,ch14.17404 Page 816 Wednesday, February 7, 2001 2:54 PM

14

Larger Web Site Examples II

"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.

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 shelve-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.

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.

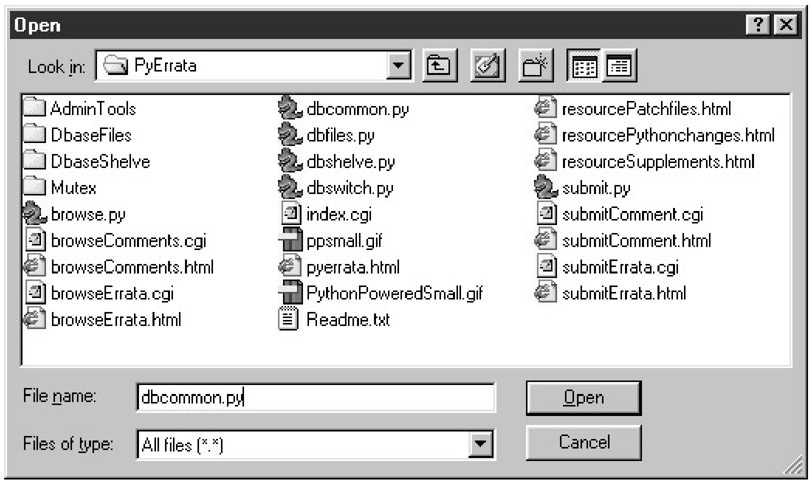
The implementation of these ideas is the PyErrata system—a web site implemented with server-side Python programs. PyErrata allows readers to post bug reports and comments about this edition of *Programming Python*, as well as view the collection of all prior posts by various sort keys. Its goal is to 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.

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.



AdminTools
DbaseFiles
DbaseShelve
Mutex
browse.py
browseComments.cgi
browseComments.html
browseErrata.cgi
browseErrata.html
dbcommon.py
dbfiles.py
dbshelve.py
dbswitch.py
index.cgi
ppsmall.gif
pyerrata.html
PythonPoweredSmall.gif
Readme.txt
resourcePatchfiles.html
resourcePythonchanges.html
resourceSupplements.html
submit.py
submitComment.cgi
submitComment.html
submitErrata.cgi
submitErrata.html

File name: dbcommon.py
Files of type: All files (*.*)

Figure 14-1. PyErrata site contents

You will find a similar structure on this book's CD-ROM. 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.

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, *Databases and Persistence*, 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 on the CD) 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.

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.

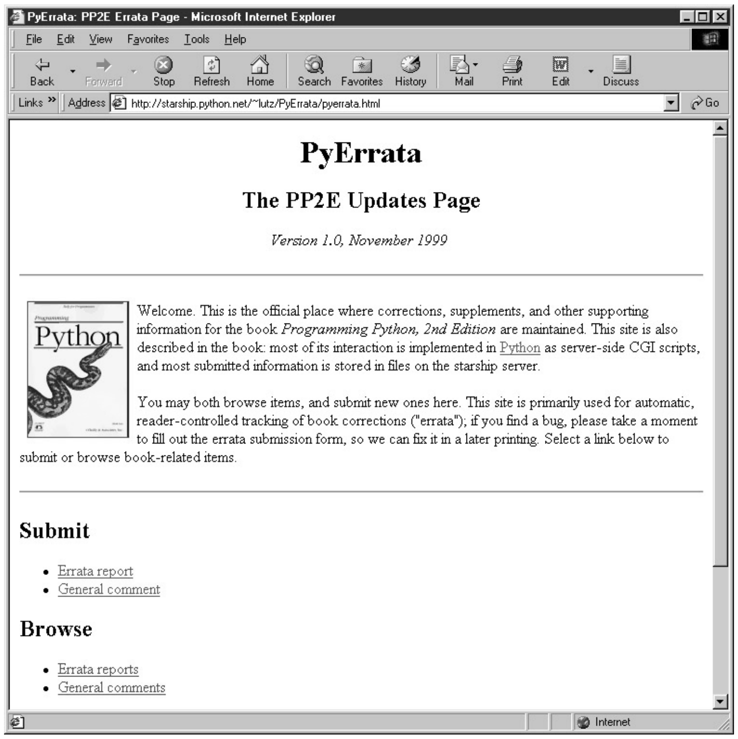
PyErrata installs as a set of HTML files and Python CGI scripts, along with a few image files. As usual, you can simply point your web browser to the system's root page to run the system live while you study this chapter. Its root page currently lives here:[*]

> *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

[*] 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.

error reports; the four main links on this page essentially provide write and read access to its databases over the Web.


PyErrata: PP2E Errata Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/pyerrata.html Go
Internet

PyErrata

The PP2E Updates Page

Version 1.0, November 1999

Welcome. This is the official place where corrections, supplements, and other supporting information for the book *Programming Python, 2nd Edition* are maintained. This site is also described in the book: most of its interaction is implemented in Python as server-side CGI scripts, and most submitted information is stored in files on the starship server.

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.

Submit

- Errata report
- General comment

Browse

- Errata reports
- General comments

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>
```

Example 14-1. PP2E\Internet\Cgi-Web\PyErrata\pyerrata.html (continued)

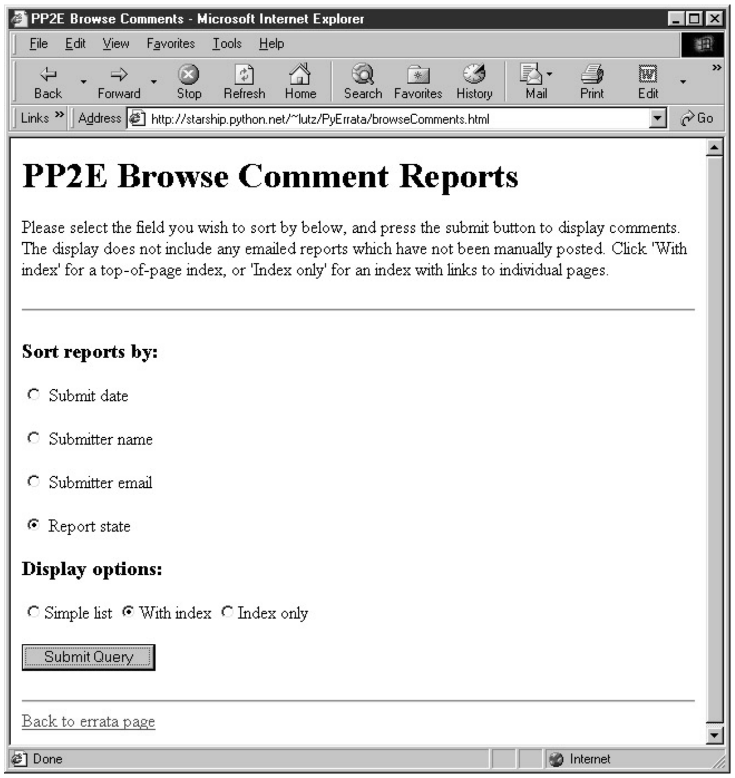
```
<A href="http://rmi.net/~lutz/about-pp.html">
<IMG src="ppsmall.gif" align=left alt="[Book Cover]" border=1 hspace=8></A>

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="../PyInternetDemos.html">More examples</A>
</BODY></HTML>
```

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.

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.


PP2E Browse Comments - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.html Go
Done
Internet

PP2E Browse Comment Reports

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.

Sort reports by:

○ Submit date

○ Submitter name

○ Submitter email

[✓ Report state]

Display options:

○ Simple list [✓ With index] ○ Index only

Submit Query

Back to errata page

Figure 14-3. Browse comments, selection page

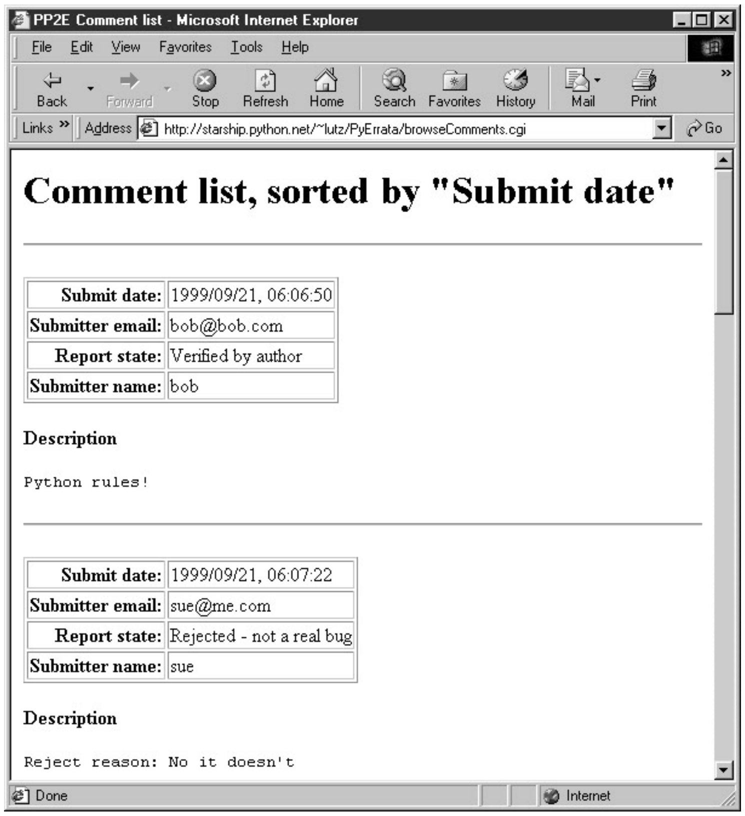
Now, the first thing you should know about PyErrata's browse feature is that it allows users to query and view the report database in multiple ways. Reports may be ordered by any report field and displayed in three different formats. The top-level browse pages essentially serve to configure a query against the report database and the presentation of its result.

,ch14.17404 Page 824 Wednesday, February 7, 2001 2:54 PM

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.


PP2E Comment list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.cgi Go
Done
Internet

Comment list, sorted by "Submit date"

Submit date:	1999/09/21, 06:06:50
Submitter email:	bob@bob.com
Report state:	Verified by author
Submitter name:	bob

Description

```
Python rules!
```

Submit date:	1999/09/21, 06:07:22
Submitter email:	sue@me.com
Report state:	Rejected - not a real bug
Submitter name:	sue

Description

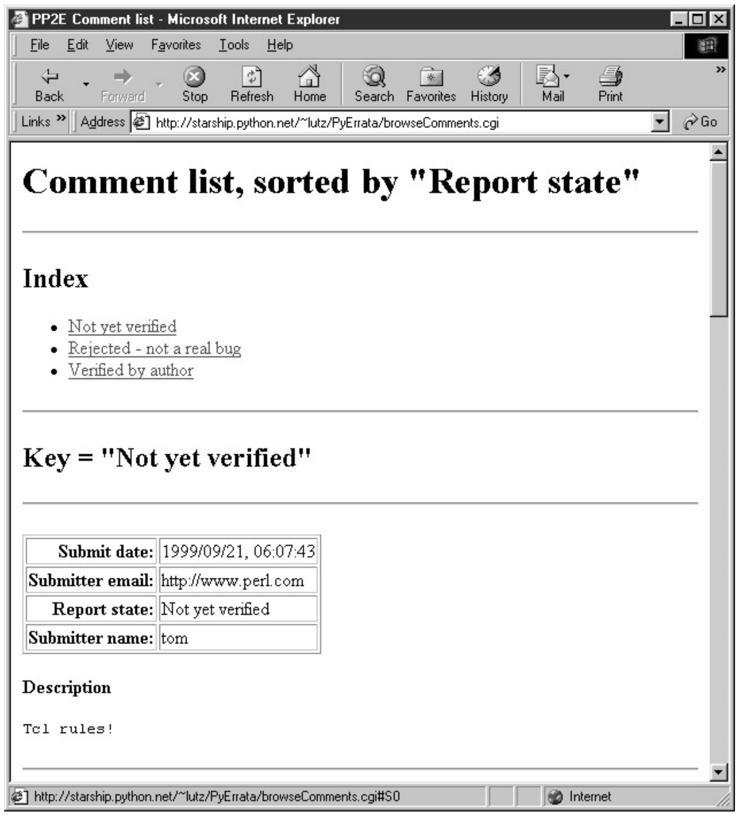
```
Reject reason: No it doesn't
```

Figure 14-4. Browse comments, "Simple list" option

,ch14.17404 Page 825 Wednesday, February 7, 2001 2:54 PM

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.


PP2E Comment list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.cgi Go
http://starship.python.net/~lutz/PyErrata/browseComments.cgi#S0
Internet

Comment list, sorted by "Report state"

Index

- Not yet verified
- Rejected - not a real bug
- Verified by author

Key = "Not yet verified"

Submit date:	1999/09/21, 06:07:43
Submitter email:	http://www.perl.com
Report state:	Not yet verified
Submitter name:	tom

Description

```
Tcl rules!
```

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.

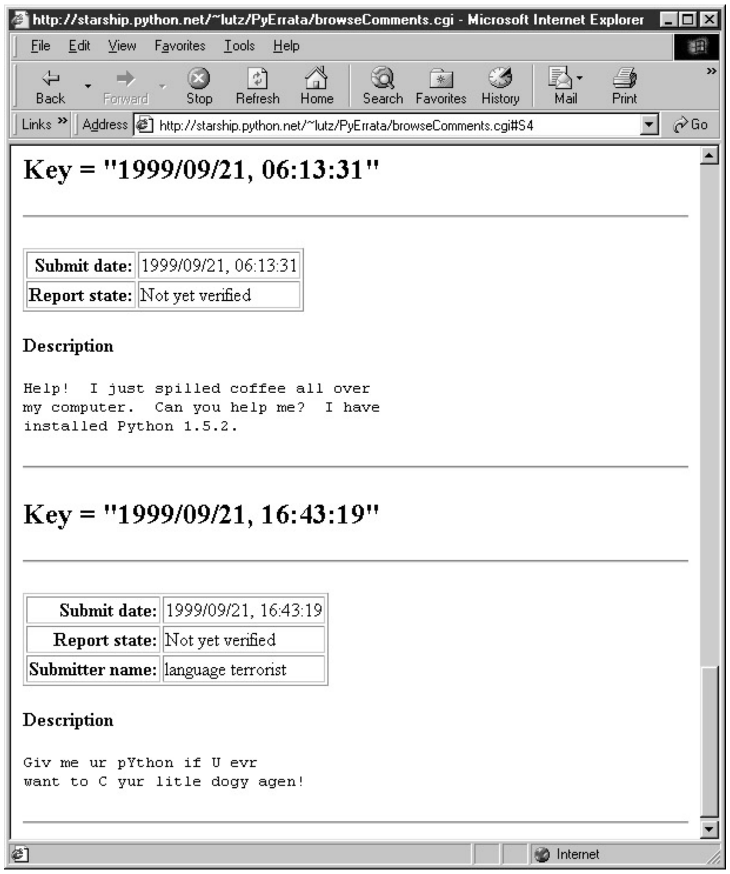
For very large databases, it may be impractical to list every record's contents on the same page; the third PyErrata display format option provides a solution. Figure 14-7 shows the page produced by the "Index only" display option, with "Submit date" chosen for report order. There are no records on this 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.

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

,ch14.17404 Page 827 Wednesday, February 7, 2001 2:54 PM


http://starship.python.net/~lutz/PyErrata/browseComments.cgi - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.cgi#S4 Go
Internet

Key = "1999/09/21, 06:13:31"

Submit date:	1999/09/21, 06:13:31
Report state:	Not yet verified

Description

```
Help!  I just spilled coffee all over
my computer.  Can you help me?  I have
installed Python 1.5.2.
```

Key = "1999/09/21, 16:43:19"

Submit date:	1999/09/21, 16:43:19
Report state:	Not yet verified
Submitter name:	language terrorist

Description

```
Giv me ur pYthon if U evr
want to C yur litle dogy agen!
```

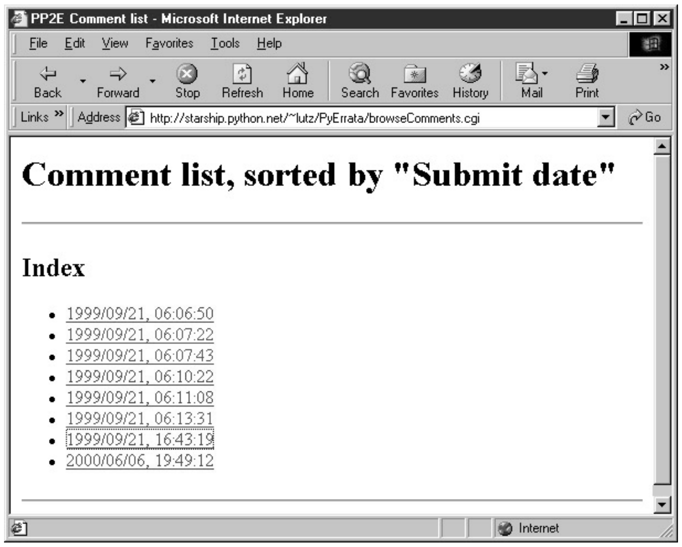
Figure 14-6. Browse comments, "With index" listing

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.[*]

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.

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

[*] 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.

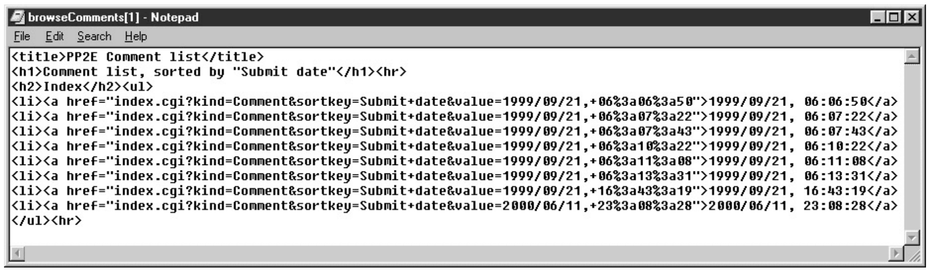

PP2E Comment list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.cgi Go
Internet

Comment list, sorted by "Submit date"

Index

- 1999/09/21, 06:06:50
- 1999/09/21, 06:07:22
- 1999/09/21, 06:07:43
- 1999/09/21, 06:10:22
- 1999/09/21, 06:11:08
- 1999/09/21, 06:13:31
- 1999/09/21, 16:43:19
- 2000/06/06, 19:49:12

Figure 14-7. Browse comments, "Index only" selection list



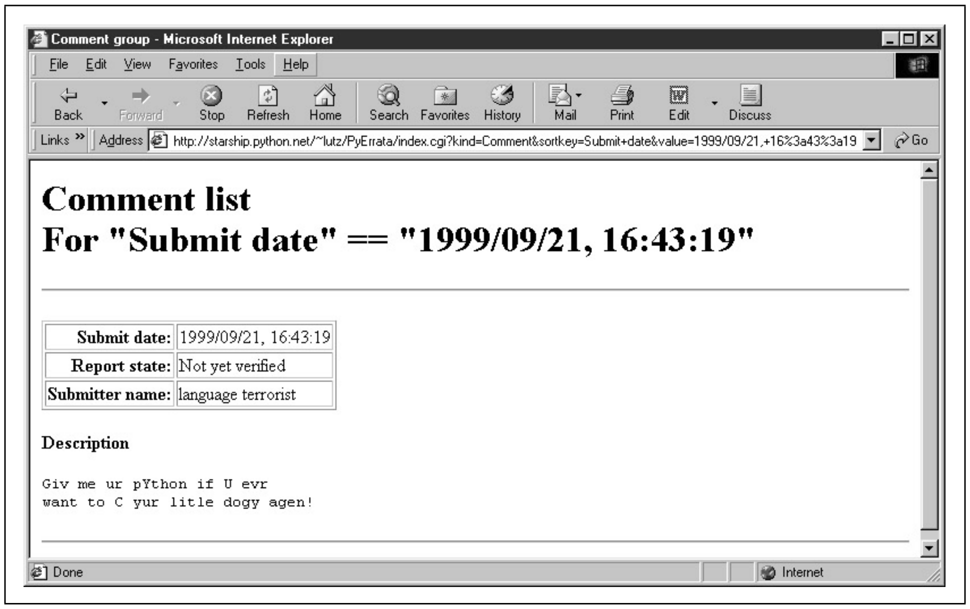
```
<title>PP2E Comment list</title>
<h1>Comment list, sorted by "Submit date"</h1><hr>
<h2>Index</h2><ul>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a06%3a50">1999/09/21, 06:06:50</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a07%3a22">1999/09/21, 06:07:22</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a07%3a43">1999/09/21, 06:07:43</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a10%3a22">1999/09/21, 06:10:22</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a11%3a08">1999/09/21, 06:11:08</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+06%3a13%3a31">1999/09/21, 06:13:31</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+16%3a43%3a19">1999/09/21, 16:43:19</a>
<li><a href="index.cgi?kind=Comment&sortkey=Submit+date&value=2000/06/11,+23%3a08%3a28">2000/06/11, 23:08:28</a>
</ul><hr>
```

Figure 14-8. PyErrata generated links code

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.

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.


Comment group - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address
Go
Done
Internet

http://starship.python.net/~lutz/PyErrata/index.cgi?kind=Comment&sortkey=Submit+date&value=1999/09/21,+16%3a43%3a19

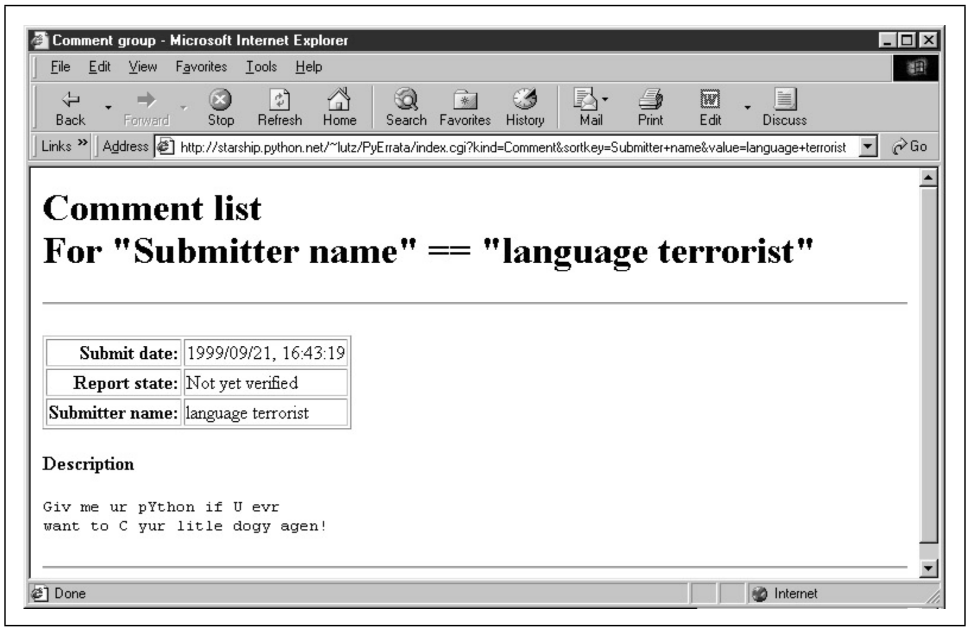
Comment list
For "Submit date" == "1999/09/21, 16:43:19"

Submit date:	1999/09/21, 16:43:19
Report state:	Not yet verified
Submitter name:	language terrorist

Description

```
Giv me ur pYthon if U evr
want to C yur litle dogy agen!
```

Figure 14-9. Browse comments, "Index only" link clicked


Comment group - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address
Go
Done
Internet

http://starship.python.net/~lutz/PyErrata/index.cgi?kind=Comment&sortkey=Submitter+name&value=language+terrorist

Comment list
For "Submitter name" == "language terrorist"

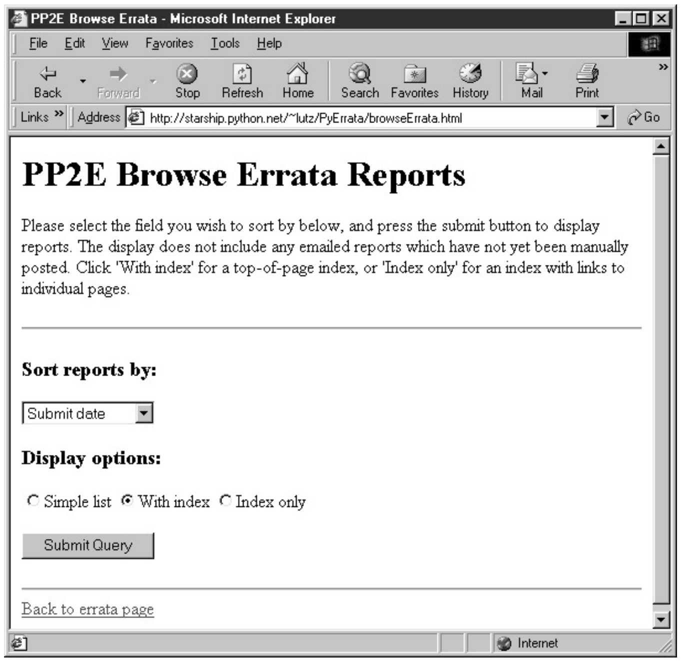
Submit date:	1999/09/21, 16:43:19
Report state:	Not yet verified
Submitter name:	language terrorist

Description

```
Giv me ur pYthon if U evr
want to C yur litle dogy agen!
```

Figure 14-10. Browse comments, "Index only" page

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.


PP2E Browse Errata - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseErrata.html Go
Internet

PP2E Browse Errata Reports

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.

Sort reports by:

Submit date

Display options:

Simple list [✓ With index] Index only

Submit Query

Back to errata page

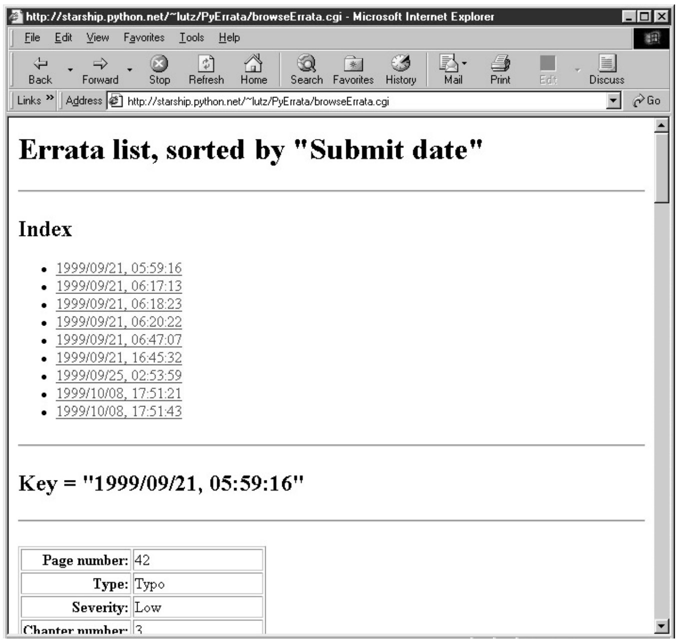
Figure 14-11. Browse errata, selection page

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.

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.

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.

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.


http://starship.python.net/~lutz/PyErrata/browseErrata.cgi - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/browseErrata.cgi Go

Errata list, sorted by "Submit date"

Index

- 1999/09/21, 05:59:16
- 1999/09/21, 06:17:13
- 1999/09/21, 06:18:23
- 1999/09/21, 06:20:22
- 1999/09/21, 06:47:07
- 1999/09/21, 16:45:32
- 1999/09/25, 02:53:59
- 1999/10/08, 17:51:21
- 1999/10/08, 17:51:43

Key = "1999/09/21, 05:59:16"

Page number:	42
Type:	Typo
Severity:	Low
Chapter number:	3

Figure 14-12. Browse errata, "With index" display

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:

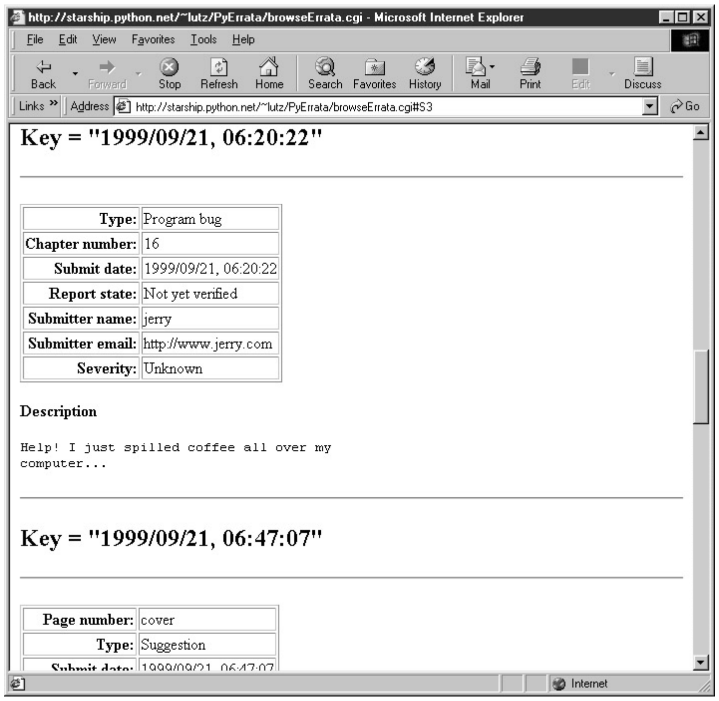
```
http://starship.python.net/~lutz/
      PyErrata/index.cgi?kind=Errata&sortkey=Chapter+number&value=16
```

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
```


http://starship.python.net/~lutz/PyErrata/browseErrata.cgi - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/browseErrata.cgi#S3 Go
Internet

Key = "1999/09/21, 06:20:22"

Type:	Program bug
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
Severity:	Unknown

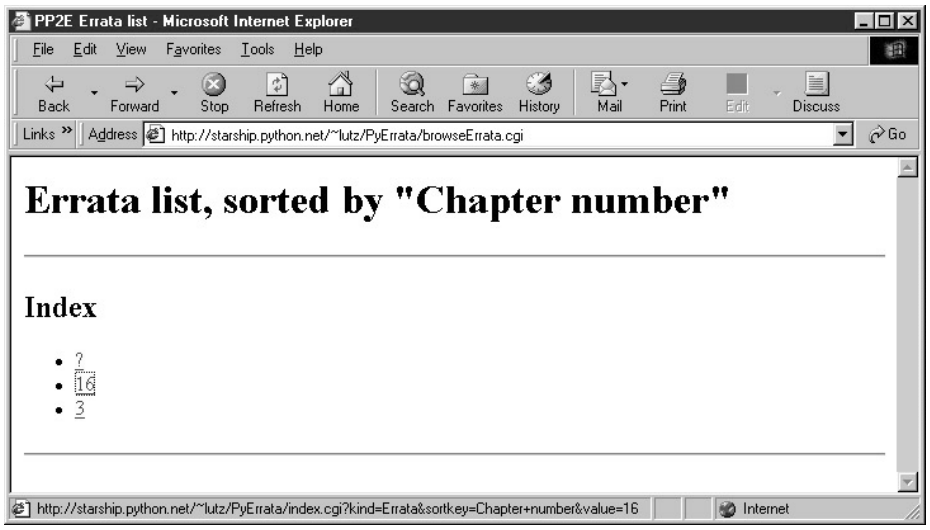
Description

```
Help! I just spilled coffee all over my
computer...
```

Key = "1999/09/21, 06:47:07"

Page number:	cover
Type:	Suggestion
Submit date:	1999/09/21, 06:47:07

Figure 14-13. Browse errata, report list

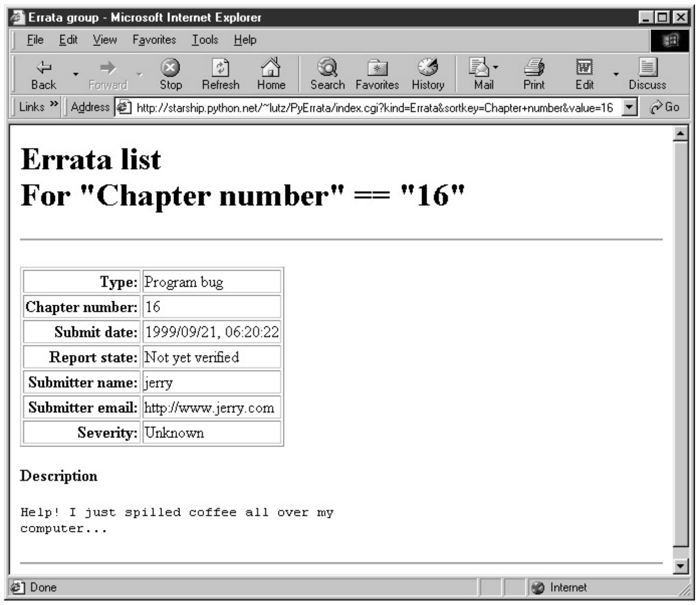

PP2E Errata list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/browseErrata.cgi Go
http://starship.python.net/~lutz/PyErrata/index.cgi?kind=Errata&sortkey=Chapter+number&value=16
Internet

Errata list, sorted by "Chapter number"

Index

- ?
- 16
- 3

Figure 14-14. Browse errata, "Index only" link page


Errata group - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address
Go
Done
Internet

http://starship.python.net/~lutz/PyErrata/index.cgi?kind=Errata&sortkey=Chapter+number&value=16

Errata list
For "Chapter number" == "16"

Type:	Program bug
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
Severity:	Unknown

Description

```
Help! I just spilled coffee all over my
computer...
```

Figure 14-15. Browse errata, "Index only" link clicked

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 the section "Submitting PyErrata Reports" later in this chapter.

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 split 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.

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
```

Example 14-4. PP2E\Internet\Cgi-Web\PyErrata\browseErrata.html (continued)

```
          <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.

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,
```

Example 14-6. PP2E\Internet\Cgi-Web\PyErrata\browse.py (continued)

```
# 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)
```

Example 14-6. PP2E\Internet\Cgi-Web\PyErrata\browse.py (continued)

```
        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 encoding

inputs = {'kind':'?', 'sortkey':'?', 'value':'?'}
for field in inputs.keys():
    if form.has_key(field):
        inputs[field] = cgi.escape(form[field].value)    # adds html encoding
```

Example 14-7. PP2E\Internet\Cgi-Web\PyErrata\index.cgi (continued)

```
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 `<A name=#`*`section`*`>` 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, *Server-Side Scripting*), 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.

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.

User Interface: Submitting Comment Reports

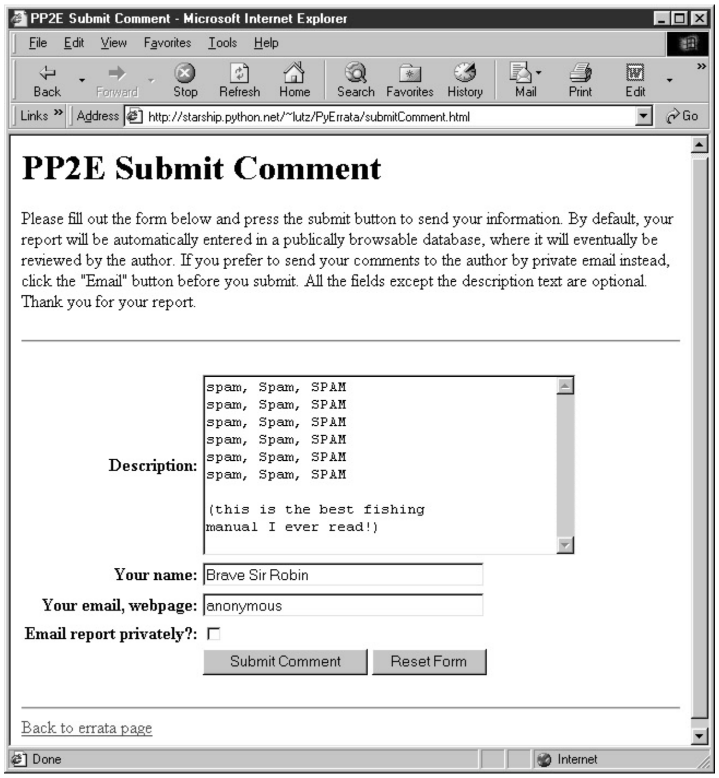
As we've seen, PyErrata supports two user functions: browsing the reports database and adding new reports to it. If you click the "General comment" link in the Submit section of the root page shown in Figure 14-2, you'll be presented with the comment submission page shown in Figure 14-16.

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.

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.

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.

,ch14.17404 Page 842 Wednesday, February 7, 2001 2:54 PM


PP2E Submit Comment - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Links Address http://starship.python.net/~lutz/PyErrata/submitComment.html Go

PP2E Submit Comment

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.

Description:

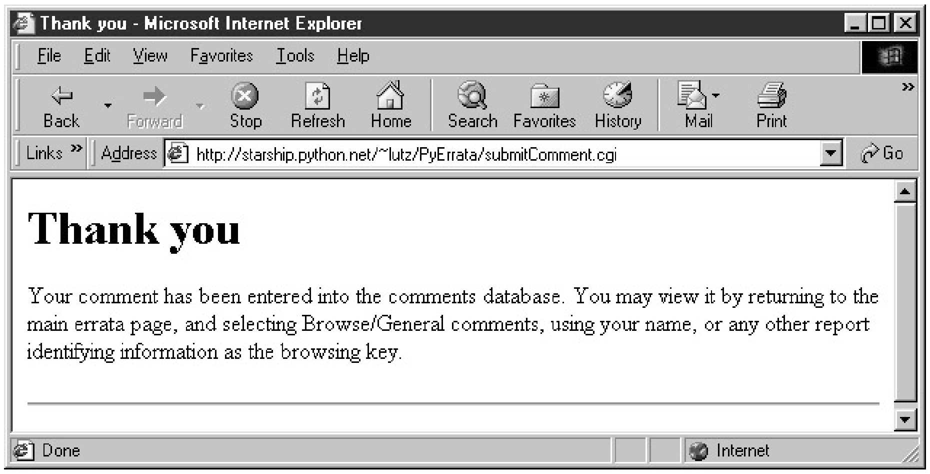
```
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM

(this is the best fishing
manual I ever read!)
```

Your name: Brave Sir Robin

Your email, webpage: anonymous

Email report privately?: ☐

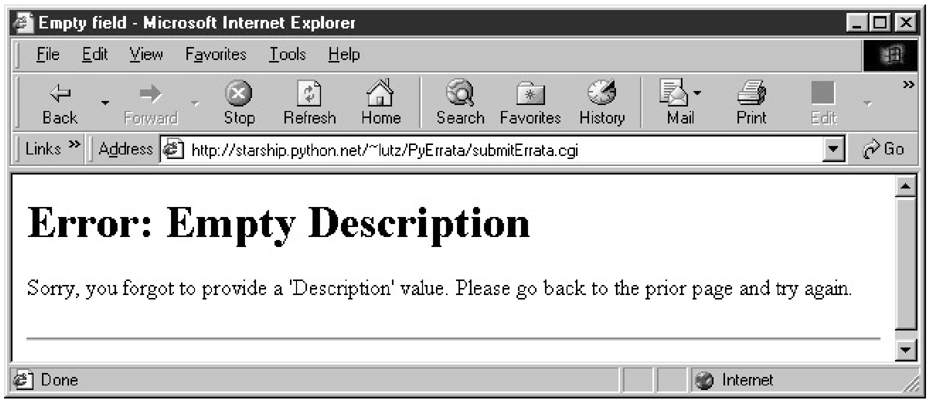
Figure 14-16. Submit comments, input page


Thank you - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/submitComment.cgi Go

Thank you

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.

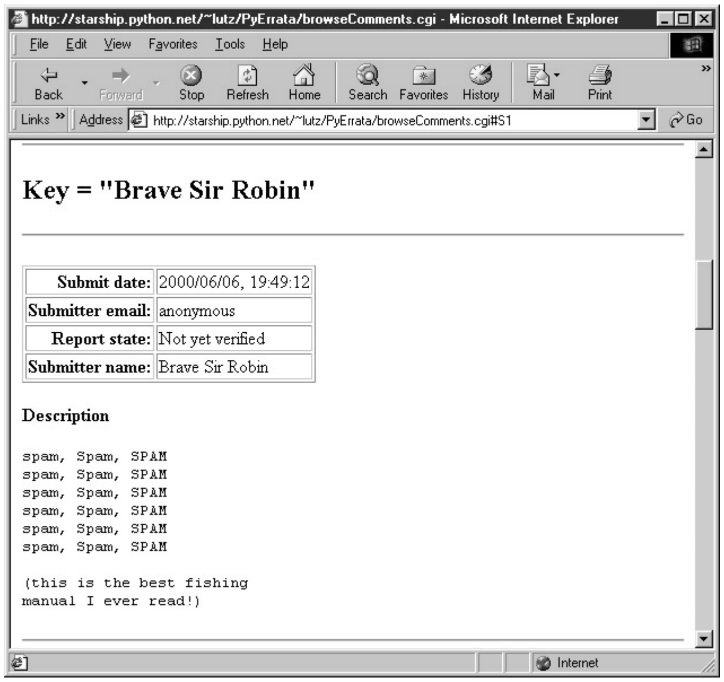
Figure 14-17. Submit comments, confirmation page


Empty field - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.cgi Go
Done
Internet

Error: Empty Description

Sorry, you forgot to provide a 'Description' value. Please go back to the prior page and try again.

Figure 14-18. Submit, missing field error page


http://starship.python.net/~lutz/PyErrata/browseComments.cgi - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseComments.cgi#S1 Go
Internet

Key = "Brave Sir Robin"

Submit date:	2000/06/06, 19:49:12
Submitter email:	anonymous
Report state:	Not yet verified
Submitter name:	Brave Sir Robin

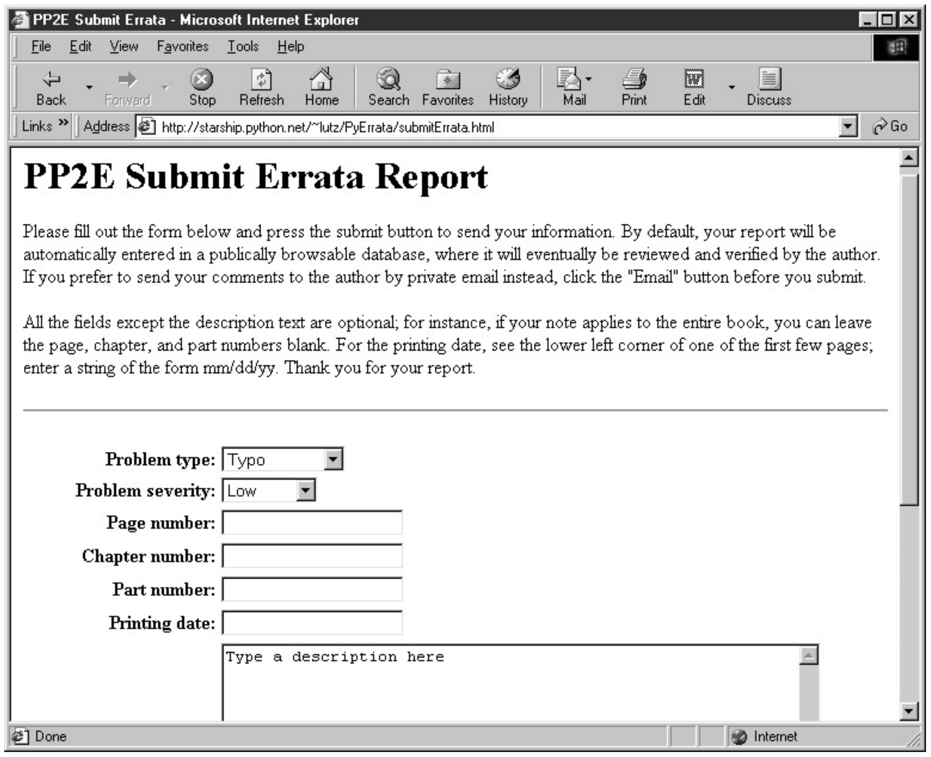
Description

```
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM
spam, Spam, SPAM

(this is the best fishing
manual I ever read!)
```

Figure 14-19. Submit comments, verifying result

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.


PP2E Submit Errata - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.html Go
Done
Internet

PP2E Submit Errata Report

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.

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.

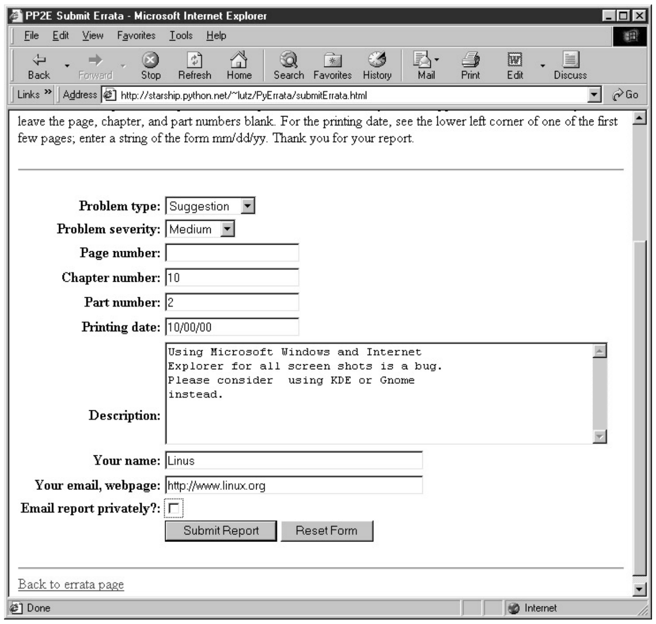
Problem type:	Typo
Problem severity:	Low
Page number:	
Chapter number:	
Part number:	
Printing date:	
	Type a description here

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.

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, not a comment.

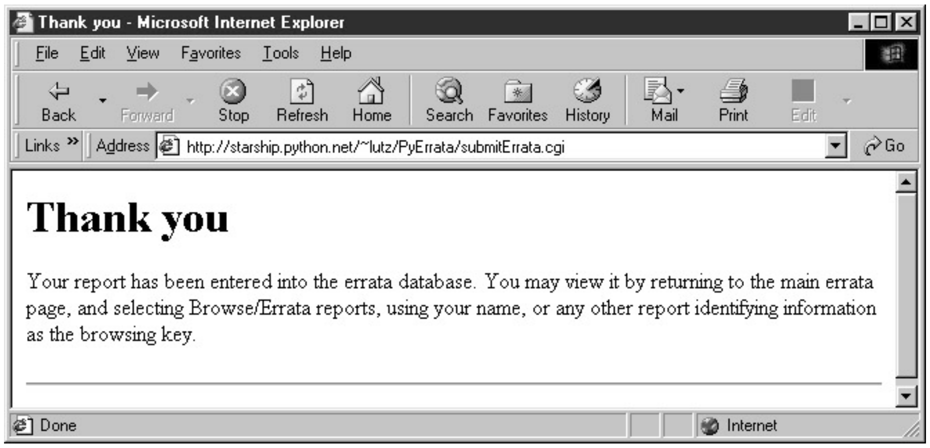
As before, we can verify a submission with the browse pages immediately after it has been confirmed. Let's bring up an index list page for submission dates and click on the new entry at the bottom (Figure 14-23). Our report is fetched from the errata database and displayed in a new page (Figure 14-24). 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


PP2E Submit Errata - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.html Go
Submit Report Reset Form
Back to errata page
Done Internet

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.

Problem type:	Suggestion
Problem severity:	Medium
Page number:	
Chapter number:	10
Part number:	2
Printing date:	10/00/00
Description:	Using Microsoft Windows and Internet Explorer for all screen shots is a bug. Please consider using KDE or Gnome instead.
Your name:	Linus
Your email, webpage:	http://www.linux.org
Email report privately?:	☐

Figure 14-21. Submit errata, input page (filled)


Thank you - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.cgi Go
Done Internet

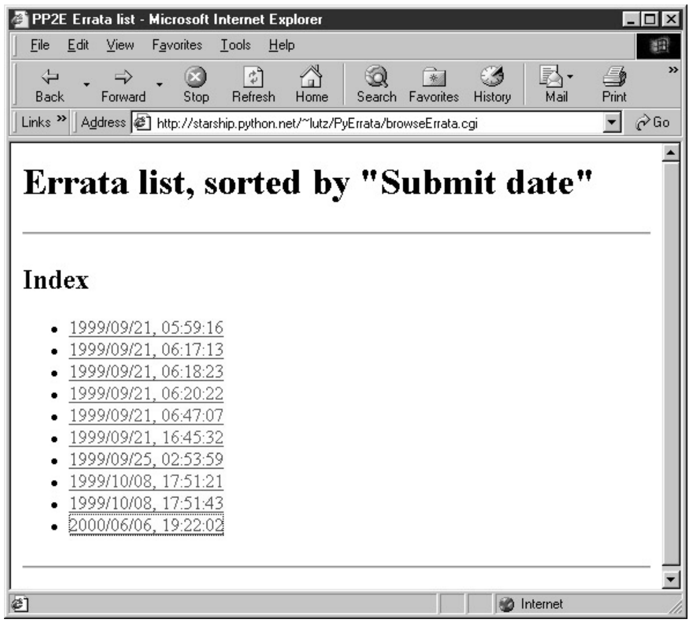
Thank you

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.

Figure 14-22. Submit errata, confirmation

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.

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,

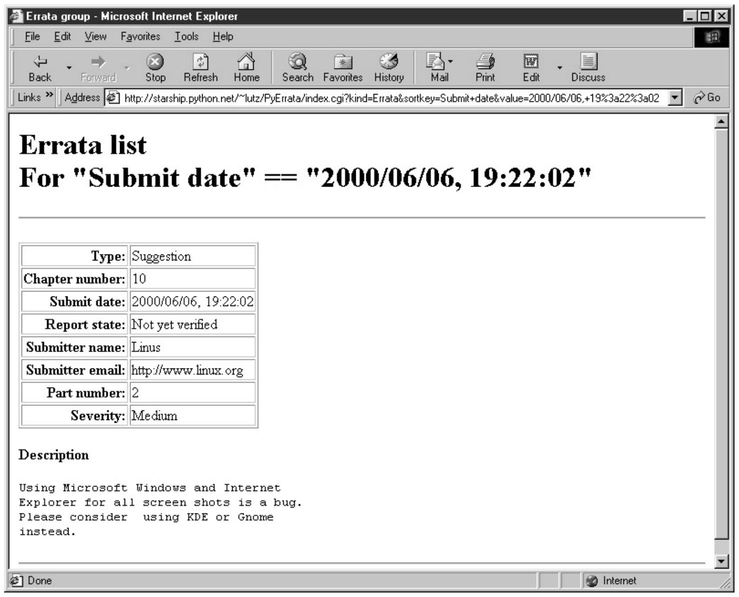

PP2E Errata list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links Address http://starship.python.net/~lutz/PyErrata/browseErrata.cgi
Go
Internet

Errata list, sorted by "Submit date"

Index

- 1999/09/21, 05:59:16
- 1999/09/21, 06:17:13
- 1999/09/21, 06:18:23
- 1999/09/21, 06:20:22
- 1999/09/21, 06:47:07
- 1999/09/21, 16:45:32
- 1999/09/25, 02:53:59
- 1999/10/08, 17:51:21
- 1999/10/08, 17:51:43
- 2000/06/06, 19:22:02

Figure 14-23. Submit errata, verify result (index)


Errata group - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/index.cgi?kind=Errata&sortkey=Submit+date&value=2000/06/06,+19%3a22%3a02
Go
Done
Internet

Errata list
For "Submit date" == "2000/06/06, 19:22:02"

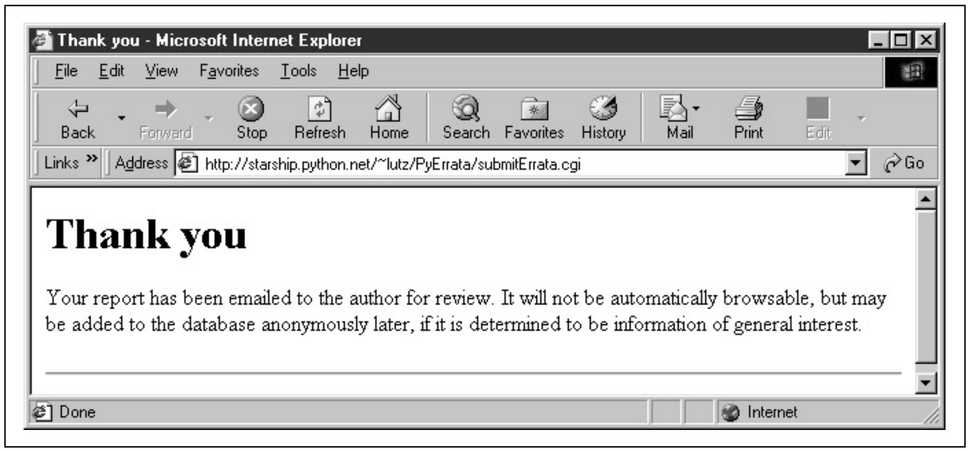
Type:	Suggestion
Chapter number:	10
Submit date:	2000/06/06, 19:22:02
Report state:	Not yet verified
Submitter name:	Linus
Submitter email:	http://www.linux.org
Part number:	2
Severity:	Medium

Description

```
Using Microsoft Windows and Internet
Explorer for all screen shots is a bug.
Please consider  using KDE or Gnome
instead.
```

Figure 14-24. Submit errata, verify result (record)

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.

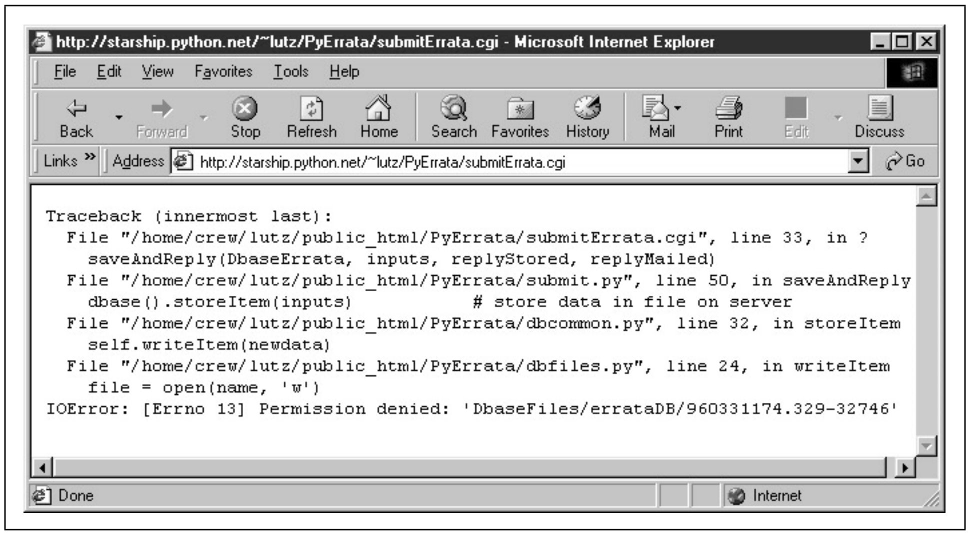

Thank you - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.cgi Go
Done
Internet

Thank you

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.

Figure 14-25. Submit errata, email mode confirmation

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`.


http://starship.python.net/~lutz/PyErrata/submitErrata.cgi - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Links Address http://starship.python.net/~lutz/PyErrata/submitErrata.cgi Go
Done
Internet

```
Traceback (innermost last):
  File "/home/crew/lutz/public_html/PyErrata/submitErrata.cgi", line 33, in ?
    saveAndReply(DbaseErrata, inputs, replyStored, replyMailed)
  File "/home/crew/lutz/public_html/PyErrata/submit.py", line 50, in saveAndReply
    dbase().storeItem(inputs)           # store data in file on server
  File "/home/crew/lutz/public_html/PyErrata/dbcommon.py", line 32, in storeItem
    self.writeItem(newdata)
  File "/home/crew/lutz/public_html/PyErrata/dbfiles.py", line 24, in writeItem
    file = open(name, 'w')
IOError: [Errno 13] Permission denied: 'DbaseFiles/errataDB/960331174.329-32746'
```

Figure 14-26. Submit errata, exception (need chmod 777 dir)

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">
  </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: within 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.

Implementation: Submitting Errata Reports

The top-level errata submission page in Figures 14-20 and 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
```

Example 14-10. PP2E\Internet\Cgi-Web\PyErrata\submitErrata.html (continued)

```
      </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()
```

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 `smptlib` module (discussed in Chapter 11, *Client-Side Scripting*) 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):                             # email data to author
    text = ''                                     # 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
```

Example 14-12. PP2E\Internet\Cgi-Web\PyErrata\submit.py (continued)

```
        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.

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 two database storage mediums—flat pickle files and shelves—as well as tools for synchronizing data access.

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, *Databases and Persistence*, is devoted exclusively to this topic and provides more in-depth coverage than we require here.* 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

* 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.

parallel, scripts must take care to ensure that database writes and reads are properly synchronized to avoid data corruption and incomplete records.

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.

Database Storage Structure

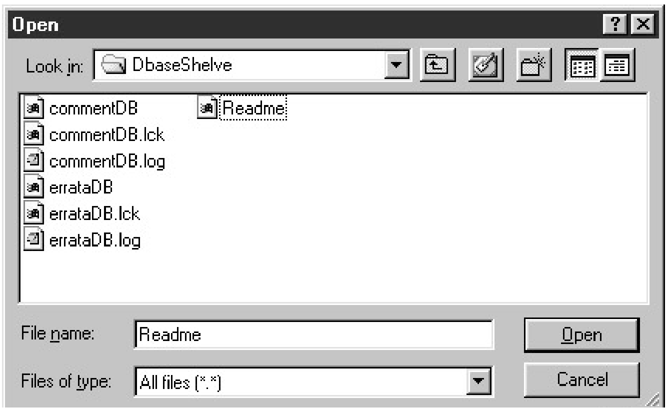
First of all, let's get a handle on what the system really stores. If you flip back to Figure 14-1, you'll notice that there are two top-level database directories: *DbaseShelve* (for the shelve mechanism) and *DbaseFiles* (for file-based storage). Each of these directories has unique contents.

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 shelve and log files as records are stored.

We'll explore the Python `shelve` module in more detail in the next part of this book, but the parts of 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
```



Look in: DbaseShelve

commentDB
commentDB.lck
commentDB.log
errataDB
errataDB.lck
errataDB.log
Readme

File name: Readme

Files of type: All files (*.*)

Figure 14-27. PyErrata shelve-based directory contents

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 shelve 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 exits, and must be explicitly opened.

Like dictionaries, each distinct value stored in a shelve 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 shelve, the system generates a unique key 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 shelve contain all the report information we're interested in, shelve keys need only be unique, not meaningful. Records are loaded by blindly iterating over the shelve's keys list.

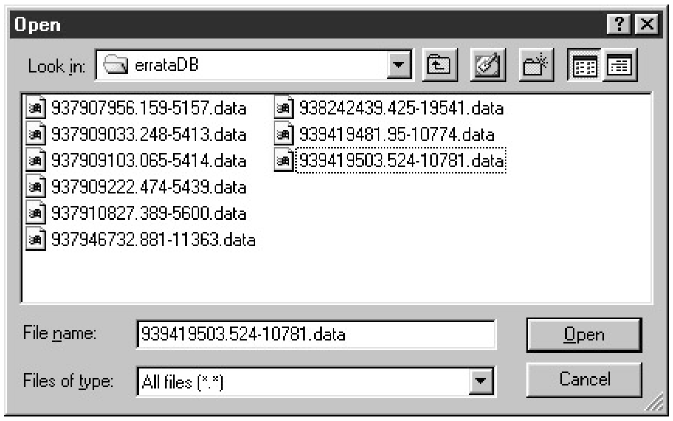
In addition to generating unique keys for records, shelves must accommodate *concurrent updates*. Because shelves are mapped to single files in the filesystem (here, `errataDB` and `commentDB`), we must synchronize all access to them in a potentially parallel process environment such as CGI scripting.

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 shelve database-storage scheme uses locks on the *.lck* files to make sure that writers (submit processes) gain exclusive access to the shelve 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 shelve.

Notice that we use a separate *.lck* file for locks, rather than locking the shelve file itself. In some systems, shelves are mapped to multiple files, and in others (e.g., GDBM), locks on the underlying shelve file are reserved for use by the DBM file-system itself. Using our own lock file subverts such reservations and is more portable among DBM implementations.

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.



```
Look in: errataDB

937907956.159-5157.data      938242439.425-19541.data
937909033.248-5413.data      939419481.95-10774.data
937909103.065-5414.data      939419503.524-10781.data
937909222.474-5439.data
937910827.389-5600.data
937946732.881-11363.data

File name:     939419503.524-10781.data
Files of type: All files (*.*)
```

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.* We'll

* 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.

study the `pickle` module in depth in Part IV of this book too, but its interfaces employed by PyErrata are simple as well:

```
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, *System Tools*, for a refresher on the `glob` module).

In a sense, this flat-file approach uses the filesystem as a shelve 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.

Database Switch

On to code listings. The first database module, shown in Example 14-13, simply selects between a file-based mechanism and shelve-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;
############################################################
```

Example 14-13. PP2E\Internet\Cgi-Web\PyErrata\dbswitch.py (continued)

```
#
# one directory per dbase, one flat pickle file per submit
#

from dbfiles import DbaseErrata, DbaseComment


#
# one shelve per dbase, one key per submit, with mutex update locks
#

# from dbshelve import DbaseErrata, DbaseComment
```

Storage-Specific Classes for Files and Shelves

The next two modules implement file- and shelve-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 define 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'
```

Example 14-14. PP2E\Internet\Cgi-Web\PyErrata\dbfiles.py (continued)

```
    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 later 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 doen'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 alock 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')
```

Example 14-15. PP2E\Internet\Cgi-Web\PyErrata\dbshelve.py (continued)

```
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'
```

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 and 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 assertion 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. This 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 and 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):            # returns a simple list
        reports = self.readTable()                    # ordered by field sort
        if field:
```

Example 14-16. PP2E\Internet\Cgi-Web\PyErrata\dbcommon.py (continued)

```
            reports.sort(lambda x, y, f=field: cmp(x[f], y[f]))
        return reports

    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 dict
        return keys, index
```

Mutual Exclusion for Shelves

We've at last reached the bottom of the PyErrata code hierarchy: code that encapsulates file locks for synchronizing shelve 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
```

Example 14-17. PP2E\Internet\Cgi-Web\PyErrata\Mutex\mutexcntl.py (continued)

```
        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)            # any number shared at once
        finally:                                    # but no exclusive actions
            self.unlockFile()                       # allow new writers to run
        return result

    def exclusiveAction(self, action, *args):
        self.lockFileWrite()                        # block if any other locks
        try:
            result = apply(action, args)            # no other actions overlap
        finally:
            self.unlockFile()                       # allow new readers/writers
        return result

    def logmsg(self, text):
        if not debugMutexCntl: return
        log = open(self.filename + '.log', 'a')       # append to the end
        log.write('%s\t%s\n' % (time.time(), text))   # output won't overwrite
        log.close()                                   # but it may intermingle

    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 the 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.

Using fcntl.flock to lock files

When we studied threads in Chapter 3, *Parallel System Tools*, 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 an 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)       # waits for lock if needed
    fcntl.flock(self.lock.fileno(), LOCK_UN)    # unlock for other processes
```

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 no 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 `threading` 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,[*] 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.

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 14-18 and 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')                      # open the lock file for fd
fcntl.flock(file.fileno(), LOCK_SH)               # block if a writer has lock
print os.getpid(), 'got read lock', time.time()   # any number of readers can run

time.sleep(3)
print 'lines so far:', os.popen('wc -l Shared.txt').read(),

print os.getpid(), 'unlocking\n'
fcntl.flock(file.fileno(), LOCK_UN)               # resume blocked writers now
```

In this simple test, locks on text file *test.lck* are used to synchronize read and write access to a text file appended by writers. The appended text file plays the role of PyErrata shelve databases, and the reader and writer scripts in Examples 14-18 and 14-19 stand in for its browse and submit script processes.

Example 14-19. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testwrite.py

```
#!/usr/bin/python

import os, fcntl, time
from FCNTL import LOCK_EX, LOCK_UN
```

[*] 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.

Example 14-19. PP2E\Internet\Cgi-Web\PyErrata\Mutex\testwrite.py (continued)

```
print os.getpid(), 'start writer', time.time()

file = open('test.lck', 'r')                      # open the lock file
fcntl.flock(file.fileno(), LOCK_EX)               # block if any read or write
print os.getpid(), 'got write lock', time.time()  # only 1 writer at a time

log = open('Shared.txt', 'a')
time.sleep(3)
log.write('%d Hello\n' % os.getpid())

print os.getpid(), 'unlocking\n'
fcntl.flock(file.fileno(), LOCK_UN)               # resume blocked read or write
```

To start a set of readers and writers running in parallel, Example 14-20 uses the Unix `fork`/`execl` call combination to launch program processes (both calls are described in Chapter 3).

Example 14-20. PP2E\Internet\Cgi-Web\PyErrata\Mutex\launch-test.py

```
#!/usr/bin/python
######################################################
# launch test program processes
# run with ./launch-test.py > launch-test.out
# try spawning reader before writer, then writer
# before reader--second process blocks till first
# unlocks in both cases; if launches 2 readers
# initially, both get lock and block writer; if
# launch 2 writers first then 2 readers, 2nd writer
# waits for first, both readers wait for both
# writers, and both readers get lock at same time;
# in test below, the first writer runs, then all
# readers run before any writer;  if readers are
# first, all run before any writer; (all on linux)
######################################################

import os

for i in range(1):
    if os.fork() == 0:
        os.execl("./testwrite.py")

for i in range(2):                     # copy this process
    if os.fork() == 0:                 # if in new child process
        os.execl("./testread.py")      # overlay with test program

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")
```

Example 14-20. PP2E\Internet\Cgi-Web\PyErrata\Mutex\launch-test.py (continued)

```
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:[*]

```
[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 a 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
```

[*] 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).

```
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. Such a sequence will avoid concurrent update problems.

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 14-21 and 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 14-21 and 14-22, because we'll run these scripts as ourself, 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
    def update(self):                                   # can run at same time
        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 follow 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
```

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 is 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. All 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 the domain of "quick and dirty" hackerage and aren't as robust as they could be. For instance, because 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 more 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.

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 14-24 and 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')
```

Display Tools

The scripts in Examples 14-26 and 14-27 produce raw dumps of each database structure's contents. Because the databases use pure Python storage mechanisms (pickles, shelves), 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 evolves. 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 Web:

```
[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...
```

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 listed 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 = ''                               # input reject reason text
    while 1:                                  # prepend to original desc
        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
```

Example 14-29. PP2E\Internet\Cgi-Web\PyErrata\AdminTools\verifyFiles.py (continued)

```
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 numbe
r': '', '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', 'Descri
ption': '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 if the system is ever abused by a frivolous user (including me, while writing examples for this book). The *shelve*-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 avoids 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.

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.

,ch14.17404 Page 880 Wednesday, February 7, 2001 2:54 PM

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*.

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- and 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 these 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 tools.

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 a 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.

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's 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;
```

Example 14-31. PP2E\Internet\PyErrata\AdminTools\dbaseindexed.py (continued)

```
# 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 fields
            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:
```

Example 14-31. PP2E\Internet\PyErrata\AdminTools\dbaseindexed.py (continued)

```
            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 lisy 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):
```

Example 14-31. PP2E\Internet\PyErrata\AdminTools\dbaseindexed.py (continued)

```
        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 s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
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': '2
000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Report state': 'Not y
et verified', 'Submitter name': 'Bob'}], 'Brian': [{'Submit date': '2000/06/13,
11:45:01', 'Page': 40, 'Submit mode': '', 'Report state': 'Not yet verified', 'S
ubmitter name': 'Brian'}]}
--------------------------------------------------------------------------------
new stuff
--------------------------------------------------------------------------------
['Bob', 'Brian']
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
--------------------------------------------------------------------------------
['Not yet verified']
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Brian'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 42, 'Submit mode': 'email', 'Rep
ort state': 'Not yet verified', 'Submitter name': 'Bob'}
--------------------------------------------------------------------------------
boundary cases
--------------------------------------------------------------------------------
{'Submit date': '2000/06/13, 11:45:01', 'Page': 38, 'Submit mode': '', 'Report s
tate': 'Not yet verified', 'Submitter name': 'Bob'}
{'Submit date': '2000/06/13, 11:45:01', 'Page': 40, 'Submit mode': '', 'Report s
tate': '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 `mutex` 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.