escaped format when received again on the server. The browser parses out escape codes and does not put them back again when uploading form data, so we don't need to undo escapes later. For example, here is part of the escaped text area sent to a browser during a Reply transaction (use your browser's View Source option to see this live):

```
<tr><th align=right>Text:
<td><textarea name=text cols=80 rows=10 readonly>
more stuff

--Mark Lutz (http://rmi.net/~lutz) [PyMailCgi 1.0]

&gt; -----Original Message-----
&gt; From: lutz@rmi.net
&gt; To: lutz@rmi.net
&gt; Date: Tue May 2 18:28:41 2000
&gt;
&gt; &lt;table&gt;&lt;textarea&gt;
&gt; &lt;/textarea&gt;&lt;/table&gt;
&gt; --Mark Lutz (http://rmi.net/~lutz) [PyMailCgi 1.0]
&gt;
&gt;
&gt; &gt; -----Original Message-----
```

After this reply is delivered, its text looks as it did before escapes (and exactly as it appeared to the user in the message edit web page):

```
more stuff

--Mark Lutz (http://rmi.net/~lutz) [PyMailCgi 1.0]

> -----Original Message-----
> From: lutz@rmi.net
> To: lutz@rmi.net
> Date: Tue May 2 18:28:41 2000
>
> <table><textarea>
> </textarea></table>
> --Mark Lutz (http://rmi.net/~lutz) [PyMailCgi 1.0]
>
>
> > -----Original Message-----
```

Did you notice the odd characters in the hidden password field of the generated HTML screen shot ([Figure 13-14](#))? It turns out that the POP password is still encrypted when placed in hidden fields of the HTML. For security, the values of a page's hidden fields can be seen with a browser's View Source option, and it's not impossible that the text of this page could be intercepted off the Net.

The password is no longer URL-encoded when put in the hidden field, though, even though it was when it appeared at the end of the smart link URL. Depending on your encryption module, the password might now contain non-printable characters when generated as a hidden field value because the browser doesn't care, as long as the field is run through `cgi.escape` like everything else added to the HTML reply stream. The `commonhtml` module is careful to route all text and headers through `cgi.escape` as the view page is constructed.

As a comparison, [Figure 13-15](#) shows what the mail message captured in [Figure 13-13](#) looks like when viewed in PyMailGui, the client-side Tkinter-based email tool from [Chapter 11](#). PyMailGui doesn't need to care about things like passing state in URLs or hidden fields (it saves state in Python variables) or escaping HTML and URL strings (there are no browsers, and no network transmission steps once mail is downloaded). It does require Python to be installed on the client, but we'll get into that in a few pages.

Figure 13-15. PyMailGui viewer, same message



13.5.4 The Message Action Pages

At this point in our hypothetical PyMailCgi web interaction, we are viewing an email message ([Figure 13-13](#)) that was chosen from the selection list page. On the message view page, selecting an action from the pull-down list and clicking the Next button invokes the script in [Example 13-9](#) on the server to perform a reply, forward, or delete operation for the selected message.

Example 13-9. PP2E\Internet\Cgi-WebPyMailCgi\onViewSubmit.cgi

```
#!/usr/bin/python
# On submit in mail view window, action selected=(fwd, reply, delete)

import cgi, string
import commonhtml, secret
from externs import pmail, mailconfig
from commonhtml import getfield

def quotetext(form):
    """
```

```
note that headers come from the prior page's form here,
not from parsing the mail message again; that means that
commonhtml.viewpage must pass along date as a hidden field
"""
quoted = '\n-----Original Message-----\n'
for hdr in ('From', 'To', 'Date'):
    quoted = quoted + '%s: %s\n' % (hdr, getfield(form, hdr))
quoted = quoted + '\n' + getfield(form, 'text')
quoted = '\n' + string.replace(quoted, '\n', '\n> ')
return quoted

form = cgi.FieldStorage( ) # parse form or url data
user, pswd, site = commonhtml.getstandardpopfields(form)

try:
    if form['action'].value == 'Reply':
        headers = {'From': mailconfig.myaddress,
                  'To': getfield(form, 'From'),
                  'Cc': mailconfig.myaddress,
                  'Subject': 'Re: ' + getfield(form, 'Subject')}
        commonhtml.editpage('Reply', headers, quotetext(form))

    elif form['action'].value == 'Forward':
        headers = {'From': mailconfig.myaddress,
                  'To': '',
                  'Cc': mailconfig.myaddress,
                  'Subject': 'Fwd: ' + getfield(form, 'Subject')}
        commonhtml.editpage('Forward', headers, quotetext(form))

    elif form['action'].value == 'Delete':
        msgnum = int(form['mnum'].value) # or string.atol, but not eval(
        commonhtml.runsilent( # mnum field is required here
            pymail.deletemessages,
            (site, user, secret.decode(pswd), [msgnum], 0) )
        commonhtml.confirmationpage('Delete')

    else:
        assert 0, 'Invalid view action requested'
except:
    commonhtml.errorpage('Cannot process view action')
```

This script receives all information about the selected message as form input field data (some hidden, some not) along with the selected action's name. The next step in the interaction depends upon the action selected:

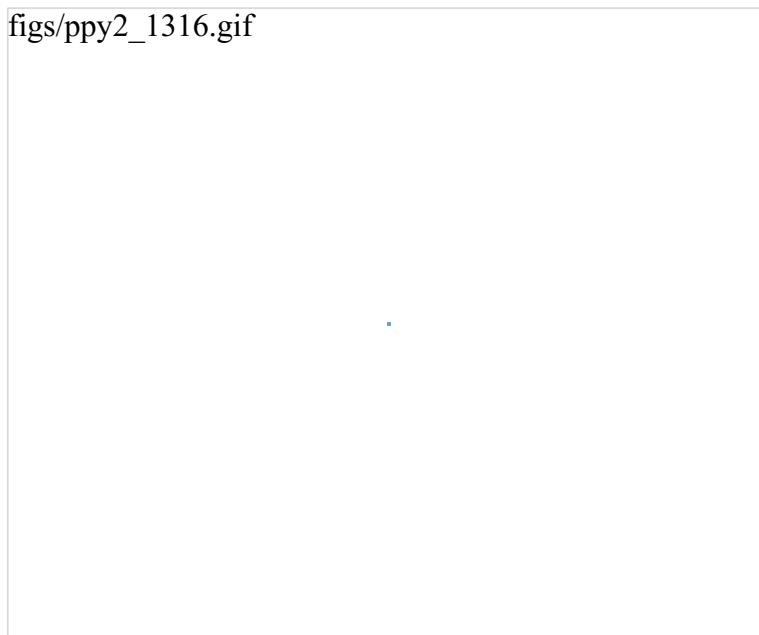
- Reply and Forward actions generate a message edit page with the original message's lines automatically quoted with a leading >.
- Delete actions trigger immediate deletion of the email being viewed, using a tool imported from the `pymail` module from [Chapter 11](#).

All these actions use data passed in from the prior page's form, but only the Delete action cares about the POP username and password and must decode the password received (it arrives here from hidden form input fields generated in the prior page's HTML).

13.5.4.1 Reply and forward

If you select Reply as the next action, the message edit page in [Figure 13-16](#) is generated by the script. Text on this page is editable, and pressing this page's Send button again triggers the send mail script we saw in [Example 13-4](#). If all goes well, we'll receive the same confirmation page we got earlier when writing new mail from scratch ([Figure 13-4](#)).

Figure 13-16. PyMailCgi reply page



Forward operations are virtually the same, except for a few email header differences. All of this busy-ness comes "for free," because Reply and Forward pages are generated by calling `commonhtml.editpage`, the same utility used to create a new mail composition page. Here, we simply pass the utility preformatted header line strings (e.g., replies add "Re:" to the subject text). We applied the same sort of reuse trick in PyMailGui, but in a different context. In PyMailCgi, one script handles three pages; in PyMailGui, one callback function handles three buttons, but the architecture is similar.

13.5.4.2 Delete

Selecting the Delete action on a message view page and pressing Next will cause the `onViewSubmit` script to immediately delete the message being viewed. Deletions are performed by calling a reusable delete utility function coded in [Chapter 11](#); the call to the utility is wrapped in a `commonhtml.runsilent` call that prevents `print` statements in the utility from showing up in the HTML reply stream (they are just status messages, not HTML code). [Figure 13-17](#) shows a delete operation in action.

Figure 13-17. PyMailCgi view page, delete selected





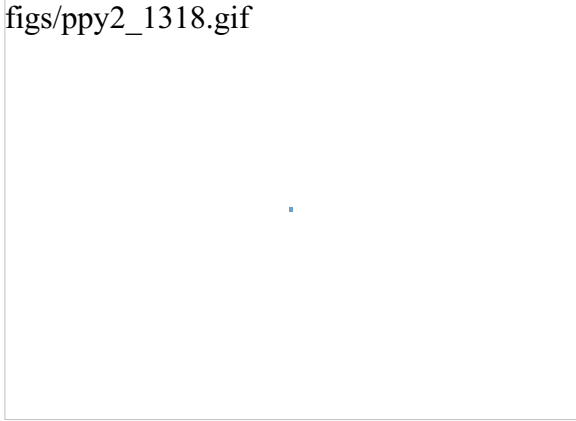
As mentioned, Delete is the only action that uses the POP account information (user, password, and site) that was passed in from hidden fields on the prior (message view) page. By contrast, the Reply and Forward actions format an edit page, which ultimately sends a message to the SMTP server; no POP information is needed or passed. But at this point in the interaction, the POP password has racked up more than a few frequent flyer miles. In fact, it may have crossed phone lines, satellite links, and continents on its journey from machine to machine. This process is illustrated here:

1. **Input (Client):** The password starts life by being typed into the login page on the client (or being embedded in an explicit URL), unencrypted. If typed into the input form in a web browser, each character is displayed as a star (*).
2. **Load index (Client to CGI server to POP server):** It is next passed from the client to the CGI server, which sends it on to your POP server in order to load a mail index. The client sends only the password, unencrypted.
3. **List page URLs (CGI server to client):** To direct the next script's behavior, the password is embedded in the mail selection list web page itself as hyperlink URL parameters, encrypted and URL-encoded.
4. **Load message (Client to CGI server to POP server):** When an email is selected from the list, the password is sent to the next script within the script's URL; the CGI script decrypts it and passes it on to the POP server to fetch the selected message.
5. **View page fields (CGI server to client):** To direct the next script's behavior, the password is embedded in the view page itself as HTML hidden input fields, encrypted and HTML-escaped.
6. **Delete (Client to CGI server to POP server):** Finally, the password is again passed from the client to CGI server, this time as hidden form field values; the CGI script decrypts it and passes it to the POP server to delete.

Along the way, scripts have passed the password between pages as both a URL parameter and an HTML hidden input field; either way, they have always passed its encrypted string, and never passed an unencrypted password and username together in any transaction. Upon a Delete request, the password must be decoded here using the `secret` module before passing it to the POP server. If the script can access the POP server again and delete the selected message, another confirmation page appears, as shown in [Figure 13-18](#).

Figure 13-18. PyMailCgi delete confirmation

figs/ppy2_1318.gif



Note that you really should click "Back to root page" after a successful deletion -- don't use your browser's Back button to return to the message selection list at this point, because the delete has changed the relative numbers of some messages in the list. PyMilGui worked around this problem by only deleting on exit, but PyMailCgi deletes mail immediately since there is no notion of "on exit." Clicking on a view link in an old selection list page may not bring up the message you think it should, if it comes after a message that was deleted.

This is a property of POP email in general: incoming mail simply adds to the mail list with higher message numbers, but deletions remove mail from arbitrary locations in the list and hence change message numbers for all mail following the ones deleted. Even PyMailGui may get some message numbers wrong if mail is deleted by another program while the GUI is open (e.g., in a second PyMailGui instance). Alternatively, both mailers could delete all email off the server as soon as it is downloaded, such that deletions wouldn't impact POP identifiers (Microsoft Outlook uses this scheme, for instance), but this requires additional mechanisms for storing deleted email persistently for later access.

One subtlety: for replies and forwards, the `onViewSubmit` mail action script builds up a >-quoted representation of the original message, with original "From:", "To:", and "Date:" header lines prepended to the mail's original text. Notice, though, that the original message's headers are fetched from the CGI form input, not by reparsing the original mail (the mail is not readily available at this point). In other words, the script gets mail header values from the form input fields of the view page. Because there is no "Date" field on the view page, the original message's date is also passed along to the action script as a hidden input field to avoid reloading the message. Try tracing through the code in this chapter's listings to see if you can follow dates from page to page.

13.6 Utility Modules

This section presents the source code of the utility modules imported and used by the page script shown above. There aren't any new screen shots to see here, because these are utilities, not CGI scripts (notice their *.py* extensions). Moreover, these modules aren't all that useful to study in isolation, and are included here primarily to be referenced as you go through the CGI scripts' code. See earlier in this chapter for additional details not repeated here.

13.6.1 External Components

When I install PyMailCgi and other server-side programs shown in this book, I simply upload the contents of the *Cgi-Web* examples directory on my laptop to the top-level web directory on my server account (*public_html*). The *Cgi-Web* directory also lives on this book's CD (see <http://examples.oreilly.com/python2>), a mirror of the one on my PC. I don't copy the entire book examples distribution to my web server, because code outside the *Cgi-Web* directory isn't designed to run on a web server.

When I first installed PyMailCgi, however, I ran into a problem: it's written to reuse modules coded in other parts of the book, and hence in other directories outside *Cgi-Web*. For example, it reuses the `mailconfig` and `pymail` modules we wrote in [Chapter 11](#), but neither lives in the CGI examples directory. Such external dependencies are usually okay, provided we use package imports or configure `sys.path` appropriately on startup. In the context of CGI scripts, though, what lives on my development machine may not be what is available on the web server machine where the scripts are installed.

To work around this (and avoid uploading the full book examples distribution to my web server) I define a directory at the top-level of *Cgi-Web* called *Extern*, to which any required external modules are copied as needed. For this system, *Extern* includes a subdirectory called *Email*, where the `mailconfig` and `pymail` modules are copied for upload to the server.

Redundant copies of files are less than ideal, but this can all be automated with install scripts that automatically copy to *Extern* and then upload *Cgi-Web* contents via FTP using Python's `ftplib` module (discussed in [Chapter 11](#)). Just in case I change this structure, though, I've encapsulated external name accesses in the utility module in [Example 13-10](#).

Example 13-10. PP2E\Internet\Cgi-Web\PyMailCgi\externs.py

```
#####  
# Isolate all imports of modules that live outside of the  
# PyMailCgi PyMailCgi directory. Normally, these would come  
# from PP2E.Internet.Email, but when I install PyMailCgi,  
# I copy just the Cgi-Web directory's contents to public_html  
# on the server, so there is no PP2E directory on the server.  
# Instead, I either copy the imports referenced in this file to  
# the PyMailCgi parent directory, or tweak the dir appended to  
# the sys.path module search path here. Because all other  
# modules get the externals from here, there is only one place  
# to change when they are relocated. This may be arguably  
# gross, but I only put Internet code on the server machine.
```

```
#####  
  
import sys  
sys.path.append('.') # see dir where Email installed on server  
from Extern import Email # assumes a ../Extern dir with Email dir  
from Extern.Email import pmail # can use names Email.pmail or pmail  
from Extern.Email import mailconfig
```

This module appends the parent directory of PyMailCgi to `sys.path` to make the *Extern* directory visible (remember, `PYTHONPATH` might be anything when CGI scripts are run as user "nobody") and preimports all external names needed by PyMailCgi into its own namespace. It also supports future changes; because all external references in PyMailCgi are made through this module, I have to change only this one file if externals are later installed differently.

As a reference, [Example 13-11](#) lists part of the external `mailconfig` module again. For PyMailCgi, it's copied to *Extern*, and may be tweaked as desired on the server (for example, the signature string differs slightly in this context). See the `pmail.py` file in [Chapter 11](#), and consider writing an automatic copy-and-upload script for the *Cgi-Web\Extern* directory a suggested exercise; it's not proved painful enough to compel me to write one of my own.

Example 13-11. PP2E\Internet\Cgi-Web\Extern\Email\mailconfig.py

```
#####  
# email scripts get server names from here:  
# change to reflect your machine/user names;  
# could get these in command line instead  
#####  
  
# SMTP email server machine (send)  
smtpservername = 'smtp.rmi.net' # or starship.python.net, 'localhost'  
  
# POP3 email server machine, user (retrieve)  
popservername = 'pop.rmi.net' # or starship.python.net, 'localhost'  
popusername = 'lutz' # password is requested when run  
  
...rest omitted  
  
# personal info used by PyMailGui to fill in forms;  
# sig-- can be a triple-quoted block, ignored if empty string;  
# addr--used for initial value of "From" field if not empty,  
  
myaddress = 'lutz@rmi.net'  
mysignature = '--Mark Lutz (http://rmi.net/~lutz) [PyMailCgi 1.0]'
```

13.6.2 POP Mail Interface

The `loadmail` utility module in [Example 13-12](#) depends on external files and encapsulates access to mail on the remote POP server machine. It currently exports one function, `loadnewmail`, which returns a list of all mail in the specified POP account; callers are unaware of whether this mail is fetched over the Net, lives in memory, or is loaded from a persistent storage medium on the CGI server machine. That is by design -- `loadmail` changes won't impact its clients.

Example 13-12. PP2E\Internet\Cgi-Web\PyMailCgi\loadmail.py

```
#####  
# mail list loader; future--change me to save mail list between  
# cgi script runs, to avoid reloading all mail each time; this  
# won't impact clients that use the interfaces here if done well;
```

```
# for now, to keep this simple, reloads all mail on each operation
#####

from commonhtml import runsilent          # suppress print's (no verbose flag)
from externs      import Email

# load all mail from number 1 up
# this may trigger an exception

def loadnewmail(mailserver, mailuser, mailpswd):
    return runsilent(Email.pymail.loadmessages,
                    (mailserver, mailuser, mailpswd))
```

It's not much to look at -- just an interface and calls to other modules. The `Email.pymail.loadmessages` function (reused here from [Chapter 11](#)) uses the Python `poplib` module to fetch mail over sockets. All this activity is wrapped in a `commonhtml.runsilent` function call to prevent `pymail` print statements from going to the HTML reply stream (although any `pymail` exceptions are allowed to propagate normally).

As it is, though, `loadmail` loads all incoming email to generate the selection list page, and reloads all email again every time you fetch a message from the list. This scheme can be horribly inefficient if you have lots of email sitting on your server; I've noticed delays on the order of a dozen seconds when my mailbox is full. On the other hand, servers can be slow in general, so the extra time taken to reload mail isn't always significant; I've witnessed similar delays on the server for empty mailboxes and simple HTML pages too.

More importantly, `loadmail` is intended only as a first-cut mail interface -- something of a usable prototype. If I work on this system further, it would be straightforward to cache loaded mail in a file, shelf, or database on the server, for example. Because the interface exported by `loadmail` would not need to change to introduce a caching mechanism, clients of this module would still work. We'll explore server storage options in the next chapter.

13.6.3 POP Password Encryption

Time to call the cops. We discussed the approach to password security adopted by `PyMailCgi` earlier. In brief, it works hard to avoid ever passing the POP account username and password across the Net together in a single transaction, unless the password is encrypted according to module `secret.py` on the server. This module can be different everywhere `PyMailCgi` is installed and can be uploaded anew at any time -- encrypted passwords aren't persistent and live only for the duration of one mail-processing interaction session.^[4] [Example 13-13](#) is the encryptor module I installed on my server while developing this book.

[4] Note that there are other ways to handle password security, beyond the custom encryption schemes described in this section. For instance, Python's `socket` module now supports the server-side portion of the OpenSSL secure sockets protocol. With it, scripts may delegate the security task to web browsers and servers. On the other hand, such schemes do not afford as good an excuse to introduce Python's standard encryption tools in this book.

Example 13-13. PP2E\Internet\Cgi-Web\PyMailCgi\secret.py

```
#####
# PyMailCgi encodes the pop password whenever it is sent to/from client over
# the net with a user name as hidden text fields or explicit url params; uses
```

```
# encode/decode functions in this module to encrypt the pswd--upload your own
# version of this module to use a different encryption mechanism; pymail also
# doesn't save the password on the server, and doesn't echo pswd as typed, but
# this isn't 100% safe--this module file itself might be vulnerable to some
# malicious users; Note: in Python 1.6, the socket module will include standard
# (but optional) support for openSSL sockets on the server, for programming
# secure Internet transactions in Python; see 1.6 socket module docs;
#####

forceReadablePassword = 0
forceRotorEncryption = 1

import time, string
dayofweek = time.localtime(time.time( ))[6]

#####
# string encoding schemes
#####

if not forceReadablePassword:
    # don't do anything by default: the urllib.quote or
    # cgi.escape calls in commonhtml.py will escape the
    # password as needed to embed in in URL or HTML; the
    # cgi module undoes escapes automatically for us;

    def stringify(old): return old
    def unstringify(old): return old

else:
    # convert encoded string to/from a string of digit chars,
    # to avoid problems with some special/nonprintable chars,
    # but still leave the result semi-readable (but encrypted);
    # some browser had problems with escaped ampersands, etc.;

    separator = '-'

    def stringify(old):
        new = ''
        for char in old:
            ascii = str(ord(char))
            new = new + separator + ascii # '-ascii-ascii-ascii'
        return new

    def unstringify(old):
        new = ''
        for ascii in string.split(old, separator)[1:]:
            new = new + chr(int(ascii))
        return new

#####
# encryption schemes
#####

if (not forceRotorEncryption) and (dayofweek % 2 == 0):
    # use our own scheme on evenly-numbered days (0=monday)
    # caveat: may fail if encode/decode over midnite boundary

    def do_encode(pswd):
        res = ''
        for char in pswd:
            res = res + chr(ord(char) + 1) # add 1 to each ascii code
        return str(res)

    def do_decode(pswd):
        res = ''
        for char in pswd:
            res = res + chr(ord(char) - 1)
```

```
        return res

else:
    # use the standard lib's rotor module to encode pswd
    # this does a better job of encryption than code above

    import rotor
    mykey = 'pymailcgi'

    def do_encode(pswd):
        robj = rotor.newrotor(mykey)                # use enigma encryption
        return robj.encrypt(pswd)

    def do_decode(pswd):
        robj = rotor.newrotor(mykey)
        return robj.decrypt(pswd)

#####
# top-level entry points
#####

def encode(pswd):
    return stringify(do_encode(pswd))                # encrypt plus string encode

def decode(pswd):
    return do_decode(unstringify(pswd))
```

This encryptor module implements two alternative encryption schemes: a simple ASCII character code mapping, and Enigma-style encryption using the standard `rotor` module. The `rotor` module implements a sophisticated encryption strategy, based on the "Enigma" encryption machine used by the Nazis to encode messages during World War II. Don't panic, though; Python's `rotor` module is much less prone to cracking than the Nazis'!

In addition to encryption, this module also implements an encoding method for already-encrypted strings. By default, the encoding functions do nothing, and the system relies on straight URL encoding. An optional encoding scheme translates the encrypted string to a string of ASCII code digits separated by dashes. Either encoding method makes non-printable characters in the encrypted string printable.

13.6.3.1 Default encryption scheme: rotor

To illustrate, let's test this module's tools interactively. First off, we'll experiment with Python's standard `rotor` module, since it's at the heart of the default encoding scheme. We import the module, make a new rotor object with a key (and optionally, a rotor count), and call methods to encrypt and decrypt:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
>>> import rotor
>>> r = rotor.newrotor('pymailcgi')                # (key, [,numrotors])
>>> r.encrypt('abc123')                            # may return non-printable chars
' \323an\021\224'

>>> x = r.encrypt('spam123')                       # result is same len as input
>>> x
'* _\344\011pY'
>>> len(x)
7
>>> r.decrypt(x)
'spam123'
```


Notice that the same rotor object can encrypt multiple strings, that the result may contain non-printable characters (printed as `\ascii` escape codes when displayed, possibly in octal form), and that the result is always the same length as the original string. Most importantly, a string encrypted with `rotor` can be decrypted in a different process (e.g., in a later CGI script) if we recreate the rotor object:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
>>> import rotor
>>> r = rotor.newrotor('pymailcgi')           # can be decrypted in new process
>>> r.decrypt('* _\344\011pY')                # use "\ascii" escapes for two chars
'spam123'
```

Our `secret` module by default simply uses `rotor` to encrypt, and does no additional encoding of its own. It relies on URL encoding when the password is embedded in a URL parameter, and HTML escaping when the password is embedded in hidden form fields. For URLs, the following sorts of calls occur:

```
>>> from secret import encode, decode
>>> x = encode('abc$#<>&+')                  # CGI scripts do this (rotor)
>>> x
' \323a\016\317\326\023\0163'

>>> import urllib                             # urllib.urlencode does this
>>> y = urllib.quote_plus(x)
>>> y
'+%d3a%0e%cf%d6%13%0e3'

>>> a = urllib.unquote_plus(y)                 # cgi.FieldStorage does this
>>> a
' \323a\016\317\326\023\0163'

>>> decode(a)                                 # CGI scripts do this (rotor)
'abc$#<>&+'
```

13.6.3.2 Alternative encryption schemes

To show how to write alternative encryptors and encoders, `secret` also includes a digits-string encoder and a character-code shuffling encryptor; both are enabled with global flag variables at the top of the module:

`forceReadablePassword`

If set to true, the encrypted password is encoded into a string of ASCII code digits separated by dashes. Defaults to false to fall back on URL and HTML escape encoding.

`forceRotorEncryption`

If set to false and the encryptor is used on an even-numbered day of the week, the simple character-code encryptor is used instead of `rotor`. Defaults to true to force rotor encryption.

To show how these alternatives work, let's set `forceReadablePassword` to 1 and `forceRotorEncryption` to 0, and `reimport`. Note that these are global variables that must be set before the module is imported (or reloaded), because they control the selection of alternative `def` statements. Only one version of each kind of function is ever made by the module:

```
C:\...\PP2E\Internet\Cgi-Web\PyMailCgi>python
```

```
>>> from secret import *
>>> x = encode('abc$#<>&+')
>>> x
'-98-99-100-37-36-61-63-39-44'

>>> y = decode(x)
>>> y
'abc$#<>&+'
```

This really happens in two steps, though -- encryption and then encoding (the top-level `encode` and `decode` functions orchestrate the two steps). Here's what the steps look like when run separately:

```
>>> t = do_encode('abc$#<>&+')           # just our encryption
>>> t
"bcd%$=?',"
>>> stringify(t)                          # add our own encoding
'-98-99-100-37-36-61-63-39-44'

>>> unstringify(x)                       # undo encoding
"bcd%$=?',"
>>> do_decode(unstringify(x))            # undo both steps
'abc$#<>&+'
```

This alternative encryption scheme merely adds 1 to the each character's ASCII code value, and the encoder inserts the ASCII code integers of the result. It's also possible to combine `rotor` encryption and our custom encoding (set both `forceReadablePassword` and `forceRotorEncryption` to 1), but URL encoding provided by `urllib` works just as well. Here are a variety of schemes in action; `secret.py` is edited and saved before each reload:

```
>>> import secret
>>> secret.encode('spam123')             # default: rotor, no extra encoding
'* _\344\011pY'

>>> reload(secret)                      # forcereadable=1, forcerotor=0
<module 'secret' from 'secret.py'>
>>> secret.encode('spam123')
'-116-113-98-110-50-51-52'

>>> reload(secret)                      # forcereadable=1, forcerotor=1
<module 'secret' from 'secret.py'>
>>> secret.encode('spam123')
'-42-32-95-228-9-112-89'
>>> ord('Y')                            # the last one is really a 'Y'
89

>>> reload(secret)                      # back to default rotor, no stringify
<module 'secret' from 'secret.pyc'>
>>> import urllib
>>> urllib.quote_plus(secret.encode('spam123'))
'%2a+_%e4%09pY'
>>> 0x2a                                 # the first is really 42, '*'
42
>>> chr(42)
'*'
```

You can provide any kind of encryption and encoding logic you like in a custom `secret.py`, as long as it adheres to the expected protocol -- encoders and decoders must receive and return a string. You can also alternate schemes by days of the week as done here (but note that this can fail if your system is being used when the clock turns over at midnight!), and so on. A few final pointers:

Other Python encryption tools

There are additional encryption tools that come with Python or are available for Python or the Web; see <http://www.python.org> and the library manual for details. Some encryption schemes are considered serious business and may be protected by law from export, but the rules change over time.

Secure sockets support

As mentioned, Python 1.6 (not yet out as I wrote this) will have standard support for OpenSSL secure sockets in the Python `socket` module. OpenSSL is an open source implementation of the secure sockets protocol (you must fetch and install it separately from Python -- see <http://www.openssl.org>). Where it can be used, this will provide a better and less limiting solution for securing information like passwords than the manual scheme we adopted here.

For instance, secure sockets allow usernames and passwords to be entered into and submitted from a single web page, thereby supporting arbitrary mail readers. The best we can do without secure sockets is to either avoid mixing unencrypted user and password values and assume that some account data and encryptors live on the server (as done here) or to have two distinct input pages or URLs (one for each value). Neither scheme is as use friendly as a secure sockets approach. Most browsers already support SSL; to add it to Python on your server, see the Python 1.6 (and beyond) library manual.

Internet security is a much bigger topic than can be addressed fully here, and we've really only scratched its surface. For additional information on security issues, consult books geared exclusively towards web programming techniques.

On my server, the *secret.py* file will be changed over time, in case snoopers watch the book's web site. Moreover, its source code cannot be viewed with the `getfile` CGI script coded in [Chapter 12](#). That means that if you run this system live, passwords in URLs and hidden form fields may look very different than seen in this book. My password will have changed by the time you read these words too, or else it would be possible to know my password from this book alone!

13.6.4 Common Utilities Module

The file *commonhtml.py*, shown in [Example 13-14](#), is the Grand Central Station of this application -- its code is used and reused by just about every other file in the system. Most of it is self-explanatory, and I've already said most of what I wanted to say about it earlier, in conjunction with the CGI scripts that use it.

I haven't talked about its debugging support, though. Notice that this module assigns `sys.stderr` to `sys.stdout`, in an attempt to force the text of Python error messages to show up in the client's browser (remember, uncaught exceptions print details to `sys.stderr`). That works sometimes in PyMailCgi, but not always -- the error text shows up in a web page only if a `page_header` call has already printed a response preamble. If you want to see all error messages, make sure you call `page_header` (or `print Content-type: lines` manually) before any other processing. This module

also defines functions that dump lots of raw CGI environment information to the browser (dumpstatepage), and that wrap calls to functions that print status messages so their output isn't added to the HTML stream (runsilent).

I'll leave the discovery of any remaining magic in this code up to you, the reader. You are hereby admonished to go forth and read, refer, and reuse.

Example 13-14. PP2E\Internet\Cgi-Web\PyMailCgi\commonhtml.py

```
#!/usr/bin/python
#####
# generate standard page header, list, and footer HTML;
# isolates html generation-related details in this file;
# text printed here goes over a socket to the client,
# to create parts of a new web page in the web browser;
# uses one print per line, instead of string blocks;
# uses urllib to escape parms in url links auto from a
# dict, but cgi.escape to put them in html hidden fields;
# some of the tools here are useful outside pymailcgi;
# could also return html generated here instead of
# printing it, so it could be included in other pages;
# could also structure as a single cgi script that gets
# and tests a next action name as a hidden form field;
# caveat: this system works, but was largely written
# during a 2-hour layover at the Chicago O'Hare airport:
# some components could probably use a bit of polishing;
# to run standalone on starship via a commandline, type
# "python commonhtml.py"; to run standalone via a remote
# web browser, rename file with .cgi and run fixcgi.py.
#####

import cgi, urllib, string, sys
sys.stderr = sys.stdout          # show error messages in browser
from externs import mailconfig   # from a package somewhere on server

# my address root
urlroot = 'http://starship.python.net/~lutz/PyMailCgi'

def pageheader(app='PyMailCgi', color='#FFFFFF', kind='main', info=''):
    print 'Content-type: text/html\n'
    print '<html><head><title>%s: %s page (PP2E)</title></head>' % (app, kind)
    print '<body bgcolor="%s"><h1>%s %s</h1><hr>' % (color, app, (info or kind))

def pagefooter(root='pymailcgi.html'):
    print '</p><hr><a href="http://www.python.org">'
    print '</a>'
    print '<a href="%s">Back to root page</a>' % root
    print '</body></html>'

def formatlink(cgiurl, parmdict):
    """
    make "%url?key=val&key=val" query link from a dictionary;
    escapes str( ) of all key and val with %xx, changes ' ' to +
    note that url escapes are different from html (cgi.escape)
    """
    parmtext = urllib.urlencode(parmdict)          # calls urllib.quote_plus
    return '%s%s' % (cgiurl, parmtext)           # urllib does all the work

def pagelistsimple(linklist):                    # show simple ordered list
    print '<ol>'
    for (text, cgiurl, parmdict) in linklist:
        link = formatlink(cgiurl, parmdict)
        text = cgi.escape(text)
```

```
        print '<li><a href="%s">\n      %s</a>' % (link, text)
    print '</ol>'

def pagelisttable(linklist):
    print '<p><table border>'
    count = 1
    for (text, cgiurl, parmdict) in linklist:
        link = formatlink(cgiurl, parmdict)
        text = cgi.escape(text)
        print '<tr><th><a href="%s">View</a> %d<td>\n %s' % (link, count, text)
        count = count+1
    print '</table>'

def listpage(linkslst, kind='selection list'):
    pageheader(kind=kind)
    pagelisttable(linkslst)
    pagefooter( )

def messagearea(headers, text, extra=''):
    print '<table border cellpadding=3>'
    for hdr in ('From', 'To', 'Cc', 'Subject'):
        val = headers.get(hdr, '?')
        val = cgi.escape(val, quote=1)
        print '<tr><th align=right>%s:' % hdr
        print '      <td><input type=text '
        print '        name=%s value="%s" %s size=60>' % (hdr, val, extra)
    print '<tr><th align=right>Text:'
    print '<td><textarea name=text cols=80 rows=10 %s>' % extra
    print '%s\n</textarea></table>' % (cgi.escape(text) or '?') # if has </>s

def viewpage(msgnum, headers, text, form):
    """
    on View + select (generated link click)
    very subtle thing: at this point, pswd was url encoded in the
    link, and then unencoded by cgi input parser; it's being embedded
    in html here, so we use cgi.escape; this usually sends nonprintable
    chars in the hidden field's html, but works on ie and ns anyhow:
    in url: ?user=lutz&mnum=3&pswd=%8cg%c2P%1e%f0%5b%c5J%1c%f3&...
    in html: <input type=hidden name=pswd value="...nonprintables..">
    could urllib.quote the html field here too, but must urllib.unquote
    in next script (which precludes passing the inputs in a URL instead
    of the form); can also fall back on numeric string fmt in secret.py
    """
    pageheader(kind='View')
    user, pswd, site = map(cgi.escape, getstandardpopfields(form))
    print '<form method=post action="%s/onViewSubmit.cgi">' % urlroot
    print '<input type=hidden name=mnum value="%s">' % msgnum
    print '<input type=hidden name=user value="%s">' % user # from page|url
    print '<input type=hidden name=site value="%s">' % site # for deletes
    print '<input type=hidden name=pswd value="%s">' % pswd # pswd encoded
    messagearea(headers, text, 'readonly')

    # onViewSubmit.quotetext needs date passed in page
    print '<input type=hidden name=Date value="%s">' % headers.get('Date', '?')
    print '<table><tr><th align=right>Action:'
    print '<td><select name=action>'
    print '      <option>Reply<option>Forward<option>Delete</select>'
    print '<input type=submit value="Next">'
    print '</table></form>' # no 'reset' needed here
    pagefooter( )

def editpage(kind, headers={}, text=''):
    # on Send, View+select+Reply, View+select+Fwd
    pageheader(kind=kind)
    print '<form method=post action="%s/onSendSubmit.cgi">' % urlroot
    if mailconfig.mysignature:
        text = '\n%s\n%s' % (mailconfig.mysignature, text)
```

```
messagearea(headers, text)
print '<input type=submit value="Send">'
print '<input type=reset value="Reset">'
print '</form>'
pagefooter( )

def errorpage(message):
    pageheader(kind='Error') # or sys.exc_type/exc_value
    exc_type, exc_value = sys.exc_info( )[:2] # but safer, thread-specif
    print '<h2>Error Description</h2><p>', message
    print '<h2>Python Exception</h2><p>', cgi.escape(str(exc_type))
    print '<h2>Exception details</h2><p>', cgi.escape(str(exc_value))
    pagefooter( )

def confirmationpage(kind):
    pageheader(kind='Confirmation')
    print '<h2>%s operation was successful</h2>' % kind
    print '<p>Press the link below to return to the main page.</p>'
    pagefooter( )

def getfield(form, field, default=''):
    # emulate dictionary get method
    return (form.has_key(field) and form[field].value) or default

def getstandardpopfields(form):
    """
    fields can arrive missing or '' or with a real value
    hard-coded in a url; default to mailconfig settings
    """
    return (getfield(form, 'user', mailconfig.popusername),
            getfield(form, 'pswd', '?'),
            getfield(form, 'site', mailconfig.popservername))

def getstandardsmtpfields(form):
    return getfield(form, 'site', mailconfig.smtpservername)

def runsilent(func, args):
    """
    run a function without writing stdout
    ex: suppress print's in imported tools
    else they go to the client/browser
    """
    class Silent:
        def write(self, line): pass
    save_stdout = sys.stdout
    sys.stdout = Silent( ) # send print to dummy obje
    try: # which has a write method
        result = apply(func, args) # try to return func result
    finally: # but always restore stdout
        sys.stdout = save_stdout
    return result

def dumpstatepage(exhaustive=0):
    """
    for debugging: call me at top of a cgi to
    generate a new page with cgi state details
    """
    if exhaustive:
        cgi.test( ) # show page with form, environ, etc.
    else:
        pageheader(kind='state dump')
        form = cgi.FieldStorage( ) # show just form fields names/values
        cgi.print_form(form)
        pagefooter( )
    sys.exit( )

def selftest(showastable=0): # make phony web page
```

```
links = [
    ('text1', urlroot + '/page1.cgi', {'a':1}),
    ('text2', urlroot + '/page1.cgi', {'a':2, 'b':'3'}),
    ('text3', urlroot + '/page2.cgi', {'x':'a b', 'y':'a<b&c', 'z':'?'}),
    ('te<>4', urlroot + '/page2.cgi', {'<x>':'', 'y':'<a>', 'z':None})]
pageheader(kind='View')
if showastable:
    pagelisttable(links)
else:
    pagelistsimple(links)
pagefooter( )

if __name__ == '__main__':
    selftest(len(sys.argv) > 1)
# when run, not imported
# html goes to stdout
```

13.7 CGI Script Trade-offs

As shown in this chapter, PyMailCgi is still something of a system in the making, but it does work as advertised: by pointing a browser at the main page's URL, I can check and send email from anywhere I happen to be, as long as I can find a machine with a web browser. In fact, any machine and browser will do: Python doesn't even have to be installed.^[5] That's not the case with the PyMailGui client-side program we wrote in [Chapter 11](#).

[5] This property can be especially useful when visiting government institutions, which seem to generally provide web browser accessibility, but restrict administrative functions and broader network connectivity to officially cleared system administrators (and international spies).

But before we all jump on the collective Internet bandwagon and utterly abandon traditional APIs like Tkinter, a few words of larger context are in order. Besides illustrating larger CGI applications in general, this example was chosen to underscore some of the trade-offs you run into when building applications to run on the Web. PyMailGui and PyMailCgi do roughly the same things, but are radically different in implementation:

- PyMailGui is a traditional user-interface program: it runs entirely on the local machine, calls out to an in-process GUI API library to implement interfaces, and talks to the Internet through sockets only when it has to (e.g., to load or send email on demand). User requests are routed immediately to callback handler functions or methods running locally, with shared variables that automatically retain state between requests. For instance, PyMailGui only loads email once, keeps it in memory, and only fetches newly arrived messages on future loads because its memory is retained between events.
- PyMailCgi, like all CGI systems, consists of scripts that reside and run on a server machine, and generate HTML to interact with a user at a web browser on the client machine. It runs only in the context of a web browser, and handles user requests by running CGI scripts remotely on the server. Unless we add a real database system, each request handler runs autonomously, with no state information except that which is explicitly passed along by prior states as hidden form fields or URL parameters. As coded, PyMailCgi must reload all email whenever it needs to process incoming email in any way.

On a basic level, both systems use the Python POP and SMTP modules to fetch and send email through sockets. But the implementation alternatives they represent have some critical ramifications that you should know about when considering delivering systems on the Web:

Performance costs

Networks are slower than CPUs . As implemented, PyMailCgi isn't nearly as fast or as

complete as PyMailGui. In PyMailCgi, every time the user clicks a submit button, the request goes across the network. More specifically, every user request incurs a network transfer overhead, every callback handler (usually) takes the form of a newly spawned process on the server, parameters come in as text strings that must be parsed out, and the lack of state information on the server between pages means that mail needs to be reloaded often. In contrast, user clicks in PyMailGui trigger in-process function calls instead of network traffic and process forks, and state is easily saved as Python in-process variables (e.g., the loaded-mail list is retained between clicks). Even with an ultra-fast Internet connection, a server-side CGI system is slower than a client-side program.^[6]

[6] To be fair, some Tkinter operations are sent to the underlying Tcl library as strings too, which must be parsed. This may change in time; but the contrast here is with CGI scripts versus GUI libraries in general, not with a particular library's implementation.

Some of these bottlenecks may be designed away at the cost of extra program complexity. For instance, some web servers use threads and process pools to minimize process creation for CGI scripts. Moreover, some state information can be manually passed along from page to page in hidden form fields and generated URL parameters, and state can be saved between pages in a concurrently accessible database to minimize mail reloads (see the PyErrata case study in [Chapter 14](#) for an example). But there's no getting past the fact that routing events over a network to scripts is much slower than calling a Python function directly.

Complexity costs

HTML isn't pretty . Because PyMailCgi must generate HTML to interact with the user in a web browser, it is also more complex (or at least, less readable) than PyMailGui. In some sense, CGI scripts embed HTML code in Python. Because the end result of this is a mixture of two very different languages, creating an interface with HTML in a CGI script can be much less straightforward than making calls to a GUI API such as Tkinter.

Witness, for example, all the care we've taken to escape HTML and URLs in this chapter's examples; such constraints are grounded in the nature of HTML. Furthermore, changing the system to retain loaded-mail list state in a database between pages would introduce further complexities to the CGI-based solution. Secure sockets (e.g., OpenSSL, to be supported in Python 1.6) would eliminate manual encryption costs, but introduce other overheads.

Functionality costs

HTML can only say so much. HTML is a portable way to specify simple pages and forms, but is poor to useless when it comes to describing more complex user interfaces. Because CGI scripts create user interfaces by writing HTML back to a browser, they are highly limited in terms of user-interface constructs.

For example, consider implementing an image-processing and animation program as CGI scripts: HTML doesn't apply once we leave the domain of fill-out forms and simple interactions. This is precisely the limitation that Java applets were designed to address -- programs that are stored on a server but pulled down to run on a client on demand, and given access to a full-featured GUI API for creating richer user interfaces. Nevertheless, strictly server-side programs are inherently limited by the constraints of HTML. The animation scripts we wrote at the end of [Chapter 8](#), for example, are well beyond the scope of server-side scripts.

Portability benefits

All you need is a browser . On the client side, at least. Because PyMailCgi runs over the Web, it can be run on any machine with a web browser, whether that machine has Python and Tkinter installed or not. That is, Python needs to be installed on only one computer: the web server machine where the scripts actually live and run. As long as you know that the users of your system have an Internet browser, installation is simple.

Python and Tkinter, you will recall, are very portable too -- they run on all major window systems (X, Windows, Mac) -- but to run a client-side Python/Tk program such as PyMailGui, you need Python and Tkinter on the client machine itself. Not so with an application built as CGI scripts: they will work on Macintosh, Linux, Windows, and any other machine that can somehow render HTML web pages. In this sense, HTML becomes a sort of portable GUI API language in CGI scripts, interpreted by your web browser. You don't even need the source code or bytecode for the CGI scripts themselves -- they run on a remote server that exists somewhere else on the Net, not on the machine running the browser.

Execution requirements

But you do need a browser. That is, the very nature of web-enabled systems can render them useless in some environments. Despite the pervasiveness of the Internet, there are still plenty of applications that run in settings that don't have web browsers or Internet access. Consider, for instance, embedded systems, real-time systems, and secure government applications. While an Intranet (a local network without external connections) can sometimes make web applications feasible in some such environments, I have recently worked at more than one company whose client sites had no web browsers to speak of. On the other hand, such clients may be more open to installing systems like Python on local machines, as opposed to supporting an internal or external network.

Administration requirements

You really need a server too . You can't write CGI-based systems at all without access to a web sever. Further, keeping programs on a centralized server creates some fairly critical administrative overheads. Simply put, in a pure client/server architecture, clients are simpler, but the server becomes a critical path resource and a potential performance bottleneck. If the centralized server goes down, you, your employees, and your customers may be knocked out of commission. Moreover, if enough clients use a shared server at the same time, the speed costs of web-based systems become even more pronounced. In fact, one could make the argument that moving towards a web server architecture is akin to stepping backwards in time -- to the time of centralized

mainframes and dumb terminals. Whichever way we step, offloading and distributing processing to client machines at least partially avoids this processing bottleneck.

So what's the best way to build applications for the Internet -- as client-side programs that talk to the Net, or as server-side programs that live and breathe on the Net? Naturally, there is no one answer to that question, since it depends upon each application's unique constraints. Moreover, there are more possible answers to it than we have proposed here; most common CGI problems already have common proposed solutions. For example:

Client-side solutions

Client- and server-side programs can be mixed in many ways. For instance, applet programs live on a server, but are downloaded to and run as client-side programs with access to rich GUI libraries (more on applets when we discuss JPython in [Chapter 15](#)). Other technologies, such as embedding JavaScript or Python directly in HTML code, also support client-side execution and richer GUI possibilities; such scripts live in HTML on the server, but run on the client when downloaded and access browser components through an exposed object model (see the discussion [Section 15.8](#) near the end of [Chapter 15](#)). The emerging Dynamic HTML (DHTML) extensions provide yet another client-side scripting option for changing web pages after they have been constructed. All of these client-side technologies add extra complexities all their own, but ease some of the limitations imposed by straight HTML.

State retention solutions

Some web application servers (e.g., Zope, described in [Chapter 15](#)) naturally support state retention between pages by providing concurrently accessible object databases. Some of these systems have a real underlying database component (e.g., Oracle and mySql); others may make use of files or Python persistent object shelves with appropriate locking (as we'll explore in the next chapter). Scripts can also pass state information around in hidden form fields and generated URL parameters, as done in PyMailCgi, or store it on the client machine itself using the standard cookie protocol.

Cookies are bits of information stored on the client upon request from the server. A cookie is created by sending special headers from the server to the client within the response HTML (`Set-Cookie: name=value`). It is then accessed in CGI scripts as the value of a special environment variable containing cookie data uploaded from the client (`HTTP_COOKIE`). Search <http://www.python.org> for more details on using cookies in Python scripts, including the freely available `cookie.py` module, which automates the cookie translation process.^[7] Cookies are more complex than program variables and are somewhat controversial (some see them as intrusive), but they can offload some simple state retention tasks.

[7] Also see the new standard `cookie` module in Python release 2.0.

HTML generation solutions

Add-ons can also take some of the complexity out of embedding HTML in Python CGI scripts, albeit at some cost to execution speed. For instance, the HTMLgen system described in [Chapter 15](#) lets programs build pages as trees of Python objects that

"know" how to produce HTML. When a system like this is employed, Python scripts deal only with objects, not the syntax of HTML itself. Other systems such as PHP and Active Server Pages (described in the same chapter) allow scripting language code to be embedded in HTML and executed on the server, to dynamically generate or determine part of the HTML that is sent back to a client in response to requests.

Clearly, Internet technology does imply some design trade-offs, and is still evolving rapidly. It is nevertheless an appropriate delivery context for many (though not all) applications. As with every design choice, you must be the judge. While delivering systems on the Web may have some costs in terms of performance, functionality, and complexity, it is likely that the significance of those overheads will diminish with time.

Chapter 13. Larger Web Site Examples I

[Section 13.1. "Things to Do When Visiting Chicago"](#)

[Section 13.2. The PyMailCgi Web Site](#)

[Section 13.3. The Root Page](#)

[Section 13.4. Sending Mail by SMTP](#)

[Section 13.5. Reading POP Email](#)

[Section 13.6. Utility Modules](#)

[Section 13.7. CGI Script Trade-offs](#)

14.1 "Typos Happen"

This chapter presents the second of two server-side Python web programming case studies. It covers the design and implementation of PyErrata, a CGI-based web site implemented entirely in Python that allows users to post book comments and error reports, and demonstrates the concepts underlying persistent database storage in the CGI world. As we'll see, this case study teaches both server-side scripting and Python development techniques.

14.2 The PyErrata Web Site

The last chapter concluded with a discussion of the downsides of deploying applications on the Web. But now that I've told you all the reasons you might not want to design systems for the Web, I'm going to completely contradict myself and present a system that cries out for a web-based implementation. This chapter presents the PyErrata web site, a Python program that lets arbitrary people on arbitrary machines submit book comments and bug reports (usually called errata) over the Web, using just a web browser.

PyErrata is in some ways simpler than the PyMailCgi case study presented in the previous chapter. From a user's perspective, PyErrata is more hierarchical than linear: user interactions are shorter and spawn fewer pages. There is also little state retention in web pages themselves in PyErrata; URL parameters pass state in only one isolated case, and no hidden form fields are generated.

On the other hand, PyErrata introduces an entirely new dimension: persistent data storage. State (error and comment reports) is stored permanently by this system on the server, either in flat pickle files or a shelf-based database. Both raise the specter of concurrent updates, since any number of users out in cyberspace may be accessing the site at the same time.

14.2.1 System Goals

Before you ponder too long over the seeming paradox of a book that comes with its own bug-reporting system, I should provide a little background. Over the last five years, I've been fortunate enough to have had the opportunity to write four books, a large chapter in a reference book, and various magazine articles and training materials. Changes in the Python world have also provided opportunities to rewrite books from the ground up. It's been both wildly rewarding and lucrative work (well, rewarding, at least).

But one of the first big lessons one learns upon initiation in the publishing business is that typos are a fact of life. Really. No matter how much of a perfectionist you are, books will have bugs. Furthermore, big books tend to have more bugs than little books, and in the technical publishing domain, readers are often sufficiently savvy and motivated to send authors email when they find those bugs.

That's a terrific thing, and helps authors weed out typos in reprints. I always encourage and appreciate email from readers. But I get lots of email -- at times, so much so that given my schedule, I find it difficult to even reply to every message, let alone investigate and act on every typo report. I get lots of other email too, and can miss a reader's typo report if I'm not careful.

About a year ago, I realized that I just couldn't keep up with all the traffic and started thinking about alternatives. One obvious way to cut down on the overhead of managing reports is to delegate responsibility -- to offload at least some report-processing tasks to the people who generate the reports. That is, I needed to somehow provide a widely available system, separate from my email account, that automates report posting and logs reports to be reviewed as time allows.

Of course, that's exactly the sort of need that the Internet is geared to. By implementing an error-reporting system as a web site, any reader can visit and log reports from any machine with a browser, whether they have Python installed or not. Moreover, those reports can be logged in a database at the web site for later inspection by both author and readers, instead of requiring manual extraction from incoming email.

The implementation of these ideas is the PyErrata system -- a web site implemented with server-side Python programs. PyErrata allows readers to post bug reports and comments about this edition of Programming Python, as well as view the collection of all prior posts by various sort keys. Its goal is to replace the traditional errata list pages I've had to maintain manually for other books in the past.

More than any other web-based example in this book, PyErrata demonstrates just how much work can be saved with a little Internet scripting. To support the first edition of this book, I hand-edited an HTML file that listed all known bugs. With PyErrata, server-side programs generate such pages dynamically from a user-populated database. Because list pages are produced on demand, PyErrata not only publishes and automates list creation, it also provides multiple ways to view report data. I wouldn't even try to reorder the first edition's static HTML file list.


PyErrata is something of an experiment in open systems, and as such is vulnerable to abuse. I still have to manually investigate reports, as time allows. But it at least has the potential to ease one of the chores that generally goes unmentioned in typical publishing contracts.

14.2.2 Implementation Overview

Like other web-based systems in this part of the book, PyErrata consists of a collection of HTML files, Python utility modules, and Python-coded CGI scripts that run on a shared server instead of on a client. Unlike those other web systems, PyErrata also implements a persistent database and defines additional directory structures to support it. [Figure 14-1](#) shows the top-level contents of the site, seen on Windows from a PyEdit Open dialog.

Figure 14-1. PyErrata site contents

figs/ppy2_1401.gif



You will find a similar structure on this book's CD-ROM (view CD-ROM content online at <http://examples.oreilly.com/python2>). To install this site on the Net, all the files and directories you see here are uploaded to the server machine and stored in a *PyErrata* subdirectory within the root of the directory that is exposed to the Web (my *public_html* directory). The top-level files of this site implement browse and submit operations as well as database interfaces. A few resource page files and images show up in this listing too, but are ignored in this book. Besides files, this site has subdirectories of its own:

- *Mutex* is a Python package that contains a mutual-exclusion utility module used for shelves, as well as test scripts for this utility model.
- *AdminTools* includes system utility scripts that are run standalone from the command line.
- *DbaseFiles* holds the file-based database, with separate subdirectories for errata and comment pickle files.
- *DbaseShelve* contains the shelve-based database, with separate shelve files for errata and comments.

We'll meet the contents of the database subdirectories later in this chapter, when exploring the database implementation.

14.2.3 Presentation Strategy

PyErrata takes logic factoring, code reuse, and encapsulation to extremes. Top-level scripts, for example, are often just a few lines long and ultimately invoke generic logic in common utility modules. With such an architecture, mixing short code segments with lots of screen shots makes it tough to trace the flow of control through the program.

To make this system easier to study, we're going to take a slightly different approach here. *PyErrata*'s implementation will be presented in three main sections corresponding to major functional areas of the system: report browsing, report submission, and database interfaces. The site root page will be shown before these three sections, but mostly just for context; it's simple, static HTML.

Within the browsing and submission sections, all user interaction models (and screen shots) are shown first, followed by all the source code used to implement that interaction. Like the *PyForm* example in [Chapter 16](#), *PyErrata* is at heart a database-access program, and its database interfaces are ultimately the core of the system. Because these interfaces encapsulate most low-level storage details, though, we'll save their presentation for last.

Although you still may have to jump around some to locate modules across functional boundaries, this organization of all the code for major chunks of the system in their own sections should help minimize page-flipping.

Use the Source, Luke

I want to insert the standard case-study caveat here: although this chapter does explain major concepts along the way, understanding the whole story is left partly up to you. As always, please consult the source code listings in this chapter (and at <http://examples.oreilly.com/python2>) for details not spelled out explicitly. I've taken this minimal approach deliberately, mostly because I assume you already know a lot about CGI scripting and the Python language by this point in the book, but also because real-world development time is spent as much on reading other people's code as on writing your own. Python makes both tasks relatively easy, but now is your chance to see how for yourself.

I also wish to confess right off that this chapter has a hidden agenda. PyErrata not only shows more server-side scripting techniques, but also illustrates common Python development concepts at large. Along the way, we focus on this system's current software architecture and point out a variety of design alternatives. Be sure to pay special attention to the way that logic has been layered into multiple abstraction levels. For example, by separating database and user-interface (page generation) code, we minimize code redundancy and cross-module dependencies and maximize code reuse. Such techniques are useful in all Python systems, web-based or not.

14.3 The Root Page

Let's start at the top. In this chapter we will study the complete implementation of PyErrata, but readers are also encouraged to visit the web site where it lives to sample the flavor of its interaction first-hand. Unlike PyMailCgi, there are no password constraints in PyErrata, so you can access all of its pages without any configuration steps.

PyErrata installs as a set of HTML files and Python CGI scripts, along with a few image files. As usual, you can simply point your web browser to the system's root page to run the system live while you study this chapter. Its root page currently lives here:^[1]

[1] But be sure to see this book's web site, <http://rmi.net/~lutz/about-pp.html>, for an updated link if the one listed here no longer works by the time you read this book. Web sites seem to change addresses faster than developers change jobs.

<http://starship.python.net/~lutz/PyErrata/pyerrata.html>

If you go to this address, your browser will be served the page shown in [Figure 14-2](#). PyErrata supports both submission and browsing of comments and error reports; the four main links on this page essentially provide write and read access to its databases over the Web.

Figure 14-2. PyErrata main page



The static HTML code file downloaded to produce this page is listed in [Example 14-1](#). The

only parts we're interested in are shown in bold: links to the submission and browsing pages for comments and errata. There is more to this page, but we're only dealing with the parts shown in the screen shot. For instance, the site will eventually also include resource page HTML files (e.g., Python resources and changes), but we'll ignore those components in this book.

Example 14-1. PP2E\Internet\Cgi-Web\PyErrata\pyerrata.html

```
<HTML><BODY>
<TITLE>PyErrata: PP2E Errata Page</TITLE>
<H1 align=center>PyErrata</H1>
<H2 align=center>The PP2E Updates Page</H2>
<P align=center><I>Version 1.0, November 1999</I></P>

<HR><P>
<A href="http://rmi.net/~lutz/about-pp.html">
<IMG src="ppsmall.gif" align=left alt="[Book Cover]" border=1 hspace=8></A>

Welcome. This is the official place where corrections, supplements,
and other supporting information for the book <I>Programming Python,
2nd Edition</I> are maintained. This site is also described in the book:
most of its interaction is implemented in
<A HREF="http://rmi.net/~lutz/about-python.html">Python</A> as server-side
CGI scripts, and most submitted information is stored in files on the starship
server.
<P>
You may both browse items, and submit new ones here. This site is primarily
used for automatic, reader-controlled tracking of book corrections ("errata");
if you find a bug, please take a moment to fill out the errata submission
form, so we can fix it in a later printing. Select a link below to submit
or browse book-related items.
</P>
<HR>

<H2>Submit</H2>
<UL>
<LI><A href="submitErrata.html">Errata report</A>
<LI><A href="submitComment.html">General comment</A>
</UL>

<H2>Browse</H2>
<UL>
<LI><A href="browseErrata.html">Errata reports</A>
<LI><A href="browseComments.html">General comments</A>
</UL>

<H2>Library</H2>
<UL>
<LI><A href="resourceSupplements.html">Supplements</A>
<LI><A href="resourcePythonchanges.html">Python changes</A>
<LI><A href="resourcePatchfiles.html">Program patch files</A>
</UL>

<HR>
<A href="http://www.python.org">
<IMG SRC="PythonPoweredSmall.gif"
ALIGN=left ALT="[Python Logo]" border=0 hspace=10></A>
<A href="http://PyInternetDemos.html">More examples</A>
</BODY></HTML>
```


14.4 Browsing PyErrata Reports

On to the first major system function: browsing report records. Before we study the code used to program browse operations, let's get a handle on the sort of user interaction it is designed to produce. If you're the sort that prefers to jump into code right away, it's okay to skip the next two sections for now, but be sure to come back here to refer to the screen shots as you study code listed later.

14.4.1 User Interface: Browsing Comment Reports

As shown in [Figure 14-2](#), PyErrata lets us browse and submit two kinds of reports: general comments and errata (bug) reports. Clicking the "General comments" link in the Browse section of the root page brings up the page shown in [Figure 14-3](#).

Figure 14-3. Browse comments, selection page



figs/ppy2_1403.gif

Now, the first thing you should know about PyErrata's browse feature is that it allows users to query and view the report database in multiple ways. Reports may be ordered by any report field and displayed in three different formats. The top-level browse pages essentially serve to configure a query against the report database and the presentation of its result.

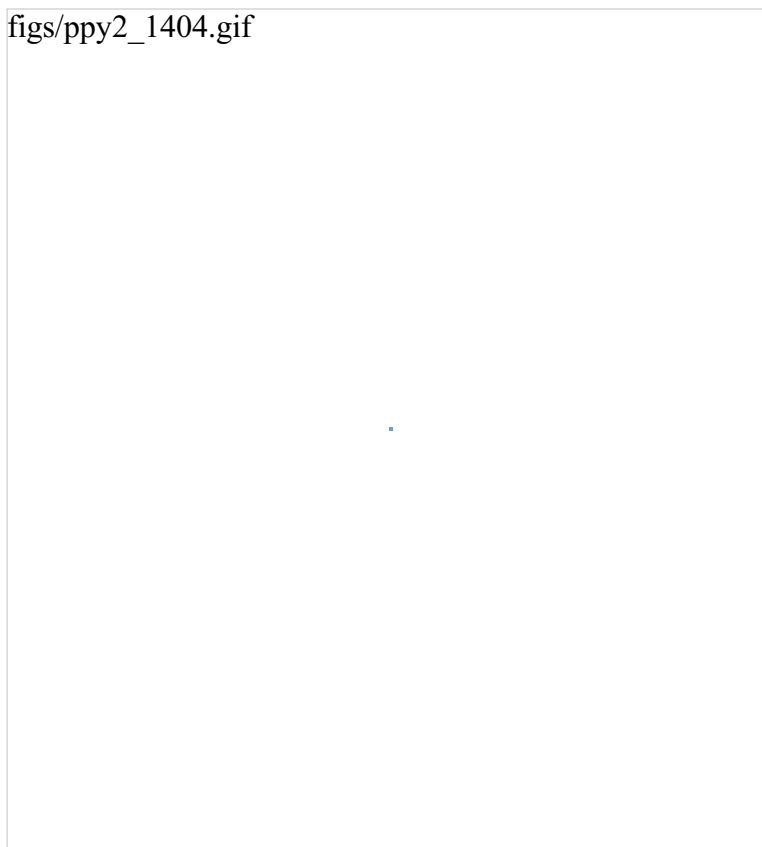
To specify an ordering, first select a sort criterion: a report field name by which report listings are ordered. Fields take the form of radio buttons on this page. To specify a report display format, select one of three option buttons:

- Simple list yields a simple sorted list page.
- With index generates a sorted list page, with hyperlinks at the top that jump to the starting point of each sort key value in the page when clicked.
- Index only produces a page containing only hyperlinks for each sort key value, which fetch and display matching records when clicked.

[Figure 14-4](#) shows the simple case produced by clicking the "Submit date" sort key button, selecting the "Simple list" display option, and pressing the Submit Query button to contact a Python script on the server. It's a scrollable list of all comment reports in the database ordered by submission date.

Figure 14-4. Browse comments, "Simple list" option

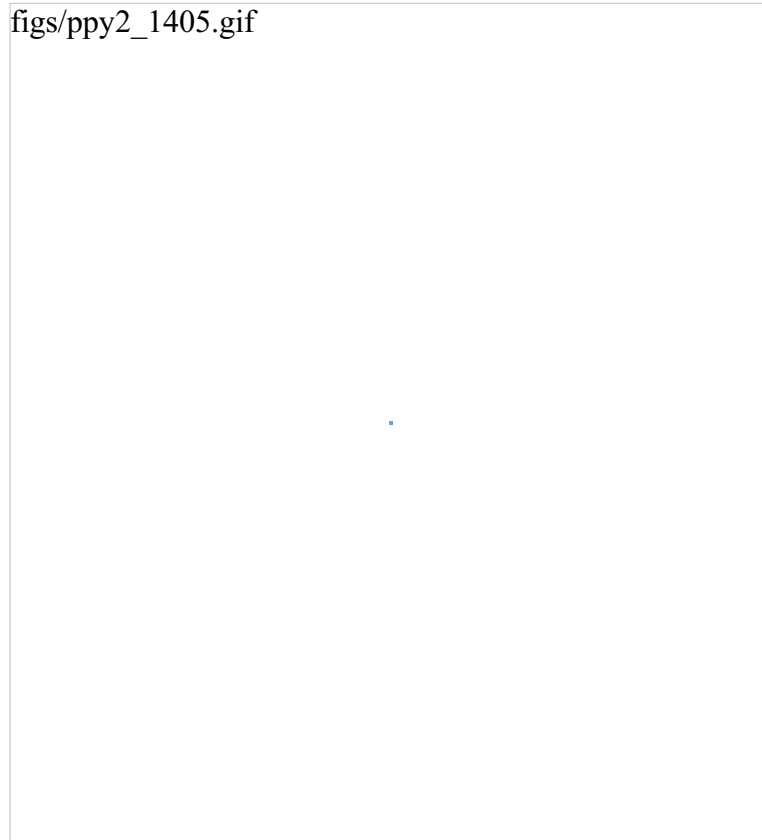
figs/ppy2_1404.gif



In all query results, each record is displayed as a table of attribute field values (as many as are present in the record) followed by the text of the record's description field. The description is typically multiple lines long, so it's shown separately and without any HTML reformatting (i.e., as originally typed). If there are multiple records in a list, they are separated by horizontal lines.

Simple lists like this work well for small databases, but the other two display options are better suited to larger report sets. For instance, if we instead pick the "With index" option, we are served up a page that begins with a list of links to other locations in the page, followed by a list of records ordered and grouped by a sort key's value. [Figure 14-5](#) shows the "With index" option being used with the "Report state" sort key.

Figure 14-5. Browse comments, "With index" option



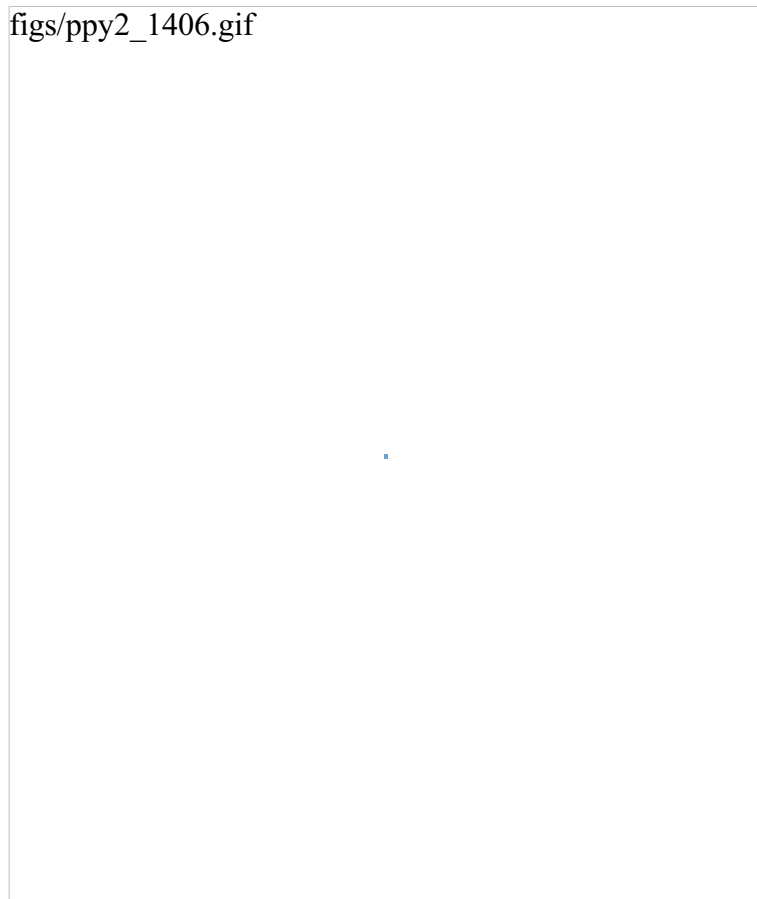
To view reports, the user can either scroll through the list or click on one of the links at the top; they follow in-page hyperlinks to sections of the report list where a given key value's records begin. Internally, these hyperlinks use `file.html#section` section-link syntax that is supported by most browsers, and in-page tags. The important parts of the generated HTML code look like this:

```
<title>PP2E Comment list</title>
<h1>Comment list, sorted by "Report state"</h1><hr>
<h2>Index</h2><ul>
<li><a href="#S0">Not yet verified</a>
<li><a href="#S1">Rejected - not a real bug</a>
<li><a href="#S2">Verified by author</a>
</ul><hr>
<h2><a name="#S0">Key = "Not yet verified"</a></h2><hr>
<p><table border>
<tr><th align=right>Submit date:<td>1999/09/21, 06:07:43
...more...
```

[Figure 14-6](#) shows the result of clicking one such link in a page sorted instead by submit date. Notice the #s4 at the end of the result's URL. We'll see how these tags are automatically generated in a moment.

Figure 14-6. Browse comments, "With index" listing

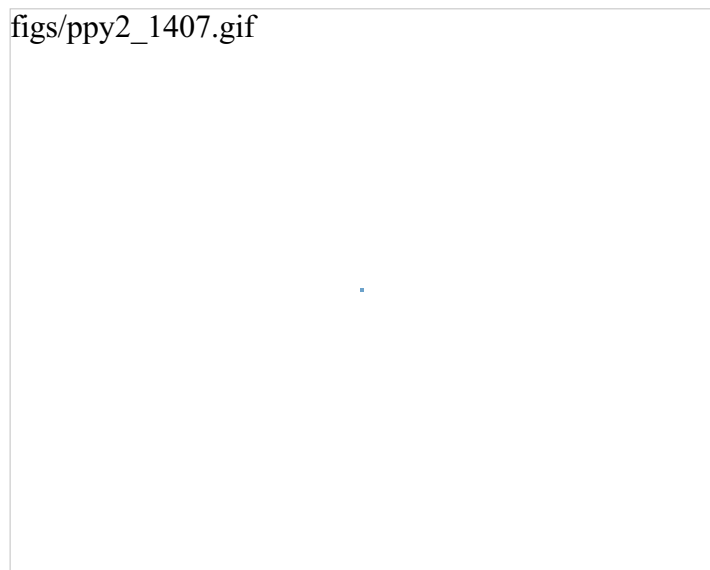
figs/ppy2_1406.gif



For very large databases, it may be impractical to list every record's contents on the same page; the third PyErrata display format option provides a solution. [Figure 14-7](#) shows the page produced by the "Index only" display option, with "Submit date" chosen for report order. There are no records on this page, just a list of hyperlinks that "know" how to fetch records with the listed key value when clicked. They are another example of what we've termed smart links -- they embed key and value information in the hyperlink's URL.

Figure 14-7. Browse comments, "Index only" selection list

figs/ppy2_1407.gif



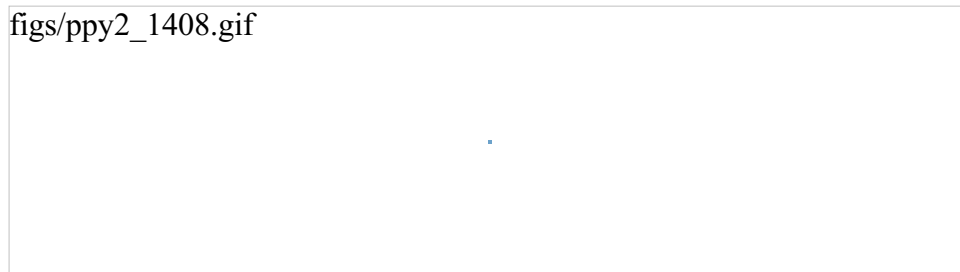
PyErrata generates these links dynamically; they look like the following, except that I've added line-feeds to make them more readable in this book:

```
<title>PP2E Comment list</title>
<h1>Comment list, sorted by "Submit date"</h1><hr>
<h2>Index</h2><ul>
<li><a href="index.cgi?kind=Comment&
                sortkey=Submit+date&
                value=1999/09/21,+06%3a06%3a50">1999/09/21, 06:06:50</a>
<li><a href="index.cgi?kind=Comment&
                sortkey=Submit+date&
                value=1999/09/21,+06%3a07%3a22">1999/09/21, 06:07:22</a>
...more...
</ul><hr>
```

Note the URL-encoded parameters in the links this time; as you'll see in the code, this is Python's `urllib` module at work again. Also notice that unlike the last chapter's PyMailCgi example, PyErrata generates minimal URLs in lists (without server and path names -- they are inferred and added by the browser from the prior page's address). If you view the generated page's source code, the underlying smart links are more obvious; [Figure 14-8](#) shows one such index page's code. ^[2]

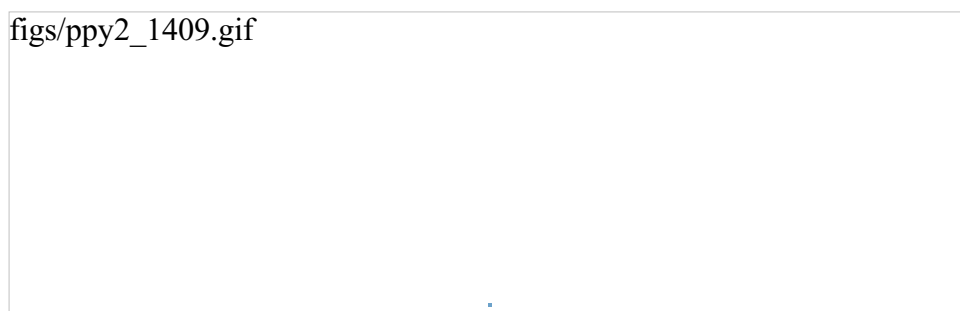
[2] Like PyMailCgi, the `&` character in the generated URLs is not escaped by PyErrata, since its parameter name doesn't clash with HTML character escape names. If yours might, be sure to use `cgi.escape` on URLs to be inserted into web pages.

Figure 14-8. PyErrata generated links code



Clicking on a link in the "Index only" page fetches and displays all records in the database with the displayed value in the displayed key field. For instance, pressing the second to last link in the index page ([Figure 14-7](#)) yields the page shown in [Figure 14-9](#). As usual, generated links appear in the address field of the result.

Figure 14-9. Browse comments, "Index only" link clicked





If we ask for an index based on field "Submitter name," we generate similar results but with different key values in the list and URLs; [Figure 14-10](#) shows the result of clicking such an index page link. This is the same record as [Figure 14-9](#), but was accessed via name key, not submit date. By treating records generically, PyErrata provides multiple ways to view and access stored data.

Figure 14-10. Browse comments, "Index only" page




14.4.2 User Interface: Browsing Errata Reports

PyErrata maintains two distinct databases -- one for general comments and one for genuine error reports. To PyErrata, records are just objects with fields; it treats both comments and errata the same, and is happy to use whatever database it is passed. Because of that, the interface for browsing errata records is almost identical to that for comments, and as we'll see in the implementation section, it largely uses the same code.

Errata reports differ, though, in the fields they contain. Because there are many more fields that can be filled out here, the root page of the errata browse function is slightly different. As seen in [Figure 14-11](#), sort fields are selected from a pull-down selection list rather than radiobuttons. Every attribute of an errata report can be used as a sort key, even if some reports have no value for the field selected. Most fields are optional; as we'll see later, reports with empty field values are shown as value ? in index lists and grouped under value `(none)` in report lists.

Figure 14-11. Browse errata, selection page


figs/ppy2_1411.gif



Once we've picked a sort order and display format and submitted our query, things look much the same as for comments (albeit with labels that say Errata instead of Comment). For instance, [Figure 14-12](#) shows the "With index" option for errata sorted by submit date.

Figure 14-12. Browse errata, "With index" display


figs/ppy2_1412.gif



Clicking one of the links on this page leads to a section of the report page list, as in [Figure 14-13](#) again, the URL at the top uses `#section` hyperlinks.

Figure 14-13. Browse errata, report list

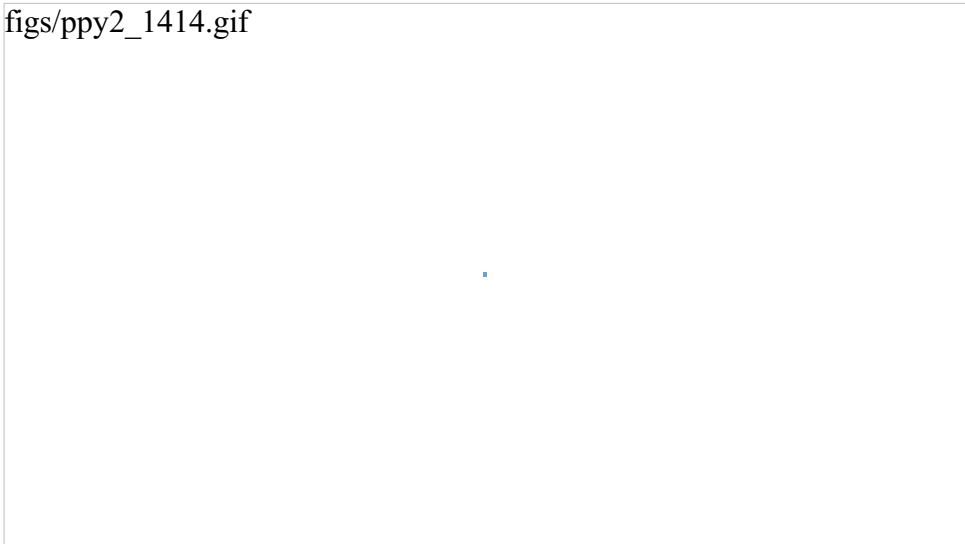
figs/ppy2_1413.gif



The "Index only" mode works the same here too: [Figure 14-14](#) shows the index page for sort field "Chapter number". Notice the "?" entry; if clicked, it will fetch and display all records with an empty chapter number field. In the display, their empty key values print as `(none)`. In the database, it's really an empty string.

Figure 14-14. Browse errata, "Index only" link page


figs/ppy2_1414.gif



Clicking on the "16" entry brings up all errata tagged with that chapter number in the database; [Figure 14-15](#) shows that only one was found this time.

Figure 14-15. Browse errata, "Index only" link clicked

figs/ppy2_1415.gif



14.4.3 Using Explicit URLs with PyErrata

Because Python's `cgi` module treats form inputs and URL parameters the same way, you can also use explicit URLs to generate most of the pages shown so far. In fact, PyErrata does too; the URL shown at the top of [Figure 14-15](#):

```
http://starship.python.net/~lutz/  
PyErrata/index.cgi?kind=Errata&sortkey=Chapter+number&value=16
```

was generated by PyErrata internally to represent a query to be sent to the next script (mostly -- the browser actually adds the first part, through *PyErrata/*). But there's nothing preventing a user (or another script) from submitting that fully specified URL explicitly to trigger a query and reply. Other pages can be fetched with direct URLs too; this one loads the index page itself:

```
http://starship.python.net/~lutz/  
PyErrata/browseErrata.cgi?key=Chapter+number&display=indexonly
```

Likewise, if you want to query the system for all comments submitted under a given name, you can either navigate through the system's query pages, or type a URL like this:

```
http://starship.python.net/~lutz/  
PyErrata/index.cgi?kind=Comment&sortkey=Submitter+name&value=Bob
```

You'll get a page with Python exception information if there are no matches for the key and value in the specified database. If you instead just want to fetch a comment list sorted by submit dates (e.g., to parse in another script), type this:

```
http://starship.python.net/~lutz/  
PyErrata/browseComments.cgi?key=Submit+date&display=list
```

If you access this system outside the scope of its form pages like this, be sure to specify a complete URL and URL-encoded parameter values. There is no notion of a prior page, and because most key values originate from values in user-provided reports, they may contain arbitrary characters.

It's also possible to use explicit URLs to submit new reports -- each field may be passed as a URL's parameter to the submit script:

```
http://starship.python.net/~lutz/  
PyErrata/submitComment.cgi?Description=spam&Submitter+name=Bob
```

but we won't truly understand what this does until we reach [Section 14.5](#) later in this chapter.

14.4.4 Implementation: Browsing Comment Reports

Okay, now that we've seen the external behavior of the browse function, let's roll up our sleeves and dig into its implementation. The following sections list and discuss the source code files that implement PyErrata browse operations. All of these live on the web server; some are static HTML files and others are executable Python scripts. As you read, remember to refer back to the user interface sections to see the sorts of pages produced by the code.

As mentioned earlier, this system has been factored for reuse: top-level scripts don't do much but call out to generalized modules with appropriate parameters. The database where submitted reports are stored is completely encapsulated as well; we'll study its implementation later in this chapter, but for now we can be mostly ignorant of the medium used to store information.

The file in [Example 14-2](#) implements the top-level comment browsing page.

Example 14-2. PP2E\Internet\Cgi-Web\PyErrata\browseComments.html

```
<html><body bgcolor="#FFFFFF">  
<title>PP2E Browse Comments</title>  
<h1>PP2E Browse Comment Reports</h1>  
  
<p>Please select the field you wish to sort by below, and press  
the submit button to display comments. The display does not include  
any emailed reports which have not been manually posted. Click  
'With index' for a top-of-page index, or 'Index only' for an index  
with links to individual pages.  
</p>  
  
<hr>  
<form method=POST action="browseComments.cgi">  
  <h3>Sort reports by:</h3>  
  
  <p><input type=radio name=key value="Submit date" checked> Submit date  
<p><input type=radio name=key value="Submitter name"> Submitter name  
<p><input type=radio name=key value="Submitter email"> Submitter email  
<p><input type=radio name=key value="Report state"> Report state  
  
  <h3>Display options:</h3>  
  <p><input type=radio name=display value="list">Simple list  
    <input type=radio name=display value="indexed" checked>With index  
    <input type=radio name=display value="indexonly">Index only  
<p><input type=submit>  
</form>  
  
<hr>  
<a href="pyerrata.html">Back to errata page</A>  
</body></html>
```

This is straight and static HTML code, as opposed to a script (there's nothing to construct dynamically here). As with all forms, clicking its submit button triggers a CGI script ([Example 14-3](#)) on the server, passing all the input fields' values.

Example 14-3. PP2E\Internet\Cgi-Web\PyErrata\browseComments.cgi

```
#!/usr/bin/python

from dbswitch import DbaseComment      # dbfiles or dbshelve
from browse import generatePage        # reuse html formatter
generatePage(DbaseComment, 'Comment')  # load data, send page
```

There's not much going on here, because all the machinery used to perform a query has been spl off to the `browse` module (shown in [Example 14-6](#)) so that it can be reused to browse errata reports too. Internally, browsing both kinds of records is handled the same way; here, we pass in only items that vary between comment and errata browsing operations. Specifically, we pass in the comment database object and a "Comment" label for use in generated pages. Module `browse` is happy to query and display records from any database we pass to it.

The `dbswitch` module used here (and listed in [Example 14-13](#)) simply selects between flat-file and shelve database mechanisms. By making the mechanism choice in a single module, we need to update only one file to change to a new medium; this CGI script is completely independent of the underlying database mechanism. Technically, the object `dbswitch.DbaseComment` is a class object, used later to construct a database interface object in the `browse` module.

14.4.5 Implementation: Browsing Errata Reports

The file in [Example 14-4](#) implements the top-level errata browse page, used to select a report sort order and display format. Fields are in a pull-down selection list this time, but otherwise this page is similar to that for comments.

Example 14-4. PP2E\Internet\Cgi-Web\PyErrata\browseErrata.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Browse Errata</title>
<h1>PP2E Browse Errata Reports</h1>

<p>Please select the field you wish to sort by below, and press
the submit button to display reports. The display does not include
any emailed reports which have not yet been manually posted. Click
'With index' for a top-of-page index, or 'Index only' for an index
with links to individual pages.
</p>

<hr>
<form method=POST action="browseErrata.cgi">
  <h3>Sort reports by:</h3>
  <select name=key>
    <option>Page number
    <option>Type
    <option>Submit date
    <option>Severity
    <option>Chapter number
    <option>Part number
    <option>Printing date
    <option>Submitter name
```

```
        <option>Submitter email
        <option>Report state
    </select>
    <h3>Display options:</h3>
    <p><input type=radio name=display value="list">Simple list
        <input type=radio name=display value="indexed" checked>With index
        <input type=radio name=display value="indexonly">Index only
    <p><input type=submit>
</form>

<hr>
<a href="pyerrata.html">Back to errata page</A>
</body></html>
```

When submitted, the form in this HTML file invokes the script in [Example 14-5](#) on the server.

Example 14-5. PP2E\Internet\Cgi-Web\PyErrata\browseErrata.cgi

```
#!/usr/bin/python

from dbswitch import DbaseErrata          # dbfiles or dbshelve
from browse import generatePage          # reuse html formatter
generatePage(DbaseErrata)                # load data, send page
```

Again, there's not much to speak of here. In fact, it's nearly identical to the comment browse script, because both use the logic split off into the `browse` module. Here, we just pass a different database for the browse logic to process.

14.4.6 Common Browse Utility Modules

To fully understand how browse operations work, we need to explore the module in [Example 14 6](#), which is used by both comment and errata browse operations.

Example 14-6. PP2E\Internet\Cgi-Web\PyErrata\browse.py

```
#####
# on browse requests: fetch and display data in new page;
# report data is stored in dictionaries on the database;
# caveat: the '#Si' section links generated for top of page
# indexes work on a recent Internet Explorer, but have been
# seen to fail on an older Netscape; if they fail, try
# using 'index only' mode, which uses url links to encode
# information for creating a new page; url links must be
# encoded with urllib, not cgi.escape (for text embedded in
# the html reply stream; IE auto changes space to %20 when
# url is clicked so '+' replacement isn't always needed,
# but urllib.quote_plus is more robust; web browser adds
# http://server-name/root-dir/PyErrata/ to indexurl;
#####

import cgi, urllib, sys, string
sys.stderr = sys.stdout          # show errors in browser
indexurl = 'index.cgi'           # minimal urls in links

def generateRecord(record):
    print '<p><table border>'
    rowhtml = '<tr><th align=right>%s:<td>%s\n'
    for field in record.keys( ):
        if record[field] != '' and field != 'Description':
            print rowhtml % (field, cgi.escape(str(record[field])))
```



```
print '</table></p>'
field = 'Description'
text = string.strip(record[field])
print '<p><b>%s</b><br><pre>%s</pre><hr>' % (field, cgi.escape(text))

def generateSimpleList(dbase, sortkey):
    records = dbase( ).loadSortedTable(sortkey)          # make list
    for record in records:
        generateRecord(record)

def generateIndexOnly(dbase, sortkey, kind):
    keys, index = dbase( ).loadIndexedTable(sortkey)     # make index links
    print '<h2>Index</h2><ul>'                          # for load on click
    for key in keys:
        html = '<li><a href="%s?kind=%s&sortkey=%s&value=%s">%s</a>'
        htmlkey = cgi.escape(str(key))
        urlkey = urllib.quote_plus(str(key))             # html or url escapes
        urlsortkey = urllib.quote_plus(sortkey)          # change spaces to '+'
        print html % (indexurl,
                      kind, urlsortkey, (urlkey or '(none)'), (htmlkey or '?'))
    print '</ul><hr>'

def generateIndexed(dbase, sortkey):
    keys, index = dbase( ).loadIndexedTable(sortkey)
    print '<h2>Index</h2><ul>'
    section = 0                                         # make index
    for key in keys:
        html = '<li><a href="#S%d">%s</a>'
        print html % (section, cgi.escape(str(key)) or '?')
        section = section + 1
    print '</ul><hr>'
    section = 0                                         # make details
    for key in keys:
        html = '<h2><a name="#S%d">Key = "%s"</a></h2><hr>'
        print html % (section, cgi.escape(str(key)))
        for record in index[key]:
            generateRecord(record)
        section = section + 1

def generatePage(dbase, kind='Errata'):
    form = cgi.FieldStorage( )
    try:
        sortkey = form['key'].value
    except KeyError:
        sortkey = None

    print 'Content-type: text/html\n'
    print '<title>PP2E %s list</title>' % kind
    print '<h1>%s list, sorted by "%s"</h1><hr>' % (kind, str(sortkey))

    if not form.has_key('display'):
        generateSimpleList(dbase, sortkey)

    elif form['display'].value == 'list':                # dispatch on display type
        generateSimpleList(dbase, sortkey)              # dict would work here too

    elif form['display'].value == 'indexonly':
        generateIndexOnly(dbase, sortkey, kind)

    elif form['display'].value == 'indexed':
        generateIndexed(dbase, sortkey)
```

This module in turn heavily depends on the top-level database interfaces we'll meet in a few moments. For now, all we need to know at this high level of abstraction is that the database exports interfaces for loading report records and sorting and grouping them by key values, and

that report records are stored away as dictionaries in the database with one key per field in the report. Two top-level interfaces are available for accessing stored reports:

- `dbase().loadSortedTable(sortkey)` loads records from the generated database interface object into a simple list, sorted by the key whose name is passed in. It returns a list of record dictionaries sorted by a record field.
- `dbase().loadIndexedTable(sortkey)` loads records from the generated database interface object into a dictionary of lists, grouped by values of the passed-in key (one dictionary entry per sort key value). It returns both a dictionary of record-dictionary lists to represent the grouping by key, as well as a sorted-keys list to give ordered access into the groups dictionary (remember, dictionaries are unordered).

The simple list display option uses the first call, and both index display options use the second to construct key-value lists and sets of matching records. We will see the implementation of these calls and record store calls later. Here, we only care that they work as advertised.

Technically speaking, any mapping for storing a report record's fields in the database will do, but dictionaries are the storage unit in the system as currently coded. This representation was chosen for good reasons:

- It blends well with the CGI form field inputs object returned by `cgi.FieldStorage`. Submit scripts simply merge form field input dictionaries into expected field dictionaries to configure a record.
- It's more direct than other representations. For instance, it's easy to generically process all fields by stepping through the record dictionary's keys list, while using classes and attribute names for fields is less direct and might require frequent `getattr` calls.
- It's more flexible than other representations. For instance, dictionary keys can have values that attribute names cannot (e.g., embedded spaces), and so map well to arbitrary form field names.

More on the database later. For the "Index only" display mode, the `browse` module generates links that trigger the script in [Example 14-7](#) when clicked. There isn't a lot to see in this file either, because most page generation is again delegated to the `generateRecord` function in the `browse` module in [Example 14-6](#). The passed-in "kind" field is used to select the appropriate database object class to query here; the passed-in sort field name and key values are then used to extract matching records returned by the database interface.

Example 14-7. PP2E\Internet\Cgi-Web\PyErrata\index.cgi

```
#!/usr/bin/python
#####
# run when user clicks on a hyperlink generated for
# index-only mode by browse.py; input parameters are
# hard-coded into the link url, but there's nothing
# stopping someone from creating a similar link on
# their own--don't eval( ) inputs (security concern);
# note that this script assumes that no data files
# have been deleted since the index page was created;
# cgi.FieldStorage undoes any urllib escapes in the
# input parameters (%xx and '+' for spaces undone);
```

```
#####

import cgi, sys, dbswitch
from browse import generateRecord
sys.stderr = sys.stdout
form = cgi.FieldStorage( ) # undoes url encodin

inputs = {'kind':'?', 'sortkey':'?', 'value':'?'}
for field in inputs.keys( ):
    if form.has_key(field):
        inputs[field] = cgi.escape(form[field].value) # adds html encoding

if inputs['kind'] == 'Errata':
    dbase = dbswitch.DbaseErrata
else:
    dbase = dbswitch.DbaseComment

print 'Content-type: text/html\n'
print '<title>%s group</title>' % inputs['kind']
print '<h1>%(kind)s list<br>For "%(sortkey)s" == "%(value)s"</h1><hr>' % inputs

keys, index = dbase( ).loadIndexedTable(inputs['sortkey'])
key = inputs['value']
if key == '(none)': key = ''
for record in index[key]:
    generateRecord(record)
```

In a sense, this `index` script is a continuation of `browse`, with a page in between. We could combine these source files with a bit more work and complexity, but their logic really must be run in distinct processes. In interactive client-side programs, a pause for user input might simply take the form of a function call (e.g., to `raw_input`); in the CGI world, though, such a pause generally requires spawning a distinct process to handle the input.

There are two additional points worth underscoring before we move on. First of all, the "With index" option has its limitations. Notice how the `browse` module generates in-page `#section` hyperlinks, and then tags each key's section in the records list with a header line that embeds an `` tag, using a counter to generate unique section labels. This all relies on the fact that the database interface knows how to return records grouped by key values (one list per key). Unfortunately, in-page links like this may not work on all browsers (they've failed on older Netscapes); if they don't work in yours, use the "Index only" option to access records by key groups.

The second point is that since all report fields are optional, the system must handle empty or missing fields gracefully. Because submit scripts (described in the next section) define a fixed set of fields for each record type, the database never really has "missing" fields in records; empty fields are simply stored as empty strings and omitted in record displays. When empty values are used in index lists, they are displayed as `?`; within key labels and URLs, they are denoted as string `(none)`, which is internally mapped to the empty string in the `index` and `browse` modules just listed (empty strings don't work well as URL parameters). This is subtle, so see these modules for more details.

A word on redundancy: notice that the list of possible sort fields displayed in the browse input pages is hardcoded into their HTML files. Because the submit scripts we'll explore next ensure that all records in a database have the same set of fields, the HTML files' lists will be redundant with records stored away in the databases.

We could in principle build up the HTML sort field lists by inspecting the keys of any record in the comment and errata databases (much as we did in the language selector example in [Chapter 12](#)), but that may require an extra database operation. These lists also partially overlap with the fields list in both submit page HTML and submit scripts, but seem different enough to warrant some redundancy.

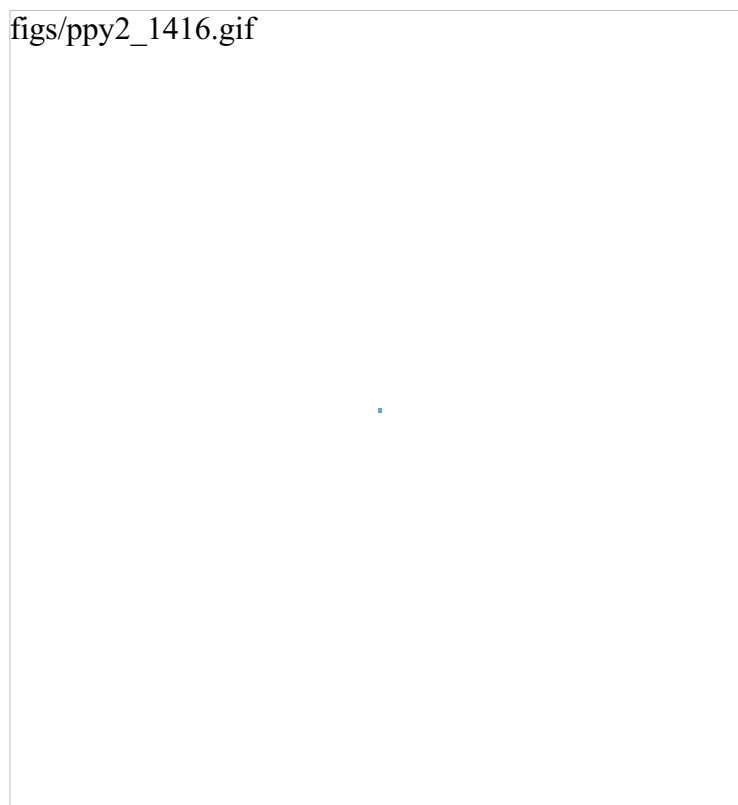
14.5 Submitting PyErrata Reports

The next major functional area in PyErrata serves to implement user-controlled submission of new comment and errata reports. As before, let's begin by getting a handle on this component's user-interface model before inspecting its code.

14.5.1 User Interface: Submitting Comment Reports

As we've seen, PyErrata supports two user functions: browsing the reports database and adding new reports to it. If you click the "General comment" link in the Submit section of the root page shown in [Figure 14-2](#), you'll be presented with the comment submission page shown in [Figure 14-16](#).

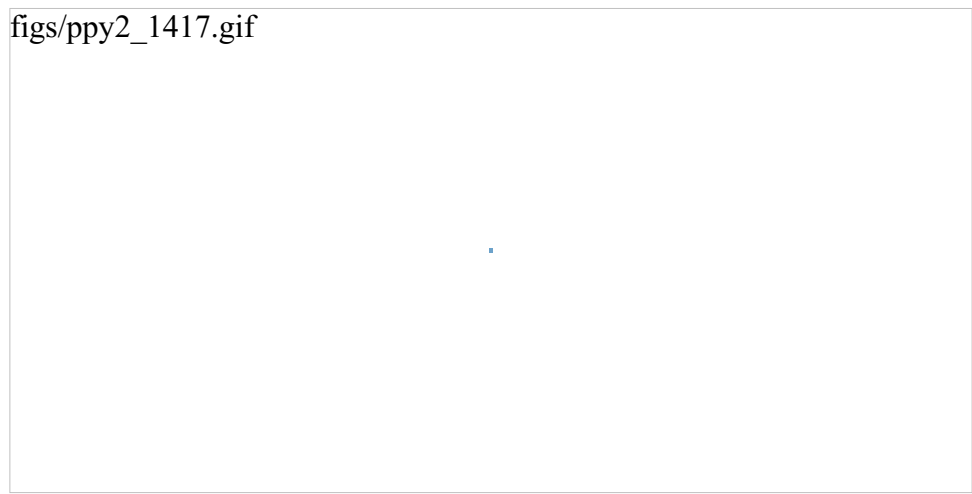
Figure 14-16. Submit comments, input page



This page initially comes up empty; the data we type into its form fields is submitted to a server-side script when we press the submit button at the bottom. If the system was able to store the data as a new database record, a confirmation like the one in [Figure 14-17](#) is reflected back to the client.

Figure 14-17. Submit comments, confirmation page

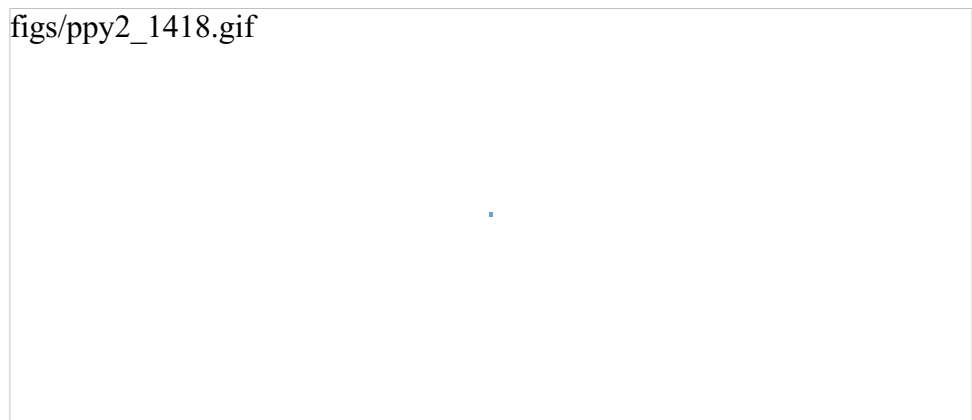
figs/ppy2_1417.gif



All fields in submit forms are optional except one; if we leave the "Description" field empty and send the form, we get the error page shown in [Figure 14-18](#) (generated during an errata submission). Comments and error reports without descriptions aren't incredibly useful, so we kick such requests out. All other report fields are stored empty if we send them empty (or missing altogether) to the submit scripts.

Figure 14-18. Submit, missing field error page

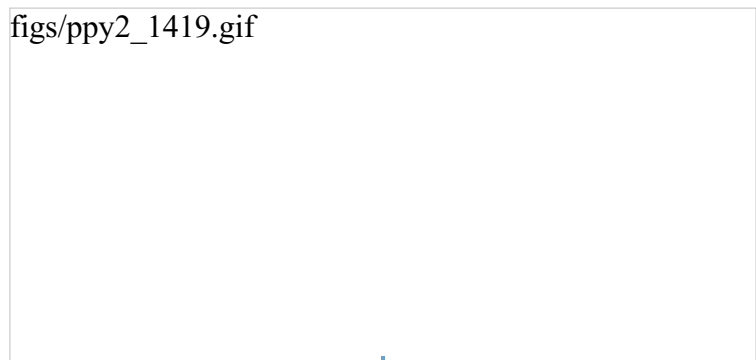
figs/ppy2_1418.gif



Once we've submitted a comment, we can go back to the browse pages to view it in the database [Figure 14-19](#) shows the one we just added, accessed by key "Submitter name" and in "With index display format mode."

Figure 14-19. Submit comments, verifying result

figs/ppy2_1419.gif





14.5.2 User Interface: Submitting Errata Reports

Here again, the pages generated to submit errata reports are virtually identical to the ones we just saw for submitting comments, as comments and errata are treated the same within the system. Both are instances of generic database records with different sets of fields. But also as before, the top-level errata submission page differs, because there are many more fields that can be filled in; [Figure 14-20](#) shows the top of this input page.


Figure 14-20. Submit errata, input page (top)



There are lots of fields here, but only the description is required. The idea is that users will fill in as many fields as they like to describe the problem; all text fields default to an empty string if no value is typed into them. [Figure 14-21](#) shows a report in action with most fields filled with relevant information.

Figure 14-21. Submit errata, input page (filled)

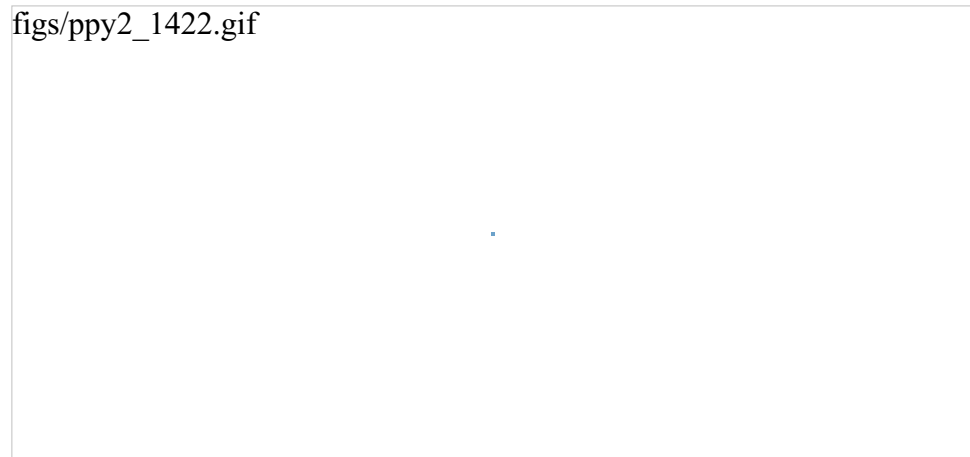
figs/ppy2_1421.gif



When we press the submit button, we get a confirmation page as before ([Figure 14-22](#)), this time with text customized to thank us for an errata instead of a comment.

Figure 14-22. Submit errata, confirmation

figs/ppy2_1422.gif



As before, we can verify a submission with the browse pages immediately after it has been confirmed. Let's bring up an index list page for submission dates and click on the new entry at the bottom ([Figure 14-23](#)). Our report is fetched from the errata database and displayed in a new page ([Figure 14-24](#)). Note that the display doesn't include a "Page number" field: we left it blank on the submit form. PyErrata displays only nonempty record fields when formatting web pages. Because it treats all records generically, the same is true for comment reports; at its core, PyErrata is a very generic system that doesn't care about the meaning of data stored in records.

Figure 14-23. Submit errata, verify result (index)

figs/ppy2_1423.gif


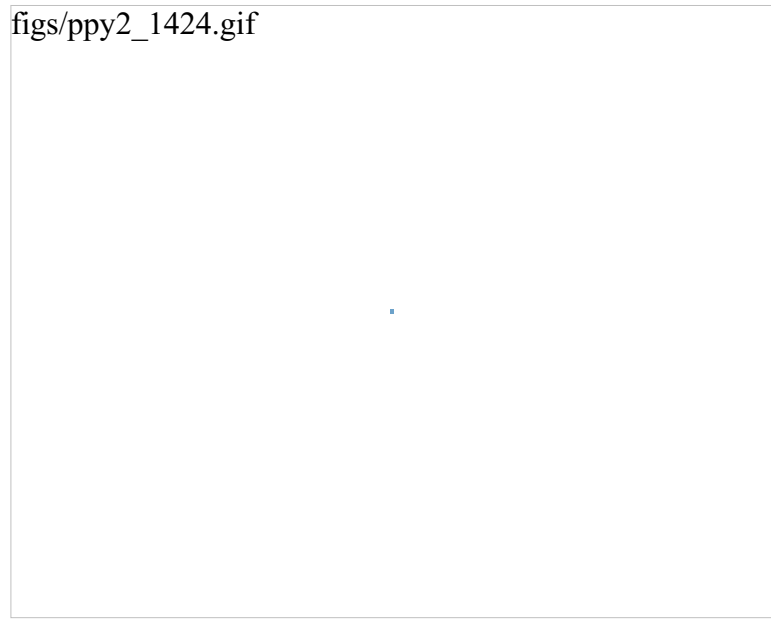


Figure 14-24. Submit errata, verify result (record)

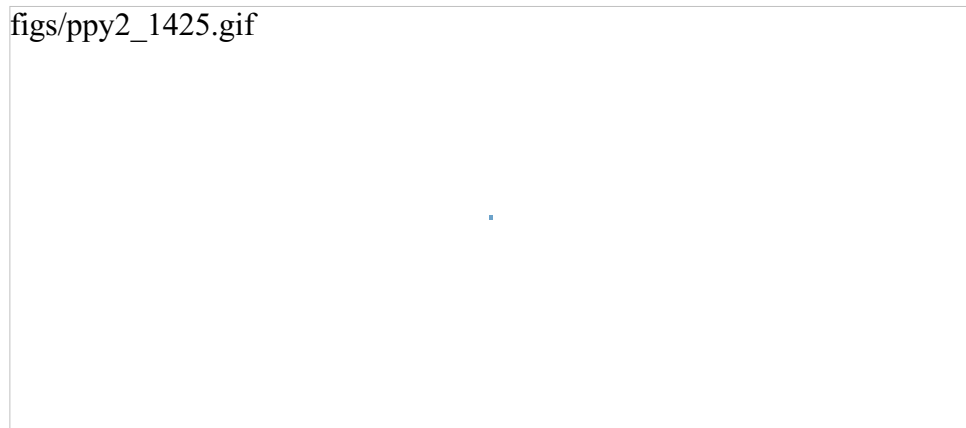
figs/ppy2_1424.gif



Because not everyone wants to post to a database viewable by everyone in the world with a browser, PyErrata also allows both comments and errata to be sent by email instead of being automatically added to the database. If we click the "Email report privately" checkbox near the bottom of the submit pages before submission, the report's details are emailed to me (their fields show up as a message in my mailbox), and we get the reply in [Figure 14-25](#).

Figure 14-25. Submit errata, email mode confirmation

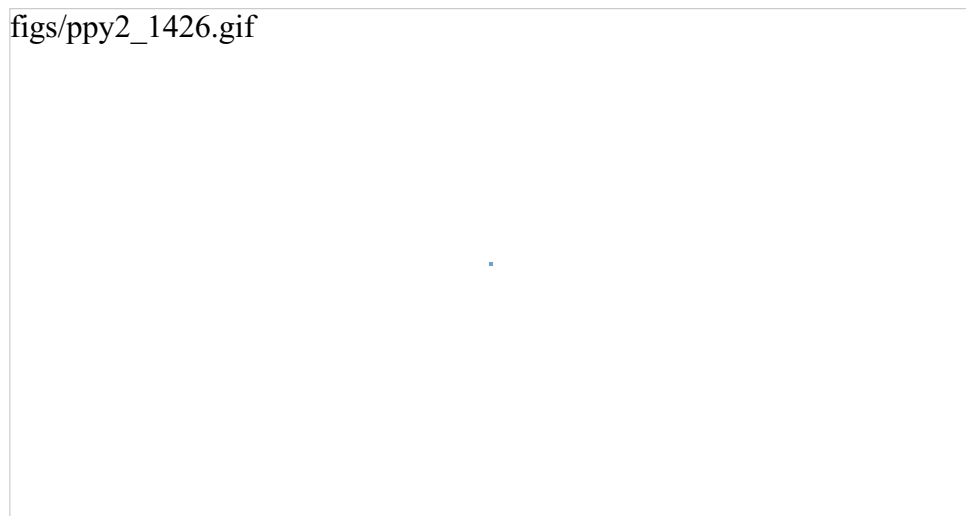
figs/ppy2_1425.gif



Finally, if the directory or shelve file that represents the database does not grant write access to everyone (remember, CGI scripts run as user "nobody"), our scripts won't be able to store the new record. Python generates an exception, which is displayed in the client's browser because PyErrata is careful to route exception text to `sys.stdout`. [Figure 14-26](#) shows an exception page I received before making the database directory in question writable with the shell command `chmod 777 DbaseFiles/errataDB`.

Figure 14-26. Submit errata, exception (need chmod 777 dir)

figs/ppy2_1426.gif



14.5.3 Implementation: Submitting Comment Reports

Now that we've seen the external behavior of PyErrata submit operations, it's time to study their internal workings. Top-level report submission pages are defined by static HTML files. [Example 14-8](#) shows the comment page's file.

Example 14-8. PP2E\Internet\Cgi-Web\PyErrata\submitComment.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Submit Comment</title>
<h1>PP2E Submit Comment</h1>
```

```
<p>Please fill out the form below and press the submit button to
send your information. By default, your report will be automatically
```

entered in a publically browsable database, where it will eventually be reviewed by the author. If you prefer to send your comments to the author by private email instead, click the "Email" button before you submit. All the fields except the description text are optional. Thank you for your report.

```
</p>
<hr>
<form method=POST action="submitComment.cgi">
  <table>
    <tr>
      <th align=right>Description:
      <td><textarea name="Description" cols=40 rows=10>Type your comment here
        </textarea>
    <tr>
      <th align=right>Your name:
      <td><input type=text size=35 name="Submitter name">
    <tr>
      <th align=right>Your email, webpage:
      <td><input type=text size=35 name="Submitter email">
    <tr>
      <th align=right>Email report privately?:
      <td><input type=checkbox name="Submit mode" value="email">
    <tr>
      <th></th>
      <td><input type=submit value="Submit Comment">
        <input type=reset value="Reset Form">
    </td>
    </tr>
  </table>
</form>

<hr>
<A href="pyerrata.html">Back to errata page</A>
</body></html>
```

The CGI script that is invoked when this file's form is submitted, shown in [Example 14-9](#), does the work of storing the form's input in the database and generating a reply page.

Example 14-9. PP2E\Internet\Cgi-Web\PyErrata\submitComment.cgi

```
#!/usr/bin/python

DEBUG=0
if DEBUG:
    import sys
    sys.stderr = sys.stdout
    print "Content-type: text/html"; print

import traceback
try:
    from dbswitch import DbaseComment          # dbfiles or dbshelve
    from submit    import saveAndReply        # reuse save logic

    replyStored = """
    Your comment has been entered into the comments database.
    You may view it by returning to the main errata page, and
    selecting Browse/General comments, using your name, or any
    other report identifying information as the browsing key."""

    replyMailed = """
    Your comment has been emailed to the author for review.
    It will not be automatically browsable, but may be added to
    the database anonymously later, if it is determined to be
    information of general use."""

    inputs = {'Description': '',          'Submit mode': '',
```

```
        'Submitter name':'', 'Submitter email:''}

    saveAndReply(DbaseComment, inputs, replyStored, replyMailed)

except:
    print "\n\n<PRE>"
    traceback.print_exc( )
```

Don't look too hard for database or HTML-generation code here; it's all been factored out to the `submit` module, listed in a moment, so it can be reused for errata submissions too. Here, we simply pass it things that vary between comment and errata submits: database, expected input fields, and reply text.

As before, the database interface object is fetched from the `switch` module to select the currently supported storage medium. Customized text for confirmation pages (`replyStored`, `replyMailed`) winds up in web pages and is allowed to vary per database.

The `inputs` dictionary in this script provides default values for missing fields and defines the format of comment records in the database. In fact, this dictionary is stored in the database: with the `submit` module, input fields from the form or an explicit URL are merged in to the `inputs` dictionary created here, and the result is written to the database as a record.

More specifically, the `submit` module steps through all keys in `inputs` and picks up values of those keys from the parsed form input object, if present. The result is that this script guarantees that records in the comments database will have all the fields listed in `inputs`, but no others. Because all submit requests invoke this script, this is true even if superfluous fields are passed in an explicit URL; only fields in `inputs` are stored in the database.

Notice that almost all of this script is wrapped in a `try` statement with an empty `except` clause. This guarantees that every (uncaught) exception that can possibly happen while our script runs will return to this `try` and run its exception handler; here, it runs the standard `traceback.print_exc` call to print exception details to the web browser in unformatted (`<PRE>`) mode.

14.5.4 Implementation: Submitting Errata Reports

The top-level errata submission page in Figures [Figure 14-20](#) and [Figure 14-21](#) is also rendered from a static HTML file on the server, listed in [Example 14-10](#). There are more input fields here but it's similar to comments.

Example 14-10. PP2E\Internet\Cgi-Web\PyErrata\submitErrata.html

```
<html><body bgcolor="#FFFFFF">
<title>PP2E Submit Errata</title>
<h1>PP2E Submit Errata Report</h1>

<p>Please fill out the form below and press the submit button to
send your information. By default, your report will be automatically
entered in a publically browsable database, where it will eventually
be reviewed and verified by the author. If you prefer to send your
comments to the author by private email instead, click the "Email"
button before you submit.

<p>All the fields except the description text are optional;
for instance, if your note applies to the entire book, you can leave
the page, chapter, and part numbers blank. For the printing date, see
```

the lower left corner of one of the first few pages; enter a string of the form mm/dd/yy. Thank you for your report.

```
</p>
<hr>
<form method=POST action="submitErrata.cgi">
  <table>
    <tr>
      <th align=right>Problem type:
      <td><select name="Type">
        <option>Typo
        <option>Grammar
        <option>Program bug
        <option>Suggestion
        <option>Other
      </select>
    <tr>
      <th align=right>Problem severity:
      <td><select name="Severity">
        <option>Low
        <option>Medium
        <option>High
        <option>Unknown
      </select>
    <tr>
      <th align=right>Page number:
      <td><input type=text name="Page number">
    <tr>
      <th align=right>Chapter number:
      <td><input type=text name="Chapter number">
    <tr>
      <th align=right>Part number:
      <td><input type=text name="Part number">
    <tr>
      <th align=right>Printing date:
      <td><input type=text name="Printing date">
    <tr>
      <th align=right>Description:
      <td><textarea name="Description" cols=60 rows=10>Type a description here
        </textarea>
    <tr>
      <th align=right>Your name:
      <td><input type=text size=40 name="Submitter name">
    <tr>
      <th align=right>Your email, webpage:
      <td><input type=text size=40 name="Submitter email">
    <tr>
      <th align=right>Email report privately?:
      <td><input type=checkbox name="Submit mode" value="email">
    <tr>
      <th></th>
      <td><input type=submit value="Submit Report">
        <input type=reset value="Reset Form">
    </table>
  </form>
  <hr>
  <A href="pyerrata.html">Back to errata page</A>
</body></html>
```

The script triggered by the form on this page, shown in [Example 14-11](#), also looks remarkably similar to the `submitComment` script shown in [Example 14-9](#). Because both scripts simply use factored-out logic in the `submit` module, all we need do here is pass in appropriately tailored confirmation pages text and expected input fields. As before, real CGI inputs are merged into the script's `inputs` dictionary to yield a database record; the stored record will contain exactly the

fields listed here.

Example 14-11. PP2E\Internet\Cgi-Web\PyErrata\submitErrata.cgi

```
#!/usr/bin/python

DEBUG=0
if DEBUG:
    import sys
    sys.stderr = sys.stdout
    print "Content-type: text/html"; print

import traceback
try:
    from dbswitch import DbaseErrata          # dbfiles or dbshelve
    from submit import saveAndReply         # reuse save logic

    replyStored = """
    Your report has been entered into the errata database.
    You may view it by returning to the main errata page, and
    selecting Browse/Errata reports, using your name, or any
    other report identifying information as the browsing key."""

    replyMailed = """
    Your report has been emailed to the author for review.
    It will not be automatically browsable, but may be added to
    the database anonymously later, if it is determined to be
    information of general interest."""

    # 'Report state' and 'Submit date' are added when written

    inputs = {'Type':'',          'Severity':'',
              'Page number':'',  'Chapter number':'', 'Part number':'',
              'Printing Date':'', 'Description':'',   'Submit mode':'',
              'Submitter name':'', 'Submitter email':''}

    saveAndReply(DbaseErrata, inputs, replyStored, replyMailed)

except:
    print "\n\n<pre>"
    traceback.print_exc( )
```

14.5.5 Common Submit Utility Module

Both comment and errata reports ultimately invoke functions in the module in [Example 14-12](#) to store to the database and generate a reply page. Its primary goal is to merge real CGI inputs into the expected inputs dictionary and post the result to the database or email. We've already described the basic ideas behind this module's code, so we don't have much new to say here.

Notice, though, that email-mode submissions (invoked when the submit page's email checkbox is checked) use an `os.popen` shell command call to send the report by email; messages arrive in my mailbox with one line per nonempty report field. This works on my Linux web server, but other mail schemes such as the `smtpLib` module (discussed in [Chapter 11](#)) are more portable.

Example 14-12. PP2E\Internet\Cgi-Web\PyErrata\submit.py

```
#####
# on submit request: store or mail data, send reply page;
# report data is stored in dictionaries on the database;
# we require a description field (and return a page with
```

```
# an error message if it's empty), even though the dbase
# mechanism could handle empty description fields--it
# makes no sense to submit a bug without a description;
#####

import cgi, os, sys, string
mailto = 'lutz@rmi.net'           # or lutz@starship.python.net
sys.stderr = sys.stdout         # print errors to browser
print "Content-type: text/html\n"

thankyouHtml = """
<TITLE>Thank you</TITLE>
<H1>Thank you</H1>
<P>%s</P>
<HR>"""

errorHtml = """
<TITLE>Empty field</TITLE>
<H1>Error: Empty %s</H1>
<P>Sorry, you forgot to provide a '%s' value.
Please go back to the prior page and try again.</P>
<HR>"""

def sendMail(inputs):
    text = ''                     # email data to author
                                # or 'mailto:' form action
    for key, val in inputs.items( ): # or smtplib.py or sendmail
        if val != '':
            text = text + ('%s = %s\n' % (key, val))
    mailcmd = 'mail -s "PP2E Errata" %s' % mailto
    os.popen(mailcmd, 'w').write(text)

def saveAndReply(dbase, inputs, replyStored, replyMailed):
    form = cgi.FieldStorage( )
    for key in form.keys( ):
        if key in inputs.keys( ):
            inputs[key] = form[key].value        # pick out entered fields

    required = ['Description']
    for field in required:
        if string.strip(inputs[field]) == '':
            print errorHtml % (field, field)    # send error page to browser
            break
    else:
        if inputs['Submit mode'] == 'email':
            sendMail(inputs)                    # email data direct to author
            print thankyouHtml % replyMailed
        else:
            dbase( ).storeItem(inputs)         # store data in file on server
            print thankyouHtml % replyStored
```

This module makes use of one additional database interface to store record dictionaries: `dbase().storeItem(inputs)`. However, we need to move on to the next section to fully understand the processing that this call implies.

Another redundancy caveat: the list of expected fields in the `inputs` dictionaries in submit scripts is the same as the input fields list in submit HTML files. In principle again, we could instead generate the HTML file's fields list using data in a common module to remove this redundancy. However, that technique may not be as directly useful here, since each field requires description text in the HTML file only.

14.6 PyErrata Database Interfaces

Now that we've seen the user interfaces and top-level implementations of browse and submit operations, this section proceeds down one level of abstraction to the third and last major functional area in the PyErrata system.

Compared to other systems in this part of the book, one of the most unique technical features of PyErrata is that it must manage persistent data. Information posted by readers needs to be logged in a database for later review. PyErrata stores reports as dictionaries, and includes logic to support different database storage mediums -- flat pickle files and shelves -- as well as tools for synchronizing database access.

14.6.1 The Specter of Concurrent Updates

There is a variety of ways for Python scripts to store data persistently: files, object pickling, object shelves, real databases, and so on. In fact, [Chapter 16](#) is devoted exclusively to this topic and provides more in-depth coverage than we require here.^[3] Those storage mediums all work in the context of server-side CGI scripts too, but the CGI environment almost automatically introduces a new challenge: concurrent updates. Because the CGI model is inherently parallel, scripts must take care to ensure that database writes and reads are properly synchronized to avoid data corruption and incomplete records.

[3] But see [Chapter 16](#) if you could use a bit of background information on this topic. The current chapter introduces and uses only the simplest interfaces of the `object`, `pickle` and `shelve` modules, and most module interface details are postponed until that later chapter.

Here's why. With PyErrata, a given reader may visit the site and post a report or view prior posts. But in the context of a web application, there is no way to know how many readers may be posting or viewing at once: any number of people may press a form's submit button at the same time. As we've seen, form submissions generally cause the HTTP server to spawn a new process to handle the request. Because these handler processes all run in parallel, if one hundred users all press submit at the same time, there will be one hundred CGI script processes running in parallel on the server, all of which may try to update (or read) the reports database at the same time.

Due to all this potential parallel traffic, server-side programs that maintain a database must somehow guarantee that database updates happen one at a time, or the database could be corrupted. The likelihood of two particular scenarios increases with the number of site users:

- **Concurrent writers:** If two processes try to update the same file at once, we may wind up with part of one process's new data intermixed with another's, lose part of one process's data, or otherwise corrupt stored data.
- **Concurrent reader and writer:** Similarly, if a process attempts to read a record that is being written by another, it may fetch an incomplete report. In effect, the database must be managed as a shared resource among all possible CGI handler processes, whether they update or not.

Constraints vary per database medium, and while it's generally okay for multiple processes to read a database at the same time, writers (and updates in general) almost always need to have exclusive access to a shared database. There is a variety of ways to make database access safe in a potentially concurrent environment such as CGI-based web sites:

Database systems

If you are willing to accept the extra complexity of using a full-blown database system in your application (e.g., Sybase, Oracle, mySql), most provide support for concurrent access in one form or another.

Central database servers

It's also possible to coordinate access to shared data stores by routing all data requests to a perpetually running manager program that you implement yourself. That is, each time a CGI script needs to hit the database, it must ask a data server program for access via a communications protocol such as socket calls.

File naming conventions

If it is feasible to store each database record in a separate flat file, we can sometimes avoid or minimize the concurrent access problems altogether by giving each flat file a distinct name. For instance, if each record's filename includes both the file's creation time and the ID of the process that created it, it will be unique for all practical purposes, since a given process updates only one particular file. In this scheme, we rely on the operating system's filesystem to make records distinct, by storing them in unique files.

File locking protocols

If the entire database is physically stored as a single file, we can use operating-system tools to lock the file during update operations. On Unix and Linux servers, exclusively locking a file will block other processes that need it until the lock is released; when used consistently by all processes, such a mechanism automatically synchronizes database accesses. Python shelves support concurrent readers but not concurrent updates, so we must add locks of our own to use them as dynamic data stores in CGI scripting.

In this section, we implement both of the last two schemes for PyErrata to illustrate concurrent data-access fundamentals.

14.6.2 Database Storage Structure

First of all, let's get a handle on what the system really stores. If you flip back to [Figure 14-1](#), you notice that there are two top-level database directories: *DbaseShelve* (for the shelve mechanism) and *DbaseFiles* (for file-based storage). Each of these directories has unique contents.

14.6.2.1 Shelve database

For shelve-based databases, the *DbaseShelve* directory's contents are shown in [Figure 14-27](#). The `commentDB` and `errataDB` files are the shelves used to store reports, and the `.lck` and `.log` files are

lock and log files generated by the system. To start a new installation from scratch, only the two `.lck` files are needed initially (and can be simply empty files); the system creates the shelves and log files as records are stored.

Figure 14-27. PyErrata shelves-based directory contents



We'll explore the Python `shelve` module in more detail in the next part of this book, but the parts it used in this chapter are straightforward. Here are the basic `shelve` interfaces we'll use in this example:

```
import shelve                                # load the standard shelve module
dbase = shelve.open('filename')             # open shelve (create if doesn't yet exist)
dbase['key'] = object                         # store almost any object in shelve file
object = dbase['key']                        # fetch object from shelve in future run
dbase.keys( )                               # list of keys stored in the shelve
dbase.close( )                              # close shelve's file
```

In other words, shelves are like dictionaries of Python objects that are mapped to an external file and so persist between program runs. Objects in a shelf are stored away and later fetched with a key. In fact, it's not inaccurate to think of shelves as dictionaries that live on after a program exit and must be explicitly opened.

Like dictionaries, each distinct value stored in a shelf must have a unique key. Because there is no field in a comment or errata report that is reliably unique among all reports, we need to generate one of our own. Record submit time is close to being unique, but there is no guarantee that two users (and hence two processes) won't submit a report in the same second.

To assign each record a unique slot in the shelf, the system generates a uniquekey string for each containing the submission time (seconds since the Unix "epoch" as a floating-point value) and the process ID of the storing CGI script. Since the dictionary values stored in the shelf contain all the report information we're interested in, shelf keys need only be unique, not meaningful. Records are loaded by blindly iterating over the shelf's keys list.

In addition to generating unique keys for records, shelves must accommodate concurrent updates. Because shelves are mapped to single files in the filesystem (here, `errataDB` and `commentDB`), we must synchronize all access to them in a potentially parallel process environment such as CGI scripting.

In its current form, the Python `shelve` module supports concurrent readers but not concurrent

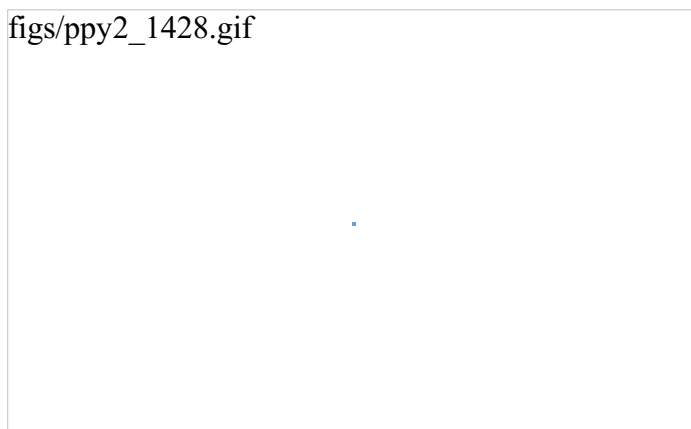
updates, so we need to add such functionality ourselves. The PyErrata implementation of the shelf database-storage scheme uses locks on the *.lck* files to make sure that writers (submit processes) gain exclusive access to the shelf before performing updates. Any number of readers may run in parallel, but writers must run alone and block all other processes -- readers and writers -- while they update the shelf.

Notice that we use a separate *.lck* file for locks, rather than locking the shelf file itself. In some systems, shelves are mapped to multiple files, and in others (e.g., GDBM), locks on the underlying shelf file are reserved for use by the DBM filesystem itself. Using our own lock file subverts such reservations and is more portable among DBM implementations.

14.6.2.2 Flat-file database

Things are different with the flat-files database medium; [Figure 14-28](#) shows the contents of the file-based errata database subdirectory, *DbaseFiles/errataDB*. In this scheme, each report is stored in a distinct and uniquely named flat file containing a pickled report-data dictionary. A similar directory exists for comments, *DbaseFiles/commentDB*. To start from scratch here, only the two subdirectories must exist; files are added as reports are submitted.

Figure 14-28. PyErrata file-based directory contents



Python's object pickler converts ("serializes") in-memory objects to and from specially coded strings in a single step, and therefore comes in handy for storing complex objects like the dictionaries PyErrata uses to represent report records.^[4] We'll also study the `pickle` module in depth in [Part IV](#), but its interfaces employed by PyErrata are simple as well:

[4] PyErrata could also simply write report record dictionaries to files with one field key and value per text line, and split lines later to rebuild the record. It could also just convert the record dictionary to its string representation with the `str` built-in function, write that string to the file manually, and convert the string back to a dictionary later with the built-in `eval` function (which may or may not be slower, due to the general parsing overhead of `eval`). As we'll see in the next part of this book, though, object pickling is a much more powerful and general approach to object storage -- it also handles things like class instance objects and shared and cyclic object references well. See table wrapper classes in the PyForm example in [Chapter 16](#) for similar topics.

```
pickle.dump(object, outputfile)          # store object in a file
object = pickle.load(inputfile)          # load object back from file
```

For flat files, the system-generated key assigned to a record follows the same format as for shelves but here it is used to name the report's file. Because of that, record keys are more apparent (we see them when listing the directory), but still don't need to convey any real information. They need only be unique for each stored record to yield a unique file. In this storage scheme, records are processed by iterating over directory listings returned by the standard `glob.glob` call on name pattern `*.data` (see [Chapter 2](#), for a refresher on the `glob` module).

In a sense, this flat-file approach uses the filesystem as a shelf and relies on the operating system to segregate records as files. It also doesn't need to care much about concurrent access issues; because generated filenames make sure that each report is stored in its own separate file, it's impossible for two submit processes to be writing the same file at once. Moreover, it's okay to read one report while another is being created; they are truly distinct files.

We still need to be careful, though, to avoid making a file visible to reader directory listings until it is complete, or else we may read a half-finished file. This case is unlikely in practice -- it can happen only if the writer still hasn't finished by the time the reader gets around to that file in its directory listing. But to avoid problems, submit scripts first write data to a temporary file, and move the temporary file to the real `*.data` name only after it is complete.

14.6.3 Database Switch

On to code listings. The first database module, shown in [Example 14-13](#), simply selects between file-based mechanism and shelf-based mechanism; we make the choice here alone to avoid impacting other files when we change storage schemes.

Example 14-13. PP2E\Internet\Cgi-Web\PyErrata\dbswitch.py

```
#####
# for testing alternative underlying database mediums;
# since the browse, submit, and index cgi scripts import
# dbase names from here only, they'll get whatever this
# module loads; in other words, to switch mediums, simply
# change the import here; eventually we could remove this
# interface module altogether, and load the best medium's
# module directly, but the best may vary by use patterns;
#####

#
# one directory per dbase, one flat pickle file per submit
#

from dbfiles import DbaseErrata, DbaseComment

#
# one shelf per dbase, one key per submit, with mutex update locks
#

# from dbshelve import DbaseErrata, DbaseComment
```

14.6.4 Storage-Specific Classes for Files and Shelves

The next two modules implement file- and shelf-based database-access objects; the classes they define are the objects passed and used in the browse and submit scripts. Both are really just subclasses of the more generic class in `dbcommon`; in [Example 14-14](#), we fill in methods that defi

storage scheme-specific behavior, but the superclass does most of the work.

Example 14-14. PP2E\Internet\Cgi-Web\PyErrata\dbfiles.py

```
#####
# store each item in a distinct flat file, pickled;
# dbcommon assumes records are dictionaries, but we don't here;
# chmod to 666 to allow admin access (else 'nobody' owns);
# subtlety: unique filenames prevent multiple writers for any
# given file, but it's still possible that a reader (browser)
# may try to read a file while it's being written, if the
# glob.glob call returns the name of a created but still
# incomplete file; this is unlikely to happen (the file
# would have to still be incomplete after the time from glob
# to unpickle has expired), but to avoid this risk, files are
# created with a temp name, and only moved to the real name
# when they have been completely written and closed;
# cgi scripts with persistent data are prone to parallel
# updates, since multiple cgi scripts may be running at once;
#####

import dbcommon, pickle, glob, os

class Dbase(dbcommon.Dbase):
    def writeItem(self, newdata):
        name = self.dirname + self.makeKey( )
        file = open(name, 'w')
        pickle.dump(newdata, file)          # store in new file
        file.close( )
        os.rename(name, name+'.data')        # visible to globs
        os.chmod(name+'.data', 0666)         # owned by 'nobody'

    def readTable(self):
        reports = []
        for filename in glob.glob(self.dirname + '*.data'):
            reports.append(pickle.load(open(filename, 'r')))
        return reports

class DbaseErrata(Dbase):
    dirname = 'DbaseFiles/errataDB/'

class DbaseComment(Dbase):
    dirname = 'DbaseFiles/commentDB/'
```

The shelve interface module listed in [Example 14-15](#) provides the same methods interface, but implements them to talk to shelves. Its class also mixes in the mutual-exclusion class to get file locking; we'll study that class's code in a few pages.

Notice that this module extends `sys.path` so that a platform-specific `fcntl` module (described in this chapter) becomes visible to the file-locking tools. This is necessary in the CGI script context only, because the module search path given to CGI user "nobody" doesn't include the platform-specific extension modules directory. Both the file and shelve classes set newly created file permissions to octal 0666, so that users besides "nobody" can read and write. If you've forgotten whom "nobody" is, see earlier discussions of permission and ownership issues in this and the previous two chapters.

Example 14-15. PP2E\Internet\Cgi-Web\PyErrata\dbshelve.py

```
#####
# store items in a shelve, with file locks on writes;
# dbcommon assumes items are dictionaries (not here);
```

```
# chmod call assumes single file per shelve (e.g., gdbm);
# shelve allows simultaneous reads, but if any program
# is writing, no other reads or writes are allowed,
# so we obtain the lock before all load/store ops
# need to chown to 0666, else only 'nobody' can write;
# this file doesn't know about fcntl, but mutex doesn't
# know about cgi scripts--one of the 2 needs to add the
# path to FCNTL module for cgi script use only (here);
# we circumvent whatever locking mech the underlying
# dbm system may have, since we acquire a lock on our own
# non-dbm file before attempting any dbm operation;
# allows multiple simultaneous readers, but writers
# get exclusive access to the shelve; lock calls in
# MutexCntl block and later resume callers if needed;
#####

# cgi runs as 'nobody' without
# the following default paths
import sys
sys.path.append('/usr/local/lib/python1.5/plat-linux2')

import dbcommon, shelve, os
from Mutex.mutexcntl import MutexCntl

class Dbase(MutexCntl, dbcommon.Dbase):
    # mix mutex, dbcommon, mine
    def safe_writeItem(self, newdata):
        dbase = shelve.open(self.filename) # got excl access: update
        dbase[self.makeKey( )] = newdata # save in shelve, safely
        dbase.close( )
        os.chmod(self.filename, 0666) # else others can't change

    def safe_readTable(self):
        reports = [] # got shared access: load
        dbase = shelve.open(self.filename) # no writers will be run
        for key in dbase.keys( ):
            reports.append(dbase[key]) # fetch data, safely
        dbase.close( )
        return reports

    def writeItem(self, newdata):
        self.exclusiveAction(self.safe_writeItem, newdata)

    def readTable(self):
        return self.sharedAction(self.safe_readTable)

class DbaseErrata(Dbase):
    filename = 'DbaseShelve/errataDB'

class DbaseComment(Dbase):
    filename = 'DbaseShelve/commentDB'
```

14.6.5 Top-Level Database Interface Class

Here, we reach the top-level database interfaces that our CGI scripts actually call. The class in [Example 14-16](#) is "abstract" in the sense that it cannot do anything by itself. We must provide or create instances of subclasses that define storage-specific methods, rather than making instances of this class directly.

In fact, this class deliberately leaves the underlying storage scheme undefined and raises assertic errors if a subclass doesn't fill in the required details. Any storage-specific class that provides `writeItem` and `readTable` methods can be plugged into this top-level class's interface model. TI

includes classes that interface with flat files, shelves, and other specializations we might add in the future (e.g., schemes that talk to full-blown SQL or object databases, or that cache data in persistent servers).

In a sense, subclasses take the role of embedded component objects here; they simply need to provide expected interfaces. Because the top-level interface has been factored out to this single class, we can change the underlying storage scheme simply by selecting a different storage-specific subclass (as in `dbswitch`); the top-level database calls remain unchanged. Moreover, changes in optimizations to top-level interfaces will likely impact this file alone.

Since this is a superclass common to storage-specific classes, we also here define record key generation methods and insert common generated attributes (submit date, initial report state) into new records before they are written.

Example 14-16. PP2E\Internet\Cgi-Web\PyErrata\dbcommon.py

```
#####
# an abstract superclass with shared dbase access logic;
# stored records are assumed to be dictionaries (or other
# mapping), one key per field; dbase medium is undefined;
# subclasses: define writeItem and readTable as appropriate
# for the underlying file medium--flat files, shelves, etc.
# subtlety: the 'Submit date' field added here could be kept
# as a tuple, and all sort/select logic will work; but since
# these values may be embedded in a url string, we don't want
# to convert from string to tuple using eval in index.cgi;
# for consistency and safety, we convert to strings here;
# if not for the url issue, tuples work fine as dict keys;
# must use fixed-width columns in time string to sort;
# this interface may be optimized in future releases;
#####

import time, os

class Dbase:

    # store

    def makeKey(self):
        return "%s-%s" % (time.time(), os.getpid( ))

    def writeItem(self, newdata):
        assert 0, 'writeItem must be customized'

    def storeItem(self, newdata):
        secsSinceEpoch = time.time( )
        timeTuple = time.localtime(secsSinceEpoch)
        y_m_d_h_m_s = timeTuple[:6]
        newdata['Submit date'] = '%s/%02d/%02d, %02d:%02d:%02d' % y_m_d_h_m_s
        newdata['Report state'] = 'Not yet verified'
        self.writeItem(newdata)

    # load

    def readTable(self):
        assert 0, 'readTable must be customized'

    def loadSortedTable(self, field=None):
        reports = self.readTable( )
        if field:
            reports.sort(lambda x, y, f=field: cmp(x[f], y[f]))
        return reports

# returns a simple list
# ordered by field sort
```



```
def loadIndexedTable(self, field):
    reports = self.readTable( )
    index = {}
    for report in reports:
        try:
            index[report[field]].append(report)    # group by field values
        except KeyError:
            index[report[field]] = [report]        # add first for this key
    keys = index.keys( )
    keys.sort( )                                  # sorted keys, groups d
    return keys, index
```

14.6.6 Mutual Exclusion for Shelves

We've at last reached the bottom of the PyErrata code hierarchy: code that encapsulates file lock for synchronizing shelf access. The class listed in [Example 14-17](#) provides tools to synchronize operations, using a lock on a file whose name is provided by systems that use the class.

It includes methods for locking and unlocking the file, but also exports higher-level methods for running function calls in exclusive or shared mode. Method `sharedAction` is used to run read operations, and `exclusiveAction` handles writes. Any number of shared actions can occur in parallel, but exclusive actions occur all by themselves and block all other action requests in parallel processes. Both kinds of actions are run in `try-finally` statements to guarantee that file locks are unlocked on action exit, normal or otherwise.

Example 14-17. PP2E\Internet\Cgi-Web\PyErrata\Mutex\mutexcntl.py

```
#####
# generally useful mixin, so a separate module;
# requires self.filename attribute to be set, and
# assumes self.filename+'.lck' file already exists;
# set mutexcntl.debugMutexCntl to toggle logging;
# writes lock log messages to self.filename+'.log';
#####

import fcntl, os, time
from FCNTL import LOCK_SH, LOCK_EX, LOCK_UN

debugMutexCntl = 1
processType = {LOCK_SH: 'reader', LOCK_EX: 'writer'}

class MutexCntl:
    def lockFile(self, mode):
        self.logPrelock(mode)
        self.lock = open(self.filename + '.lck')    # lock file in this process
        fcntl.flock(self.lock.fileno( ), mode)     # waits for lock if needed
        self.logPostlock( )

    def lockFileRead(self):                        # allow > 1 reader: shared
        self.lockFile(LOCK_SH)                    # wait if any write lock

    def lockFileWrite(self):                       # writers get exclusive lock
        self.lockFile(LOCK_EX)                    # wait if any lock: r or w

    def unlockFile(self):
        self.logUnlock( )
        fcntl.flock(self.lock.fileno( ), LOCK_UN) # unlock for other processes

    def sharedAction(self, action, *args):        # higher level interface
        self.lockFileRead( )                       # block if a write lock
```

```
try:
    result = apply(action, args)
finally:
    self.unlockFile( )
return result

def exclusiveAction(self, action, *args):
    self.lockFileWrite( )
    try:
        result = apply(action, args)
    finally:
        self.unlockFile( )
    return result

def logmsg(self, text):
    if not debugMutexCntl: return
    log = open(self.filename + '.log', 'a')
    log.write('%s\t%s\n' % (time.time( ), text))
    log.close( )

def logPrelock(self, mode):
    self.logmsg('Requested: %s, %s' % (os.getpid( ), processType[mode]))
def logPostlock(self):
    self.logmsg('Aquired: %s' % os.getpid( ))
def logUnlock(self):
    self.logmsg('Released: %s' % os.getpid( ))
```

This file lock management class is coded in its own module by design, because it is potentially worth reusing. In PyErrata, shelve database classes mix it in with multiple inheritance to implement mutual exclusion for database writers.

This class assumes that a lockable file exists as name `self.filename` (defined in client classes) with a `.lck` extension; like all instance attributes, this name can vary per client of the class. If a global variable is true, the class also optionally logs all lock operations in a file of the same name as the lock, but with a `.log` extension.

Notice that the log file is opened in a append mode; on Unix systems, this mode guarantees that log file text written by each process appears on a line of its own, not intermixed (multiple copies of this class may write to the log from parallel CGI script processes). To really understand how this class works, though, we need to say more about Python's file-locking interface.

14.6.6.1 Using `fcntl.flock` to lock files

When we studied threads in [Chapter 3](#), we saw that the Python thread module includes a mutual-exclusion lock mechanism that can be used to synchronize threads' access to shared global memory resources. This won't usually help us much in the CGI environment, however, because each database request generally comes from a distinct process spawned by the HTTP server to handle incoming request. That is, thread locks work only within the same process, because all threads run within a single process.

For CGI scripts, we usually need a locking mechanism that spans multiple processes instead. On Unix systems, the Python standard library exports a tool based on locking files, and therefore may be used across process boundaries. All of this magic happens in these two lines in the PyErrata `mutex` class:

```
fcntl.flock(self.lock.fileno( ), mode)
fcntl.flock(self.lock.fileno( ), LOCK_UN)
```

The `fcntl.flock` call in the standard Python library attempts to acquire a lock associated with a file, and by default blocks the calling process if needed until the lock can be acquired. The call accepts a file descriptor integer code (the `stdio` file object's `fileno` method returns one for us) and a mode flag defined in standard module `fcntl`, which takes one of three values in our system:

- `LOCK_EX` requests an exclusive lock, typically used for writers. This lock is granted only if other locks are held (exclusive or shared) and blocks all other lock requests (exclusive or shared) until the lock is released. This guarantees that exclusive lock holders run alone.
- `LOCK_SH` requests a shared lock, typically used for readers. Any number of processes can hold shared locks at the same time, but one is granted only if no exclusive lock is held, and new exclusive lock requests are blocked until all shared locks are released.
- `LOCK_UN` unlocks a lock previously acquired by the calling process so that other processes can acquire locks and resume execution.

In database terms, the net effect is that readers wait only if a write lock is held by another process and writers wait if any lock is held -- read or write. Though used to synchronize processes, this scheme is more complex and powerful than the simple acquire/release model for locks in the Python `thread` module, and is different from the class tools available in the higher-level `thread` module. However, it could be emulated by both these thread modules.

`fcntl.flock` internally calls out to whatever file-locking mechanism is available in the underlying operating system,^[5] and therefore you can consult the corresponding Unix or Linux manpage for more details. It's also possible to avoid blocking if a lock can't be acquired, and there are other synchronization tools in the Python library (e.g., "fifos"), but we will ignore such options here.

[5] Locking mechanisms vary per platform and may not exist at all. For instance, the `flock` call is not currently supported on Windows as of Python 1.5.2, so you may need to replace this call with a platform-specific alternative on some server machines.

14.6.6.2 Mutex test scripts

To help us understand the PyErrata synchronization model, let's get a better feel for the underlying file-locking primitives by running a few simple experiments. Examples [Example 14-18](#) and [Example 14-19](#) implement simple reader and writer processes using the `flock` call directly instead of our class. They request shared and exclusive locks, respectively.

Example 14-18. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testread.py

```
#!/usr/bin/python

import os, fcntl, time
from fcntl import LOCK_SH, LOCK_UN
print os.getpid(), 'start reader', time.time( )

file = open('test.lck', 'r')
fcntl.flock(file.fileno( ), LOCK_SH)
print os.getpid(), 'got read lock', time.time( ) # open the lock file for fd
# block if a writer has lock
# any number of readers can
```



```
for i in range(2):
    if os.fork( ) == 0:
        os.execl("./testwrite.py")           # same, but start writers

for i in range(2):
    if os.fork( ) == 0:
        os.execl("./testread.py")

for i in range(1):
    if os.fork( ) == 0:
        os.execl("./testwrite.py")
```

Comments in this script give the results for running its logic various ways on Linux. Pragmatic note: after copying these files over from Windows in an FTP'd `tar` file, I first had to give them executable permissions and convert them from DOS to Unix line-feed format before Linux would treat them as executable programs:^[6]

[6] The `+x` syntax in the `chmod` shell command here means "set the executable bit" in the file's permission bit-string for "self", the current user. At least on my machine, `chmod` accepts both the integer bit-strings used earlier and symbolic forms like this. Note that we run these tests on Linux because the Python `os.fork` call doesn't work on Windows, at least as of Python 1.5.2. It may eventually, but for now Windows scripts use `os.spawnv` instead (see [Chapter 3](#) for details).

```
[mark@toy .../PyErrata/Mutex]$ chmod +x *.py
[mark@toy .../PyErrata/Mutex]$ python $X/PyTools/fixeoln_all.py tounix "*.py"
__init__.py
launch-mutex-simple.py
launch-mutex.py
launch-test.py
mutexcntl.py
testread-mutex.py
testread.py
testwrite-mutex.py
testwrite.py
```

Once they've been so configured as executables, we can run all three of these scripts from the Linux command line. The reader and writer scripts access a *Shared.txt* file, which is meant to simulate shared resource in a real parallel application (e.g., a database in the CGI realm):

```
[mark@toy ...PyErrata/Mutex]$ ./testwrite.py
1010 start writer 960919842.773
1010 got write lock 960919842.78
1010 unlocking

[mark@toy ...PyErrata/Mutex]$ ./testread.py
1013 start reader 960919900.146
1013 got read lock 960919900.153
lines so far:      132 Shared.txt
1013 unlocking
```

The `launch-test` script simply starts a batch of the reader and writer scripts that run as parallel processes to simulate a concurrent environment (e.g., web browsers contacting a CGI script all at once):

```
[mark@toy ...PyErrata/Mutex]$ python launch-test.py
```

```
1016 start writer 960919933.206
1016 got write lock 960919933.213
1017 start reader 960919933.416
1018 start reader 960919933.455
1022 start reader 960919933.474
1021 start reader 960919933.486
1020 start writer 960919933.497
1019 start writer 960919933.508
1023 start writer 960919933.52
1016 unlocking

1017 got read lock 960919936.228
1018 got read lock 960919936.234
1021 got read lock 960919936.24
1022 got read lock 960919936.246
lines so far:      133 Shared.txt
1022 unlocking

lines so far:      133 Shared.txt
1018 unlocking

lines so far:      133 Shared.txt
1017 unlocking

lines so far:      133 Shared.txt
1021 unlocking

1019 got write lock 960919939.375
1019 unlocking

1020 got write lock 960919942.379
1020 unlocking

1023 got write lock 960919945.388
1023 unlocking
```

This output is a bit cryptic; most lines list process ID, text, and system time, and each process inserts a three-second delay (via `time.sleep`) to simulate real activities. If you look carefully, you'll notice that all processes start at roughly the same time, but access to the shared file is synchronized into this sequence:

1. One writer grabs the file first.
2. Next, all readers get it at the same time, three seconds later.
3. Finally, all other writers get the file one after another, three seconds apart.

The net result is that writer processes always access the file alone while all others are blocked. So a sequence will avoid concurrent update problems.

14.6.6.3 Mutex class test scripts

To test our mutex class outside the scope of PyErrata, we simply rewrite these scripts to hook into the class's interface. The output of Examples [Example 14-21](#) and [Example 14-22](#) is similar to the raw `fcntl` versions shown previously, but an additional log file is produced to help trace lock operations.

Example 14-21. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testread-mutex.py

```
#!/usr/bin/python
import os, time
from mutexcntl import MutexCntl

class app(MutexCntl):
    def go(self):
        self.filename = 'test'
        print os.getpid( ), 'start mutex reader'
        self.sharedAction(self.report)                # can report with others
                                                    # but not during update

    def report(self):
        print os.getpid( ), 'got read lock'
        time.sleep(3)
        print 'lines so far:', os.popen('wc -l Shared.txt').read( ),
        print os.getpid( ), 'unlocking\n'

if __name__ == '__main__': app().go( )
```

Unlike PyErrata, we don't need to change `sys.path` to allow `FCNTL` imports in the `mutexcntl` module in Examples [Example 14-21](#) and [Example 14-22](#), because we'll run these scripts as ourselves not the CGI user "nobody" (my path includes the directory where `FCNTL` lives).

Example 14-22. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testwrite-mutex.py

```
#!/usr/bin/python
import os, time
from mutexcntl import MutexCntl

class app(MutexCntl):
    def go(self):
        self.filename = 'test'
        print os.getpid( ), 'start mutex writer'
        self.exclusiveAction(self.update)            # must do this alone;
                                                    # no update or report
                                                    # can run at same time

    def update(self):
        print os.getpid( ), 'got write lock'
        log = open('Shared.txt', 'a')
        time.sleep(3)
        log.write('%d Hello\n' % os.getpid( ))
        print os.getpid( ), 'unlocking\n'

if __name__ == '__main__': app().go( )
```

The launcher is the same as [Example 14-20](#), but [Example 14-23](#) starts multiple copies of the class based readers and writers. Run [Example 14-23](#) on your server with various process counts to fool the locking mechanism.

Example 14-23. PP2E\Internet\Cgi-Web\PyErrata\launch-mutex.py

```
#!/usr/bin/python
# launch test program processes
# same, but start mutexcntl clients

import os

for i in range(1):
    if os.fork( ) == 0:
        os.execl("./testwrite-mutex.py")

for i in range(2):
    if os.fork( ) == 0:
```

```
        os.execl("./testread-mutex.py")

for i in range(2):
    if os.fork( ) == 0:
        os.execl("./testwrite-mutex.py")

for i in range(2):
    if os.fork( ) == 0:
        os.execl("./testread-mutex.py")

for i in range(1):
    if os.fork( ) == 0:
        os.execl("./testwrite-mutex.py")
```

The output of the class-based test is more or less the same. Processes start up in a different order but the synchronization behavior is identical -- one writer writes, all readers read, then remaining writers write one at a time:

```
[mark@toy .../PyErrata/Mutex]$ python launch-mutex.py
1035 start mutex writer
1035 got write lock
1037 start mutex reader
1040 start mutex reader
1038 start mutex writer
1041 start mutex reader
1039 start mutex writer
1036 start mutex reader
1042 start mutex writer
1035 unlocking

1037 got read lock
1041 got read lock
1040 got read lock
1036 got read lock
lines so far:      137 Shared.txt
1036 unlocking

lines so far:      137 Shared.txt
1041 unlocking

lines so far:      137 Shared.txt
1040 unlocking

lines so far:      137 Shared.txt
1037 unlocking

1038 got write lock
1038 unlocking

1039 got write lock
1039 unlocking

1042 got write lock
1042 unlocking
```

All times have been removed from launcher output this time, because our mutex class automatically logs lock operations in a separate file, with times and process IDs; the three-second sleep per process is more obvious in this format:

```
[mark@toy .../PyErrata/Mutex]$ cat test.log
```



```
960920109.518 Requested: 1035, writer
960920109.518 Aquired: 1035
960920109.626 Requested: 1040, reader
960920109.646 Requested: 1038, writer
960920109.647 Requested: 1037, reader
960920109.661 Requested: 1041, reader
960920109.674 Requested: 1039, writer
960920109.69 Requested: 1036, reader
960920109.701 Requested: 1042, writer
960920112.535 Released: 1035
960920112.542 Aquired: 1037
960920112.55 Aquired: 1041
960920112.557 Aquired: 1040
960920112.564 Aquired: 1036
960920115.601 Released: 1036
960920115.63 Released: 1041
960920115.657 Released: 1040
960920115.681 Released: 1037
960920115.681 Aquired: 1038
960920118.689 Released: 1038
960920118.696 Aquired: 1039
960920121.709 Released: 1039
960920121.716 Aquired: 1042
960920124.728 Released: 1042
```

Finally, this is what the shared text file looks like after all these processes have exited stage left. Each writer simply added a line with its process ID; it's not the most amazing of parallel process results, but if you pretend that this is our PyErrata shelve-based database, these tests seem much more meaningful:

```
[mark@toy .../PyErrata/Mutex]$ cat Shared.txt
1010 Hello
1016 Hello
1019 Hello
1020 Hello
1023 Hello
1035 Hello
1038 Hello
1039 Hello
1042 Hello
```

14.7 Administrative Tools

Now that we have finished implementing a Python-powered, web-enabled, concurrently accessible report database, and published web pages and scripts that make that database accessible to the cyberworld at large, we can sit back and wait for reports to come in. Or almost; there still no way for the site owner to view or delete records offline. Moreover, all records are tagged as "not yet verified" on submission, and must somehow be verified or rejected.

This section lists a handful of tersely documented PyErrata scripts that accomplish such tasks. A are Python programs shipped in the top-level *AdminTools* directory and are assumed to be run from a shell command line on the server (or other machine, after database downloads). They implement simple database content dumps, database backups, and database state-changes and deletions for use by the errata site administrator.

These tasks are infrequent, so not much work has gone into these tools. Frankly, some fall into tl domain of "quick and dirty" hackage and aren't as robust as they could be. For instance, becau these scripts bypass the database interface classes and speak directly to the underlying file structures, changes in the underlying file mechanisms will likely break these tools. Also in a mor polished future release, these tools might instead sprout GUI- or web-based user interfaces to support over-the-net administration. For now, such extensions are left as exercises for the ambitious reader.

14.7.1 Backup Tools

System backup tools simply spawn the standard Unix `tar` and `gzip` command-line programs to copy databases into single compressed files. You could write a shell script for this task too, but Python works just as well, as shown in Examples [Example 14-24](#) and [Example 14-25](#).

Example 14-24. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\backupFiles.py

```
#!/usr/bin/python
import os
os.system('tar -cvf DbaseFiles.tar ../DbaseFiles')
os.system('gzip DbaseFiles.tar')
```

Example 14-25. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\backupShelve.py

```
#!/usr/bin/python
import os
os.system('tar -cvf DbaseShelve.tar ../DbaseShelve')
os.system('gzip DbaseShelve.tar')
```

14.7.2 Display Tools

The scripts in Examples [Example 14-26](#) and [Example 14-27](#) produce raw dumps of each databas

structure's contents. Because the databases use pure Python storage mechanisms (pickles, shelve these scripts can work one level below the published database interface classes; whether they should depends on how much code you're prepared to change when your database model evolve Apart from printing generated record filenames and shelve keys, there is no reason that these scripts couldn't be made less brittle by instead calling the database classes' `loadSortedTable` methods. Suggested exercise: do better.

Example 14-26. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\dumpFiles.py

```
#!/usr/bin/python
import glob, pickle

def dump(kind):
    print '\n', kind, '='*60, '\n'
    for file in glob.glob("../DbaseFiles/%s/*.data" % kind):
        print '\n', '-'*60
        print file
        print pickle.load(open(file, 'r'))

dump('errataDB')
dump('commentDB')
```

Example 14-27. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\dumpShelve.py

```
#!/usr/bin/python
import shelve
e = shelve.open('../DbaseShelve/errataDB')
c = shelve.open('../DbaseShelve/commentDB')

print '\n', 'Errata', '='*60, '\n'
print e.keys( )
for k in e.keys( ): print '\n', k, '-'*60, '\n', e[k]

print '\n', 'Comments', '='*60, '\n'
print c.keys( )
for k in c.keys( ): print '\n', k, '-'*60, '\n', c[k]
```

Running these scripts produces the following sorts of results (truncated at 80 characters to fit in this book). It's not nearly as pretty as the web pages generated for the user in PyErrata, but could be piped to other command-line scripts for further offline analysis and processing. For instance, the dump scripts' output could be sent to a report-generation script that knows nothing of the We

```
[mark@toy ../Internet/Cgi-Web/PyErrata/AdminTools]$ python dumpFiles.py

errataDB =====

-----
../DbaseFiles/errataDB/937907956.159-5157.data
{'Page number': '42', 'Type': 'Typo', 'Severity': 'Low', 'Chapter number': '3'}.
-----
...more...

commentDB =====

-----
../DbaseFiles/commentDB/937908410.203-5352.data
{'Submit date': '1999/09/21, 06:06:50', 'Submitter email': 'bob@bob.com',...
```

```
-----
...more...

[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python dumpShelve.py

Errata =====

['938245136.363-20046', '938244808.434-19964']

938245136.363-20046 -----
{'Page number': '256', 'Type': 'Program bug', 'Severity': 'High', 'Chapter nu..

938244808.434-19964 -----
{'Page number': 'various', 'Type': 'Suggestion', 'Printing Date': '', 'Chapte..

Comments =====

['938245187.696-20054']

938245187.696-20054 -----
{'Submit date': '1999/09/25, 03:39:47', 'Submitter email': 'bob@bob.com', 'Re..
```

14.7.3 Report State-Change Tools

Our last batch of command-line tools allows the site owner to mark reports as verified or rejected and to delete reports altogether. The idea is that someone will occasionally run these scripts offline, as time allows, to change states after investigating reports. And this is the end to our quest for errata automation: the investigation process itself is assumed to require both time and brains.

There are no interfaces in the database's classes for changing existing reports, so these scripts can at least make a case for going below the classes to the physical storage mediums. On the other hand, the classes could be extended to support such update operations too, with interfaces that could also be used by future state-change tools (e.g., web interfaces).

To minimize some redundancy, let's first define state-change functions in a common module list in [Example 14-28](#), so they may be shared by both the file and shelve scripts.

Example 14-28. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifycommon.py

```
#####
# put common verify code in a shared module for consistency and
# reuse; could also generalize dbase update scan, but this helps
#####

def markAsVerify(report):
    report['Report state'] = 'Verified by author'

def markAsReject(report):
    reason = ''
    while 1:
        try:
            line = raw_input('reason>')
        except EOFError:
            break
    reason = reason + line + '\n'
    report['Report state'] = 'Rejected - not a real bug'
    report['Description'] = ('Reject reason: ' + reason +
        '\n[Original description=>]\n' + report['Description'])
```

To process state changes on the file-based database, we simply iterate over all the pickle files in the database directories, as shown in [Example 14-29](#).

Example 14-29. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifyFiles.py

```
#!/usr/bin/python
#####
# report state change and deletion operations;
# also need a tool for anonymously publishing reports
# sent by email that are of general interest--for now,
# they can be entered with the submit forms manually;
# this is text-based: the idea is that records can be
# browsed in the errata page first (sort by state to
# see unverified ones), but an edit gui or web-based
# verification interface might be very useful to add;
#####

import glob, pickle, os
from verifycommon import markAsVerify, markAsReject

def analyse(kind):
    for file in glob.glob("../DbaseFiles/%s/*.data" % kind):
        data = pickle.load(open(file, 'r'))
        if data['Report state'] == 'Not yet verified':
            print data
            if raw_input('Verify?') == 'y':
                markAsVerify(data)
                pickle.dump(data, open(file, 'w'))
            elif raw_input('Reject?') == 'y':
                markAsReject(data)
                pickle.dump(data, open(file, 'w'))
            elif raw_input('Delete?') == 'y':
                os.remove(file) # same as os.unlink

print 'Errata...'; analyse('errataDB')
print 'Comments...'; analyse('commentDB')
```

When run from the command line, the script displays one report's contents at a time and pauses after each to ask if it should be verified, rejected, or deleted. Here is the beginning of one file database verify session, shown with line wrapping so you can see what I see (it's choppy but compact):

```
[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python verifyFiles.py
Errata...
{'Page number': '12', 'Type': 'Program bug', 'Printing Date': '', 'Chapter number': '', 'Submit date': '1999/09/21, 06:17:13', 'Report state': 'Not yet verified', 'Submitter name': 'Lisa Lutz', 'Submitter email': '', 'Description': '1 + 1 = 2, not 3...\\015\\012', 'Submit mode': '', 'Part number': '', 'Severity': 'High'}
Verify?n
Reject?n
Delete?n
{'Page number': '', 'Type': 'Program bug', 'Printing Date': '', 'Chapter number': '16', 'Submit date': '1999/09/21, 06:20:22', 'Report state': 'Not yet verified', 'Submitter name': 'jerry', 'Submitter email': 'http://www.jerry.com', 'Description': 'Help! I just spilled coffee all over my\\015\\012computer...\\015\\012', 'Submit mode': '', 'Part number': '', 'Severity': 'Unknown'}
Verify?n
Reject?y
reason>It's not Python's fault
reason>(ctrl-d)
...more...
```

Verifications and rejections change records, but deletions actually remove them from the system. In `verifycommon`, a report rejection prompts for an explanation and concatenates it to the original

description. Deletions delete the associated file with `os.remove`; this feature may come in handy the system is ever abused by a frivolous user (including me, while writing examples for this book). The shelfe -based version of the verify script looks and feels similar, but deals in shelves instead of flat files, as shown in [Example 14-30](#).

Example 14-30. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifyShelve.py

```
#!/usr/bin/python
#####
# like verifyFiles.py, but do it to shelves;
# caveats: we should really obtain a lock before shelve
# updates here, and there is some scan logic redundancy
#####

import shelve
from verifycommon import markAsVerify, markAsReject

def analyse(dbase):
    for k in dbase.keys( ):
        data = dbase[k]
        if data['Report state'] == 'Not yet verified':
            print data
            if raw_input('Verify?') == 'y':
                markAsVerify(data)
                dbase[k] = data
            elif raw_input('Reject?') == 'y':
                markAsReject(data)
                dbase[k] = data
            elif raw_input('Delete?') == 'y':
                del dbase[k]

print 'Errata...'; analyse(shelve.open('../DbaseShelve/errataDB'))
print 'Comments...'; analyse(shelve.open('../DbaseShelve/commentDB'))
```

Note that the `verifycommon` module helps ensure that records are marked consistently and avoid some redundancy. However, the file and shelve verify scripts still look very similar; it might be better to further generalize the notion of database update scans by moving this logic into the storage-specific database interface classes shown earlier.

Short of doing so, there is not much we can do about the scan-logic redundancy or storage-structure dependencies of the file and shelve verify scripts. The existing load-list database class methods won't help, because they don't provide the generated filename and shelve key details we need to rewrite records here. To make the administrative tools more robust, some database class redesign would probably be in order -- which seems as good a segue to the next section as any.

14.8 Designing for Reuse and Growth

I admit it: PyErrata may be thrifty, but it's also a bit self-centered. The database interfaces presented in the prior sections work as planned and serve to separate all database processing from CGI scripting details. But as shown in this book, these interfaces aren't as generally reusable as they could be; moreover, they are not yet designed to scale up to larger database applications.

Let's wrap up this chapter by donning our software code review hats for just a few moments and exploring some design alternatives for PyErrata. In this section, I highlight the PyErrata database interface's obstacles to general applicability, not as self-deprecation, but to show how programming decisions can impact reusability.

Something else is going on in this section too. There is more concept than code here, and the code that is here is more like an experimental design than a final product. On the other hand, because that design is coded in Python, it can be run to test the feasibility of design alternatives; as we've seen, Python can be used as a form of executable pseudocode.

14.8.1 Reusability

As we saw, code reuse is pervasive within PyErrata: top-level calls filter down to common `browse` and `submit` modules, which in turn call database classes that reuse a common module. But what about sharing PyErrata code with other systems? Although not designed with generality in mind PyErrata's database interface modules could almost be reused to implement other kinds of file- or shelve-based databases outside the context of PyErrata itself. However, we need a few more tweaks to turn these interfaces into widely useful tools.

As is, `shelve` and `file-directory` names are hardcoded into the storage-specific subclass modules, but another system could import and reuse their `Dbase` classes and provide different directory names. Less generally, though, the `dbcommon` module adds two attributes to all new records (`submit-time` and `report-state`) that may or may not be relevant outside PyErrata. It also assumes that stored values are mappings (dictionaries), but that is less PyErrata-specific.

If we were to rewrite these classes for more general use, it would make sense to first repackage the four `DbaseErrata` and `DbaseComment` classes in modules of their own (they are very specific instances of file and shelve databases). We would probably also want to somehow relocate `dbcommon`'s insertion of `submit-time` and `report-state` attributes from the `dbcommon` module to the four classes themselves (these attributes are specific to PyErrata databases). For instance, we might define a new `DbasePyErrata` class that sets these attributes and is a mixed-in superclass to the four PyErrata storage-specific database classes:

```
# in new module
class DbasePyErrata:
    def storeItem(self, newdata):
        secsSinceEpoch      = time.time( )
        timeTuple           = time.localtime(secsSinceEpoch)
        y_m_d_h_m_s         = timeTuple[:6]
        newdata['Submit date'] = '%s/%02d/%02d, %02d:%02d:%02d' % y_m_d_h_m_s
        newdata['Report state'] = 'Not yet verified'
        self.writeItem(newdata)

# in dbshelve
```

```
class Dbase(MutexCntl, dbcommon.Dbase):
    # as is

# in dbfiles
class Dbase(dbcommon.Dbase):
    # as is

# in new file module
class DbaseErrata(DbasePyErrata, dbfiles.Dbase):
    dirname = 'DbaseFiles/errataDB/'
class DbaseComment(DbasePyErrata, dbfiles.Dbase):
    dirname = 'DbaseFiles/commentDB/'

# in new shelve module
class DbaseErrata(DbasePyErrata, dbshelve.Dbase):
    filename = 'DbaseShelve/errataDB'
class DbaseComment(DbasePyErrata, dbshelve.Dbase):
    filename = 'DbaseShelve/commentDB'
```

There are more ways to structure this than we have space to cover here. The point is that by factoring out application-specific code, `dbshelve` and `dbfiles` modules not only serve to keep `PyErrata` interface and database code distinct, but also become generally useful data-storage tool

14.8.2 Scalability

`PyErrata`'s database interfaces were designed for this specific application's storage requirements alone and don't directly support very large databases. If you study the database code carefully, you'll notice that submit operations update a single item, but browse requests load entire report databases all at once into memory. This scheme works fine for the database sizes expected in `PyErrata`, but performs badly for larger data sets. We could extend the database classes to handle larger data sets too, but they would likely require new top-level interfaces altogether.

Before I stopped updating it, the static HTML file used to list errata from the first edition of this book held just some 60 reports, and I expect a similarly small data set for other books and editions. With such small databases, it's reasonable to load an entire database into memory (i.e., into Python lists and dictionaries) all at once, and frequently. Indeed, the time needed to transfer web page containing 60 records across the Internet likely outweighs the time it takes to load 60 report files or shelve keys on the server.

On the other hand, the database may become too slow if many more reports than expected are posted. There isn't much we could do to optimize the "Simple list" and "With index" display options, since they really do display all records. But for the "Index only" option, we might be able to change our classes to load only records having a selected value in the designated report field.

For instance, we could work around database load bottlenecks by changing our classes to implement delayed loading of records: rather than returning the real database, load requests could return objects that look the same but fetch actual records only when needed. Such an approach might require no changes in the rest of the system's code, but may be complex to implement.

14.8.2.1 Multiple shelve field indexing

Perhaps a better approach would be to define an entirely new top-level interface for the "Index only" option -- one that really does load only records matching a field value query. For instance, rather than storing all records in a single shelve, we could implement the database as a set of index shelves, one per record field, to associate records by field values. Index shelve keys would be

values of the associated field; shelve values would be lists of records having that field value. The shelve entry lists might contain either redundant copies of records, or unique names of flat files holding the pickled record dictionaries, external to the index shelves (as in the current flat-file model).

For example, the PyErrata comment database could be structured as a directory of flat files to hold pickled report dictionaries, together with five shelves to index the values in all record fields (submitter-name, submitter-email, submit-mode, submit-date, report-state). In the report-state shelve, there would be one entry for each possible report state (verified, rejected, etc.); each entry would contain a list of records with just that report-state value. Field value queries would be fast but store and load operations would become more complex:

- To store a record in such a scheme, we would first pickle it to a uniquely named flat file, then insert that file's name into lists in all five shelves, using each field's value as shelve key.
- To load just the records matching a field/value combination, we would first index that field shelve on the value to fetch a filename list, and step through that list to load matching records only, from flat pickle files.

Let's take the leap from hypothetical to concrete, and prototype these ideas in Python. If you're following closely, you'll notice that what we're really talking about here is an extension to the flat file database structure, one that merely adds index shelves. Hence, one possible way to implement the model is as a subclass of the current flat-file classes. [Example 14-31](#) does just that, as proof of the design concept.

Example 14-31. PP2E\Internet\PyErrata\AdminTools\dbaseindexed.py

```
#####
# add field index shelves to flat-file database mechanism;
# to optimize "index only" displays, use classes at end of this file;
# change browse, index, submit to use new loaders for "Index only" mode;
# minor nit: uses single lock file for all index shelve read/write ops;
# storing record copies instead of filenames in index shelves would be
# slightly faster (avoids opening flat files), but would take more space;
# falls back on original brute-force load logic for fields not indexed;
# shelve.open creates empty file if doesn't yet exist, so never fails;
# to start, create DbaseFilesIndex/{commentDB,errataDB}/indexes.lck;
#####

import sys; sys.path.insert(0, '..')          # check admin parent dir first
from Mutex import mutexcntl                 # fcntl path okay: not 'nobody'
import dbfiles, shelve, pickle, string, sys

class Dbase(mutexcntl.MutexCntl, dbfiles.Dbase):
    def makeKey(self):
        return self.cachedKey
    def cacheKey(self):                       # save filename
        self.cachedKey = dbfiles.Dbase.makeKey(self) # need it here too
        return self.cachedKey

    def indexName(self, fieldname):
        return self.dirname + string.replace(fieldname, ' ', '-')

    def safeWriteIndex(self, fieldname, newdata, recfilename):
        index = shelve.open(self.indexName(fieldname))
        try:
            keyval = newdata[fieldname]          # recs have all field
```

```
        reclist = index[keyval]                                # fetch, mod, rewrite
        reclist.append(recfilename)                          # add to current list
        index[keyval] = reclist
    except KeyError:
        index[keyval] = [recfilename]                        # add to new list

def safeLoadKeysList(self, fieldname):
    if fieldname in self.indexfields:
        keys = shelve.open(self.indexName(fieldname)).keys( )
        keys.sort( )
    else:
        keys, index = self.loadIndexedTable(fieldname)
    return keys

def safeLoadByKey(self, fieldname, fieldvalue):
    if fieldname in self.indexfields:
        dbase = shelve.open(self.indexName(fieldname))
        try:
            index = dbase[fieldvalue]
            reports = []
            for filename in index:
                pathname = self.dirname + filename + '.data'
                reports.append(pickle.load(open(pathname, 'r')))
            return reports
        except KeyError:
            return []
    else:
        key, index = self.loadIndexedTable(fieldname)
        try:
            return index[fieldvalue]
        except KeyError:
            return []

# top-level interfaces (plus dbcommon and dbfiles)

def writeItem(self, newdata):
    # extend to update indexes
    filename = self.cacheKey( )
    dbfiles.Dbase.writeItem(self, newdata)
    for fieldname in self.indexfields:
        self.exclusiveAction(self.safeWriteIndex,
                             fieldname, newdata, filename)

def loadKeysList(self, fieldname):
    # load field's keys list only
    return self.sharedAction(self.safeLoadKeysList, fieldname)

def loadByKey(self, fieldname, fieldvalue):
    # load matching recs list only
    return self.sharedAction(self.safeLoadByKey, fieldname, fieldvalue)

class DbaseErrata(Dbase):
    dirname      = 'DbaseFilesIndexed/errataDB/'
    filename     = dirname + 'indexes'
    indexfields  = ['Submitter name', 'Submit date', 'Report state']

class DbaseComment(Dbase):
    dirname      = 'DbaseFilesIndexed/commentDB/'
    filename     = dirname + 'indexes'
    indexfields  = ['Submitter name', 'Report state']    # index just these

#
# self-test
#

if __name__ == '__main__':
    import os
```

```
dbase = DbaseComment( )
os.system('rm %s*' % dbase.dirname) # empty dbase dir
os.system('echo > %s.lck' % dbase.filename) # init lock file

# 3 recs; normally have submitter-email and description, not page
# submit-date and report-state are added auto by rec store method
records = [{'Submitter name': 'Bob', 'Page': 38, 'Submit mode': ''},
           {'Submitter name': 'Brian', 'Page': 40, 'Submit mode': ''},
           {'Submitter name': 'Bob', 'Page': 42, 'Submit mode': 'email'}]
for rec in records: dbase.storeItem(rec)

dashes = '-'*80
def one(item):
    print dashes; print item
def all(list):
    print dashes
    for x in list: print x

one('old stuff')
all(dbase.loadSortedTable('Submitter name')) # load flat list
all(dbase.loadIndexedTable('Submitter name')) # load, grouped
#one(dbase.loadIndexedTable('Submitter name')[0])
#all(dbase.loadIndexedTable('Submitter name')[1]['Bob'])
#all(dbase.loadIndexedTable('Submitter name')[1]['Brian'])

one('new stuff')
one(dbase.loadKeysList('Submitter name')) # bob, brian
all(dbase.loadByKey('Submitter name', 'Bob')) # two recs match
all(dbase.loadByKey('Submitter name', 'Brian')) # one rec mathces
one(dbase.loadKeysList('Report state')) # all match
all(dbase.loadByKey('Report state', 'Not yet verified'))

one('boundary cases')
all(dbase.loadByKey('Submit mode', '')) # not indexed: load
one(dbase.loadByKey('Report state', 'Nonesuch')) # unknown value: []
try: dbase.loadByKey('Nonesuch', 'Nonesuch') # bad fields: exc
except: print 'Nonesuch failed'
```

This module's code is something of an executable prototype, but that's much of the point here. The fact that we can actually run experiments coded in Python helps pinpoint problems in a model early on.

For instance, I had to redefine the `makeKey` method here to cache filenames locally (they are needed for index shelves too). That's not quite right, and if I were to adopt this database interface I would probably change the file class to return generated filenames, not discard them. Such misfits can often be uncovered only by writing real code -- a task that Python optimizes by design.

If this module is run as a top-level script, its self-test code at the bottom of the file executes with the following output. I don't have space to explain it in detail, but try to match it up with the module's self-test code to trace how queries are satisfied with and without field indexes:

```
[mark@toy .../Internet/Cgi-Web/PyErrata/AdminTools]$ python dbaseindexed.py
-----
old stuff
-----
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Re
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report
tate': 'Not yet verified', 'Submitter name': 'Brian'}
-----
['Bob', 'Brian']
{'Bob': [{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '',
```

```
'Report state': 'Not yet verified', 'Submitter name': 'Bob'}, {'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}], 'Brian': [{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Brian'}]}
```

new stuff

['Bob', 'Brian']

{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}

{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Brian'}

['Not yet verified']

{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Brian'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}

boundary cases

{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'Submitter name': 'Brian'}

[]

Nonesuch failed

```
[mark@toy .../PyErrata/AdminTools]$ ls DbaseFilesIndexed/commentDB/
960918301.263-895.data  960918301.506-895.data  Submitter-name  indexes.log
960918301.42-895.data  Report-state             indexes.lck
```

```
[mark@toy .../PyErrata/AdminTools]$ more DbaseFilesIndexed/commentDB/indexes.log
960918301.266  Requested: 895, writer
960918301.266  Aquired: 895
960918301.36   Released: 895
960918301.36   Requested: 895, writer
960918301.361  Aquired: 895
960918301.419  Released: 895
960918301.422  Requested: 895, writer
960918301.422  Aquired: 895
960918301.46   Released: 895
...more...
```

One drawback to this interface is that it works only on a machine that supports the `fcntl.flock` call (notice that I ran the previous test on Linux). If you want to use these classes to support indexed file/shelve databases on other machines, you could delete or stub out this call in the `mut` module to do nothing and return. You won't get safe updates if you do, but many applications don't need to care:

```
try:
    import fcntl
    from FCNTL import *
except ImportError:
    class fakeFcntl:
        def flock(self, fileno, flag): return
    fcntl = fakeFcntl( )
    LOCK_SH = LOCK_EX = LOCK_UN = 0
```

You might instead instrument `MutexCntl.lockFile` to do nothing in the presence of a command line argument `flag`, mix in a different `MutexCntl` class at the bottom that does nothing on lock calls, or hunt for platform-specific locking mechanisms (e.g., the Windows extensions package exports a Windows-only file locking call; see its documentation for details).

Regardless of whether you use locking or not, the `dbaseindexed` flat-files plus multiple-shelve indexing scheme can speed access by keys for large databases. However, it would also require changes to the top-level CGI script logic that implements "Index only" displays, and so is not without seams. It may also perform poorly for very large databases, as record information would span multiple files. If pressed, we could finally extend the database classes to talk to a real database system such as Oracle, MySQL, PostGres, or Gadfly (described in [Chapter 16](#)).

All of these options are not without trade-offs, but we have now come dangerously close to stepping beyond the scope of this chapter. Because the PyErrata database modules were designed with neither general applicability nor broad scalability in mind, additional mutations are left as suggested exercises.

Chapter 14. Larger Web Site Examples II

[Section 14.1. "Typos Happen"](#)

[Section 14.2. The PyErrata Web Site](#)

[Section 14.3. The Root Page](#)

[Section 14.4. Browsing PyErrata Reports](#)

[Section 14.5. Submitting PyErrata Reports](#)

[Section 14.6. PyErrata Database Interfaces](#)

[Section 14.7. Administrative Tools](#)

[Section 14.8. Designing for Reuse and Growth](#)

15.1 "Surfing on the Shoulders of Giants"

This chapter concludes our look at Python Internet programming by exploring a handful of Internet-related topics and packages. We've covered many Internet topics in the previous five chapters -- socket basics, client and server-side scripting tools, and programming full-blown web sites with Python. Yet we still haven't seen many of Python's standard built-in Internet modules in action. Moreover, there is a rich collection of third-party extensions for scripting the Web with Python that we have not touched on at all.

In this chapter, we explore a grab-bag of additional Internet-related tools and third-party extensions of interest to Python Internet developers. Along the way, we meet larger Internet packages, such as HTMLgen, JPython, Zope, PSP, Active Scripting, and Grail. We'll also study standard Python tools useful to Internet programmers, including Python's restricted execution mode, XML support, COM interfaces, and techniques for implementing proprietary servers. In addition to their practical uses, these systems demonstrate just how much can be achieved by wedding a powerful object-oriented scripting language such as Python to the Web.

Before we start, a disclaimer: none of these topics is presented in much detail here, and undoubtedly some interesting Internet systems will not be covered at all. Moreover, the Internet evolves at lightning speed, and new tools and techniques are certain to emerge after this edition is published; indeed, most of the systems in this chapter appeared in the five years after the first edition of this book was written, and the next five years promise to be just as prolific. As always, the standard moving-target caveat applies: read the Python library manual's Internet section for details we've skipped, and stay in touch with the Python community at <http://www.python.org> for information about extensions not covered due to a lack of space or a lack of clairvoyance.

15.10 Rolling Your Own Servers in Python

Most of the Internet modules we looked at in the last few chapters deal with client-side interface such as FTP and POP, or special server-side protocols such as CGI that hide the underlying serv itself. If you want to build servers in Python by hand, you can do so either manually or by using higher-level tools.

15.10.1 Coding Solutions

We saw the sort of code needed to build servers manually in [Chapter 10](#). Python programs typically implement servers either by using raw socket calls with threads, forks, or selects to handle clients in parallel, or by using the `SocketServer` module.

In either case, to serve requests made in terms of higher-level protocols such as FTP, NNTP, and HTTP, you must listen on the protocol's port and add appropriate code to handle the protocol's message conventions. If you go this route, the client-side protocol modules in Python's standard library can help you understand the message conventions used. You may also be able to uncover protocol server examples in the Demos and Tools directories of the Python source distribution an on the Net at large (search <http://www.python.org>). See prior chapters for more details on writin socket-based servers.

As a higher-level interface, Python also comes with precoded HTTP web protocol server implementations, in the form of three standard modules. `BaseHTTPServer` implements the server itself; this class is derived from the standard `SocketServer.TCPServer` class. `SimpleHTTPServer` and `CGIHTTPServer` implement standard handlers for incoming HTTP requests; the former handles simple web page file requests, while the latter also runs referenced CGI scripts on the server machine by forking processes.

For example, to start a CGI-capable HTTP server, simply run Python code like that shown in [Example 15-19](#) on the server machine.

Example 15-19. PP2E\Internet\Other\webserver.py

```
#!/usr/bin/python
#####
# implement a HTTP server in Python which
# knows how to run server-side CGI scripts;
# change root dir for your server machine
#####

import os
from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler
os.chdir("/home/httpd/html") # run in html root dir
srvraddr = ("", 80) # my hostname, portnumbe
srvrobj = HTTPServer(srvraddr, CGIHTTPRequestHandler)
srvrobj.serve_forever( ) # run as perpetual dem
```


This assumes that you have appropriate permissions to run such a script, of course; see the Python library manual for more details on pre-coded HTTP server and request handler modules. Once you have your server running, you can access it in any web browser or by using either the Python `httplib` module, which implements the client side of the HTTP protocol, or the Python `urllib` module, which provides a file-like interface to data fetched from a named URL address (see the `urllib` examples in [Chapter 11](#)Chapter 11, and [Chapter 13](#), and use a URL of the form "http://.. to access HTTP documents).

15.10.2 Packaged Solutions

Finally, you can deploy full-blown, open source, and Python-friendly web servers and tools that are freely available on the Net. These may change over time too, but here are a few current options:

Medusa, asyncore

The Medusa system (<http://www.nightmare.com/medusa>) is an architecture for building long-running, high-performance network servers in Python, and is used in several mission-critical systems. Beginning in Python 1.5.2, the core of Medusa is now standard in Python in the form of the `asyncore` and `asynchat` library modules. These standard modules may be used by themselves to build high-performance network servers, based on an asynchronous multiplexing, single-process model. They use an event loop built using the `select` system call presented in [Chapter 10](#) of this book to provide concurrency without spawning threads or processes, and are well-suited to handling short-lived transactions. See the Python library for details. The complete Medusa system (not shipped with Python) also provides pre-coded HTTP and FTP servers; it is free for noncommercial use, but requires a license otherwise.

Zope

If you are doing any server-side work at all, be sure to consider the Zope open source web application server, described earlier in this chapter and at <http://www.zope.org>. Zope provides a full-featured web framework that implements an object model that is well beyond standard server-side CGI scripting. The Zope world has also developed full-blown servers (e.g., Zserver).

Mailman

If you are looking for email list support, be sure to explore the GNU mailing list manager, otherwise known as Mailman. Written in Python, Mailman provides a robust, quick, and feature-rich email discussion list tool. Mailman allows users to subscribe over the Web, supports web-based administration, and provides mail-to-news gateways and integrated spam prevention (spam of the junk mail variety, that is). At this time, <http://www.list.org> is the place to find more Mailman details.

Apache

If you are adventurous, you may be interested in the highly configurable Apache open source web server. Apache is one of the dominant servers used on the Web today, despite its free nature. Among many other things, it supports running Python server-side scripts in a variety of modes; see the site <http://www.apache.org> for details on Apache itself.

PyApache

If you use Apache, also search the Python web site for information on the PyApache Apache server module (sometimes called `mod_pyapache`), which embeds a Python interpreter inside Apache to speed up the process of launching Python server-side scripts. CGI scripts are passed to the embedded interpreter directly, avoiding interpreter startup costs. PyApache also opens up the possibility of scripting Apache's internal components.

mod_python

As I wrote this chapter, another package for embedding Python within the Apache web server appeared on the open source landscape: `mod_python`, available at <http://www.modpython.org>. According to its release notes, `mod_python` also allows Python to be embedded in Apache, with a substantial boost in performance and added flexibility. The beta release announcement for this system appeared on `comp.lang.python` the very week that this section was written, so check the Web for its current status.

Be sure to watch <http://www.python.org> for new developments on the server front, as well as lat breaking advances in Python web scripting techniques in general.

15.2 Zope: A Web Publishing Framework

Zope is an open source web-application server and toolkit, written in and customizable with Python. It is a server-side technology that allows web designers to implement sites and applications by publishing Python object hierarchies on the Web. With Zope, programmers can focus on writing objects, and let Zope handle most of the underlying HTTP and CGI details. If you are interested in implementing more complex web sites than the form-based interactions we've seen in the last three chapters, you should investigate Zope: it can obviate many of the tasks that web scripters wrestle with on a daily basis.

Sometimes compared to commercial web toolkits such as ColdFusion, Zope is made freely available over the Internet by a company called Digital Creations and enjoys a large and very active development community. Indeed, many attendees at a recent Python conference were attracted by Zope, which had its own conference track. The use of Zope has spread so quickly that many Pythonistas now look to it as Python's "killer application" -- a system so good that it naturally pushes Python into the development spotlight. At the least, Zope offers a new, higher-level way of developing sites for the Web, above and beyond raw CGI scripting.^[1]

[1] Over the years, observers have also pointed to other systems as possible Python "killer applications," including Grail, Python's COM support on Windows, and JPython. I hope they're all right, and fully expect new killers to arise after this edition is published. But at the time that I write this, Zope is attracting an astonishing level of interest among both developers and investors.

15.2.1 Zope Components

Zope began life as a set of tools (part of which was named "Bobo") placed in the public domain by Digital Creations. Since then, it has grown into a large system with many components, a growing body of add-ons (called "products" in Zope parlance), and a fairly steep learning curve. We can't do it any sort of justice in this book, but since Zope is one of the most popular Python-based applications at this writing, I'd be remiss if I didn't provide a few details here.

In terms of its core components, Zope includes the following parts:

Zope Object Request Broker (ORB)

At the heart of Zope, the ORB dispatches incoming HTTP requests to Python objects and returns results to the requestor, working as a perpetually running middleman between the HTTP CGI world and your Python objects. The Zope ORB is described further in the next section.

HTML document templates

Zope provides a simple way to define web pages as templates, with values automatically inserted from Python objects. Templates allow an object's HTML representation to be defined independently of the object's implementation. For instance, values of attributes in a class instance object may be automatically plugged into a template's text by name. Template coders need not be Python coders, and vice versa.

Object database

To record data persistently, Zope comes with a full object-oriented database system for storing Python objects. The Zope object database is based on the Python `pickle` serialization module we'll meet in the next part of this book, but adds support for transactions, lazy object fetches (sometimes called delayed evaluation), concurrent access, and more. Objects are stored and retrieved by key, much as they are with Python's standard `shelve` module, but classes must subclass an imported `Persistent` superclass, and object stores are instances of an imported `PickleDictionary` object. Zope starts and commits transactions at the start and end of HTTP requests.

Zope also includes a management framework for administrating sites, as well as a product API used to package components. Zope ships with these and other components integrated into a whole system, but each part can be used on its own as well. For instance, the Zope object database can be used in arbitrary Python applications by itself.

15.2.2 What's Object Publishing?

If you're like me, the concept of publishing objects on the Web may be a bit vague at first glance, but it's fairly simple in Zope: the Zope ORB automatically maps URLs requested by HTTP into calls on Python objects. Consider the Python module and function in [Example 15-1](#).

Example 15-1. PP2E\Internet\Other\messages.py

```
"A Python module published on the Web by Zope"

def greeting(size='brief', topic='zope'):
    "a published Python function"
    return 'A %s %s introduction' % (size, topic)
```

This is normal Python code, of course, and says nothing about Zope, CGI, or the Internet at large. We may call the function it defines from the interactive prompt as usual:

```
C:\...\PP2E\Internet\Other>python
>>> import messages
>>> messages.greeting( )
'A brief zope introduction'

>>> messages.greeting(size='short')
'A short zope introduction'

>>> messages.greeting(size='tiny', topic='ORB')
'A tiny ORB introduction'
```

But if we place this module file, along with Zope support files, in the appropriate directory on a server machine running Zope, it automatically becomes visible on the Web. That is, the function becomes a published object -- it can be invoked through a URL, and its return value

becomes a response page. For instance, if placed in a `cgi-bin` directory on a server called `myserver.net`, the following URLs are equivalent to the three calls above:

```
http://www.myserver.net/cgi-bin/messages/greeting
http://www.myserver.net/cgi-bin/messages/greeting?size=short
http://www.myserver.net/cgi-bin/messages/greeting?size=tiny&topic=ORB
```

When our function is accessed as a URL over the Web this way, the Zope ORB performs two feats of magic:

- The URL is automatically translated into a call to the Python function. The first part of the URL after the directory path (`messages`) names the Python module, the second (`greeting`) names a function or other callable object within that module, and any parameters after the `?` become keyword arguments passed to the named function.
- After the function runs, its return value automatically appears in a new page in your web browser. Zope does all the work of formatting the result as a valid HTTP response.

In other words, URLs in Zope become remote function calls, not just script invocations. The functions (and methods) called by accessing URLs are coded in Python, and may live at arbitrary places on the Net. It's as if the Internet itself becomes a Python namespace, with one module directory per server.

Zope is a server-side technology based on objects, not text streams; the main advantage of this scheme is that the details of CGI input and output are handled by Zope, while programmers focus on writing domain objects, not on text generation. When our function is accessed with a URL, Zope automatically finds the referenced object, translates incoming parameters to function call arguments, runs the function, and uses its return value to generate an HTTP response. In general, a URL like:

```
http://servername/dirpath/module/object1/object2/method?arg1=val1&arg2=val2
```

is mapped by the Zope ORB running on `servername` into a call to a Python object in a Python module file installed in `dirpath`, taking the form:

```
module.object1.object2.method(arg1=val1, arg2=val2)
```

The return value is formatted into an HTML response page sent back to the client requestor (typically a browser). By using longer paths, programs can publish complete hierarchies of objects; Zope simply uses Python's generic object-access protocols to fetch objects along the path.

As usual, a URL like those listed here can appear as the text of a hyperlink, typed manually into a web browser, or used in an HTTP request generated by a program (e.g., using Python's `urllib` module in a client-side script). Parameters are listed at the end of these URLs directly, but if you post information to this URL with a form instead, it works the same way:

```
<form action="http://www.myserver.net/cgi-bin/messages/greeting" method=POST>
  Size: <input type=text name=size>
  Topic: <input type=text name=topic value=zope>
  <input type=submit>
</form>
```

Here, the `action` tag references our function's URL again; when the user fills out this form

and presses its submit button, inputs from the form sent by the browser magically show up as arguments to the function again. These inputs are typed by the user, not hardcoded at the end of a URL, but our published function doesn't need to care. In fact, Zope recognizes a variety of parameter sources and translates them all into Python function or method arguments: form inputs, parameters at the end of URLs, HTTP headers and cookies, CGI environment variables, and more.

This just scratches the surface of what published objects can do, though. For instance, published functions and methods can use the Zope object database to save state permanently, and Zope provides many more advanced tools such as debugging support, precoded HTTP servers for use with the ORB, and finer-grained control over responses to URL requestors.

For all things Zope, visit <http://www.zope.org>. There, you'll find up-to-date releases, as well as documentation ranging from tutorials to references to full-blown Zope example sites. Also see this book's CD (view CD-ROM content online at <http://examples.oreilly.com/python2>) for a copy of the Zope distribution, current as of the time we went to press.

.	Python creator Guido van Rossum and his Pythonlabs team of core Python developers have moved from BeOpen to Digital Creations, home of the Zope framework introduced here. Although Python itself remains an open source system, Guido's presence at Digital Creations is seen as a strategic move that will foster future growth of both Zope and Python.
---	--

15.3 HTMLgen: Web Pages from Objects

One of the things that makes CGI scripts complex is their inherent dependence on HTML: they must embed and generate legal HTML code to build user interfaces. These tasks might be easier if the syntax of HTML were somehow removed from CGI scripts and handled by an external tool.

HTMLgen is a third-party Python tool designed to fill this need. With it, programs build web pages by constructing trees of Python objects that represent the desired page and "know" how to format themselves as HTML. Once constructed, the program asks the top of the Python object tree to generate HTML for itself, and out comes a complete, legally formatted HTML web page.

Programs that use HTMLgen to generate pages need never deal with the syntax of HTML; instead, they can use the higher-level object model provided by HTMLgen and trust it to do the formatting step. HTMLgen may be used in any context where you need to generate HTML. It is especially suited for HTML generated periodically from static data, but can also be used for HTML creation in CGI scripts (though its use in the CGI context incurs some extra speed costs). For instance, HTMLgen would be ideal if you run a nightly job to generate web pages from database contents. HTMLgen can also be used to generate documents that don't live on the Web at all; the HTML code it produces works just as well when viewed offline.

15.3.1 A Brief HTMLgen Tutorial

We can't investigate HTMLgen in depth here, but let's look at a few simple examples to sample the flavor of the system. HTMLgen is shipped as a collection of Python modules that must be installed on your machine; once it's installed, you simply import objects from the HTMLgen module corresponding to the tag you wish to generate, and make instances:

```
C:\Stuff\HTMLgen\HTMLgen>python
>>> from HTMLgen import *
>>> p = Paragraph("Making pages from objects is easy\n")
>>> p
<HTMLgen.Paragraph instance at 7dbb00>
>>> print p
<P>Making pages from objects is easy
</P>
```

Here, we make a HTMLgen.Paragraph object (a class instance), passing in the text to be formatted. All HTMLgen objects implement `__str__` methods and can emit legal HTML code for themselves. When we print the Paragraph object, it emits an HTML paragraph construct. HTMLgen objects also define `append` methods, which do the right thing for the object type; Paragraphs simply add appended text to the end of the text block:

```
>>> p.append("Special < characters > are & escaped")
>>> print p
<P>Making pages from objects is easy
Special &lt; characters &gt; are &amp; escaped</P>
```

Notice that HTMLgen escaped the special characters (e.g., `<` means `<`) so that they are

legal HTML; you don't need to worry about writing either HTML or escape codes yourself. HTMLgen has one class for each HTML tag; here is the `List` object at work, creating an ordered list:

```
>>> choices = ['python', 'tcl', 'perl']
>>> print List(choices)
<UL>
<LI>python
<LI>tcl
<LI>perl
</UL>
```

In general, HTMLgen is smart about interpreting data structures you pass to it. For instance, embedded sequences are automatically mapped to the HTML code for displaying nested lists:

```
>>> choices = ['tools', ['python', 'c++'], 'food', ['spam', 'eggs']]
>>> l = List(choices)
>>> print l
<UL>
<LI>tools
  <UL>
    <LI>python
    <LI>c++
  </UL>
<LI>food
  <UL>
    <LI>spam
    <LI>eggs
  </UL>
</UL>
```

Hyperlinks are just as easy: simply make and print an `Href` object with the link target and text. (The text argument can be replaced by an image, as we'll see later in [Example 15-3](#).)

```
>>> h = Href('http://www.python.org', 'python')
>>> print h
<A HREF="http://www.python.org">python</A>
```

To generate HTML for complete pages, we create one of the HTML document objects, append its component objects, and print the document object. HTMLgen emits a complete page's code, ready to be viewed in a browser:

```
>>> d = SimpleDocument(title='My doc')
>>> p = Paragraph('Web pages made easy')
>>> d.append(p)
>>> d.append(h)
>>> print d
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<!-- This file generated using Python HTMLgen module. -->
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen 2.2.2">
  <TITLE>My doc</TITLE>
</HEAD>
<BODY>
<P>Web pages made easy</P>

<A HREF="http://www.python.org">python</A>

</BODY> </HTML>
```


There are other kinds of document classes, including a `SeriesDocument` that implements a standard layout for pages in a series. `SimpleDocument` is simple indeed: it's essentially a container for other components, and generates the appropriate wrapper HTML code. HTMLgen also provides classes such as `Table`, `Form`, and so on, one for each kind of HTML construct.

Naturally, you ordinarily use HTMLgen from within a script, so you can capture the generated HTML in a file or send it over an Internet connection in the context of a CGI application (remember, printed text goes to the browser in the CGI script environment). The script in [Example 15-2](#) does roughly what we just did interactively, but saves the printed text in a file.

Example 15-2. PP2E\Internet\Other\htmlgen101.py

```
import sys
from HTMLgen import *

p = Paragraph('Making pages from objects is easy.\n')
p.append('Special < characters > are & escaped')

choices = ['tools', ['python', 'c++'], 'food', ['spam', 'eggs']]
l = List(choices)

s = SimpleDocument(title="HTMLgen 101")
s.append(Heading(1, 'Basic tags'))
s.append(p)
s.append(l)
s.append(HR( ))
s.append(Href('http://www.python.org', 'Python home page'))

if len(sys.argv) == 1:
    print s                # send html to sys.stdout or real file
else:
    open(sys.argv[1], 'w').write(str(s))
```

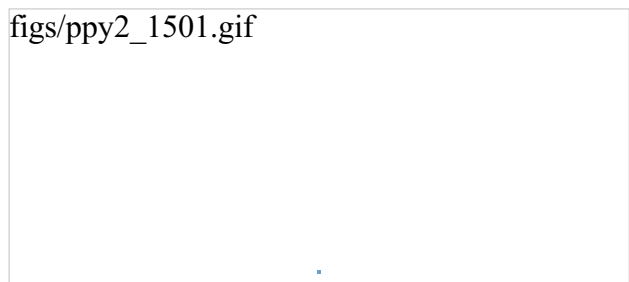
This script also uses the `HR` object to format a horizontal line, and `Heading` to insert a header line. It either prints HTML to the standard output stream (if no arguments are listed) or writes HTML to an explicitly named file; the `str` built-in function invokes object `__str__` methods just as `print` does. Run this script from the system command line to make a file, using one of the following:

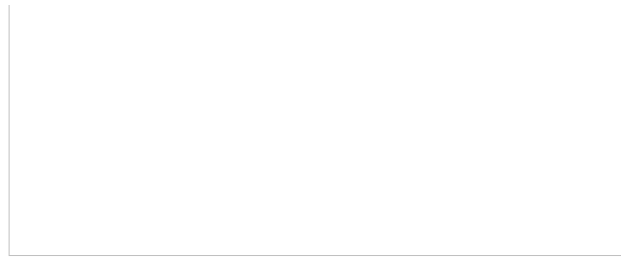
```
C:\...\PP2E\Internet\Other>python htmlgen101.py > htmlgen101.html
C:\...\PP2E\Internet\Other>python htmlgen101.py htmlgen101.html
```

Either way, the script's output is a legal HTML page file, which you can view in your favorite browser by typing the output filename in the address field or clicking on the file in your file explorer. Either way, it will look a lot like [Figure 15-1](#).

Figure 15-1. Viewing htmlgen101.py output in a browser

figs/ppy2_1501.gif





See file *htmlgen101.html* in the examples distribution if you wish to inspect the HTML generated to describe this page directly (it looks much like the prior document's output). [Example 15-3](#) shows another script that does something less hardcoded: it constructs a web page to display its own source code.

Example 15-3. PP2E\Internet\Other\htmlgen101-b.py

```
import sys
from HTMLgen import *
d = SimpleDocument(title="HTMLgen 101 B")

# show this script
text = open('htmlgen101-b.py', 'r').read( )
d.append(Heading(1, 'Source code'))
d.append(Paragraph( PRE(text) ))

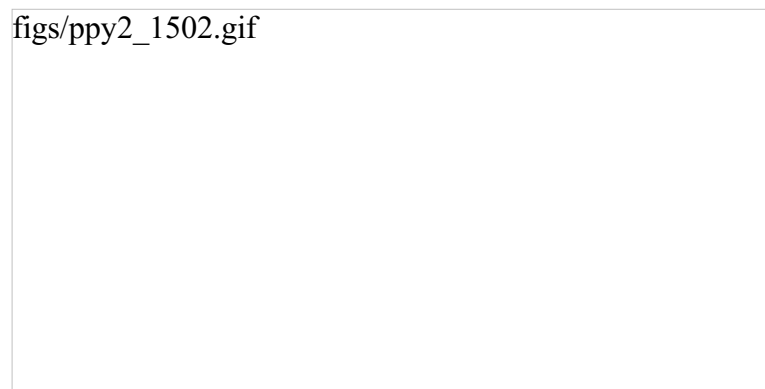
# add gif and links
site = 'http://www.python.org'
gif = 'PythonPoweredSmall.gif'
image = Image(gif, alt='picture', align='left', hspace=10, border=0)

d.append(HR( ))
d.append(Href(site, image))
d.append(Href(site, 'Python home page'))

if len(sys.argv) == 1:
    print d
else:
    open(sys.argv[1], 'w').write(str(d))
```

We use the `PRE` object here to specify preformatted text, and the `Image` object to generate code to display a GIF file on the generated page. Notice that HTML tag options such as `alt` and `align` are specified as keyword arguments when making `HTMLgen` objects. Running this script and pointing a browser at its output yields the page shown in [Figure 15-2](#); the image at the bottom is also a hyperlink, because it was embedded inside an `Href` object.

Figure 15-2. Viewing htmlgen101-b.py output in a browser





And that (along with a few nice advanced features) is all there is to using HTMLgen. Once you become familiar with it, you can construct web pages by writing Python code, without ever needing to manually type HTML tags again. Of course, you still must write code with HTMLgen instead of using a drag-and-drop page layout tool, but that code is incredibly simple and supports the addition of more complex programming logic where needed to construct pages dynamically.

In fact, now that you're familiar with HTMLgen, you'll see that many of the HTML files shown earlier in this book could have been simplified by recoding them to use HTMLgen instead of direct HTML code. The earlier CGI scripts could have used HTMLgen as well, albeit with additional speed overheads -- printing text directly is faster than generating it from object trees, though perhaps not significantly so (CGI scripts are generally bound to network speeds, not CPU speed).

HTMLgen is open source software, but it is not a standard part of Python and must therefore be installed separately. You can find a copy of HTMLgen on this book's CD (see <http://examples.oreilly.com/python2>), but the Python web site should have its current location and version. Once installed, simply add the HTMLgen path to your PYTHONPATH variable setting to gain access to its modules. For more documentation about HTMLgen, see the package itself: its *html* subdirectory includes the HTMLgen manual in HTML format.

15.4 JPython (Jython): Python for Java

JPython (recently renamed "Jython") is an entirely distinct implementation of the Python programming language that allows programmers to use Python as a scripting component in Java-based applications. In short, JPython makes Python code look like Java, and consequently offers variety of technology options inherited from the Java world. With JPython, Python code may be run as client-side applets in web browsers, as server-side scripts, and in a variety of other roles. JPython is distinct from other systems mentioned in this section in terms of its scope: while it is based on the core Python language we've seen in this book, it actually replaces the underlying implementation of that language rather than augmenting it.^[2]

[2] At this writing, JPython is the second implementation of the Python language. By contrast, the standard, original implementation of Python is sometimes now referred to as "CPython," because it is implemented in ANSI C. Among other things, the JPython implementation is driving a clearer definition of the Python language itself, independent of a particular implementation's effects. A new Python implementation for Microsoft's C#.NET environment is also on the way (see later in this chapter) and may further drive a definition of what it means to be Python.

This section briefly explores JPython and highlights some of the reasons you may or may not want to use it instead of the standard Python implementation. Although JPython is primarily of interest to programmers writing Java-based applications, it underscores integration possibilities and language definition issues that merit the attention of all Python users. Because JPython is Java-centric, you need to know something about Java development to make the most sense of JPython and this book doesn't pretend to teach that in the next few pages. For more details, interested readers should consult other materials, including JPython documentation at <http://www.jython.org>.

The JPython port is now called "Jython." Although you are likely to still see it called by its original JPython name on the Net (and in this book) for some time, the new Jython title will become more common as time goes by.

15.4.1 A Quick Introduction to JPython

Functionally speaking, JPython is a collection of Java classes that run Python code. It consists of a Python compiler, written in Java, that translates Python scripts to Java bytecodes so they can be executed by a Java virtual machine -- the runtime component that executes Java programs and is used by major web browsers. Moreover, JPython automatically exposes all Java class libraries for use in Python scripts. In a nutshell, here's what comes with the JPython system:

Python-to-Java-bytecode compiler

JPython always compiles Python source code into Java bytecode and passes it to a Java virtual machine (JVM) runtime engine to be executed. A command-line compiler program `jpythonc`, is also able to translate Python source code files into Java `.class` and `.jar` files,

which can then be used as Java applets, beans, servlets, and so on. To the JVM, Python code run through JPython looks the same as Java code. Besides making Python code work on a JVM, JPython code also inherits all aspects of the Java runtime system, including Java's garbage collection and security models. `jpythonc` also imposes Java source file class rules:

Access to Java class libraries (extending)

JPython uses Java's reflection API (runtime type information) to expose all available Java class libraries to Python scripts. That is, Python programs written for the JPython system can call out to any resident Java class automatically simply by importing it. The Python-to-Java interface is completely automatic and remarkably seamless -- Java class libraries appear as though they are coded in Python. Import statements in JPython scripts may refer to either JPython modules or Java class libraries. For instance, when a JPython script imports `java.awt`, it gains access to all the tools available in the `awt` library. JPython internally creates a "dummy" Python module object to serve as an interface to `awt` at import time. This dummy module consists of hooks for dispatching calls from JPython code to Java class methods and automatically converting datatypes between Java and Python representations as needed. To JPython scripts, Java class libraries look and feel exactly like normal Python modules (albeit with interfaces defined by the Java world).

Unified object model

JPython objects are actually Java objects internally. In fact, JPython implements Python types as instances of a Java `PyObject` class. By contrast, C Python classes and types are still distinct in the current release. For instance, in JPython, the number 123 is an instance of the `PyInteger` Java class, and you can specify things like `[].__class__` since all objects are class instances. That makes data mapping between languages simple: Java can process Python objects automatically, because they are Java objects. JPython automatically converts types between languages according to a standard type map as needed to call out to Java libraries, and selects among overloaded Java method signatures.

API for running Python from Java (embedding)

JPython also provides interfaces that allow Java programs to execute JPython code. As for embedding in C and C++, this allows Java applications to be customized by bits of dynamically written JPython code. For instance, JPython ships with a Java `PythonInterpreter` class, which allows Java programs to create objects that function as Python namespaces for running Python code. Each `PythonInterpreter` object is roughly a Python module, with methods such as `exec` (a string of Python code), `execfile` (a Python filename), and `get` and `set` methods for assigning Python global variables. Because Python objects are really instances of a Java `PyObject` class, an enclosing Java layer can access and process data created by Python code naturally.

Interactive Python command line

Like the standard Python implementation, JPython comes with an interactive command line that runs code immediately after it is typed. JPython's `jpython` program is equivalent to the `python` executable we've been using in this book; without arguments, it starts an interactive session. Among other things, this allows JPython programmers to import and test class components actually written in Java. This ability alone is compelling enough to interest many Java programmers.

Interface automations

Java libraries are somewhat easier to use in JPython code than in Java. That's because JPython automates some of the coding steps Java implies. For instance, callback handlers for Java GUI libraries may be simple Python functions, even though Java coders need to provide methods in fully specified classes (Java does not have first-class function objects). JPython also makes Java class data members accessible as both Python attribute names (*object.name*) and object constructor keyword arguments (*name=value*); such Python syntax is translated into calls to `getName` and `setName` accessor methods in Java classes. We'll see these automation tricks in action in the following examples. You don't have to use any of these (and they may confuse Java programmers at first glance), but they further simplify coding in JPython, and give Java class libraries a more Python-like flavor.

The net effect of all this is that JPython allows us to write Python programs that can run on any Java-aware machine -- in particular, in the context of most web browsers. More importantly, because Python programs are translated into Java bytecodes, JPython provides an incredibly seamless and natural integration between the two languages. Both walk and talk in terms of the Java model, so calls across language boundaries are trivial. With JPython's approach, it's even possible to subclass a Java class in Python and vice versa.

So why go to all this trouble to mix Python into Java environments? The most obvious answer is that JPython makes Java components easier to use: JPython scripts are typically a fraction of the size of their Java equivalents, and much less complex. More generally, the answer is really the same as it is for C and C++ environments: Python, as an easy-to-use, object-oriented scripting language, naturally complements the Java programming language.

By now, it is clear to most people that Java is too complex to serve as a scripting or rapid-development tool. But this is exactly where Python excels; by adding Python to the mix with JPython, we add a scripting component to Java systems, exactly as we do when integrating Python with C or C++. For instance, we can use JPython to quickly prototype Java systems, test Java classes interactively, and open up Java systems for end-user customization. In general, adding Python to Java development can significantly boost programmer productivity, just as it does for C and C++ systems.

JPython Versus the Python C API

Functionally, JPython is primarily an integration system: it allows us to mix Python with Java components. We also study ways to integrate Python with C and C++ components in the next part of this book. It's worth noting that we need different techniques to integrate Python with Java (such as the JPython compiler), because Java is a somewhat closed system: it prefers an all-Java mix. The C and C++ integration tools are generally less restrictive in terms of language assumptions, and any C-compatible language components will do. Java's strictness is partly due to its security goals, but the net effect is to foster integration techniques that are specific to Java alone.

On the other hand, because Java exposes runtime type information through its reflection API, JPython can largely automate the conversions and dispatching needed to access Java components from Python scripts; Python code simply imports and calls Java components. When mixing Python with C or C++, we must provide a "glue" code layer that integrates the two languages explicitly. Some of this can be automated (with the SWIG system we'll meet later in this text). No glue code is required in JPython, however, because JPython's (and Java's) developers have done all the linkage work already, in a generic fashion. It is also possible to mix in C/C++ components with Java via its native call interface (JNI), but this can be cumbersome and may cancel out Java's reported portability and security benefits.

15.4.2 A Simple JPython Example

Once a Python program is compiled with JPython, it is all Java: the program is translated to Java bytecodes, it uses Java classes to do its work, and there is no Python left except for the original source code. Because the compiler tool itself is also written in Java, JPython is sometimes called "100% pure Java." That label may be more profound to marketers than programmers, though, because JPython scripts are still written using standard Python syntax. For instance, [Example 15](#) is a legal JPython program, derived from an example originally written by Guido van Rossum.

Example 15-4. PP2E\Internet\Other\jpython.py

```
#####
# implement a simple calculator in JPython;
# evaluation runs a full expression all at
# once using the Python eval( ) built-in--
# JPython's compiler is present at run-time
#####

from java import awt                # get access to Java class libraries
from pwt import swing              # they look like Python modules here

labels = ['0', '1', '2', '+',      # labels for calculator buttons
          '3', '4', '5', '-',      # will be used for a 4x4 grid
          '6', '7', '8', '*',
          '9', '.', '=', '/' ]

keys = swing.JPanel(awt.GridLayout(4, 4)) # do Java class library magic
display = swing.JTextField( )             # Python data auto-mapped to Ja

def push(event):                         # callback for regular keys
    display.replaceSelection(event.actionCommand)

def enter(event):                         # callback for the '=' key
    display.text = str(eval(display.text)) # use Python eval( ) to run expr
    display.selectAll( )

for label in labels:                     # build up button widget grid
    key = swing.JButton(label)           # on press, invoke Python funcs
    if label == '=':
        key.actionPerformed = enter
    else:
        key.actionPerformed = push
    keys.add(key)

panel = swing.JPanel(awt.BorderLayout( )) # make a swing panel
panel.add("North", display)              # text plus key grid in middle
```

```
panel.add("Center", keys)
swing.test(panel) # start in a GUI viewer
```

The first thing you should notice is that this is genuine Python code -- JPython scripts use the same core language that we've been using all along in this book. That's good news, both because Python is such an easy language to use and because you don't need to learn a new, proprietary scripting language to use JPython. It also means that all of Python's high-level language syntax and tools are available. For example, in this script, the Python `eval` built-in function is used to parse and evaluate constructed expressions all at once, saving us from having to write an expression evaluator from scratch.

15.4.3 Interface Automation Tricks

The previous calculator example also illustrates two interface automations performed by JPython: function callback and attribute mappings. Java programmers may have already noticed that this example doesn't use classes. Like standard Python and unlike Java, JPython supports but does not impose OOP. Simple Python functions work fine as callback handlers. In [Example 15-4](#), assigning `key.actionPerformed` to a Python function object has the same effect as registering an instance of a class that defines a callback handler method:

```
def push(event):
    ...
key = swing.JButton(label)
key.actionPerformed = push
```

This is noticeably simpler than the more Java-like:

```
class handler(awt.event.ActionListener):
    def actionPerformed(self, event):
        ...
key = swing.JButton(label)
key.addActionListener(handler( ))
```

JPython automatically maps Python functions to the Java class method callback model. Java programmers may now be wondering why we can assign to something named `key.actionPerformed` in the first place. JPython's second magic feat is to make Java data members look like simple object attributes in Python code. In abstract terms, JPython code of the form:

```
X = Object(argument)
X.property = value + X.property
```

is equivalent to the more traditional and complex Java style:

```
X = Object(argument)
X.setProperty(value + X.getProperty( ))
```

That is, JPython automatically maps attribute assignments and references to Java accessor method calls by inspecting Java class signatures (and possibly Java BeanInfo files if used). Moreover, properties can be assigned with keyword arguments in object constructor calls, such that:

```
X = Object(argument, property=value)
```

is equivalent to both this more traditional form:


```
X = Object(argument)
X.setProperty(value)
```

as well as the following, which relies on attribute name mapping:

```
X = Object(argument)
X.property = value
```

We can combine both callback and property automation for an even simpler version of the callback code snippet:

```
def push(event):
    ...
key = swing.JButton(label, actionPerformed=push)
```

You don't need to use these automation tricks, but again, they make JPython scripts simpler, and that's most of the point behind mixing Python with Java.

15.4.4 Writing Java Applets in JPython

I would be remiss if I didn't include a brief example of JPython code that more directly masquerades as a Java applet: code that lives on a server machine but is downloaded to and run on the client machine when its Internet address is referenced. Most of the magic behind this is subclassing the appropriate Java class in a JPython script, demonstrated in [Example 15-5](#).

Example 15-5. PP2E\Internet\Other\jpython-applet.py

```
#####
# a simple java applet coded in Python
#####

from java.applet import Applet                                # get java superclass

class Hello(Applet):
    def paint(self, gc):                                       # on paint callback
        gc.drawString("Hello applet world", 20, 30)          # draw text message

if __name__ == '__main__':                                    # if run standalone
    import pawt                                                # get java awt lib
    pawt.test(Hello( ))                                       # run under awt loop
```

The Python class in this code inherits all the necessary applet protocol from the standard Java `Applet` superclass, so there is not much new to see here. Under JPython, Python classes can always subclass Java classes, because Python objects really are Java objects when compiled and run. The Python-coded `paint` method in this script will be automatically run from the Java AWT event loop as needed; it simply uses the passed-in `gc` user-interface handle object to draw a text message.

If we use JPython's `jpythonc` command-line tool to compile this into a Java `.class` file and properly store that file on a web server, it can then be used exactly like applets written in Java. Because most web browsers include a JVM, this means that such Python scripts may be used as client-side programs that create sophisticated user-interface devices within the browser, and so on.

15.4.5 JPython Trade-offs

Depending on your background, though, the somewhat less good news about JPython is that even though the calculator and applet scripts discussed here are straight Python code, the libraries they use are different than what we've seen so far. In fact, the library calls employed are radically different. The calculator, for example, relies primarily on imported Java class libraries, not standard Python libraries. You really need to understand Java's `awt` and `swing` libraries to make sense of its code, and this library skew between language implementations becomes more acute as programs grow larger. The applet example is even more Java-bound: it depends both on Java user interface libraries and Java applet protocols.

If you are already familiar with Java libraries, this isn't an issue at all, of course. But because most of the work performed by realistic programs is done by using libraries, the fact that most JPython code relies on very different libraries makes compatibility with standard Python less potent than may seem at first glance. To put that more strongly, apart from very trivial core language examples, many JPython programs won't run on the standard Python interpreter, and many standard Python programs won't work under JPython.

Generally, JPython presents a number of trade-offs, partly due to its relative immaturity as of this writing. I want to point out up front that JPython is indeed an excellent Java scripting tool -- arguably the best one available, and most of its trade-offs are probably of little or no concern to Java developers. For instance, if you are coming to JPython from the Java world, the fact that Java libraries are at the heart of JPython scripts may be more asset than downside. But if you are presented with a choice between the standard and Java-based Python language implementations, some of JPython's implications are worth knowing about:

JPython is not yet fully compatible with the standard Python language

At this writing, JPython is not yet totally compatible with the standard Python language, as defined by the original C implementation. In subtle ways, the core Python language itself works differently in JPython. For example, until very recently, assigning file-like objects to the standard input `sys.stdin` failed, and exceptions were still strings, not class objects. The list of incompatibilities (viewable at <http://www.jython.org>) will likely shrink over time, but will probably never go away completely. Moreover, new language features are likely to show up later in JPython than in the standard C-based implementation.

JPython requires programmers to learn Java development too

Language syntax is only one aspect of programming. The library skew mentioned previously is just one example of JPython's dependence on the Java system. Not only do you need to learn Java libraries to get real work done in JPython, but you also must come grips with the Java programming environment in general. Many standard Python libraries have been ported to JPython, and others are being adopted regularly. But major Python tools such as Tkinter GUIs may show up late or never in JPython (and instead are replaced with Java tools).^[3] In addition, many core Python library features cannot be supported in JPython, because they would violate Java's security constraints. For example, the `os.system` call for running shell commands may never become available in JPython.

[3] But see the note at the end of the later section on Grail; an early port of Tkinter for JPython is already available on the Net.

JPython applies only where a JVM is installed or shipped

You need the Java runtime to run JPython code. This may sound like a non-issue given the pervasiveness of the Internet, but I have very recently worked in more than one company for which delivering applications to be run on JVMs was not an option. Simply put, there was no JVM to be found at the customer's site. In such scenarios, JPython is either not an option, or will require you to ship a JVM with your application just to run your compiled JPython code. Shipping the standard Python system with your products is completely free shipping a JVM may require licensing and fees. This may become less of a concern as robust open source JVMs appear. But if you wish to use JPython today and can't be sure that your clients will be able to run your systems in Java-aware browsers (or other JVM components), you should consider the potential costs of shipping a Java runtime system with your products.^[4]

[4] Be sure you can get a JVM to develop those products too! Installing JPython on Windows 98 while writing this book proved painful, not because of JPython, but because I also had to come to grips with Java commands to run during installation, and track down and install a JVM other than the one provided by Microsoft. Depending on your platform, you may be faced with JPython's Java-dependence even before you type your first line of code.

JPython doesn't support Python extension modules written in C or C++

At present, no C or C++ extension modules written to work with the C Python implementation will work with JPython. This is a major impediment to deploying JPython outside the scope of applications run in a browser. To date, the half-million-strong Python user community has developed thousands of extensions written for C Python, and these constitute much of the substance of the Python development world. JPython's current alternative is to instead expose Java class libraries and ask programmers to write new extensions in Java. But this dismisses a vast library of prior and future Python art. In principle, C extensions could be supported by Java's native call interface, but it's complex, has not been done, and can negate Java portability and security.

JPython is noticeably slower than C Python

Today, Python code generally runs slower under the JPython implementation. How much slower depends on what you test, which JVM you use to run your test, whether a just-in-time (JIT) compiler is available, and which tester you cite. Posted benchmarks have run the gamut from 1.7 times slower than C Python, to 10 times slower, and up to 100 times slower. Regardless of the exact number, the extra layer of logic JPython requires to map Python to the Java execution model adds speed overheads to an already slow JVM and makes it unlikely that JPython will ever be as fast as the C Python implementation. Given that C Python is already slower than compiled languages like C, the additional slowness of JPython makes it less useful outside the realm of Java scripting. Furthermore, the Swing GUI library used by JPython scripts is powerful, but generally considered to be the slowest and largest of all Python GUI options. Given that Python's Tkinter library is a portable and standard GUI solution, Java's proprietary user-interface tools by themselves are probably not reason enough to use the JPython implementation.

JPython is less robust than C Python

At this writing, JPython is substantially more buggy than the standard C implementation of the language. This is certainly due to its younger age and smaller user base and varies from JVM to JVM, but you are more likely to hit snags in JPython. In contrast, C Python has been amazingly bug-free since its introduction in 1990.

JPython may be less portable than C Python

It's also worth noting that as of this writing, the core Python language is far more portable than Java (despite marketing statements to the contrary). Because of that, deploying standard Python code with the Java-based JPython implementation may actually lessen its portability. Naturally, this depends on the set of extensions you use, but standard Python runs today on everything from handheld PDAs and PCs to Cray supercomputers and IBM mainframes.

Some incompatibilities between JPython and standard Python can be very subtle. For instance, JPython inherits all of the Java runtime engine's behavior, including Java security constraints and garbage collection. Java garbage collection is not based on standard Python's reference count scheme, and therefore can automatically collect cyclic objects.^[5] It also means that some common Python programming idioms won't work. For example, it's typical in Python to code file-processing loops in this form:

[5] But Python 2.0's garbage collector can now collect cyclic objects too. See the 2.0 release notes and [Appendix A](#).

```
for filename in bigfilenamelist:
    text = open(filename).read( )
    dostuffwith(text)
```

That works because files are automatically closed when garbage-collected in standard Python, so we can be sure that the file object returned by the `open` call will be immediately garbage collected (it's a temporary, so there are no more references as soon as we call `read`). It won't work in JPython, though, because we can't be sure when the temporary file object will be reclaimed. To avoid running out of file descriptors, we usually need to code this differently for JPython:

```
for filename in bigfilenamelist:
    file = open(filename)
    text = file.read( )
    dostuffwith(text)
    file.close( )
```

You may face a similar implementation mismatch if you assume that output files are immediately closed: `open(name, 'w').write(bytes)` collects and closes the temporary file object and hence flushes the bytes out to the file under the standard C implementation of Python only, while JPython instead collects the file object at some arbitrary time in the future. In addition to such file closing concerns, Python `__del__` class destructors are never called in JPython, due to complications associated with object termination.

15.4.6 Picking Your Python

Because of concerns such as those just mentioned, the JPython implementation of the Python language is probably best used only in contexts where Java integration or web browser interoperability are crucial design concerns. You should always be the judge, of course, but the standard C implementation seems better suited to most other Python applications. Still, that leaves a very substantial domain to JPython -- almost all Java systems and programmers can benefit from adding JPython to their tool sets.

JPython allows programmers to write programs that use Java class libraries in a fraction of the code and complexity required by Java-coded equivalents. Hence, JPython excels as an extension language for Java-based systems, especially those that will run in the context of web browsers. Because Java is a standard component of most web browsers, JPython scripts will often run automatically without extra install steps on client machines. Furthermore, even Java-coded applications that have nothing to do with the Web can benefit from JPython's ease of use; its seamless integration with Java class libraries makes JPython simply the best Java scripting and testing tool available today.

For most other applications, though, the standard Python implementation, possibly integrated with C and C++ components, is probably a better design choice. The resulting system will likely run faster, cost less to ship, have access to all Python extension modules, be more robust and portable and be more easily maintained by people familiar with standard Python.

On the other hand, I want to point out again that the trade-offs listed here are mostly written from the Python perspective; if you are a Java developer looking for a scripting tool for Java-based systems, many of these detriments may be of minor concern. And to be fair, some of JPython's problems may be addressed in future releases; for instance, its speed will probably improve over time. Yet even as it exists today, JPython clearly makes an ideal extension-language solution for Java-based applications, and offers a much more complete Java scripting solution than those currently available for other scripting languages.^[6]

[6] Other scripting languages have addressed Java integration by reimplementing a Java virtual machine in the underlying scripting language or by integrating their original C implementations with Java using the Java native call interface. Neither approach is anywhere near as seamless and powerful as generating real Java bytecode.

For more details, see the JPython package included on this book's CD (see <http://examples.oreilly.com/python2>), and consult the JPython home page, currently maintained at <http://www.jython.org>. At least one rumor has leaked concerning an upcoming JPython book as well, so check <http://www.python.org> for developments on this front. See also the sidebar later in this chapter about the new Python implementation for the C#/.NET environment on Windows. It seems likely that there will be three Pythons to choose from very soon (not just two), and perhaps more in the future. All will likely implement the same core Python language we've used in this text, but may emphasize alternative integration schemes, application domains, development environments, and so on.

15.5 Grail: A Python-Based Web Browser

I briefly mentioned the Grail browser near the start of [Chapter 10](#). Many of Python's Internet tools date back to and reuse the work that went into Grail, a full-blown Internet web browser that:

- Is written entirely in Python
- Uses the Tkinter GUI API to implement its user interface and render pages
- Downloads and runs Python/Tkinter scripts as client-side applets

As mentioned earlier, Grail was something of a proof-of-concept for using Python to code large-scale Internet applications. It implements all the usual Internet protocols and works much like common browsers such as Netscape and Internet Explorer. Grail pages are implemented with the Tk text widgets that we met in the GUI part of this book.

More interestingly, the Grail browser allows applets to be written in Python. Grail applets are simply bits of Python code that live on a server but are run on a client. If an HTML document references a Python class and file that live on a server machine, Grail automatically downloads the Python code over a socket and runs it on the client machine, passing it information about the browser's user interface. The downloaded Python code may use the passed-in browser context information to customize the user interface, add new kinds of widgets to it, and perform arbitrary client-side processing on the local machine. Roughly speaking, Python applets in Grail serve the same purposes as Java applets in common Internet browsers: they perform client-side tasks that are too complex or slow to implement with other technologies such as server-side CGI scripts and generated HTML.

15.5.1 A Simple Grail Applet Example

Writing Grail applets is remarkably straightforward. In fact, applets are really just Python/Tkinter programs; with a few exceptions, they don't need to "know" about Grail at all. Let's look at a short example; the code in [Example 15-6](#) simply adds a button to the browser, which changes its appearance each time it's pressed (its bitmap is reconfigured in the button callback handler).

There are two components to this page definition: an HTML file and the Python applet code it references. As usual, the *grail.html* HTML file that follows describes how to format the web page when the HTML's URL address is selected in a browser. But here, the `APP` tag also specifies a Python applet (class) to be run by the browser. By default, the Python module is assumed to have the same name as the class and must be stored in the same location (URL directory) as the HTML file that references it. Additional `APP` tag options can override the applet's default location.

Example 15-6. PP2E\Internet\Other\grail.html

```
<HEAD>
<TITLE>Grail Applet Test Page</TITLE>
</HEAD>
<BODY>
<H1>Test an Applet Here!</H1>
Click this button!
<APP CLASS=Question>
</BODY>
```

The applet file referenced by the HTML is a Python script that adds widgets to the Tkinter-based Grail browser. Applets are simply classes in Python modules. When the `APP` tag is encountered in the HTML, the Grail browser downloads the `Question.py` source code module (Example 15-7) and makes an instance of its `Question` class, passing in the browser widget as the master (parent). The master is the hook that lets applets attach new widgets to the browser itself; applets extend the GUI constructed by the HTML in this way.

Example 15-7. PP2E\Internet\Other\Question.py

```
# Python applet file: Question.py
# in the same location (URL) as the html file
# that references it; adds widgets to browser;

from Tkinter import *

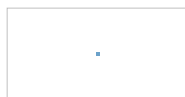
class Question:
    def __init__(self, parent):
        self.button = Button(parent,
                              bitmap='question',
                              command=self.action)
        self.button.pack( )

    def action(self):
        if self.button['bitmap'] == 'question':
            self.button.config(bitmap='questhead')
        else:
            self.button.config(bitmap='question')

if __name__ == '__main__':
    root = Tk( )
    button = Question(root)
    root.mainloop( )
```

Notice that nothing in this class is Grail- or Internet-specific; in fact, it can be run (and tested) standalone as a Python/Tkinter program. Figure 15-3 is what it looks like if run standalone on Windows (with a `Tk` application root object as the master); when run by Grail (with the browser/page object as the master), the button appears as part of the web page instead. Either way, its bitmap changes on each press.

Figure 15-3. Running a Grail applet standalone



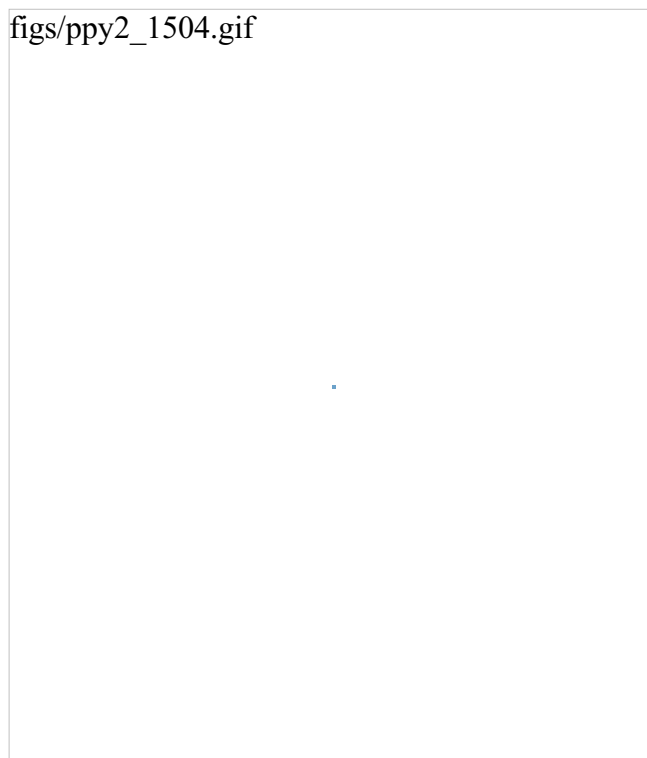
In effect, Grail applets are simply Python modules that are linked into HTML pages by using the `APP` tag. The Grail browser downloads the source code identified by an `APP` tag and runs it

locally on the client during the process of creating the new page. New widgets added to the page (like the button here) may run Python callbacks on the client later, when activated by the user.

Applets interact with the user by creating one or more arbitrary Tk widgets. Of course, the previous example is artificial; but notice that the button's callback handler could do anything we can program in Python: updating persistent information, popping up new user interaction dialogs, calling C extensions, etc. However, by working in concert with Python's restricted execution mode (discussed later) applets can be prevented from performing potentially unsafe operations, like opening local files and talking over sockets.

[Figure 15-4](#) shows a screen shot of Grail in action, hinting at what's possible with Python code downloaded to and run on a client. It shows the animated "game of life" demo; everything you see here is implemented using Python and the Tkinter GUI interface. To run the demo, you need to install Python with the Tk extension and download the Grail browser to run locally on your machine or copy it off the CD. Then point your browser to a URL where any Grail demo lives.

Figure 15-4. A Grail applet demo



Having said all that, I should add that Grail is no longer formally maintained, and is now used primarily for research purposes (Guido never intended for Grail to put Netscape or Microsoft out of business). You can still get it for free (find it at <http://www.python.org>) and use it for surfing the Web or experimenting with alternative web browser concepts, but it is not the active project it was a few years ago.

If you want to code web browser applets in Python, the more common approach today is to use the JPython system described previously to compile your scripts into Java applet bytecode files, and use Java libraries for your scripts' user-interface portions. Embedding Python code in HTML with the Active Scripting extension described later in this chapter is yet another way to integrate client-side code.

Alas, this advice may change over time too. For instance, if Tkinter is ever ported to JPython, you will be able to build GUIs in applet files with Tkinter, rather than with Java class libraries. In fact, as I wrote this, an early release of a complete Java JNI implementation of the Python built-in `_tkinter` module (which allows JPython scripts to import and use the Tkinter module in the standard Python library) was available on the Net at <http://jtkinter.sourceforge.net>. Whether this makes Tkinter a viable GUI option under JPython or not, all current approaches are subject to change. Grail, for instance, was a much more prominent tool when the first edition of this book was written. As ever, be sure to keep in touch with developments in the Python community at <http://www.python.org>; clairvoyance isn't all it's cracked up to be.

15.6 Python Restricted Execution Mode

In prior chapters, I've been careful to point out the dangers of running arbitrary Python code that was shipped across the Internet. There is nothing stopping a malicious user, for instance, from sending a string such as `os.system('rm *')` in a form field where we expect a simple number; running such a code string with the built-in `eval` function or `exec` statement may, by default, really work -- it might just delete all the files in the server or client directory where the calling Python script runs!

Moreover, a truly malicious user can use such hooks to view or download password files, and otherwise access, corrupt, or overload resources on your machine. Alas, where there is a hole, there is probably a hacker. As I've cautioned, if you are expecting a number in a form, you should use simpler string conversion tools such as `int` or `string.atoi` instead of interpreting field contents as Python program syntax with `eval`.

But what if you really want to run Python code transmitted over the Net? For instance, you may wish to put together a web-based training system that allows users to run code from a browser. It is possible to do this safely, but you need to use Python's restricted execution mode tools when you ask Python to run the code. Python's restricted execution mode support is provided in two standard library modules, `rexec` and `bastion`. `rexec` is the primary interface to restricted execution, while `bastion` can be used to restrict and monitor access to object attributes.

On Unix systems, you can also use the standard `resource` module to limit things like CPU time and memory consumption while the code is running. Python's library manual goes into detail on these modules, but let's take a brief look at `rexec` here.

15.6.1 Using rexec

The restricted execution mode implemented by `rexec` is optional -- by default, all Python code runs with full access to everything available in the Python language and library. But when we enable restricted mode, code executes in what is commonly called a "sandbox" model -- access to components on the local machine is limited. Operations that are potentially unsafe are either disallowed or must be approved by code you can customize by subclassing. For example, the script in [Example 15-8](#) runs a string of program code in a restricted environment and customizes the default `rexec` class to restrict file access to a single, specific directory.

Example 15-8. PP2E\Internet\Other\restricted.py

```
#!/usr/bin/python
import rexec, sys
Test = 1
if sys.platform[:3] == 'win':
    SafeDir = r'C:\temp'
else:
    SafeDir = '/tmp/'

def commandLine(prompt='Input (ctrl+z=end) => '):
    input = ''
    while 1:
        try:
```

```
        input = input + raw_input(prompt) + '\n'
    except EOFError:
        break
    print # clear for Windows
    return input

if not Test:
    import cgi # run on the web? - code from form
    form = cgi.FieldStorage( ) # else input interactively to test
    input = form['input'].value
else:
    input = commandLine( )

# subclass to customize default rules: default=write modes disallowed
class Guard(rexec.RExec):
    def r_open(self, name, mode='r', bufsz=-1):
        if name[:len(SafeDir)] != SafeDir:
            raise SystemError, 'files outside %s prohibited' % SafeDir
        else:
            return open(name, mode, bufsz)

# limit system resources (not available on Windows)
if sys.platform[:3] != 'win':
    import resource # at most 5 cpu seconds
    resource.setrlimit(resource.RLIMIT_CPU, (5, 5))

# run code string safely
guard = Guard( )
guard.r_exec(input) # ask guard to check and do opens
```

When we run Python code strings with this script on Windows, safe code works as usual, and we can read and write files that live in the *C:\temp* directory, because our custom `Guard` class's `r_open` method allows files with names beginning with "C:\temp" to proceed. The default `r_open` in `rexec.RExec` allows all files to be read, but all write requests fail. Here, we type code interactively for testing, but it's exactly as if we received this string over the Internet in a CGI script's form field:

```
C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => x = 5
Input (ctrl+z=end) => for i in range(x): print 'hello%d' % i,
Input (ctrl+z=end) => hello0 hello1 hello2 hello3 hello4

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\temp\rexec.txt', 'w').write('Hello rexec\n')
Input (ctrl+z=end) =>

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => print open(r'C:\temp\rexec.txt', 'r').read( )
Input (ctrl+z=end) => Hello rexec
```

On the other hand, attempting to access files outside the allowed directory will fail in our custom class, as will inherently unsafe things such as opening sockets, which `rexec` always makes out of bounds by default:

```
C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\stuff\mark\hack.txt', 'w').write('BadStuff\n')
```

```
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)      # ask guard to check and do opens
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 1, in ?
  File "restricted.py", line 30, in r_open
    raise SystemError, 'files outside %s prohibited' % SafeDir
SystemError: files outside C:\temp prohibited

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => open(r'C:\stuff\mark\secret.py', 'r').read( )
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)      # ask guard to check and do opens
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 1, in ?
  File "restricted.py", line 30, in r_open
    raise SystemError, 'files outside %s prohibited' % SafeDir
SystemError: files outside C:\temp prohibited

C:\...\PP2E\Internet\Other>python restricted.py
Input (ctrl+z=end) => from socket import *
Input (ctrl+z=end) => s = socket(AF_INET, SOCK_STREAM)
Input (ctrl+z=end) => Traceback (innermost last):
  File "restricted.py", line 41, in ?
    guard.r_exec(input)      # ask guard to check and do opens
  ...part ommitted...
  File "C:\Program Files\Python\Lib\ihooks.py", line 324, in load_module
    exec code in m.__dict__
  File "C:\Program Files\Python\Lib\plat-win\socket.py", line 17, in ?
    _realsocketcall = socket
NameError: socket
```

And what of that nasty `rm *` problem? It's possible in normal Python mode like everything else, but not when running in restricted mode. Python makes some potentially dangerous attributes of the `os` module, such as `system` (for running shell commands), disallowed in restricted mode:

```
C:\temp>python
>>> import os
>>> os.system('ls -l rexec.txt')
-rwxrwxrwa  1 0      0          13 May  4 15:45 rexec.txt
0
>>>
C:\temp>python %X%\Part2\internet\other\restricted.py
Input (ctrl+z=end) => import os
Input (ctrl+z=end) => os.system('rm *.*')
Input (ctrl+z=end) => Traceback (innermost last):
  File "C:\PP2ndEd\examples\Part2\internet\other\restricted.py", line 41, in ?
    guard.r_exec(input)      # ask guard to check and do opens
  File "C:\Program Files\Python\Lib\rexec.py", line 253, in r_exec
    exec code in m.__dict__
  File "<string>", line 2, in ?
AttributeError: system
```

Internally, restricted mode works by taking away access to certain APIs (imports are controlled, for example) and changing the `__builtins__` dictionary in the module where the restricted code runs to reference a custom and safe version of the standard `__builtin__` built-in names scope. For instance, the custom version of name `__builtins__.open` references a restricted version of the standard file open function. `rexec` also keeps customizable lists of safe built-in modules, safe `os` and `sys` module attributes, and more. For the rest of this story, see the Python library manual.

Restricted execution mode is not necessarily tied to Internet scripting. It can be useful any time you need to run Python code of possibly dubious origin. For instance, we will use Python's `eval` and `exec` built-ins to evaluate arithmetic expressions and input commands in a calculator program later in the book. Because user input is evaluated as executable code in this context, there is nothing preventing a user from knowingly or unknowingly entering code that can do damage when run (e.g., they might accidentally type Python code that deletes files). However, the risk of running raw code strings becomes more prevalent in applications that run on the Web, since they are inherently open to both use and abuse. Although JPython inherits the underlying Java security model, pure Python systems such as Zope, Grail, and custom CGI scripts can all benefit from restricted execution of strings sent over the Net.

15.7 XML Processing Tools

Python ships with XML parsing support in its standard library and plays host to a vigorous XML special-interest group. XML (eXtended Markup Language) is a tag-based markup language for describing many kinds of structured data. Among other things, it has been adopted in roles such as a standard database and Internet content representation by many companies. As an object-oriented scripting language, Python mixes remarkably well with XML's core notion of structured document interchange, and promises to be a major player in the XML arena.

XML is based upon a tag syntax familiar to web page writers. Python's `xml1ib` library module includes tools for parsing XML. In short, this XML parser is used by defining a subclass of an `XMLParser` Python class, with methods that serve as callbacks to be invoked as various XML structures are detected. Text analysis is largely automated by the library module. This module's source code, file `xml1ib.py` in the Python library, includes self-test code near the bottom that gives additional usage details. Python also ships with a standard HTML parser, `html1ib`, that works on similar principles and is based upon the `sgml1ib` SGML parser module.

Unfortunately, Python's XML support is still evolving, and describing it is well beyond the scope of this book. Rather than going into further details here, I will instead point you to sources for more information:

Standard library

First off, be sure to consult the Python library manual for more on the standard library's XML support tools. At the moment, this includes only the `xml1ib` parser module, but may expand over time.

PyXML SIG package

At this writing, the best place to find Python XML tools and documentation is at the XML SIG (Special Interest Group) web page at <http://www.python.org> (click on the "SIGs" link near the top). This SIG is dedicated to wedding XML technologies with Python, and publishes a free XML tools package distribution called PyXML. That package contains tools not yet part of the standard Python distribution, including XML parsers implemented in both C and Python, a Python implementation of SAX and DOM (the XML Document Object Model), a Python interface to the `Expat` parser, sample code, documentation, and a test suite.

Third-party tools

You can also find free, third-party Python support tools for XML on the Web by following links at the XML SIGs web page. These include a DOM implementation for CORBA environments (4DOM) that currently supports two ORBs (ILU and Fnorb) and much more.

Documentation

As I wrote these words, a book dedicated to XML processing with Python was on the eve of its publication; check the books list at <http://www.python.org> or your favorite book outlet for details.

Given the rapid evolution of XML technology, I wouldn't wager on any of these resources being up to date a few years after this edition's release, so be sure to check Python's web site for more recent developments on this front.

In fact, the XML story changed substantially between the time I wrote this section and when I finally submitted it to O'Reilly. In Python 2.0, some of the tools described here as the PyXML SIG package have made their way into a standard `xml` module package in the Python library. In other words, they ship and install with Python itself; see the Python 2.0 library manual for more details. O'Reilly has a book in the works on this topic called Python and XML.

15.8 Windows Web Scripting Extensions

Although this book doesn't cover the Windows-specific extensions available for Python in detail, a quick look at Internet scripting tools available to Windows programmers is in order here. On Windows, Python can be used as a scripting language for both the Active Scripting and Active Server Pages systems, which provide client- and server-side control of HTML-based applications. More generally, Python programs can also take the role of COM and DCOM clients and servers on Windows.

You should note that at this point in time, everything in this section works only on Microsoft tools, and HTML embedding runs only under Internet Explorer (on the client) and Internet Information Server (on the server). If you are interested in portability, other systems in this chapter may address your needs better (see JPython's client-side applets, PSP's server-side scripting support, and Zope's server-side object publishing model). On the other hand, if portability isn't a concern, the following techniques provide powerful ways to script both sides of a web conversation.

15.8.1 Active Scripting: Client-Side Embedding

Active Scripting is a technology that allows scripting languages to communicate with hosting applications. The hosting application provides an application-specific object model API, which exposes objects and functions for use in the scripting language programs.

In one of its more common roles, Active Scripting provides support that allows scripting language code embedded in HTML pages to communicate with the local web browser through an automatically exposed object model API. Internet Explorer, for instance, utilizes Active Scripting to export things such as global functions and user-interface objects for use in scripts embedded in HTML. With Active Scripting, Python code may be embedded in a web page's HTML between special tags; such code is executed on the client machine and serves the same roles as embedded JavaScript and VBScript.

15.8.1.1 Active Scripting basics

Unfortunately, embedding Python in client-side HTML works only on machines where Python is installed and Internet Explorer is configured to know about the Python language (by installing the `win32all` extension package discussed in a moment). Because of that, this technology doesn't apply to most of the browsers in cyberspace today. On the other hand, if you can configure the machines on which a system is to be delivered, this is a nonissue.

Before we get into a Python example, let's look at the way standard browser installations handle other languages embedded in HTML. By default, IE (Internet Explorer) knows about JavaScript (really, Microsoft's JScript implementation of it) and VBScript (a Visual Basic derivative), so you can embed both of those languages in any delivery scenario. For instance, the HTML file in [Example 15-9](#) embeds JavaScript code, the default IE scripting language on my PC.

Example 15-9. PP2E\Internet\Other\activescript-js.html


```
<HTML>
<BODY>
<H1>Embedded code demo: JavaScript</H1>
<SCRIPT>

// popup 3 alert boxes while this page is
// being constructed on client side by IE;
// JavaScript is the default script language,
// and alert is an automatically exposed name

function message(i) {
    if (i == 2) {
        alert("Finished!");
    }
    else {
        alert("A JavaScript-generated alert => " + i);
    }
}

for (count = 0; count < 3; count += 1) {
    message(count);
}

</SCRIPT>
</BODY></HTML>
```

All the text between the `<SCRIPT>` and `</SCRIPT>` tags in this file is JavaScript code. Don't worry about its syntax -- this book isn't about JavaScript, and we'll see a simpler Python equivalent in a moment. The important thing to know is how this code is used by the browser.

When a browser detects a block of code like this while building up a new page, it locates the appropriate interpreter, tells the interpreter about global object names, and passes the code to the interpreter for execution. The global names become variables in the embedded code and provide links to browser context. For instance, the name `alert` in the code block refers to a global function that creates a message box. Other global names refer to objects that give access to the browser's user interface: window objects, document objects, and so on.

This HTML file can be run on the local machine by clicking on its name in a file explorer. It can also be stored on a remote server and accessed via its URL in a browser. Whichever way you start it, three pop-up alert boxes created by the embedded code appear during page construction. [Figure 15-5](#) shows one under IE.

Figure 15-5. IE running embedded JavaScript code



The VBScript (Visual Basic) version of this example appears in [Example 15-10](#), so you can compare and contrast.^[7] It creates three similar pop-ups when run, but the windows say "VBScript" everywhere. Notice the `Language` option in the `<SCRIPT>` tag here; it must be used to declare the language to IE (in this case, VBScript) unless your embedded scripts speak in its default tongue. In the JavaScript version in [Example 15-9](#), `Language` wasn't needed, because JavaScript was the default language. Other than this declaration, IE doesn't care what language you insert between `<SCRIPT>` and `</SCRIPT>`; in principle, Active Scripting is a language-neutral scripting engine.

[7] Again, feel free to ignore most of this example's syntax. I'm not going to teach either JavaScript or VBScript syntax in this book, nor will I tell you which of the three versions of this example is clearer (though you can probably guess my preference). The first two versions are included partly for comparison by readers with a web development background.

Example 15-10. PP2E\Internet\Other\activescript-vb.html

```
<HTML>
<BODY>
<H1>Embedded code demo: VBScript</H1>
<SCRIPT Language=VBScript>

' do the same but with embedded VBScript;
' the Language option in the SCRIPT tag
' tells IE which interpreter to use

sub message(i)
  if i = 2 then
    alert("Finished!")
  else
    alert("A VBScript-generated alert => " & i)
  end if
end sub

for count = 0 to 2 step 1
  message(count)
next

</SCRIPT>
</BODY></HTML>
```

So how about putting Python code in that page, then? Alas, we need to do a bit more first. Although IE is language-neutral in principle, it does support some languages better than others, at least today. Moreover, other browsers may be more rigid and not support the Active Scripting concept at all.

For example, on my machine and with my installed browser versions (IE 5, Netscape 4), the previous JavaScript example works on both IE and Netscape, but the Visual Basic version works only on IE. That is, IE directly supports VBScript and JavaScript, while Netscape handles only JavaScript. Neither browser as installed can run embedded Python code, even though Python itself is already installed on my machine. There's more to do before we can replace the JavaScript and VBScript code in our HTML pages with Python.

15.8.1.2 Teaching IE about Python

To make the Python version work, you must do more than simply installing Python on your PC. You must also register Python with IE. Luckily, this is mostly an automatic process, thanks to the work of the Python Windows extension developers; you merely need to install a package of Windows extensions.

Here's how this works. The tool to perform the registration is part of the Python Win32 extensions, which are not included in the standard Python system. To make Python known to IE, you must:

1. First install the standard Python distribution your PC (you should have done this already by now -- simply double-click the Python self-installer).
2. Then download and install the `win32all` package separately from <http://www.python.org> (you can also find it at <http://examples.oreilly.com/python2>).^[8]

[8] However, you may not need to download the `win32all` package. The ActivePython Python distribution available from ActiveState (<http://www.activestate.com>), for example, comes with the Windows extensions package.

The `win32all` package includes the `win32COM` extensions for Python, plus the PythonWin IDE (a simple interface for editing and running Python programs, written with the MFC interfaces in `win32all`) and lots of other Windows-specific tools not covered in this book. The relevant point here is that installing `win32all` automatically registers Python for use in IE. If needed, you can also perform this registration manually by running the following Python script file located in the `win32` extensions package: `python\win32comext\axscript\client\pyscript.py`.

Once you've registered Python with IE this way, Python code embedded in HTML works just like our JavaScript and VBScript examples -- IE presets Python global names to expose its object model and passes the embedded code to your Python interpreter for execution. [Example 15-11](#) shows our alerts example again, programmed with embedded Python code.

Example 15-11. PP2E\Internet\Other\activescript-py.html

```
<HTML>
<BODY>
<H1>Embedded code demo: Python</H1>
<SCRIPT Language=Python>

# do the same but with python, if configured;
# embedded python code shows three alert boxes
# as page is loaded; any Python code works here,
# and uses auto-imported global funcs and objects

def message(i):
    if i == 2:
        alert("Finished!")
    else:
        alert("A Python-generated alert => %d" % i)
```

```
for count in range(3): message(count)

</SCRIPT>
</BODY></HTML>
```

[Figure 15-6](#) shows one of the three pop-ups you should see when you open this file in IE after installing the `win32all` package (you can simply click on the file's icon in Windows' file explorer to open it). Note that the first time you access this page, IE may need to load Python, which could induce an apparent delay on slower machines; later accesses generally start up much faster because Python has already been loaded.

Figure 15-6. IE running embedded Python code



Regrettably, this example still works only on IE Version 4 and higher, and not on the Netscape browser on my machine (and reportedly fails on Netscape 6 and Mozilla as well). In other words (at least today and barring new browser releases), not only is Active Scripting a Windows-only technology, but using it as a client-side web browser tool for Python works only on machines where Python is installed and registered to IE, and even then only under IE.

It's also worth knowing that even when you do get Python to work under IE, your Python code runs in restricted mode, with limited access to machine resources (e.g., your code can't open sockets -- see the `rexec` module description earlier in this chapter for details). That's probably what you want when running code downloaded over the Net, and can be changed locally (the implementation is coded in Python), but it limits the utility and scope of your Python scripts embedded in HTML.

The good news is that this does work -- with a simple configuration step, Python code can be embedded in HTML and be made to run under IE, just like JavaScript and VBScript. For many applications, the Windows and IE-only constraint is completely acceptable. Active Scripting is a straightforward way to add client-side Python scripting for web browsers, especially when you can control the target delivery environment. For instance, machines running on an Intranet within a company may have well-known configurations. In such scenarios, Active Scripting lets developers apply all the power of Python in their client-side scripts.

15.8.2 Active Server Pages: Server-Side Embedding

Active Server Pages (ASPs) use a similar model: Python code is embedded in the HTML that defines a web page. But ASP is a server-side technology; embedded Python code runs on the server machine and uses an object-based API to dynamically generate portions of the HTML that is ultimately sent back to the client-side browser. As we saw in the last three chapters, Python server-side CGI scripts embed and generate HTML, and deal with raw inputs and output streams. By contrast, server-side ASP scripts are embedded in HTML and use a higher-level object mode to get their work done.

Just like client-side Active Scripting, ASP requires you to install Python and the Python Windows extensions package. But because ASP runs embedded code on the server, you need to configure Python only on one machine. Like CGI scripts in general, this generally makes Python ASP scripting much more widely applicable, as you don't need Python support on every client. Unlike CGI scripts, however, ASP requires you to run Microsoft's IIS (Internet Information Server) today.

15.8.2.1 A short ASP example

We can't discuss ASP in any real detail here, but here's an example of what an ASP file looks like when Python code is embedded:

```
<HTML><BODY>
<SCRIPT RunAt=Server Language=Python>
#
# code here is run at the server
#
</SCRIPT>
</BODY></HTML>
```

As before, code may be embedded inside `SCRIPT` tag pairs. This time, we tell ASP to run the code at the server with the `RunAt` option; if omitted, the code and its tags are passed through to the client and run by IE (if configured properly). ASP also recognizes code enclosed in `<%` and `%>` delimiters and allows a language to be specified for the entire page. This form is more handy if there are multiple chunks of code in a page, as shown in [Example 15-12](#).

Example 15-12. PP2E\Internet\Other\asp-py.asp

```
<HTML><BODY>
<%@ Language=Python %>

<%
#
# Python code here, using global names Request (input), Response (output), etc.
#
Response.Write("Hello ASP World from URL %s" %
               Request.ServerVariables("PATH_INFO"))
%>
</BODY></HTML>
```

However the code is marked, ASP executes it on the server after passing in a handful of named objects that the code may use to access input, output and server context. For instance, the automatically imported `Request` and `Response` objects give access to input and output context. The code here calls a `Response.Write` method to send text back to the browser on the client (much like a print statement in a simple Python CGI script), as well as `Request.ServerVariables` to access environment variable information. To make this script run

live, you'll need to place it in the proper directory on a server machine running IIS with ASP support.

15.8.3 The COM Connection

At their core, both IE and IIS are based on the COM (Component Object Model) integration system -- they implement their object APIs with standard COM interfaces and look to the rest of the world like any other COM object. From a broader perspective, Python can be used as both a scripting and implementation language for any COM object. Although the COM mechanism used to run Python code embedded within HTML is automated and hidden, it can also be employed explicitly to make Python programs take the role of both COM clients and servers. COM is a general integration technology and is not strictly tied to Internet scripting, but a brief introduction here might help demystify some of the Active Scripting magic behind HTML embedding.

15.8.3.1 A brief introduction to COM

COM is a Microsoft technology for language-neutral component integration. It is sometimes marketed as ActiveX, partially derived from a system called OLE, and is the technological heart of the Active Scripting system we met earlier.^[9] COM also sports a distributed extension known as DCOM that allows communicating objects to be run on remote machines. Implementing DCOM often simply involves running through Windows registry configuration steps to associate servers with machines on which they run.

[9] Roughly, OLE (Object Linking and Embedding) was a precursor to COM, and Active Scripting is just a technology that defines COM interfaces for activities such as passing objects to arbitrary programming language interpreters by name. Active Scripting is not much more than COM itself with a few extensions, but acronym- and buzzword-overload seem to run rampant in the Windows development world.

Operationally, COM defines a standard way for objects implemented in arbitrary languages to talk to each other, using a published object model. For example, COM components can be written in and used by programs written in Visual Basic, Visual C++, Delphi, PowerBuilder, and Python. Because the COM indirection layer hides the differences between all the languages involved, it's possible for Visual Basic to use an object implemented in Python and vice versa.

Moreover, many software packages register COM interfaces to support end-user scripting. For instance, Microsoft Excel publishes an object model that allows any COM-aware scripting language to start Excel and programmatically access spreadsheet data. Similarly, Microsoft Word can be scripted through COM to automatically manipulate documents. COM's language-neutrality means that programs written in any programming language with a COM interface, including Visual Basic and Python, can be used to automate Excel and Word processing.

Of most relevance to this chapter, Active Scripting also provides COM objects that allow scripts embedded in HTML to communicate with Microsoft's Internet Explorer (on the client) and Internet Information Server (on the server). Both systems register their object models with Windows such that they can be invoked from any COM-aware language. For example, when Internet Explorer extracts and executes Python code embedded in HTML, some Python variable names are automatically preset to COM object components that give access to IE context and to (`alert` in [Example 15-11](#)). Calls to such components from Python code are automatically routed through COM back to IE.

15.8.3.2 Python COM clients

With the `win32all` Python extension package installed, though, we can also write Python programs that serve as registered COM servers and clients, even if they have nothing to do with the Internet at all. For example, the Python program in [Example 15-13](#) acts as a client to the Microsoft Word COM object.

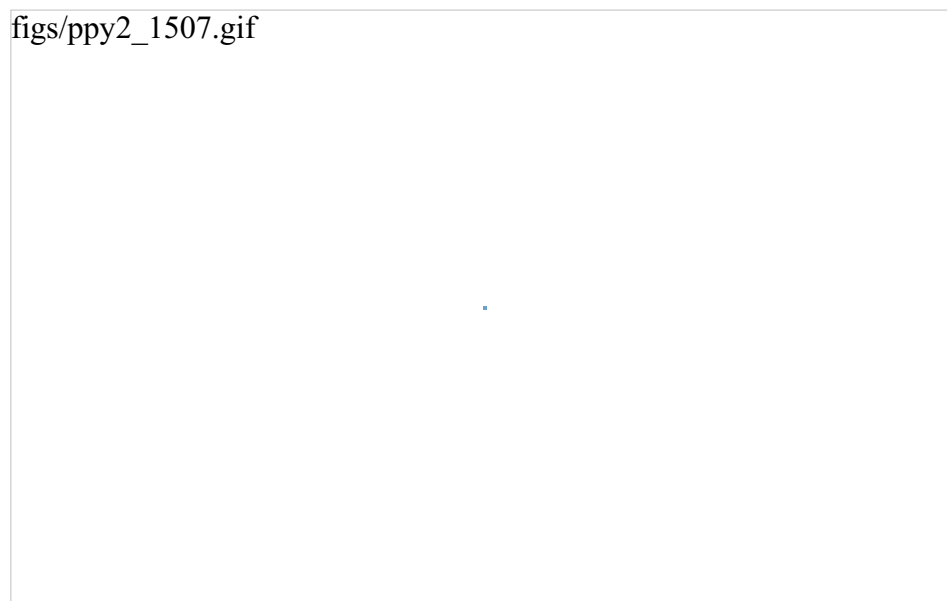
Example 15-13. PP2E\Internet\Other\Com\comclient.py

```
#####  
# a COM client coded in Python: talk to MS-Word via its COM object  
# model; uses either dynamic dispatch (run-time lookup/binding),  
# or the static and faster type-library dispatch if makepy.py has  
# been run; install the windows win32all extensions package to use  
# this interface; Word runs hidden unless Visible is set to 1 (and  
# Visible lets you watch, but impacts interactive Word sessions);  
#####  
  
from sys import argv  
docdir = 'C:\\temp\\'  
if len(argv) == 2: docdir = argv[1]           # ex: comclient.py a:\  
  
from win32com.client import Dispatch       # early or late binding  
word = Dispatch('Word.Application')      # connect/start word  
word.Visible = 1                          # else word runs hidden  
  
# create and save new doc file  
newdoc = word.Documents.Add( )           # call word methods  
spot = newdoc.Range(0,0)  
spot.InsertBefore('Hello COM client world!') # insert some text  
newdoc.SaveAs(docdir + 'pycom.doc')      # save in doc file  
newdoc.SaveAs(docdir + 'copy.doc')  
newdoc.Close( )  
  
# open and change a doc file  
olddoc = word.Documents.Open(docdir + 'copy.doc')  
finder = word.Selection.Find  
finder.text = 'COM'  
finder.Execute( )  
word.Selection.TypeText('Automation')  
olddoc.Close( )  
  
# and so on: see Word's COM interface specs
```

This particular script starts Microsoft Word -- known as `word.Application` to scripting clients - if needed, and converses with it through COM. That is, calls in this script are automatically routed from Python to Microsoft Word and back. This code relies heavily on calls exported by Word, which are not described in this book. Armed with documentation for Word's object API, though, we could use such calls to write Python scripts that automate document updates, insert and replace text, create and print documents, and so on.

For instance, [Figure 15-7](#) shows the two Word *.doc* files generated when the previous script is run on Windows: both are new files, and one is a copy of the other with a text replacement applied. The interaction that occurs while the script runs is more interesting: because Word's `Visible` attribute is set to 1, you can actually watch Word inserting and replacing text, saving files, etc., in response to calls in the script. (Alas, I couldn't quite figure out how to paste a movie clip in this book.)

Figure 15-7. Word files generated by Python COM client



In general, Python COM client calls may be dispatched either dynamically by run-time look-up in the Windows registry, or statically using type libraries created by running a Python utility script at development time (*makepy.py*). These dispatch modes are sometimes called late and early dispatch binding, respectively. Dynamic (late) dispatch skips a development step but is slower when clients are running, due to all the required look-ups.^[10]

[10] Actually, `makepy` can also be executed at runtime now, so you may no longer need to manually run it during development. See the `makepy` documentation available in the latest Windows extensions package for breaking details.

Luckily, we don't need to know which scheme will be used when we write client scripts. The `Dispatch` call used in [Example 15-13](#) to connect to Word is smart enough to use static binding if server type libraries exist, or dynamic binding if they do not. To force dynamic binding and ignore any generated type libraries, replace the first line with this:

```
from win32com.client.dynamic import Dispatch # always late binding
```

However calls are dispatched, the Python COM interface performs all the work of locating the named server, looking up and calling the desired methods or attributes, and converting Python datatypes according to a standard type map as needed. In the context of Active Scripting, the underlying COM model works the same way, but the server is something like IE or IIS (not Word), the set of available calls differs, and some Python variables are preassigned to COM server objects. The notions of "client" and "server" can become somewhat blurred in these scenarios, but the net result is similar.

15.8.3.3 Python COM servers

Python scripts can also be deployed as COM servers, and provide methods and attributes that are accessible to any COM-aware programming language or system. This topic is too complex to cover well here, but exporting a Python object to COM is mostly just a matter of providing a set of class attributes to identify the server and utilizing the proper `win32com` registration utility calls.

[Example 15-14](#) is a simple COM server coded in Python as a class.

Example 15-14. PP2E\Internet\Other\Com\comserver.py

```
#####
# a COM server coded in Python; the _reg_ class attributes
# give registry parameters, and others list methods and attrs;
# for this to work, you must install Python and the win32all
# package, this module file must live on your Python path,
# and the server must be registered to COM (see code at end);
# run pythoncom.CreateGuid( ) to make your own _reg_clsid_key;
#####

import sys
from win32com.server.exception import COMException          # what to raise
import win32com.server.util                                # server tools
globhellos = 0

class MyServer:

    # com info settings
    _reg_clsid_      = '{1BA63CC0-7CF8-11D4-98D8-BB74DD3DDE3C}'
    _reg_desc_       = 'Example Python Server'
    _reg_progid_     = 'PythonServers.MyServer'              # external name
    _reg_class_spec_ = 'comserver.MyServer'                 # internal name
    _public_methods_ = ['Hello', 'Square']
    _public_attrs_   = ['version']

    # python methods
    def __init__(self):
        self.version = 1.0
        self.hellos  = 0
    def Square(self, arg):                                  # exported methods
        return arg ** 2
    def Hello(self):                                       # global variables
        global globhellos                                  # retain state, bu
        globhellos = globhellos + 1                        # self vars don't
        self.hellos = self.hellos + 1
        return 'Hello COM server world [%d, %d]' % (globhellos, self.hellos)

# registration functions
def Register(pyclass=MyServer):
    from win32com.server.register import UseCommandLine
    UseCommandLine(pyclass)
def Unregister(classid=MyServer._reg_clsid_):
    from win32com.server.register import UnregisterServer
    UnregisterServer(classid)

if __name__ == '__main__':                                # register server if file run or clicked
    Register( )                                           # unregisters if --unregister cmd-line arg
```

As usual, this Python file must be placed in a directory in Python's module search path. Besides the server class itself, the file includes code at the bottom to automatically register and unregister the server to COM when the file is run:

- To register a server, simply call the `UseCommandLine` function in the `win32com.server.register` package and pass in the Python server class. This function uses all the special class attribute settings to make the server known to COM. The file is set to automatically call the registration tools if it is run by itself (e.g., when clicked in a file explorer).
- To unregister a server, simply pass an `--unregister` argument on the command line when

running this file. When run this way, the script automatically calls `UseCommandLine` again unregister the server; as its name implies, this function inspects command-line arguments and knows to do the right thing when `--unregister` is passed. You can also unregister servers explicitly with the `UnregisterServer` call demonstrated near the end of this script though this is less commonly used.

Perhaps the more interesting part of this code, though, is the special class attribute assignments at the start of the Python class. These class annotations can provide server registry settings (the `_reg_` attributes), accessibility constraints (the `_public_` names), and more. Such attributes are specific to the Python COM framework, and their purpose is to configure the server.

For example, the `_reg_class_spec_` is simply the Python module and class names separated by period. If set, the resident Python interpreter uses this attribute to import the module and create an instance of the Python class it defines, when accessed by a client.^[11]

[11] But note that the `_reg_class_spec_` attribute is no longer strictly needed, and not specifying it avoids a number of PYTHONPATH issues. Because such settings are prone to change, you should always consult the latest Windows extensions package reference manuals for details on this and other class annotation attributes.

Other attributes may be used to identify the server in the Windows registry. The `_reg_clsids_` attribute, for instance, gives a globally unique identifier (GUID) for the server and should vary in every COM server you write. In other words, don't use the value in this script. Instead, do what I did to make this ID, and paste the result returned on your machine into your script:^[12]

[12] The `A: />` prompt shows up here only because I copied the COM scripts to a floppy so I could run them on a machine with the `win32all` extension installed. You should be able to run from the directory where these scripts live in the examples tree.

```
A:\>python
>>> import pythoncom
>>> pythoncom.CreateGuid( )
<iid:{1BA63CC0-7CF8-11D4-98D8-BB74DD3DDE3C}>
```

GUIDs are generated by running a tool shipped with the Windows extensions package; simply import and call the `pythoncom.CreateGuid()` function and insert the returned text in the script. Windows uses the ID stamped into your network card to come up with a complex ID that is likely to be unique across servers and machines. The more symbolic Program ID string, `_reg_progid_` can be used by clients to name servers too, but is not as likely to be unique.

The rest of the server class is simply pure-Python methods, which implement the exported behavior of the server; that is, things to be called from clients. Once this Python server is annotated, coded, and registered, it can be used in any COM-aware language. For instance, programs written in Visual Basic, C++, Delphi, and Python may access its public methods and attributes through COM; of course, other Python programs can also simply import this module, but the point of COM is to open up components for even wider reuse.^[13]

[13] But you should be aware of a few type rules. In Python 1.5.2, Python-coded COM servers must be careful to use a fixed number of function arguments, and convert passed-in strings with the `str` built-in function. The latter of these constraints

arises because COM passes strings as Unicode strings. Because Python 1.6 and 2.0 now support both Unicode and normal strings, though, this constraint should disappear soon. When using COM as a client (i.e., code that calls COM), you may pass a string or Unicode object, and the conversion is done automatically; when coding a COM server (i.e., code called by COM), strings are always passed in as Unicode objects.

15.8.3.3.1 Using the Python server from a Python client

Let's put this Python COM server to work. The Python script in [Example 15-15](#) tests the server two ways: first by simply importing and calling it directly, and then by employing Python's client-side COM interfaces shown earlier to invoke it less directly. When going through COM, the `PythonServers.MyServer` symbolic program ID we gave the server (by setting class attribute `_reg_progid_`) can be used to connect to this server from any language (including Python).

Example 15-15. PP2E\Internet\Other\Com\comserver-test.py

```
#####
# test the Python-coded COM server from Python two ways
#####

def testViaPython( ):
    from comserver import MyServer
    object = MyServer( )
    print object.Hello( )
    print object.Square(8)
    print object.version

def testViaCom( ):
    from win32com.client import Dispatch
    server = Dispatch('PythonServers.MyServer')
    print server.Hello( )
    print server.Square(12)
    print server.version

if __name__ == '__main__':
    testViaPython( )
    testViaCom( )
```

If we've properly configured and registered the Python COM server, we can talk to it by running this Python test script. In the following, we run the server and client files from an MS-DOS console box (though they can usually be run by mouse clicks as well). The first command runs the server file by itself to register the server to COM; the second executes the test script to exercise the server both as an imported module (`testViaPython`) and as a server accessed through COM (`testViaCom`):

```
A:\>python comserver.py
Registered: PythonServers.MyServer
```

```
A:\>python comserver-test.py
```

```
Hello COM server world [1, 1]
64
1.0
Hello COM server world [2, 1]
144
1.0
Hello COM server world [3, 1]
144
1.0
```

```
A:\>python comserver.py --unregister
Unregistered: PythonServers.MyServer
```

Notice the two numbers at the end of the `Hello` output lines: they reflect current values of a global variable and a server instance attribute. Global variables in the server's module retain state as long as the server module is loaded; by contrast, each COM `Dispatch` (and Python class) call makes a new instance of the server class, and hence new instance attributes. The third command unregisters the server in COM, as a cleanup step. Interestingly, once the server has been unregistered, it's no longer usable, at least not through COM:

```
A:\>python comserver-test.py
Hello COM server world [1, 1]
64
1.0
Traceback (innermost last):
  File "comserver-test.py", line 21, in ?
    testViaCom( ) # com object retains
  File "comserver-test.py", line 14, in testViaCom
    server = Dispatch('PythonServers.MyServer') # use Windows register
    ...more deleted...
pywintypes.com_error: (-2147221005, 'Invalid class string', None, None)
```

15.8.3.3.2 Using the Python server from a VB client

The `comserver-test.py` script just listed demonstrates how to use a Python COM server from a Python COM client. Once we've created and registered a Python COM server, though, it's available to any language that sports a COM interface. For instance, [Example 15-16](#) shows the sort of code we write to access the Python server from Visual Basic. Clients coded in other languages (e.g., Delphi or Visual C++) are analogous, but syntax and instantiation calls may vary.

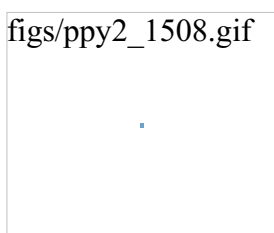
Example 15-16. PP2E\Internet\Other\Com\comserver-test.bas

```
Sub runpyserver( )
    ' use python server from vb client
    ' alt-f8 in word to start macro editor
    Set server = CreateObject("PythonServers.MyServer")
    hello1 = server.hello( )
    square = server.square(32)
    pyattr = server.Version
    hello2 = server.hello( )
    sep = Chr(10)
    Result = hello1 & sep & square & sep & pyattr & sep & hello2
    MsgBox Result
End Sub
```

The real trick (at least for someone as naive about VB as this author) is how to make this code go. Because VB is embedded in Microsoft Office products such as Word, one approach is to test this code in the context of those systems. Try this: start Word, press Alt and F8 together, and you'll wind up in the Word macro dialog. There, enter a new macro name, press Create, and you'll find yourself in a development interface where you can paste and run the VB code just shown.

I don't teach VB tools in this book, so you'll need to consult other documents if this fails on your end. But it's fairly simple once you get the knack -- running the VB code in this context produce the Word pop-up box in [Figure 15-8](#), showing the results of VB calls to our Python COM server. Global variable and instance attribute values at the end of both `Hello` reply messages are the same this time, because we make only one instance of the Python server class: in VB, by calling `CreateObject`, with the program ID of the desired server.

Figure 15-8. VB client running Python COM server



But because we've now learned how to embed VBScript in HTML pages, another way to kick off the VB client code is to put it in a web page and rely on IE to launch it for us. The bulk of the HTML file in [Example 15-17](#) is the same as the Basic file shown previously, but tags have been added around the code to make it a bona fide web page.

Example 15-17. PP2E\Internet\Other\Com\comserver-test-vbs.html

```
<HTML><BODY>
<P>Run Python COM server from VBScript embedded in HTML via IE</P>
<SCRIPT Language=VBScript>

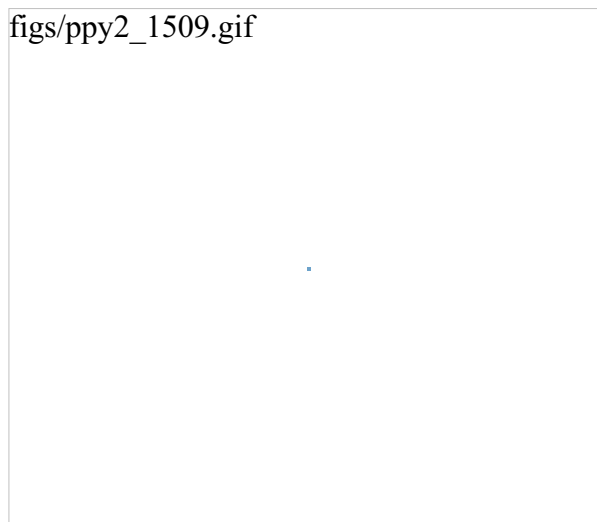
Sub runpyserver( )
    ' use python server from vb client
    ' alt-f8 in word to start macro editor
    Set server = CreateObject("PythonServers.MyServer")
    hello1 = server.hello( )
    square = server.square(9)
    pyattr = server.Version
    hello2 = server.hello( )
    sep = Chr(10)
    Result = hello1 & sep & square & sep & pyattr & sep & hello2
    MsgBox Result
End Sub

runpyserver( )

</SCRIPT>
</BODY></HTML>
```

There is an incredible amount of routing going on here, but the net result is similar to running the VB code by itself. Clicking on this file starts Internet Explorer (assuming it is registered to handle HTML files), which strips out and runs the embedded VBScript code, which in turn calls out to the Python COM server. That is, IE runs VBScript code that runs Python code -- a control flow spanning three systems, an HTML file, a Python file, and the IE implementation. With COM, it just works. [Figure 15-9](#) shows IE in action running the HTML file above; the pop-up box is generated by the embedded VB code as before.

Figure 15-9. IE running a VBScript client running a Python COM server



If your client code runs but generates a COM error, make sure that the `win32all` package has been installed, that the server module file is in a directory on Python's path, and that the server file has been run by itself to register the server with COM. If none of that helps, you're probably already beyond the scope of this text. (Please see additional Windows programming resources for more details.)

15.8.3.4 The bigger COM picture

So what does writing Python COM servers have to do with the Internet motif of this chapter? After all, Python code embedded in HTML simply plays the role of COM client to IE or IIS systems that usually run locally. Besides showing how such systems work their magic, I've presented this topic here because COM, at least in its grander world view, is also about communicating over networks.

Although we can't get into details in this text, COM's distributed extensions make it possible to implement Python-coded COM servers to run on machines that are arbitrarily remote from clients. Although largely transparent to clients, COM object calls like those in the preceding client script may imply network transfers of arguments and results. In such a configuration, COM may be used as a general client/server implementation model and a replacement for technologies such as RPC (Remote Procedure Calls).

For some applications, this distributed object approach may even be a viable alternative to Python's other client and server-side scripting tools we've studied in this part of the book. Moreover, even when not distributed, COM is an alternative to the lower-level Python/C integration techniques we'll meet later in this book.

Once its learning curve is scaled, COM is a straightforward way to integrate arbitrary components and provides a standardized way to script and reuse systems. However, COM also implies a level of dispatch indirection overhead and is a Windows-only solution at this writing. Because of that, it is generally not as fast or portable as some of the other client/server and C integration schemes discussed in this book. The relevance of such trade-offs varies per application.

As you can probably surmise, there is much more to the Windows scripting story than we cover here. If you are interested in more details, O'Reilly's *Python Programming on Win32* provides an excellent presentation of these and other Windows development topics. Much of the effort that goes into writing scripts embedded in HTML involves using the exposed object model APIs, which are deliberately skipped in this book; see Windows documentation sources for more detail.

The New C# Python Compiler

Late-breaking news: a company called ActiveState (<http://www.activestate.com>) announced a new compiler for Python after this chapter was completed. This system (tentatively titled Python.NET) is a new, independent Python language implementation like the JPython system described earlier in this chapter, but compiles Python scripts for use in the Microsoft C# language environment and .NET framework (a software component system based on XML that fosters cross-language interoperability). As such, it opens the door to other Python web scripting roles and modes in the Windows world.

If successful, this new compiler system promises to be the third Python implementation (with JPython and the standard C implementation) and an exciting development for Python in general. Among other things, the C#-based port allows Python scripts to be compiled to binary *.exe* files and developed within the Visual Studio IDE. As in the JPython Java-based implementation, scripts are coded using the standard Python core language presented in this text, and translated to be executed by the underlying C# system. Moreover, .NET interfaces are automatically integrated for use in Python scripts: Python classes may subclass, act as, and use .NET components.

Also like JPython, this new alternative implementation of Python has a specific target audience and will likely prove to be of most interest to developers concerned with C# and .NET framework integration. ActiveState also plans to roll out a whole suite of Python development products besides this new compiler; be sure to watch the Python and ActiveState web sites for more details.

15.9 Python Server Pages

Though still somewhat new at this writing, Python Server Pages (PSP) is a server-side technology that embeds JPython code inside HTML. PSP is a Python-based answer to other server-side embedded scripting approaches.

The PSP scripting engine works much like Microsoft's Active Server Pages (ASP, described earlier) and Sun's Java Server Pages (JSP) specification. At the risk of pushing the acronym tolerance envelope, PSP has also been compared to PHP, a server-side scripting language embedded in HTML. All of these systems, including PSP, embed scripts in HTML and run them on the server to generate the response stream sent back to the browser on the client; scripts interact with an exposed object model API to get their work done. PSP is written in pure Java, however, and so is portable to a wide variety of platforms (ASP applications can be run only on Microsoft platforms).

PSP uses JPython as its scripting language, reportedly a vastly more appropriate choice for scripting web sites than the Java language used in Java Server Pages. Since JPython code is embedded under PSP, scripts have access to the large number of Python and JPython tools and add-ons from within PSPs. In addition, scripts may access all Java libraries, thanks to JPython's Java integration support.

We can't cover PSP in detail here; but for a quick look, [Example 15-18](#), adapted from an example in the PSP documentation, illustrates the structure of PSPs.

Example 15-18. PP2E\Internet\Other\hello.psp

```
[$[
# Generate a simple message page with the client's IP address
]$
<HTML><HEAD>
<TITLE>Hello PSP World</TITLE>
</HEAD>
<BODY>
$[include banner.psp]$
<H1>Hello PSP World</H1>
<BR>
$[
Response.write("Hello from PSP, %s." % (Request.server["REMOTE_ADDR"]) )
]$
<BR>
</BODY></HTML>
```

A page like this would be installed on a PSP-aware server machine and referenced by URL from a browser. PSP uses `[$[` and `]$` delimiters to enclose JPython code embedded in HTML; anything outside these pairs is simply sent to the client browser, while code within these markers is executed. The first code block here is a JPython comment (note the `#` character); the second is an `include` statement that simply inserts another PSP file's contents.

The third piece of embedded code is more useful. As in Active Scripting technologies, Python code embedded in HTML uses an exposed object API to interact with the execution context -- in this case, the `Response` object is used to write output to the client's browser (much like a `print` in a CGI script), and `Request` is used to access HTTP headers for the request. The

`Request` object also has a `params` dictionary containing `GET` and `POST` input parameters, as well as a `cookies` dictionary holding cookie information stored on the client by a PSP application.

Notice that the previous example could have just as easily been implemented with a Python CGI script using a Python `print` statement, but PSP's full benefit becomes clearer in large pages that embed and execute much more complex JPython code to produce a response.

PSP runs as a Java servlet and requires the hosting web site to support the Java Servlet API, all of which is beyond the scope of this text. For more details about PSP, visit its web site, currently located at <http://www.ciobriefings.com/psp>, but search <http://www.python.org> for other links if this one changes over time.

Chapter 15. Advanced Internet Topics

[Section 15.1. "Surfing on the Shoulders of Giants"](#)

[Section 15.2. Zope: A Web Publishing Framework](#)

[Section 15.3. HTMLgen: Web Pages from Objects](#)

[Section 15.4. JPython \(.Jython\): Python for Java](#)

[Section 15.5. Grail: A Python-Based Web Browser](#)

[Section 15.6. Python Restricted Execution Mode](#)

[Section 15.7. XML Processing Tools](#)

[Section 15.8. Windows Web Scripting Extensions](#)

[Section 15.9. Python Server Pages](#)

[Section 15.10. Rolling Your Own Servers in Python](#)

16.1 "Give Me an Order of Persistence, but Hold the Pickles"

So far in this book, we've used Python in the system programming, GUI development, and Internet scripting domains -- three of Python's most common applications. In the next three chapters, we're going to take a quick look at other major Python programming topics: persistent data, data structure techniques, and text and language processing tools. None of these are covered exhaustively (each could easily fill a book alone), but we'll sample Python in action in these domains and highlight their core concepts. If any of these chapters spark your interest, additional resources are readily available in the Python world.

16.2 Persistence Options in Python

In this chapter, our focus is on persistent data -- the kind that outlives a program that creates it. That's not true by default for objects a script constructs; things like lists, dictionaries, and even class instance objects live in your computer's memory and are lost as soon as the script ends. To make data longer-lived, we need to do something special. In Python programming, there are at least five traditional ways to save information between program executions:

- Flat files: storing text and bytes
- DBM keyed files: keyed access to strings
- Pickled objects: serializing objects to byte streams
- Shelve files: storing pickled objects in DBM keyed files
- Database systems: full-blown SQL and object database systems

We studied Python's simple (or "flat") file interfaces in earnest in [Chapter 2](#), and have been using them ever since. Python provides standard access to both the `stdio` filesystem (through the built-in `open` function), as well as lower-level descriptor-based files (with the built-in `os` module). For simple data storage tasks, these are all that many scripts need. To save for use in a future program run, simply write data out to a newly opened file on your computer and read it back from that file later. As we've seen, for more advanced tasks, Python also supports other file-like interfaces such as pipes, fifos, and sockets.

Since we've already explored flat files, I won't say more about them here. The rest of this chapter introduces the remaining topics on the list earlier in this section. At the end, we'll also meet a GUI program for browsing the contents of things like shelves and DBM files. Before that, though, we need to learn what manner of beast these are.

16.3 DBM Files

Flat files are handy for simple persistence tasks, but are generally geared towards a sequential processing mode. Although it is possible to jump around to arbitrary locations with `seek` calls, flat files don't provide much structure to data beyond the notion of bytes and text lines.

DBM files, a standard tool in the Python library for database management, improve on that by providing key-based access to stored text strings. They implement a random-access, single-key view on stored data. For instance, information related to objects can be stored in a DBM file using a unique key per object and later can be fetched back directly with the same key. DBM files are implemented by a variety of underlying modules (including one coded in Python), but if you have Python, you have a DBM.

16.3.1 Using DBM Files

Although DBM filesystems have to do a bit of work to map chunks of stored data to keys for fast retrieval (technically, they generally use a technique called hashing to store data in files), your scripts don't need to care about the action going on behind the scenes. In fact, DBM is one of the easiest ways to save information in Python -- DBM files behave so much like in-memory dictionaries that you may forget you're actually dealing with a file. For instance, given a DBM file object:

- Indexing by key fetches data from the file.
- Assigning to an index stores data in the file.

DBM file objects also support common dictionary methods such as keys-list fetches and tests, and key deletions. The DBM library itself is hidden behind this simple model. Since it is so simple, let's jump right into an interactive example that creates a DBM file and shows how the interface works:

```
% python
>>> import anydbm                               # get interface: dbm, gdbm, ndbm,..
>>> file = anydbm.open('movie', 'c')           # make a dbm file called 'movie'
>>> file['Batman'] = 'Pow!'                     # store a string under key 'Batman'
>>> file.keys( )                                # get the file's key directory
['Batman']
>>> file['Batman']                              # fetch value for key 'Batman'
'Pow!'

>>> who = ['Robin', 'Cat-woman', 'Joker']
>>> what = ['Bang!', 'Splat!', 'Wham!']
>>> for i in range(len(who)):
...     file[who[i]] = what[i]                 # add 3 more "records"
...
>>> file.keys( )
['Joker', 'Robin', 'Cat-woman', 'Batman']
>>> len(file), file.has_key('Robin'), file['Joker']
(4, 1, 'Wham!')
>>> file.close( )                               # close sometimes required
```

Internally, importing `anydbm` automatically loads whatever DBM interface is available in your

Python interpreter, and opening the new DBM file creates one or more external files with names that start with the string "movie" (more on the details in a moment). But after the import and open, a DBM file is virtually indistinguishable from a dictionary. In effect, the object called `file` here can be thought of as a dictionary mapped to an external file called `movie`.

Unlike normal dictionaries, though, the contents of `file` are retained between Python program runs. If we come back later and restart Python, our dictionary is still available. DBM files are like dictionaries that must be opened:

```
% python
>>> import anydbm
>>> file = anydbm.open('movie', 'c')           # open existing dbm file
>>> file['Batman']
'Pow!'

>>> file.keys( )                               # keys gives an index list
['Joker', 'Robin', 'Cat-woman', 'Batman']
>>> for key in file.keys( ): print key, file[key]
...
Joker Wham!
Robin Bang!
Cat-woman Splat!
Batman Pow!

>>> file['Batman'] = 'Ka-Boom!'                 # change Batman slot
>>> del file['Robin']                           # delete the Robin entry
>>> file.close( )                             # close it after changes
```

Apart from having to import the interface and open and close the DBM file, Python programs don't have to know anything about DBM itself. DBM modules achieve this integration by overloading the indexing operations and routing them to more primitive library tools. But you'd never know that from looking at this Python code -- DBM files look like normal Python dictionaries, stored on external files. Changes made to them are retained indefinitely:

```
% python
>>> import anydbm                               # open dbm file again
>>> file = anydbm.open('movie', 'c')
>>> for key in file.keys( ): print key, file[key]
...
Joker Wham!
Cat-woman Splat!
Batman Ka-Boom!
```

As you can see, this is about as simple as it can be. [Table 16-1](#) lists the most commonly used DBM file operations. Once such a file is opened, it is processed just as though it were an in-memory Python dictionary. Items are fetched by indexing the file object by key and stored by assigning to a key.

Table 16-1. DBM File Operations

Python Code	Action	Description
<code>import anydbm</code>	Import	Get dbm, gdbm ,... whatever is installed
<code>file = anydbm.open('filename', 'c')</code>	Open ^[1]	Create or open an existing DBM file
<code>file['key'] = 'value'</code>	Store	Create or change the entry for <code>key</code>
<code>value = file['key']</code>	Fetch	Load the value for entry <code>key</code>
<code>count = len(file)</code>	Size	Return the number of entries stored
<code>index = file.keys()</code>	Index	Fetch the stored keys list

<code>found = file.has_key('key')</code>	Query	See if there's an entry for <code>key</code>
<code>del file['key']</code>	Delete	Remove the entry for <code>key</code>
<code>file.close()</code>	Close	Manual close, not always needed

[1] In Python versions 1.5.2 and later, be sure to pass a string `c` as a second argument when calling `anydbm.open`, to force Python to create the file if it does not yet exist, and simply open it otherwise. This used to be the default behavior but is no longer. You do not need the `c` argument when opening shelves discussed ahead - they still use an "open or create" mode by default if passed no open mode argument. Other open mode strings can be passed to `anydbm` (e.g., `n` to always create the file, and `r` for read only -- the new default); see the library reference manuals for more details.

Despite the dictionary-like interface, DBM files really do map to one or more external files. For instance, the underlying `gdbm` interface writes two files, `movie.dir` and `movie.pag`, when a GDBM file called `movie` is made. If your Python was built with a different underlying keyed-file interface, different external files might show up on your computer.

Technically, module `anydbm` is really an interface to whatever DBM-like filesystem you have available in your Python. When creating a new file, `anydbm` today tries to load the `dbhash`, `gdbm`, and `dbm` keyed-file interface modules; Python's without any of these automatically fall back on an all-Python implementation called `dumbdbm`. When opening an already-existing DBM file, `anydbm` tries to determine the system that created it with the `whichdb` module instead. You normally don't need to care about any of this, though (unless you delete the files your DBM creates).

Note that DBM files may or may not need to be explicitly closed, per the last entry in [Table 16-1](#). Some DBM files don't require a close call, but some depend on it to flush changes out to disk. On such systems, your file may be corrupted if you omit the close call. Unfortunately, the default DBM on the 1.5.2 Windows Python port, `dbhash` (a.k.a., `bsddb`), is one of the DBM systems that requires a close call to avoid data loss. As a rule of thumb, always close your DBM files explicitly after making changes and before your program exits, to avoid potential problems. This rule extends by proxy to shelves, a topic we'll meet later in this chapter.

16.4 Pickled Objects

Probably the biggest limitation of DBM keyed files is in what they can store: data stored under a key must be a simple text string. If you want to store Python objects in a DBM file, you can sometimes manually convert them to and from strings on writes and reads (e.g., with `str` and `eval` calls), but this only takes you so far. For arbitrarily complex Python objects like class instances, you need something more. Class instance objects, for example, cannot be later recreated from their standard string representations.

The Python `pickle` module, a standard part of the Python system, provides the conversion step needed. It converts Python in-memory objects to and from a single linear string format, suitable for storing in flat files, shipping across network sockets, and so on. This conversion from object to string is often called serialization -- arbitrary data structures in memory are mapped to a serial string form. The string representation used for objects is also sometimes referred to as a byte-stream, due to its linear format.

16.4.1 Using Object Pickling

Pickling may sound complicated the first time you encounter it, but the good news is that Python hides all the complexity of object-to-string conversion. In fact, the pickle module's interfaces are incredibly simple to use. The following list describes a few details of this interface.

```
P = pickle.Pickler(file)
```

Make a new pickler for pickling to an open output file object `file`.

```
P.dump(object)
```

Write an object onto the pickler's file/stream.

```
pickle.dump(object, file)
```

Same as the last two calls combined: pickle an object onto an open file.

```
U = pickle.Unpickler(file)
```

Make an unpickler for unpickling from an open input file object `file`.

```
object = U.load( )
```

Read an object from the unpickler's file/stream.

```
object = pickle.load(file)
```

Same as the last two calls combined: unpickle an object from an open file.

```
string = pickle.dumps(object)
```

Return the pickled representation of `object` as a character string.


```
object = pickle.loads( string)
```

Read an object from a character string instead of a file.

`Pickler` and `Unpickler` are exported classes. In all of these, `file` is either an open file object or any object that implements the same attributes as file objects:

- `Pickler` calls the file's `write` method with a string argument.
- `Unpickler` calls the file's `read` method with a byte count, and `readline` without arguments.

Any object that provides these attributes can be passed in to the "file" parameters. In particular, `file` can be an instance of a Python class that provides the read/write methods. This lets you map pickled streams to in-memory objects, for arbitrary use. It also lets you ship Python objects across a network, by providing sockets wrapped to look like files in pickle calls at the sender and unpickle calls at the receiver (see [Making Sockets Look Like Files](#) in [Chapter 10](#), for more details).

In more typical use, to pickle an object to a flat file, we just open the file in write-mode, and call the `dump` function; to unpickle, reopen and call `load`:

```
% python
>>> import pickle
>>> table = {'a': [1, 2, 3], 'b': ['spam', 'eggs'], 'c':{'name':'bob'}}
>>> mydb = open('dbase', 'w')
>>> pickle.dump(table, mydb)

% python
>>> import pickle
>>> mydb = open('dbase', 'r')
>>> table = pickle.load(mydb)
>>> table

{'b': ['spam', 'eggs'], 'a': [1, 2, 3], 'c': {'name': 'bob'}}
```

To make this process simpler still, the module in [Example 16-1](#) wraps pickling and unpickling calls in functions that also open the files where the serialized form of the object is stored.

Example 16-1. PP2E\Dbase\filepickle.py

```
import pickle

def saveDbase(filename, object):
    file = open(filename, 'w')
    pickle.dump(object, file)          # pickle to file
    file.close( )                     # any file-like object will do

def loadDbase(filename):
    file = open(filename, 'r')
    object = pickle.load(file)        # unpickle from file
    file.close( )                     # recreates object in memory
    return object
```

To store and fetch now, simply call these module functions:

```
C:\...\PP2E\Dbase>python
```

```
>>> from filepickle import *
>>> L = [0]
>>> D = {'x':0, 'y':L}
>>> table = {'A':L, 'B':D}           # L appears twice
>>> saveDbase('myfile', table)     # serialize to file

C:\...\PP2E\Dbase>python
>>> from filepickle import *
>>> table = loadDbase('myfile')    # reload/unpickle
>>> table
{'B': {'x': 0, 'y': [0]}, 'A': [0]}
>>> table['A'][0] = 1
>>> saveDbase('myfile', table)

C:\...\PP2E\Dbase>python
>>> from filepickle import *
>>> print loadDbase('myfile')      # both L's updated as expected
{'B': {'x': 0, 'y': [1]}, 'A': [1]}
```

Python can pickle just about anything, except compiled code objects, instances of classes that do not follow importability rules we'll meet later, and instances of some built-in and user-defined types that are coded in C or depend upon transient operating system states (e.g., open file objects cannot be pickled). A `PicklingError` is raised if an object cannot be pickled.

Refer to Python's library manual for more information on the pickler. And while you are flipping (or clicking) through that manual, be sure to also see the entries for the `cPickle` module -- a reimplement of `pickle` coded in C for faster performance. Also check out `marshal`, a module that serializes an object, too, but can only handle simple object types. If available in your Python, the `shelve` module automatically chooses the `cPickle` module for faster serialization, not `pickle`. I haven't explained `shelve` yet, but I will now.