CHAPTER 15

The PyMailGUI Client

"Use the Source, Luke"

The preceding chapter introduced Python's client-side Internet tool set—the standard library modules available for email, FTP, network news, and more, from within a Python script. This chapter picks up where the last one left off and presents a complete client-side example—PyMailGUI, a Python program that sends, receives, composes, and parses Internet email messages.

Although the end result is a working program that you can actually use for your email, this chapter also has a few additional agendas worth noting before we get started:

Client-side scripting
: For one thing, PyMailGUI implements a full-featured GUI that runs on your machine, and communicates with your mail servers when necessary. As such, it is a network client program that further illustrates some of the preceding chapter's topics, and it will help us contrast server-side solutions introduced in the next chapter.

Code reuse
: Additionally, PyMailGUI ties together a number of the utility modules we've been writing in the book so far, and demonstrates the power of code reuse in the process—it uses a thread module to allow mail transfers to overlap in time, a set of mail modules to process message content and route it across networks, a window protocol module to handle icons, and so on. Moreover, it inherits the power of tools in the Python standard library, such as the `email` package; message construction and parsing, for example, is trivial here.

Programming in the large
: And finally, because PyMailGUI is a relatively large-scale program (at least as Python programs go), it shows by example some of the code structuring techniques that come in handy once we leave the realm of the small. Object-oriented programming (OOP) and modular design work well here to divide the system in smaller, self-contained units.

Ultimately, though, PyMailGUI serves to illustrate just how far the combination of GUIs, networking, and Python can take us. Like all Python programs, this system is *scriptable*—once you've learned its general structure, you can easily change it to work as you like, by modifying its source code. And like all Python programs, this one is *portable* across platforms—you can run it on any system with Python and a network connection, without having to change its code. Such advantages become automatic when your software is coded in an open source, portable, and readable language like Python.

Source Code Modules

This chapter is something of a self-study exercise. Because PyMailGUI is fairly large and mostly applies concepts we've already learned, we won't go into much detail about its actual code. Instead, it is listed for you to read on your own. I encourage you to study the source and comments and to run this program on your own to get a feel for its operation. Also, be sure to refer back to the modules we introduced earlier in the book and are reusing here, to gain a full understanding of the system. For reference, here are the major examples that will see new action in this chapter:

Example 14-21: PP3E.Internet.Email.mailtools (package)
: Server sends and receives, parsing, construction (Chapter 14)

Example 11-17: PP3E.Gui.Tools.threadtools.py
: Thread queue management for GUI callbacks (Chapter 11)

Example 11-13: PP3E.Gui.Tools.windows.py
: Border configuration for top-level window (Chapter 11)

Example 12-4: PP3E.Gui.TextEditor.textEditor.py
: Text widget used in mail view windows, and in some pop ups (Chapter 12)

Some of these modules in turn use additional examples we coded earlier, but that are not imported by PyMailGUI itself (`textEditor`, for instance, uses `guimaker` to create its windows and toolbar). We'll also be coding new modules here. The following new modules are intended to be useful in other programs:

`utilities.py`
: Various pop-up windows, written for general use

`messagecache.py`
: A cache that keeps track of mail already loaded

`wraplines.py`
: A utility for wrapping long lines of messages

`mailconfig.py`
: User configuration parameters—server names, fonts, and so on (augmented here)

Finally, the following are new modules coded in this chapter and are specific to the PyMailGUI program:

`SharedNames.py`
: Program-wide globals used by multiple files

`ViewWindows.py`
: The implementation of View, Write, Reply, and Forward message view windows

`ListWindows.py`
: The implementation of mail-server and local-file message list windows

`PyMailGuiHelp2.py`
: User-oriented help text, opened by the main window's bar button

`PyMailGui2.py`
: The main, top-level file of the program, run to launch the main window

All told, PythonMailGUI comprises the nine new modules in the preceding two lists and is composed of some 2,200 lines of source code (including comments, whitespace, and 530 lines of help text). This doesn't include the four other book examples in the previous list that are reused in PyMailGUI, which themselves constitute 1,600 additional lines.* This is the largest example we'll see in this book, but you shouldn't be deterred by its size. Because it uses modular and OOP techniques, the code is simpler than you may think:

- Python's modules allow us to divide the system into files that have a cohesive purpose, with minimal coupling between them—code is easier to locate and understand if your modules have a logical, self-contained structure.
- Python's OOP support allows us to factor code for reuse, and avoid redundancy—as you'll see, code is customized, not repeated, and the classes we will code reflect the actual components of the GUI to make them easy to follow.

For instance, the implementation of mail list windows is easy to read and change, because it has been factored into a common shared superclass, which is customized by subclasses for mail-server and save-file lists; since these are mostly just variations on a theme, most of the code appears in just one place. Similarly, the code that implements the message view window is a superclass shared by write, reply, and forward composition windows; subclasses simply tailor it for writing rather than viewing.

Although we'll deploy these techniques in the context of a mail processing program here, such techniques will apply to any nontrivial program you'll write in Python.

* And remember: you would have to multiply these line counts by a factor of four or five to get the equivalent in a language like C or C++. If you've done much programming, you probably recognize that the fact that we can implement a fairly full-featured mail processing program in roughly 2,000 lines of code speaks volumes about the power of the Python language and its libraries. For comparison, the original version of this program from the second edition of this book was just 725 lines in 3 new modules, but also was very limited—it did not support PyMailGUI2's attachments, thread overlap, local mail files, and so on.

To help get you started, the `PyMailGuiHelp2.py` module listed last in this chapter includes a help text string that describes how this program is used, as well as its major features. Experimenting with the system, while referring to its code, is probably the best and quickest way to uncover its secrets.

Why PyMailGUI?

PyMailGUI is a Python program that implements a client-side email processing user interface with the standard Tkinter GUI toolkit. It is presented both as an instance of Python Internet scripting and as a realistically scaled example that ties together other tools we've already seen, such as threads and Tkinter GUIs.

Like the `pymail` console-based program we wrote in Chapter 14, PyMailGUI runs entirely on your local computer. Your email is fetched from and sent to remote mail servers over sockets, but the program and its user interface run locally. As a result, PyMailGUI is called an email client: like `pymail`, it employs Python's client-side tools to talk to mail servers from the local machine. Unlike `pymail`, though, PyMailGUI is a full-featured user interface: email operations are performed with point-and-click operations and advanced mail processing such as attachments and save files is supported.

Like many examples presented in this text, PyMailGUI is a practical, useful program. In fact, I run it on all kinds of machines to check my email while traveling around the world teaching Python classes. Although PyMailGUI won't put Microsoft Outlook out of business anytime soon, it has two key pragmatic features that have nothing to do with email itself: portability and scriptability, which are attractive features in their own right and they merit a few additional words here.

It's portable

PyMailGUI runs on any machine with sockets and a Python with Tkinter installed. Because email is transferred with the Python libraries, any Internet connection that supports Post Office Protocol (POP) and Simple Mail Transfer Protocol (SMTP) access will do. Moreover, because the user interface is coded with Tkinter, PyMailGUI should work, unchanged, on Windows, the X Window System (Unix, Linux), and the Macintosh (classic and OS X).

Microsoft Outlook may be a more feature-rich package, but it has to be run on Windows, and more specifically, on a single Windows machine. Because it generally deletes email from a server as it is downloaded and stores it on the client, you cannot run Outlook on multiple machines without spreading your email across all those machines. By contrast, PyMailGUI saves and deletes email only on request, and so it is a bit friendlier to people who check their email in an ad hoc fashion on arbitrary computers.

It's scriptable

PyMailGUI can become anything you want it to be because it is fully programmable. In fact, this is the real killer feature of PyMailGUI and of open source software like Python in general—because you have full access to PyMailGUI's source code, you are in complete control of where it evolves from here. You have nowhere near as much control over commercial, closed products like Outlook; you generally get whatever a large company decided you need, along with whatever bugs that company might have introduced.

As a Python script, PyMailGUI is a much more flexible tool. For instance, we can change its layout, disable features, and add completely new functionality quickly by changing its Python source code. Don't like the mail-list display? Change a few lines of code to customize it. Want to save and delete your mail automatically as it is loaded? Add some more code and buttons. Tired of seeing junk mail? Add a few lines of text processing code to the load function to filter spam. These are just a few examples. The point is that because PyMailGUI is written in a high-level, easy-to-maintain scripting language, such customizations are relatively simple, and might even be fun.

At the end of the day, because of such features, this is a realistic Python program that I actually *use*—both as a primary email tool and as a fallback option when my ISP's webmail system goes down (which, as I mentioned in the prior chapter, has a way of happening at the worst possible times).* Python scripting is an enabling skill to have.

It's also worth mentioning that PyMailGUI achieves its portability and scriptability, and implements a full-featured email interface along the way, in roughly 2,200 lines of program code. It may not have all the bells and whistles of some commercial products, but the fact that it gets as close as it does in so few lines of code is a testament to the power of both the Python language and its libraries.

Running PyMailGUI

Of course, to script PyMailGUI on your own, you'll need to be able to run it. PyMailGUI requires only a computer with some sort of Internet connectivity (a PC with a broadband or dial-up account will do) and an installed Python with the Tkinter extension enabled. The Windows port of Python has this capability, so Windows PC users should be able to run this program immediately by clicking its icon.

Two notes on running the system: first, you'll want to change the file *mailconfig.py* in the program's source directory to reflect your account's parameters, if you wish to send or receive mail from a live server; more on this as we interact with the system.

* In fact, my ISP's webmail send system went down the very day I had to submit this edition of the book to my publisher! No worries—I fired up PyMailGUI and used it to send the book as attachment files through a different server. In a sense, this book submitted itself.

Second, you can still experiment with the system without a live Internet connection—for a quick look at message view windows, use the main window's Open buttons to open saved-mail files stored in the program's *SavedMail* directory. In fact, the PyDemos launcher script at the top of the book's examples directory forces PyMailGUI to open saved-mail files by passing filenames on the command line.

Presentation Strategy

PyMailGUI is easily the largest program in this book, but it doesn't introduce many library interfaces that we haven't already seen in this book. For instance:

- The PyMailGUI interface is built with Python's Tkinter, using the familiar listboxes, buttons, and text widgets we met earlier.
- Python's `email` package is applied to pull-out headers, text, and attachments of messages, and to compose the same.
- Python's POP and SMTP library modules are used to fetch, send, and delete mail over sockets.
- Python threads, if installed in your Python interpreter, are put to work to avoid blocking during potentially overlapping, long-running mail operations.

We're also going to reuse the `TextEditor` object we wrote in Chapter 12 to view and compose messages, the `mailtools` package's tools we wrote in Chapter 14 to load and delete mail from the server, and the `mailconfig` module strategy introduced in Chapter 14 to support end-user settings. PyMailGUI is largely an exercise in combining existing tools.

On the other hand, because this program is so long, we won't exhaustively document all of its code. Instead, we'll begin by describing how PyMailGUI works from an end user's perspective—a brief demo of its windows in action. After that, we'll list the system's new source code modules without many additional comments, for further study.

Like most of the longer case studies in this book, this section assumes that you already know enough Python to make sense of the code on your own. If you've been reading this book linearly, you should also know enough about Tkinter, threads, and mail interfaces to understand the library tools applied here. If you get stuck, you may wish to brush up on the presentation of these topics earlier in the book.

New in This Edition

The 2.1 version of PyMailGUI presented in this third edition of the book is a complete rewrite of the 1.0 version of the prior edition. The main script in the second edition's version was only some 500 lines long, and was really something of a toy or prototype, written mostly to serve as a book example. In this edition, PyMailGUI is a much more realistic and full-featured program that can be used for day-to-day email

processing. It has grown to 2,200 source lines (3,800 including related modules that are reused). Among its new weapons are these:

- MIME multipart mails with attachments may be both viewed and composed.
- Mail transfers are no longer blocking, and may overlap in time.
- Mail may be saved and processed offline from a local file.
- Message parts may now be opened automatically within the GUI.
- Multiple messages may be selected for processing in list windows.
- Initial downloads fetch mail headers only; full mails are fetched on request.
- View window headers and list window columns are configurable.
- Deletions are performed immediately, not delayed until program exit.
- Most server transfers report their progress in the GUI.
- Long lines are intelligently wrapped in viewed and quoted text.
- Fonts and colors in list and view windows may be configured by the user.
- Authenticating SMTP mail-send servers that require login are supported.
- Sent messages are saved in a local file, which may be opened in the GUI.
- View windows intelligently pick a main text part to be displayed.
- Already fetched mail headers and full mails are cached for speed.
- Date strings and addresses in composed mails are formatted properly.
- View windows now have quick-access buttons for attachments/parts (2.1).
- Inbox out-of-sync errors are detected on deletes, and on index and mail loads (2.1).
- Save-mail file loads and deletes are threaded, to avoid pauses for large files (2.1).

The last three items on this list were added in version 2.1; the rest were part of the 2.0 rewrite. Some of these changes were made simple by growth in standard library tools (e.g., support for attachments is straightforward with the new `email` package), but most represent changes in PyMailGUI itself. There have also been a few genuine fixes: addresses are parsed more accurately, and date and time formats in sent mails are now standards conforming, because these tasks use new tools in the `email` package.

Although there is still room for improvement (see the list at the end of this chapter), the program provides a full-featured interface, represents the most substantial example in this book, and serves to demonstrate a realistic application of the Python language. As its users often attest, Python may be fun to work with, but it's also useful for writing practical and nontrivial software.

A PyMailGUI Demo

PyMailGUI is a multiwindow interface. It consists of the following:

- A main mail-server list window opened initially, for online mail processing
- One or more mail save-file list windows for offline mail processing
- One or more mail-view windows for viewing and editing messages
- Text editor windows for displaying the system's source code
- Nonblocking busy state pop-up dialogs
- Assorted pop-up dialogs for opened message parts, help, and more

Operationally, PyMailGUI runs as a set of parallel threads, which may overlap in time: one for each active server transfer, and one for each active offline save file load or deletion. PyMailGUI supports mail save files, automatic saves of sent messages, configurable fonts and colors, viewing and adding attachments, main message text extraction, and much more.

To make this case study easier to understand, let's begin by seeing what PyMailGUI actually does—its user interaction and email processing functionality—before jumping into the Python code that implements that behavior. As you read this part, feel free to jump ahead to the code listings that appear after the screenshots, but be sure to read this section too; this is where some subtleties of PyMailGUI's design are explained. After this section, you are invited to study the system's Python source code listings on your own for a better and more complete explanation than can be crafted in English.

Getting Started

PyMailGUI is a Python/Tkinter program, run by executing its top-level script file, *PyMailGui.py*. Like other Python programs, PyMailGUI can be started from the system command line by clicking on its filename icon in a file explorer interface, or by pressing its button in the PyDemos or PyGadgets launcher bar. However it is started, the first window PyMailGUI presents is shown in Figure 15-1. Notice the "PY" window icon: this is the handiwork of window protocol tools we wrote earlier in this book.

This is the PyMailGUI main window—every operation starts here. It consists of:

- A help button (the bar at the top)
- A clickable email list area for fetched emails (the middle section)
- A button bar at the bottom for processing messages selected in the list area

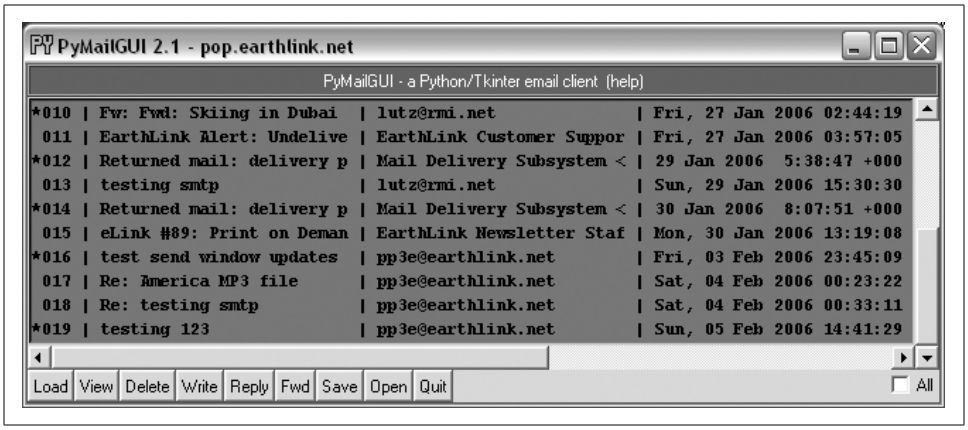

Figure 15-1. PyMailGUI main server list window

In normal operation, users load their email, select an email from the list area by clicking on it, and press a button at the bottom to process it. No mail messages are shown initially; we need to first load them with the Load button—a simple password input dialog is displayed, a busy dialog appears that counts down message headers being loaded to give a status indication, and the index is filled with messages ready to be selected.

PyMailGUI's list windows, such as the one in Figure 15-1, display mail header details in fixed-width columns, up to a maximum size. Mails with attachments are prefixed with a "*" in mail index list windows, and fonts and colors in PyMailGUI windows may be customized by the user in the `mailconfig` configuration file. You can't tell in this black-and white book, but by default, mail index lists are Indian red, view windows are a shade of purple, pop-up PyEdit windows are light cyan, and help is steel blue; you can change most of these as you like (see Example 9-11 for help with color definition strings).

List windows allow multiple messages to be selected at once—the action selected at the bottom of the window is applied to all selected mails. For instance, to view many mails, select them all and press View; each will be fetched and displayed in its own view window. Use Ctrl-Click and Shift-Click to select more than one (the standard Windows multiple selection operations apply—try it).

Before we go any further, though, let's press the help bar at the top of the list window in Figure 15-1 to see what sort of help is available; Figure 15-2 shows the help window popup that appears.

The main part of this window is simply a block of text in a scrolled-text widget, along with two buttons at the bottom. The entire help text is coded as a single triple-quoted string in the Python program. We could get fancier and spawn a web browser

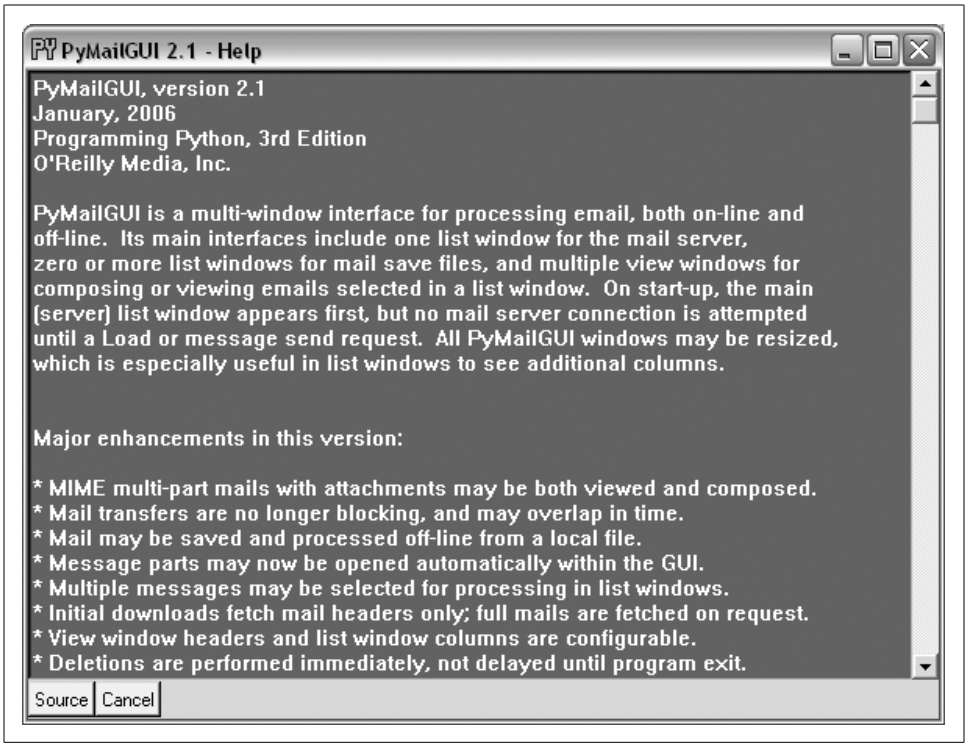

Figure 15-2. PyMailGUI help popup

to view HTML-formatted help, but simple text does the job here.* The Cancel button makes this nonmodal (i.e., nonblocking) window go away; more interestingly, the Source button pops up PyEdit text editor viewer windows for all the source files of PyMailGUI's implementation; Figure 15-3 captures one of these (there are many—this is intended as a demonstration, not as a development environment). Not every program shows you its source code, but PyMailGUI follows Python's open source motif.

When a message is selected in the mail list window, PyMailGUI downloads its full text (if it has not yet been downloaded in this session), and an email viewer window appears, as captured in Figure 15-4. View windows are built in response to actions in list windows; this is described next.

* The standard library's `webbrowser` module would help for HTML-based help. Actually, the help display started life even less fancy: it originally displayed help text in a standard information box pop up, generated by the Tkinter `showinfo` call used earlier in the book. This worked fine on Windows (at least with a small amount of help text), but it failed on Linux because of a default line-length limit in information pop-up boxes; lines were broken so badly as to be illegible.

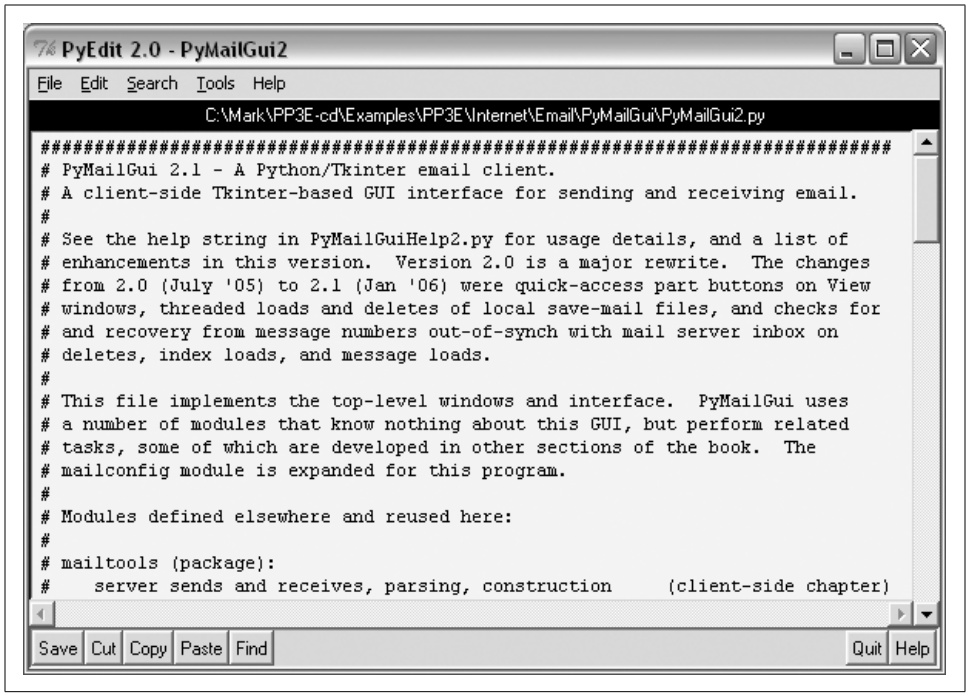

Figure 15-3. PyMailGUI source code viewer window

- The top portion consists of action buttons ("Parts" to list all message parts, "Split" to save and open parts using a selected directory, and "Cancel" to remove this nonmodal window) along with a section for displaying email header lines ("From:", "To:", and so on).
- In the middle, a row of quick-access buttons for opening message parts, including attachments, appears. When clicked, PyMailGUI opens known and generally safe parts according to their type (media types may open in a web browser, text parts in PyEdit, Windows document types per the Windows Registry, and so on).
- The bulk of this window is just another reuse of the `TextEditor` class object we wrote in Chapter 12 for the PyEdit program—PyMailGUI simply attaches an instance of `TextEditor` to every view and compose window in order to get a full-featured text editor component for free. In fact, much on this window is implemented by `TextEditor`, not by PyMailGUI.

For instance, if we pick the Tools menu of the text portion of this window and select its Info entry, we get the standard PyEdit `TextEditor` object's file text statistics box—the same popup we'd get in the standalone PyEdit text editor and in the PyView image view programs we wrote in Chapter 12 (see Figure 15-5).

In fact, this is the third reuse of `TextEditor` in this book: PyEdit, PyView, and now PyMailGUI all present the same text-editing interface to users, simply because they all use the same `TextEditor` object and code. PyMailGUI both attaches instances of this class for mail viewing and editing, and pops up instances for source-code viewing. For mail views, PyMailGUI customizes text fonts and colors per its own configuration module.

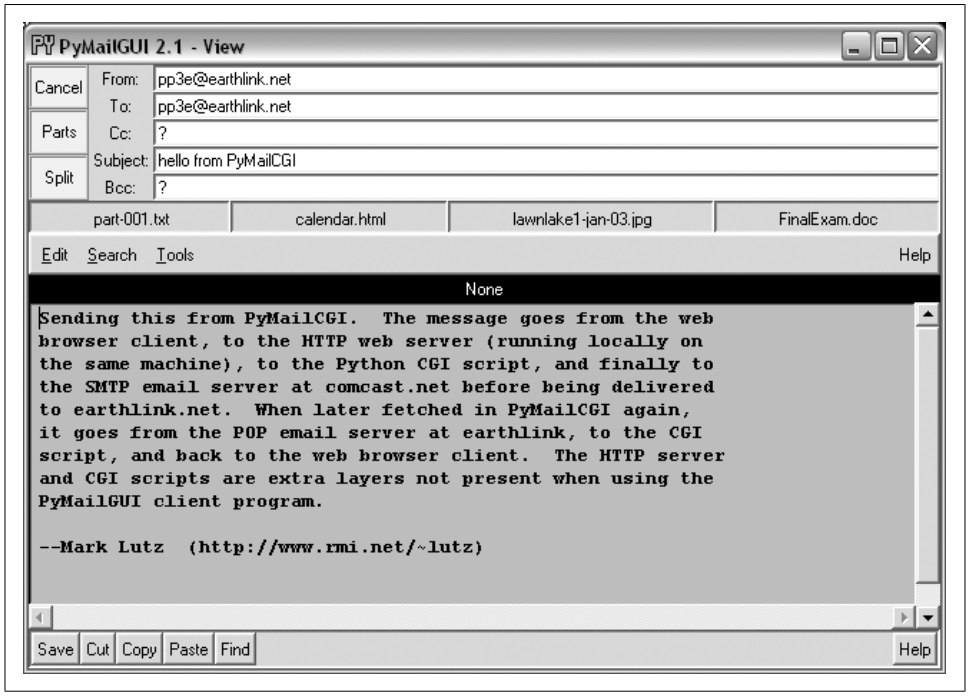

Figure 15-4. PyMailGUI view window

To display email, PyMailGUI inserts its text into an attached `TextEditor` object; to compose email, PyMailGUI presents a `TextEditor` and later fetches all its text to ship over the Net. Besides the obvious simplification here, this code reuse makes it easy to pick up improvements and fixes—any changes in the `TextEditor` object are automatically inherited by PyMailGUI, PyView, and PyEdit. In the current version, for instance, PyMailGUI supports edit undo and redo, just because PyEdit now does, too.

Loading Mail

Now, let's go back to the PyMailGUI main server list window, and click the Load button to retrieve incoming email over the POP protocol. PyMailGUI's load function gets account parameters from the `mailconfig` module listed later in this chapter,

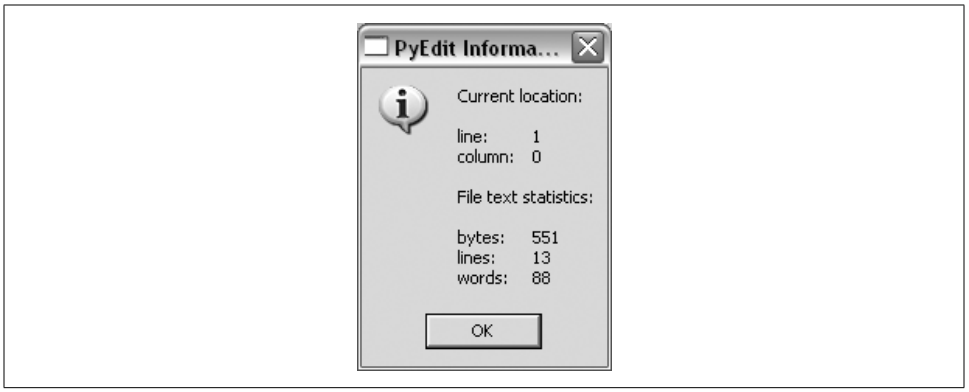

Figure 15-5. PyMailGUI attached PyEdit info box

so be sure to change this file to reflect your email account parameters (i.e., server names and usernames) if you wish to use PyMailGUI to read your own email.

The account password parameter merits a few extra words. In PyMailGUI, it may come from one of two places:

Local file
: If you put the name of a local file containing the password in the `mailconfig` module, PyMailGUI loads the password from that file as needed.

Popup dialog
: If you don't put a password filename in `mailconfig` (or if PyMailGUI can't load it from the file for whatever reason), PyMailGUI will instead ask you for your password anytime it is needed.

Figure 15-6 shows the password input prompt you get if you haven't stored your password in a local file. Note that the password you type is not shown—a `show='*'` option for the `Entry` field used in this popup tells Tkinter to echo typed characters as stars (this option is similar in spirit to both the `getpass` console input module we met earlier in the prior chapter, and an HTML `type=password` option we'll meet in a later chapter). Once entered, the password lives only in memory on your machine; PyMailGUI itself doesn't store it anywhere in a permanent way.

Also notice that the local file password option requires you to store your password unencrypted in a file on the local client computer. This is convenient (you don't need to retype a password every time you check email), but it is not generally a good idea on a machine you share with others; leave this setting blank in `mailconfig` if you prefer to always enter your password in a popup.

Once PyMailGUI fetches your mail parameters and somehow obtains your password, it will next attempt to pull down the header text of all your incoming email from your inbox on your POP email server. On subsequent loads, only newly arrived mails are loaded, if any.

Figure 15-6. PyMailGUI password input dialog

To save time, PyMailGUI fetches message header text only to populate the list window. The full text of messages is fetched later only when a message is selected for viewing or processing, and then only if the full text has not yet been fetched during this session. PyMailGUI reuses the load-mail tools in the `mailtools` module of Chapter 14 to fetch message header text, which in turn uses Python's standard `poplib` module to retrieve your email.

Threading Model

Ultimately, mail fetches run over sockets on relatively slow networks. While the download is in progress, the rest of the GUI remains active—you may compose and send other mails at the same time, for instance. To show its progress, the nonblocking dialog of Figure 15-7 is displayed when the mail index is being fetched.

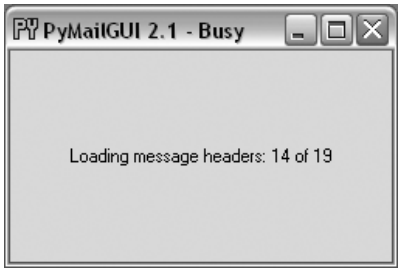

Figure 15-7. Nonblocking progress indicator: Load

In general, all server transfers display such dialogs. Figure 15-8 shows the busy dialog displayed while a full text download of five selected and uncached mails is in progress, in response to a View action. After this download finishes, all five pop up in view windows.

Such server transfers, and other long-running operations, are run in threads to avoid blocking the GUI. They do not disable other actions from running in parallel, as long as those actions would not conflict with a currently running thread. Multiple mail fetches and sends can overlap in time, for instance, and can run in parallel with the GUI itself—the GUI responds to moves, redraws, and resizes during the transfers.

On systems without threads, PyMailGUI instead goes into a blocked state during such long-running operations (it stubs out the thread-spawn operation to perform a simple function call). Because the GUI is essentially dead without threads, covering

Figure 15-8. Nonblocking progress indicator: View

and uncovering the GUI during a mail load on such platforms will erase or otherwise distort its contents. Threads are enabled by default on most platforms that Python run (including Windows), so you probably won't see such oddness on your machine.

One implementation note: as we learned earlier in this book, only the thread that creates windows should generally update them. As a result, PyMailGUI takes care to not do anything related to the user interface within threads that load, send, or delete email. Instead, the main GUI thread continues responding to user interface events and updates, and uses a timer-based event to watch a queue for exit callbacks to be added by threads, using thread tools implemented earlier in the book. Upon receipt, the GUI thread pulls the callback off the queue and dispatches it to modify the GUI (e.g., to display a fetched message, update the mail index list, or close an email composition window).

Load Server Interface

Because the load operation is really a socket operation, PyMailGUI automatically connects to your email server using whatever connectivity exists on the machine on which it is run. For instance, if you connect to the Net over a modem and you're not already connected, Windows automatically pops up the standard connection dialog. On a broadband connection, the interface to your email server is normally automatic.

After PyMailGUI finishes loading your email, it populates the main window's scrolled listbox with all of the messages on your email server and automatically scrolls to the most recently received message. Figure 15-9 shows what the main window looks like after resizing; the text area in the middle grows and shrinks with the window.

Technically, the Load button fetches all your mail's header text the first time it is pressed, but it fetches only newly arrived email headers on later presses. PyMailGUI keeps track of the last email loaded, and requests only higher email numbers on later loads. Already loaded mail is kept in memory, in a Python list, to avoid the cost of

downloading it again. PyMailGUI does not delete email from your server when it is loaded; if you really want to not see an email on a later load, you must explicitly delete it.

Entries in the main list show just enough to give the user an idea of what the message contains—each entry gives the concatenation of portions of the message's "Subject:", "From:", "Date:", and other header lines, separated by | characters and prefixed with the message's POP number (e.g., there are 19 emails in this list). Columns are aligned by determining the maximum size needed for any entry, up to a fixed maximum, and the set of headers displayed can be configured in the `mailconfig` module. Use the horizontal scroll or expand the window to see additional header details.

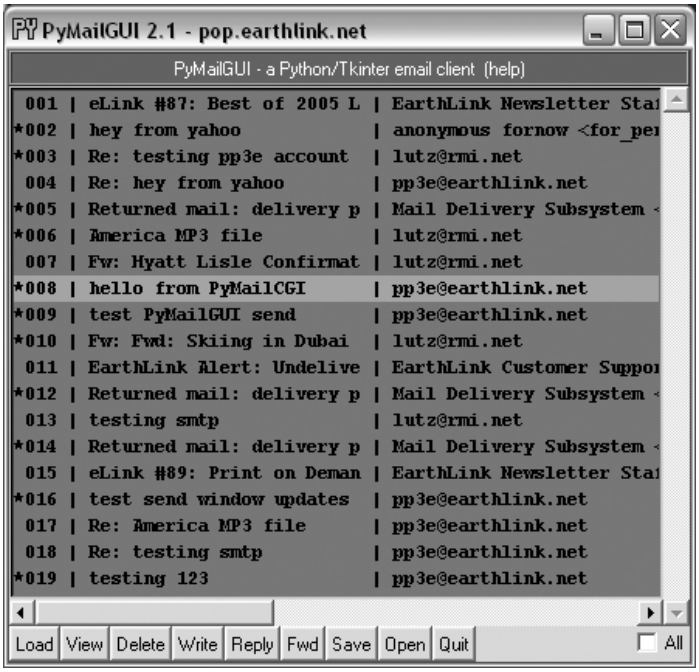

Figure 15-9. PyMailGUI main window resized

As we've seen, a lot of magic happens when downloading email—the client (the machine on which PyMailGUI runs) must connect to the server (your email account machine) over a socket and transfer bytes over arbitrary Internet links. If things go wrong, PyMailGUI pops up standard error dialog boxes to let you know what happened. For example, if you typed an incorrect username or password for your account (in the `mailconfig` module or in the password pop up), you'll see the message in Figure 15-10. The details displayed here are just the Python exception type and exception data.

Figure 15-10. PyMailGUI invalid password error box

Offline Processing with Save and Open

To save mails in a local file for offline processing, select the desired messages in any mail list window and press the Save action button (any number of messages may be selected for saving). A standard file-selection dialog appears, like that in Figure 15-11, and the mails are saved to the end of the chosen text file.

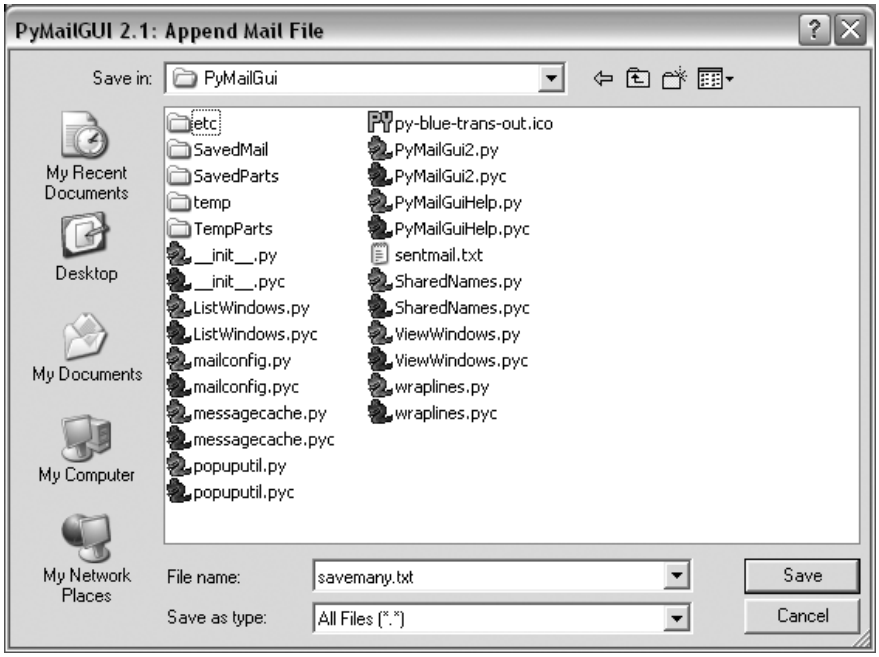

Figure 15-11. Save mail selection dialog

To view saved emails later, select the Open action at the bottom of any list window and pick your save file in the selection dialog. A new mail index list window appears for the save file and it is filled with your saved messages eventually—there may be a slight delay for large save files, because of the work involved. PyMailGUI runs file

loads and deletions in threads to avoid blocking the rest of the GUI; these threads can overlap with operations on other open save-mail files, server transfer threads, and the GUI at large.

While a mail save file is being loaded in a parallel thread, its window title is set to "Loading..." as a status indication; the rest of the GUI remains active during the load (you can fetch and delete server messages, view mails in other files, write new messages, and so on). The window title changes to the loaded file's name after the load is finished. Once filled, a message index appears in the save file's window, like the one captured in Figure 15-12 (this window also has three mails selected for processing).

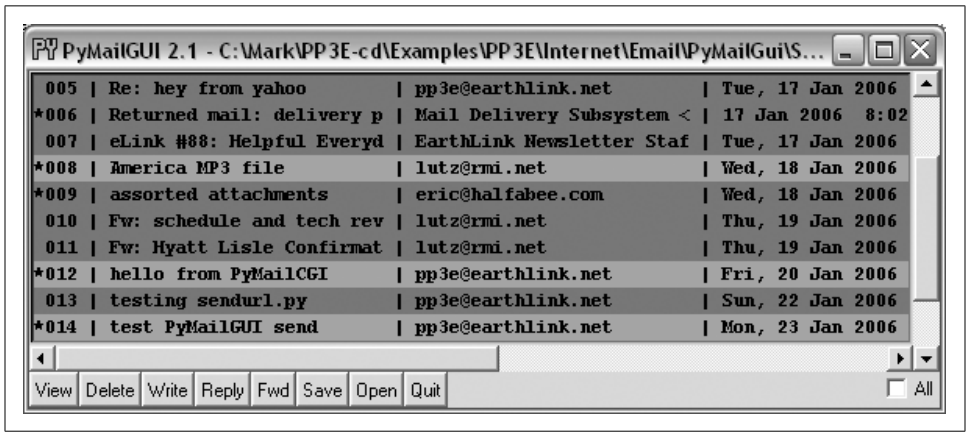

Figure 15-12. List window for mail save file, multiple selections

In general, there may be one server mail list window and any number of save-mail file list windows open at any time. Save-mail file list windows like that in Figure 15-12 can be opened at any time, even before fetching any mail from the server. They are identical to the server's inbox list window, but there is no help bar and no Load action button, and all other action buttons are mapped to the save file, not to the server.

For example, View opens the selected message in a normal mail view window identical to that in Figure 15-4, but the mail originates from the local file. Similarly, Delete removes the message from the save file, instead of from the server's inbox. Deletions from save-mail files are also run in a thread, to avoid blocking the rest of the GUI—the window title changes to "Deleting..." during the delete as a status indicator. Status indicators for loads and deletions in the server inbox window use popups instead, because the wait is longer and there is progress to display (see Figure 15-7).

Technically, saves always append raw message text to the chosen file; the file is opened in `'a'` mode, which creates the file if needed, and writes at its end. The Save and Open operations are also smart enough to remember the last directory you selected; the file dialogs begin navigation there the next time you press Save or Open.

You may also save mails from a saved file's window—use Save and Delete to move mails from file to file. In addition, saving to a file whose window is open for viewing automatically updates that file's list window in the GUI. This is also true for the automatically written sent-mail save file, described in the next section.

Sending Email and Attachments

Once we've loaded email from the server or opened a local save file, we can process our messages with the action buttons at the bottom of list windows. We can also send new emails at any time, even before a load or open. Pressing the Write button in a list window generates a mail composition window; one has been captured in Figure 15-13.

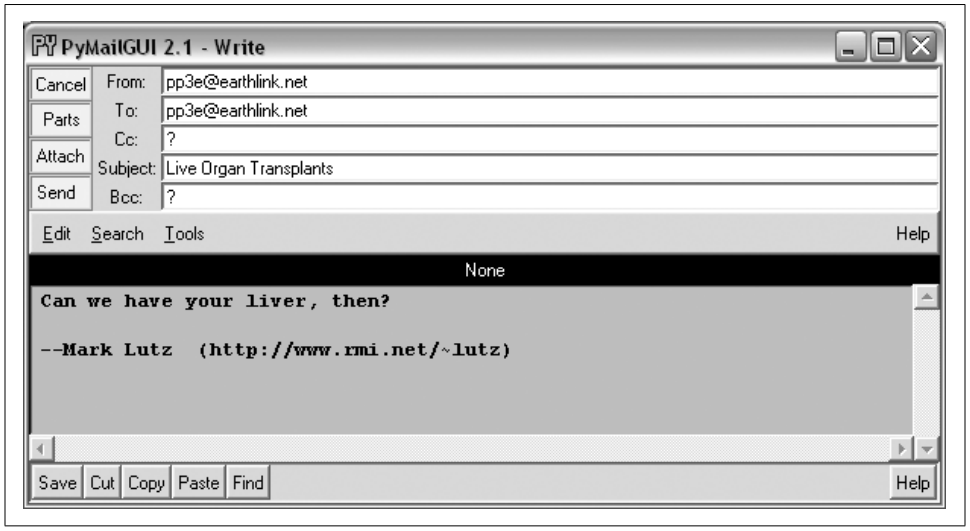

Figure 15-13. PyMailGUI write-mail compose window

This window is much like the message view window we saw in Figure 15-4, except there are no quick-access part buttons in the middle (this window is a new mail). It has fields for entering header line details, and an attached `TextEditor` object for writing the body of the new email.

For write operations, PyMailGUI automatically fills the "From:" line and inserts a signature text line ("--Mark..."), from your `mailconfig` module settings. You can change these to any text you like in the GUI, but the defaults are filled in automatically from your `mailconfig`. When the mail is sent, an `email.Utils` call handles date and time formatting in the `mailtools` module in Chapter 14.

There is also a new set of action buttons in the upper left here: Cancel closes the window (if verified), and Send delivers the mail—when you press the Send button, the text you typed into the body of this window is mailed to the addresses you typed into

the "To:", "Cc:", and "Bcc:" lines, using Python's `smtplib` module. PyMailGUI adds the header fields you type as mail header lines in the sent message. To send to more than one address, separate them with a ";" character in header fields. In this mail, I fill in the "To:" header with my own email address in order to send the message to myself for illustration purposes.

Also in compose windows, the Attach button issues a file selection dialog for attaching a file to your message, as in Figure 15-14. The Parts button pops up a dialog displaying files already attached, like that in Figure 15-15. When your message is sent, the text in the edit portion of the window is sent as the main message text, and any attached part files are sent as attachments properly encoded according to their type.

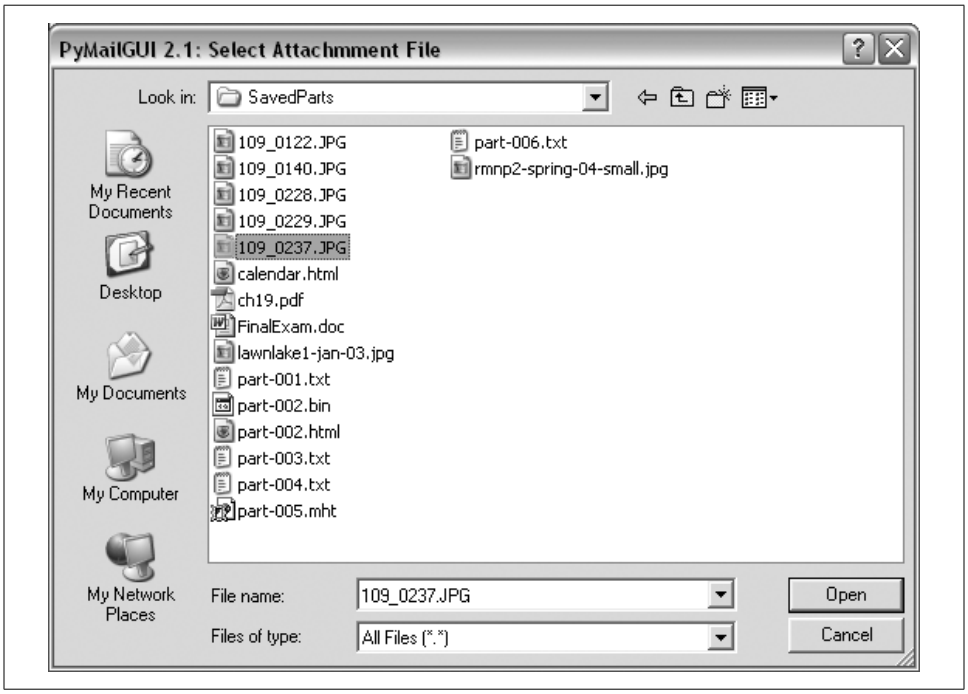

Figure 15-14. Attachment file dialog for Attach

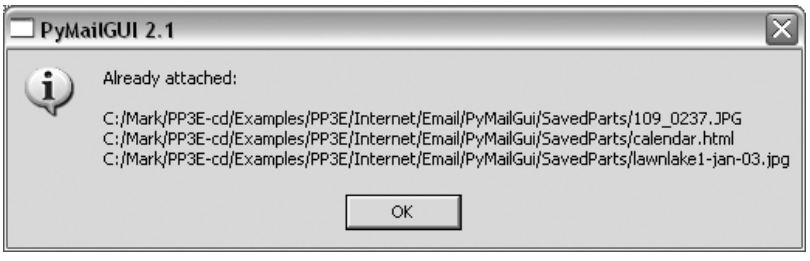

Figure 15-15. Attached parts list dialog for Parts

As we've seen, `smtplib` ultimately sends bytes to a server over a socket. Since this can be a long-running operation, PyMailGUI delegates this operation to a spawned thread, too. While the send thread runs, a nonblocking wait window appears and the entire GUI stays alive; redraw and move events are handled in the main program thread while the send thread talks to the SMTP server, and the user may perform other tasks in parallel.

You'll get an error popup if Python cannot send a message to any of the target recipients for any reason, and the mail composition window will pop up so that you may try again or save its text for later use. If you don't get an error popup, everything worked correctly, and your mail will show up in the recipients' mailboxes on their email servers. Since I sent the earlier message to myself, it shows up in mine the next time I press the main window's Load button, as we see in Figure 15-16.

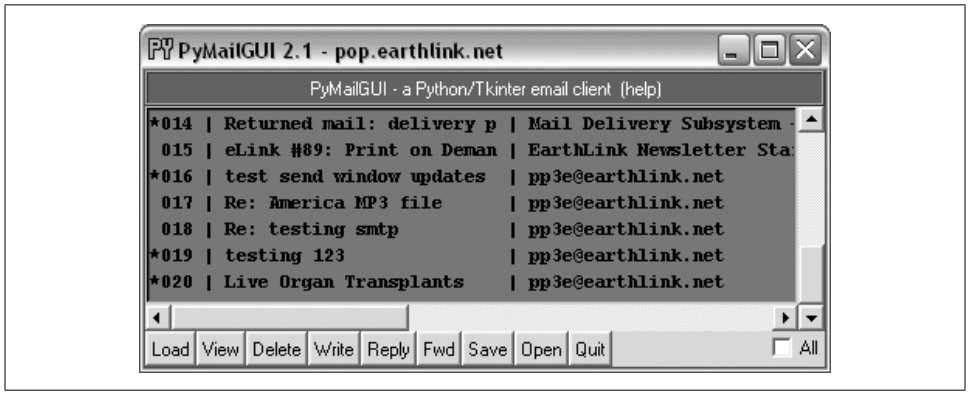

Figure 15-16. PyMailGUI main window after, loading sent mail

If you look back to the last main window shot, you'll notice that there is only one new email now—PyMailGUI is smart enough to download only the one new massage's header text and tack it onto the end of the loaded email list. Mail send operations automatically save sent mails in a save file that you name in your configuration module; use Open to view sent messages in offline mode and Delete to clean up the sent mail file if it grows too large (you can also Save from the sent-mail file to another file).

Viewing Email and Attachments

Now let's view the mail message that was sent and received. PyMailGUI lets us view email in formatted or raw mode. First, highlight (single-click) the mail you want to see in the main window, and press the View button. After the full message text is downloaded (unless it is already cached), a formatted mail viewer window like that shown in Figure 15-17 appears. If multiple messages are selected, the View button will download all that are not already cached (i.e., that have not already been

fetched) and will pop up a view window for each. Like all long-running operations, full message downloads are run in parallel threads to avoid blocking.

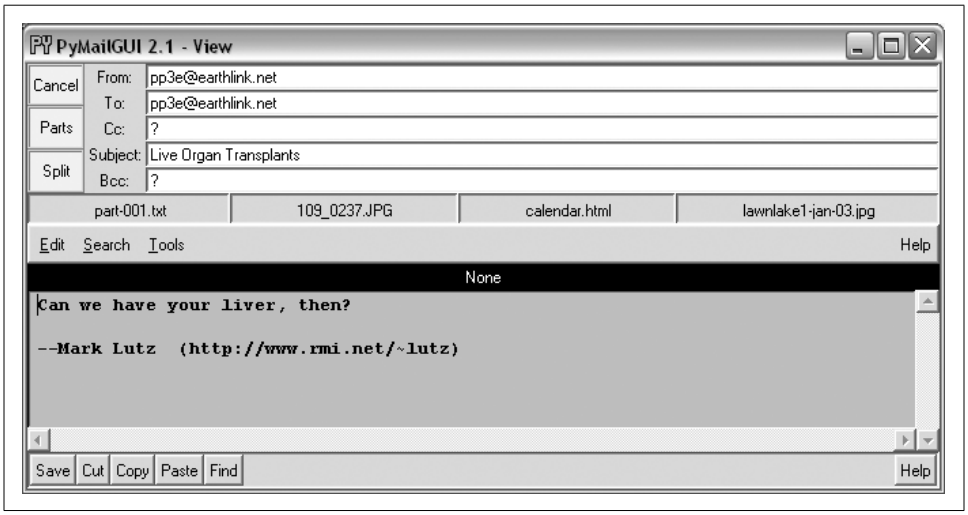

Figure 15-17. PyMailGUI view incoming mail window

Python's `email` module is used to parse out header lines from the raw text of the email message; their text is placed in the fields in the top right of the window. The message's main text is fetched from its body and stuffed into a new `TextEditor` object for display (it is also displayed in a web browser automatically if it is HTML text). PyMailGUI uses heuristics to extract the main text of the message to display, if there is one; it does not blindly show the entire raw text of the mail.

Any other parts of the message attached are displayed and opened with quick-access buttons in the middle. They are also listed by the Parts popup dialog, and they can be saved and opened all at once with Split. Figure 15-18 shows this window's Parts list popup, and Figure 15-19 displays this window's Split dialog in action.

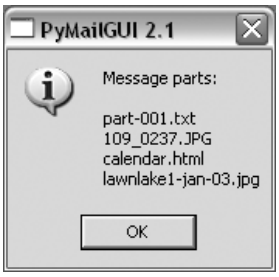

Figure 15-18. Parts dialog listing all message parts

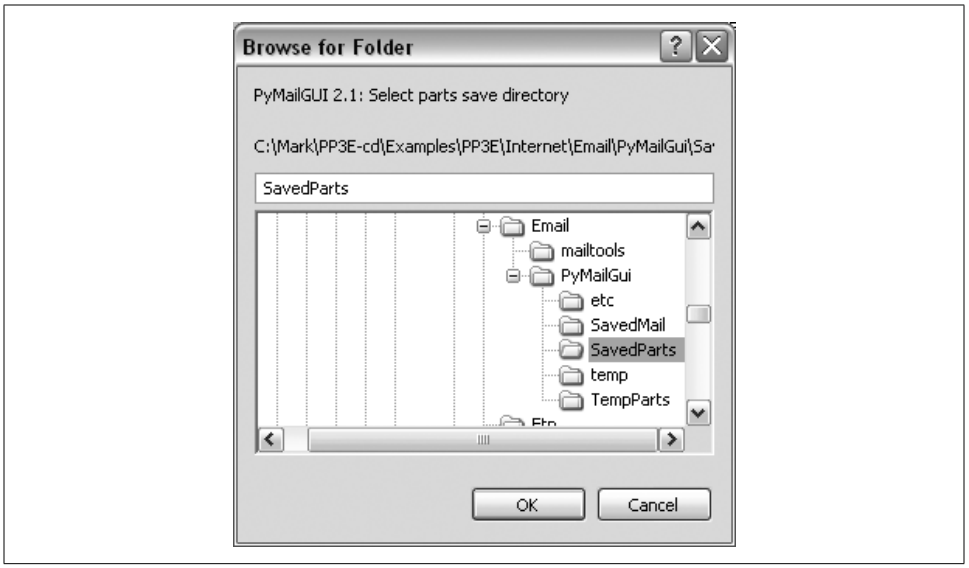

Figure 15-19. Split dialog selection

When the Split dialog in Figure 15-19 is submitted, all message parts are saved to the directory you select, and known parts are automatically opened. Individual parts are also automatically opened by the row of quick-access buttons labeled with the part's filename in the middle of the view window, after being saved to a temporary directory.

For instance, Figure 15-20 shows one of the image parts open on my Windows laptop, in a standard image viewer on that platform; other platforms may open this in a web browser instead. Click the image filename's quick-access button in Figure 15-17 to view it immediately, or run Split to open all parts at once.

By this point, the photo attachment displayed in Figure 15-20 has really gotten around: it has been encoded, attached, and sent, and then fetched, parsed, and decoded. Along the way, it has moved through multiple machines—from the client to the SMTP server, to the POP server, and back to the client.

In terms of user interaction, we attached the image to the email in Figure 15-13 using the dialog in Figure 15-14 before we sent the email. To access it later, we selected the email for viewing in Figure 15-16 and clicked on its quick-access button in Figure 15-17. PyMailGUI encoded it in base64 form, inserted it in the email's text, and later extracted and decoded it to get the original photo. With Python email tools, this just works.

Note that the main message text counts as a mail part, too—when selected, it opens in a PyEdit window, like that captured in Figure 15-21, from which it can be processed and saved (you can also save the main mail text with the Save button in the View window itself). The main part is included, because not all mails have a text

Figure 15-20. PyMailGUI opening image parts in a viewer or browser

part. For messages that have only HTML for their main text part, PyMailGUI displays the HTML text in its window, and opens a web browser to view the mail with its HTML formatting.

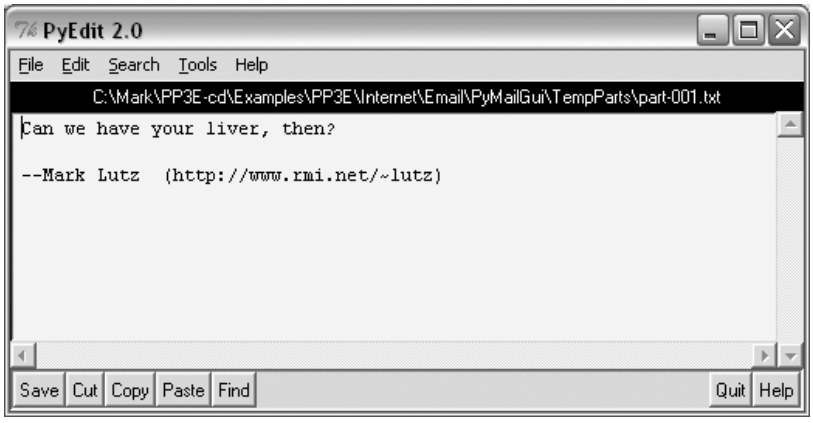

Figure 15-21. Main text part opened in PyEdit

PyMailGUI also opens HTML and XML parts in a web browser and uses the Windows Registry to open well-known Windows document types. For example, *.doc*, *.xls*, and *.pdf* files usually open, respectively, in Word, Excel, and Adobe Reader. Figure 15-22 captures the response to the *calendar.html* quick-access part button in Figure 15-17 on my Windows laptop (Firefox is my default web browser).

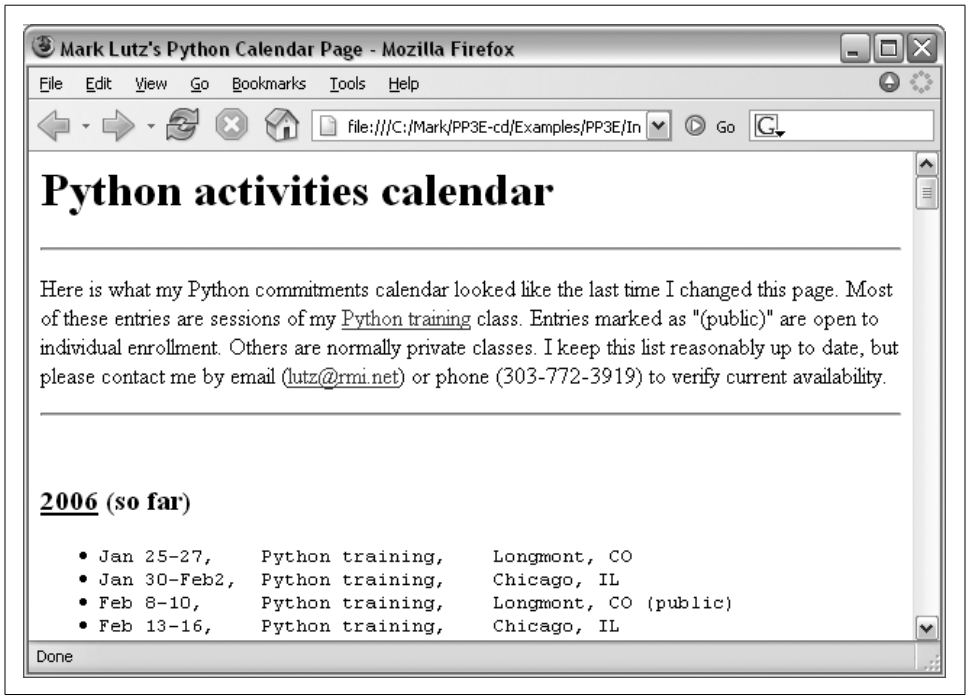

Figure 15-22. Attached HTML part opened in a web browser

The quick-access buttons in the middle of the Figure 15-17 view window are a more direct way to open parts than Split—you don't need to select a save directory, and you can open just the part you want to view. The Split button, though, allows all parts to be opened in a single step; allows you to choose where to save parts; and supports an arbitrary number of parts. Files that cannot be opened automatically because of their type can be inspected in the local save directory, after both Split and quick-access button selections (popup dialogs name the directory to use).

After a fixed maximum number of parts, the quick-access row ends with a button labeled "...", which simply runs Split to save and open additional parts when selected. Figure 15-23 captures one such message in the GUI; this was message 10 in Figure 15-9, if you're keeping track—a very complex mail, with 5 photos and 12 total parts.

Like much of PyMailGUI's behavior, the maximum number of part buttons to display in view windows can be configured in the `mailconfig.py` user settings module.

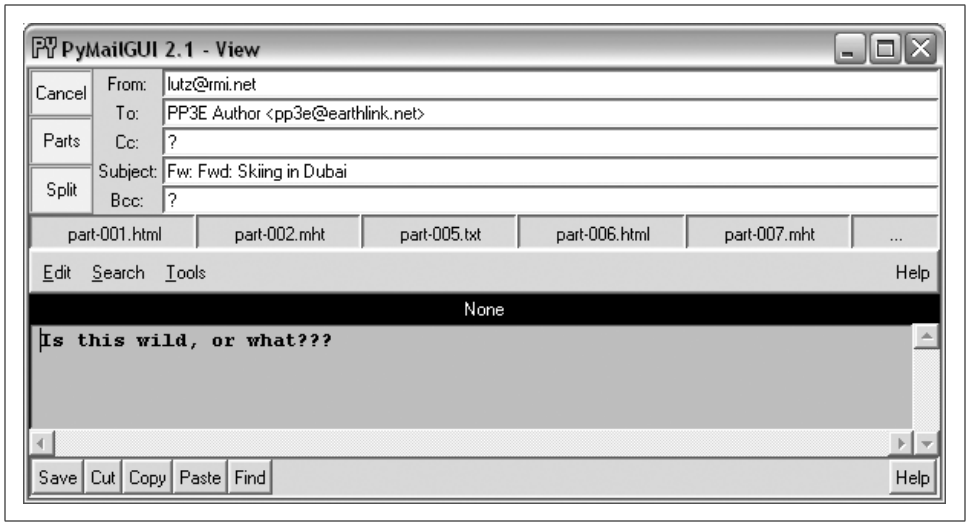

Figure 15-23. View window for a mail with many parts

That setting specified five buttons in Figure 15-23. Figure 15-24 shows a different mail with many attachments being viewed; the part buttons setting has been changed to a maximum of eight (this mail has seven parts). The setting can be higher, but at some point the buttons may become unreadable (use Split instead).

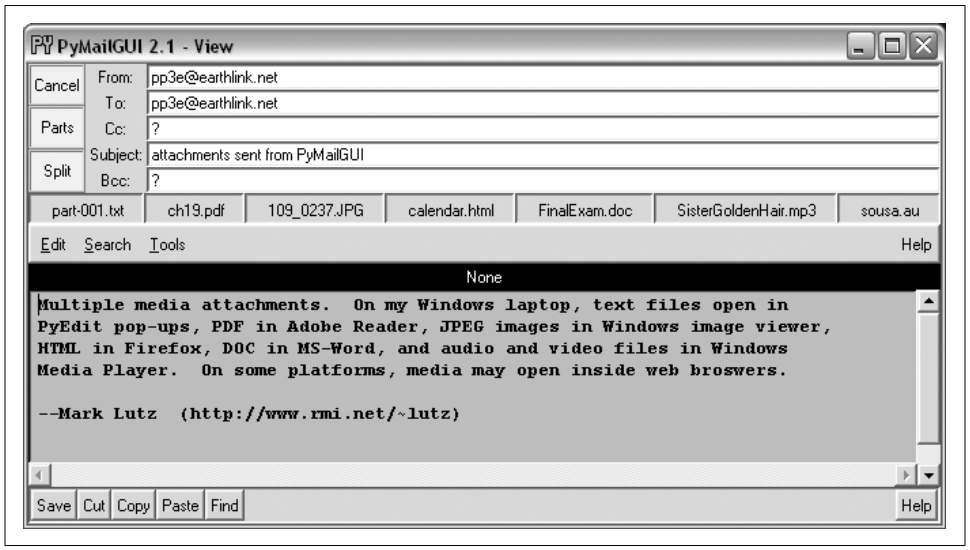

Figure 15-24. View window with part buttons setting increased

Figures 15-25 and 15-26 show what happens when the *sousa.au* and *ch19.pdf* buttons in Figure 15-24 are pressed on my Windows laptop. The results vary per

machine; the audio file opens in Windows Media Player, the MP3 file opens in iTunes instead, and some platforms may open such files directly in a web browser.

Figure 15-25. An audio part opened by PyMailGUI

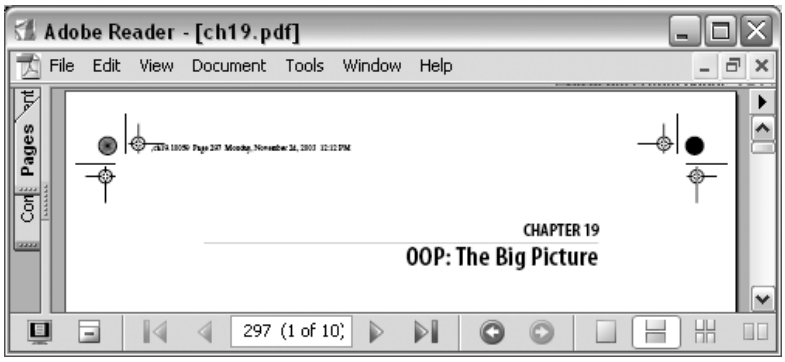

Figure 15-26. A PDF part opened in PyMailGUI

Besides the nicely formatted view window, PyMailGUI also lets us see the raw text of a mail message. Double-click on a message's entry in the main window's list to bring up a simple unformatted display of the mail's raw text (its full text is downloaded in a thread if it hasn't yet been fetched and cached). The raw version of the mail I sent to myself in Figure 15-17 is shown in Figure 15-27.

This raw text display can be useful to see special mail headers not shown in the formatted view. For instance, the optional "X-Mailer:" header in the raw text display identifies the program that transmitted a message; PyMailGUI adds it automatically,

Figure 15-27. PyMailGUI raw mail text view window

along with standard headers like "From:" and "To:". Other headers are added as the mail is transmitted: the "Received:" headers name machines that the message was routed through on its way to our email server, and "Content-Type:" is added and parsed by Python's `email` package.

And really, the raw text form is all there is to an email message—it's what is transferred from machine to machine when mail is sent. The nicely formatted display of the GUI's view windows simply parses out and decodes components from the mail's raw text with standard Python tools, and places them in the associated fields of the display.

Email Replies and Forwards

Besides allowing users to read and write email, PyMailGUI also lets users forward and reply to incoming email sent from others. To reply to an email, select its entry in the main window's list and click the Reply button. If I reply to the mail I just sent to myself (arguably narcissistic, but demonstrative), the mail composition window shown in Figure 15-28 appears.

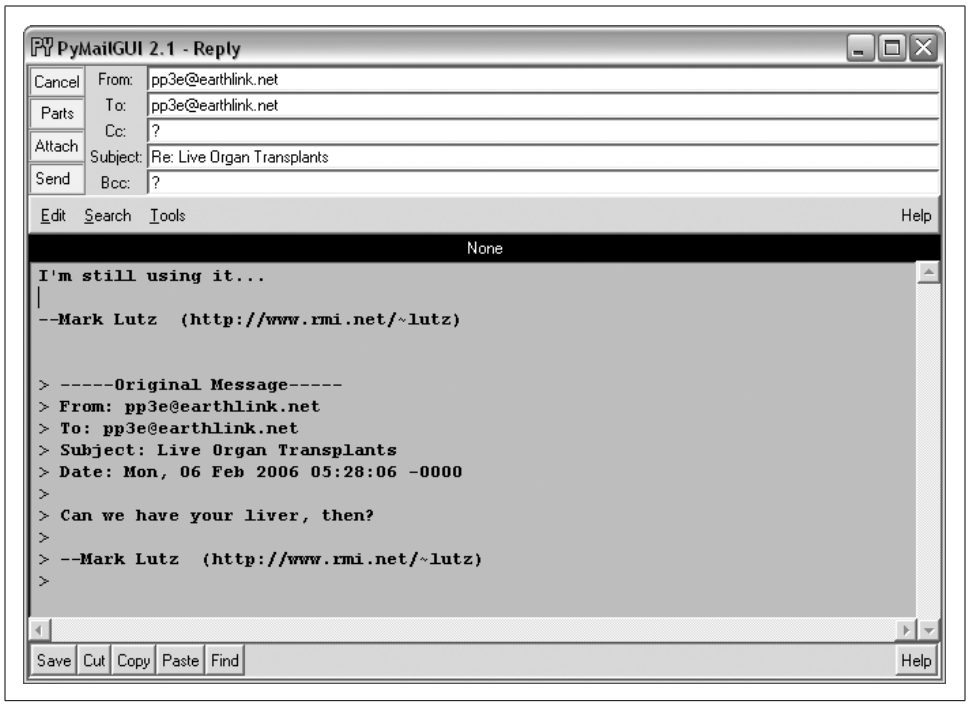

Figure 15-28. PyMailGUI reply compose window

This window is identical in format to the one we saw for the Write operation, except that PyMailGUI fills in some parts automatically:

- The "From" line is set to your email address in your `mailconfig` module.
- The "To:" line is initialized to the original message's "From" address (we're replying to the original sender, after all). An `email.Utils` call processes the "To:".
- The "Subject:" line is set to the original message's subject line, prepended with a "Re:", the standard follow-up subject line form (unless it already has one).
- The body of the reply is initialized with the signature line in `mailconfig`, along with the original message's text. The original message text is quoted with > characters and is prepended with a few header lines extracted from the original message to give some context.

Luckily, all of this is much easier than it may sound. Python's standard `email` module extracts all of the original message's header lines, and a single string `replace` method call does the work of adding the > quotes to the original message body. I simply type what I wish to say in reply (the initial paragraph in the mail's text area) and press the Send button to route the reply message to the mailbox on my mail server again. Physically sending the reply works the same as sending a brand-new

message—the mail is routed to your SMTP server in a spawned send-mail thread, and the send-mail wait popup appears while the thread runs.

Forwarding a message is similar to replying: select the message in the main window, press the Fwd button, and fill in the fields and text area of the popped-up composition window. Figure 15-29 shows the window created to forward the mail we originally wrote and received.

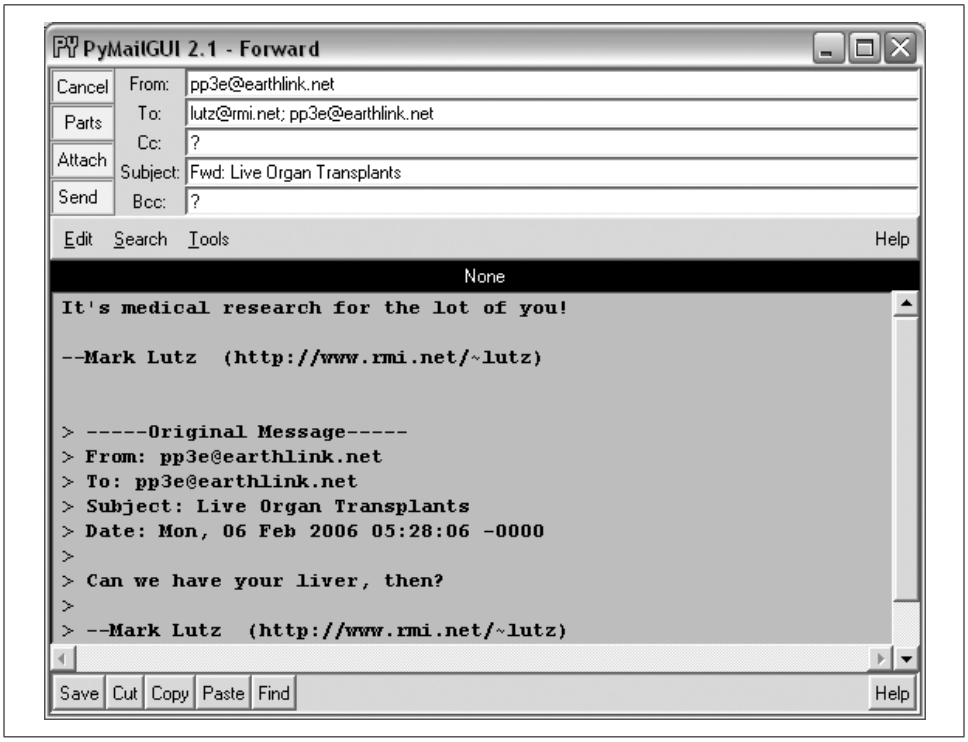

Figure 15-29. PyMailGUI forward compose window

Much like with replies, "From:" is filled from `mailconfig`, the original text is automatically quoted in the message body again, and the subject line is preset to the original message's subject, prepended with the string "Fwd:". I have to fill in the "To:" line manually, though, because this is not a direct reply (it doesn't necessarily go back to the original sender).

Notice that I'm forwarding this message to two different addresses; multiple recipient addresses are separated with a semicolon (;) in the "To:", "Cc:", and "Bcc:" header fields. The Send button in this window fires the forwarded message off to all addresses listed in these headers.

I've now written a new message, replied to it, and forwarded it. The reply and forward were sent to my email address, too; if we press the main window's Load

button again, the reply and forward messages should show up in the main window's list. In Figure 15-30, they appear as messages 22 and 21 (the order they appear in may depend on timing issues at your server).

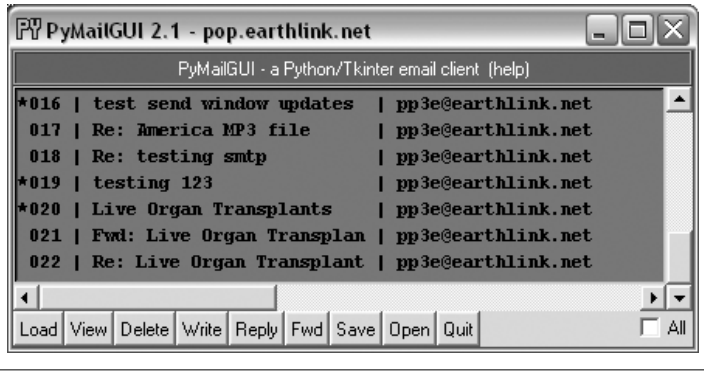

Figure 15-30. PyMailGUI mail list after sends and load

Keep in mind that PyMailGUI runs on the local computer, but the messages you see in the main window's list actually live in a mailbox on your email server machine. Every time we press Load, PyMailGUI downloads but does not delete newly arrived email from the server to your computer. The three messages we just wrote (20 through 22) will also appear in any other email program you use on your account (e.g., in Outlook, or in a web mail interface). PyMailGUI does not automatically delete messages as they are downloaded, but simply stores them in your computer's memory for processing. If we now select message 21 and press View, we see the forward message we sent, as in Figure 15-31. This message went from my machine to a remote email server and was downloaded from there into a Python list from which it is displayed.

Figure 15-32 shows what the forward message's raw text looks like; again, double-click on a main window's entry to display this form. The formatted display in Figure 15-31 simply extracts bits and pieces out of the text shown in the raw display form.

Deleting Email

So far, we've covered every action button on list windows except for Delete and the All checkbox. The All checkbox simply toggles from selecting all messages at once or deselecting all (View, Delete, Reply, Fwd, and Save action buttons apply to all currently selected messages). PyMailGUI also lets us delete messages from the server permanently, so that we won't see them the next time we access our inbox.

Delete operations are kicked off the same way as Views and Saves; just press the Delete button instead. In typical operation, I eventually delete email I'm not interested in, and save and delete emails that are important. We met Save earlier in this demo.

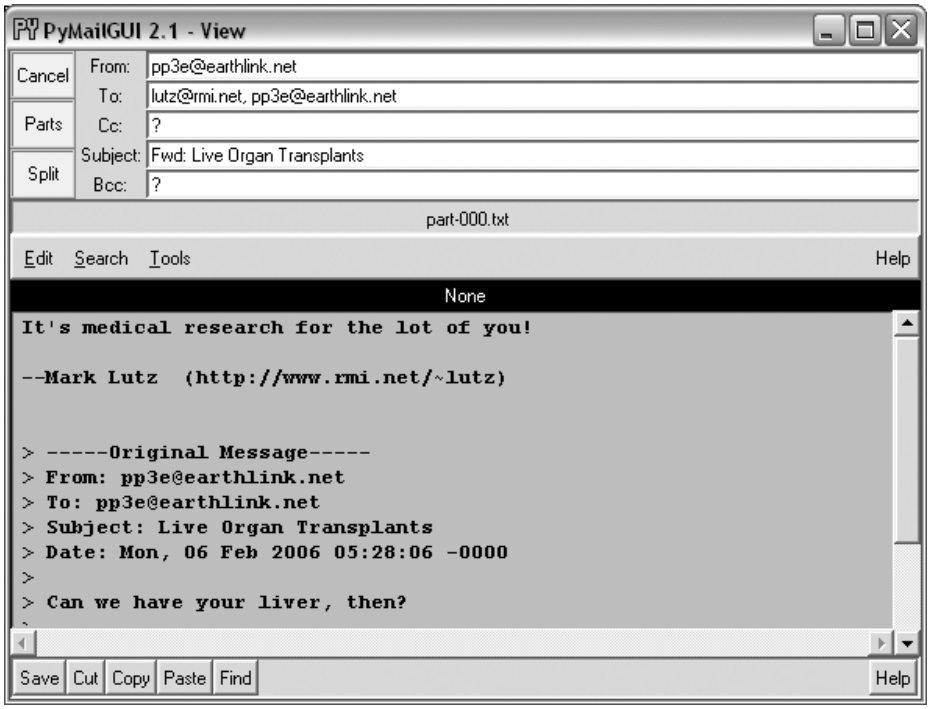

Figure 15-31. PyMailGUI view forwarded mail

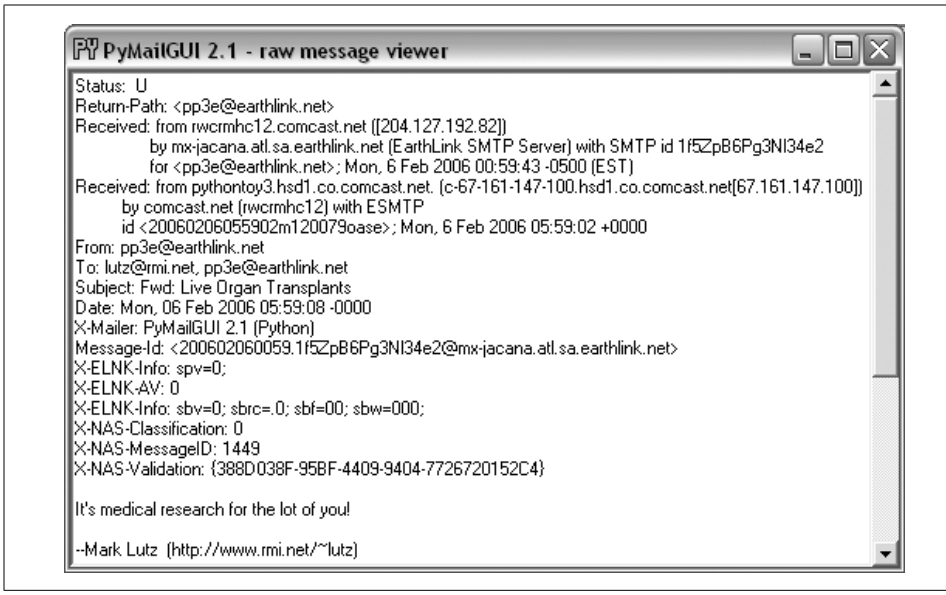

Figure 15-32. PyMailGUI view forwarded mail, raw

Like View, Save, and other operations, Delete can be applied to one or more messages. Deletes happen immediately, and like all server transfers they are run in a nonblocking thread but are performed only if you verify the operation in a popup, such as the one shown in Figure 15-33.

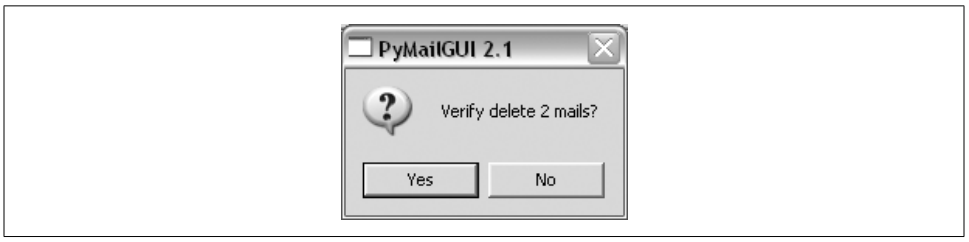

Figure 15-33. PyMailGUI delete verification on quit

By design, no mail is ever removed automatically: you will see the same messages the next time PyMailGUI runs. It deletes mail from your server only when you ask it to, and then only if verified in the last popup shown (this is your last chance to prevent permanent mail removal). After the deletions are performed, the mail index is updated, and the GUI session continues.

Deletions disable mail loads and other deletes while running and cannot be run in parallel with loads or other deletes already in progress because they may change POP message numbers and thus modify the mail index list. Messages may still be composed during a deletion, however, and offline save files may be processed.

POP Message Numbers and Synchronization

Deletions are complicated by POP's message-numbering scheme. We learned about the potential for synchronization errors between the server's inbox and the fetched email list in Chapter 14, when studying the `mailtools` package PyMailGUI uses (near Example 14-21). In brief, POP assigns each message a relative sequential number, starting from one, and these numbers are passed to the server to fetch and delete messages. The server's inbox is normally locked while a connection is held so that a series of deletions can be run as an atomic operation; no other inbox changes occur until the connection is closed.

However, message number changes also have some implications for the GUI itself. It's all right if new mail arrives while we're displaying the result of a prior download—the new mail is assigned higher numbers, beyond what is displayed on the client. But if we delete a message in the middle of a mailbox after the index has been loaded from the mail server, the numbers of all messages after the one deleted change (they are decremented by one). As a result, some message numbers might no longer be valid if deletions are made while viewing previously loaded email.

To work around this, PyMailGUI adjusts all the displayed numbers after a Delete by simply removing the entries for deleted mails from its index list and mail cache. However, this adjustment is not enough to keep the GUI in sync with the server's inbox if the inbox is modified at a time other than after the end, by deletions in another email client (even in another PyMailGUI session) or by deletions performed by the mail server itself (e.g., messages determined to be undeliverable and automatically removed from the inbox).

To handle these cases, PyMailGUI uses the safe deletion and synchronization tests in `mailtools`. That module uses mail header matching to detect mail list and server inbox synchronization errors. For instance, if another email client has deleted a message prior to the one to be deleted by PyMailGUI, `mailtools` catches the problem and cancels the deletion, and an error popup like the one in Figure 15-34 is displayed.

Figure 15-34. Safe deletion test detection of inbox difference

Similarly, index loads and message fetches run a synchronization test in `mailtools`, as well. Figure 15-35 captures the error generated if a message has been deleted in another client since we last loaded the server index window.

Figure 15-35. Synchronization error on after delete in another client

In both error cases, the inbox is automatically reloaded by PyMailGUI immediately after the error popup is dismissed. This scheme ensures that PyMailGUI won't delete or display the wrong message, in the rare case that the server's inbox is changed without its knowledge. See `mailtools` in Chapter 14 for more on synchronization tests.

Multiple Windows and Status Messages

Note that PyMailGUI is really meant to be a multiple-window interface—a detail not made obvious by the earlier screenshots. For example, Figure 15-36 shows PyMailGUI with the main server list window, two save-file list windows, two message view windows, and help. All these windows are nonmodal; that is, they are all active and independent, and do not block other windows from being selected. This interface looks slightly different on Linux and the Mac, but it has the same functionality.

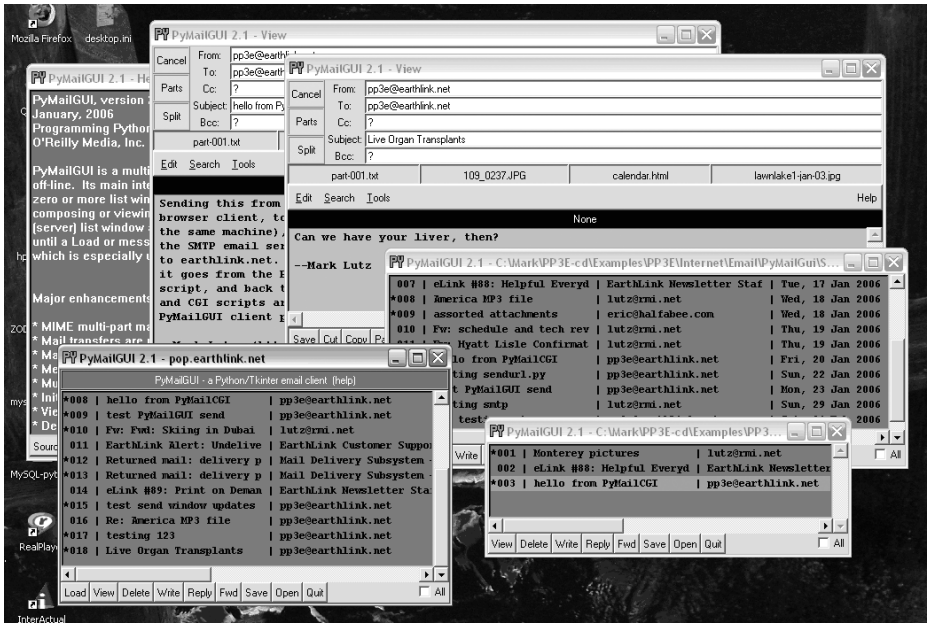

Figure 15-36. PyMailGUI multiple windows and text editors

In general, you can have any number of mail view or compose windows up at once, and cut and paste between them. This matters, because PyMailGUI must take care to make sure that each window has a distinct text-editor object. If the text-editor object were a global, or used globals internally, you'd likely see the same text in each window (and the Send operations might wind up sending text from another window). To avoid this, PyMailGUI creates and attaches a new `TextEditor` instance to each view and compose window it creates, and associates the new editor with the Send button's callback handler to make sure we get the right text.

Finally, PyMailGUI prints a variety of status messages as it runs, but you see them only if you launch the program from the system command-line console window (e.g., a DOS box on Windows or an xterm on Linux), or by double-clicking on its filename icon (its main script is a *.py*, not a *.pyw*). On Windows, you won't see these

messages when PyMailGUI is started from another program, such as the PyDemos or PyGadgets launcher bar GUIs. These status messages print server information, show mail loading status, and trace the load, store, and delete threads that are spawned along the way. If you want PyMailGUI to be more verbose, launch it from a command line and watch:

```
C:\...\PP3E\Internet\Email\PyMailGui>PyMailGui2.py
user: pp3e
loading headers
Connecting...
+OK NGPopper vEL_6_10 at earthlink.net ready<7044.1139211610@poptawny.atl.sa.
earthlink.net>
load headers exit
synch check
Connecting...
+OK NGPopper vEL_6_10 at earthlink.net ready <22399.1139211626@pop-tawny.atl.sa.
earthlink.net>
Same headers text
load 16
Connecting...
+OK NGPopper vEL_6_10 at earthlink.net ready <27558.1139211627@pop-tawny.atl.sa.
earthlink.net>
Sending to...['pp3e@earthlink.net']
From: pp3e@earthlink.net
To: pp3e@earthlink.net
Subject: Fwd: Re: America MP3 file
Date: Mon, 06 Feb 2006 07:41:05 -0000
X-Mailer: PyMailGUI 2.1 (Python)

--Mark Lutz  (http://www.rmi.net/~lutz)

> -----Original Message-----
> From: pp3e@earthlink.net
>
Send exit
```

You can also double-click on the *PyMailGui.py* filename in your file explorer GUI and monitor the popped-up DOS console box on Windows. Console messages are mostly intended for debugging, but they can also be used to help understand the system's operation.

For more details on using PyMailGUI, see its help display, or read the help string in the module `PyMailGuiHelp.py`, listed in Example 15-9 in the next section.

PyMailGUI Implementation

Last but not least, we get to the code. PyMailGUI consists of the nine new modules listed at the start of this chapter; the source code for these modules is listed in this section.

Code Reuse

Besides the code here, PyMailGUI also gets a lot of mileage out of reusing modules we wrote earlier and won't repeat here: `mailtools` for mail loads, composition, parsing, and delete operations; `threadtools` for managing server and local file access threads; the GUI section's `TextEditor` for displaying and editing mail message text; and so on.

In addition, standard Python modules and packages such as `poplib`, `smtplib`, and `email` hide most of the details of pushing bytes around the Net and extracting and building message components. As usual, the `Tkinter` standard library module also implements GUI components in a portable fashion.

Code Structure

As mentioned earlier, PyMailGUI applies code factoring and OOP to leverage code reuse. For instance, list view windows are implemented as a common superclass that codes most actions, along with one subclass for the server inbox list window and one for local save-file list windows. The subclasses customize the common superclass for their specific mail media.

This design reflects the operation of the GUI itself—server list windows load mail over POP, and save-file list windows load from local files. The basic operation of list window layout and actions, though, is similar for both and is shared in the common superclass to avoid redundancy and simplify the code. Message view windows are similarly factored: a common view window superclass is reused and customized for write, reply, and forward view windows.

To make the code easier to follow, it is divided into two main modules that reflect the structure of the GUI—one for the implementation of list window actions and one for view window actions. If you are looking for the implementation of a button that appears in a mail view or edit window, for instance, see the view window module and search for a method whose name begins with the word *on*—the convention used for callback handler methods. Button text can also be located in name/callback tables used to build the windows. Actions initiated on list windows are coded in the list window module instead.

In addition, the message cache is split off into an object and module of its own, and potentially reusable tools are coded in importable modules (e.g., line wrapping and utility popups). PyMailGUI also includes a main module that defines startup window classes, a module that contains the help text as a string, and the `mailconfig` user settings module (a version specific to PyMailGUI is used here).

The next few sections list all of PyMailGUI's code for you to study; as you read, refer back to the demo earlier in this chapter and run the program live to map its behavior back to its code. PyMailGUI also includes a `__init__.py` file so that it can be used as

a package—some of its modules may be useful in other programs. The `__init__.py` is empty in this package, so we omit it here.

PyMailGui2: The Main Module

Example 15-1 defines the file run to start PyMailGUI. It implements top-level list windows in the system—combinations of PyMailGUI's application logic and the window protocol superclasses we wrote earlier in the text. The latter of these define window titles, icons, and close behavior.

The main documentation is also in this module, as well as command-line logic—the program accepts the names of one or more save-mail files on the command line, and automatically opens them when the GUI starts up. This is used by the PyDemos launcher, for example.

Example 15-1. PP3E\Internet\Email\PyMailGui\PyMailGui2.py

```
################################################################################
# PyMailGui 2.1 - A Python/Tkinter email client.
# A client-side Tkinter-based GUI interface for sending and receiving email.
#
# See the help string in PyMailGuiHelp2.py for usage details, and a list of
# enhancements in this version.  Version 2.0 is a major rewrite.  The changes
# from 2.0 (July '05) to 2.1 (Jan '06) were quick-access part buttons on View
# windows, threaded loads and deletes of local save-mail files, and checks for
# and recovery from message numbers out-of-synch with mail server inbox on
# deletes, index loads, and message loads.
#
# This file implements the top-level windows and interface.  PyMailGui uses
# a number of modules that know nothing about this GUI, but perform related
# tasks, some of which are developed in other sections of the book.  The
# mailconfig module is expanded for this program.
#
# Modules defined elsewhere and reused here:
#
# mailtools (package):
#    server sends and receives, parsing, construction     (client-side chapter)
# threadtools.py
#    thread queue manangement for GUI callbacks           (GUI tools chapter)
# windows.py
#    border configuration for top-level windows           (GUI tools chapter)
# textEditor.py
#    text widget used in mail view windows, some pop ups   (GUI programs chapter)
#
# Generally useful modules defined here:
#
# popuputil.py
#    help and busy windows, for general use
# messagecache.py
#    a cache that keeps track of mail already loaded
# wraplines.py
```

Example 15-1. PP3E\Internet\Email\PyMailGui\PyMailGui2.py (continued)

```
#    utility for wrapping long lines of messages
# mailconfig.py
#    user configuration parameters: server names, fonts, etc.
#
# Program-specific modules defined here:
#
# SharedNames.py
#    objects shared between window classes and main file
# ViewWindows.py
#    implementation of view, write, reply, forward windows
# ListWindows.py
#    implementation of mail-server and local-file list windows
# PyMailGuiHelp.py
#    user-visible help text, opened by main window bar
# PyMailGui2.py
#    main, top-level file (run this), with main window types
##############################################################################

import mailconfig, sys
from SharedNames import appname, windows
from ListWindows import PyMailServer, PyMailFile


##############################################################################
# Top-level window classes
# View, Write, Reply, Forward, Help, BusyBox all inherit from PopupWindow
# directly: only usage;  askpassword calls PopupWindow and attaches;  order
# matters here!--PyMail classes redef some method defaults in the Window
# classes, like destroy and okayToExit: must be leftmost;  to use
# PyMailFileWindow standalone, imitate logic in PyMailCommon.onOpenMailFile;
##############################################################################

# uses icon file in cwd or default in tools dir
srvrname = mailconfig.popservername or 'Server'

class PyMailServerWindow(PyMailServer, windows.MainWindow):
    def __init__(self):
        windows.MainWindow.__init__(self, appname, srvrname)
        PyMailServer.__init__(self)

class PyMailServerPopup(PyMailServer, windows.PopupWindow):
    def __init__(self):
        windows.PopupWindow.__init__(self, appname, srvrnane)
        PyMailServer.__init__(self)

class PyMailServerComponent(PyMailServer, windows.ComponentWindow):
    def __init__(self):
        windows.ComponentWindow.__init__(self)
        PyMailServer.__init__(self)

class PyMailFileWindow(PyMailFile, windows.PopupWindow):
    def __init__(self, filename):
```

Example 15-1. PP3E\Internet\Email\PyMailGui\PyMailGui2.py (continued)

```
        windows.PopupWindow.__init__(self, appname, filename)
        PyMailFile.__init__(self, filename)


################################################################################
# when run as a top-level program: create main mail-server list window
################################################################################

if __name__ == '__main__':
    rootwin = PyMailServerWindow( )              # open server window
    if sys.argv > 1:
        for savename in sys.argv[1:]:
            rootwin.onOpenMailFile(savename)     # open save file windows (demo)
        rootwin.lift( )                          # save files loaded in threads
    rootwin.mainloop( )
```

SharedNames: Program-Wide Globals

The module in Example 15-2 implements a shared, system-wide namespace that collects resources used in most modules in the system, and defines global objects that span files. This allows other files to avoid redundantly repeating common imports, and encapsulates the locations of package imports; it is the only file that must be updated if paths change in the future. Using globals can make programs harder to understand in general (the source of some names is not as clear), but it is reasonable if all such names are collected in a single expected module such as this one (because there is only one place to search for unknown names).

Example 15-2. PP3E\Internet\Email\PyMailGui\SharedNames.py

```
################################################################################
# objects shared by all window classes and main file: program-wide globals
################################################################################

# used in all window, icon titles
appname  = 'PyMailGUI 2.1'

# used for list save, open, delete; also for sent messages file
saveMailSeparator = 'PyMailGUI' + ('-'*60) + 'PyMailGUI\n'

# currently viewed mail save files; also for sent-mail file
openSaveFiles     = {}                 # 1 window per file,{name:win}

# standard library services
import sys, os, email, webbrowser
from Tkinter       import *
from tkFileDialog  import SaveAs, Open, Directory
from tkMessageBox  import showinfo, showerror, askyesno

# reuse book examples
from PP3E.Gui.Tools      import windows      # window border, exit protocols
from PP3E.Gui.Tools      import threadtools  # thread callback queue checker
```

Example 15-2. PP3E\Internet\Email\PyMailGui\SharedNames.py (continued)

```
from PP3E.Internet.Email import mailtools    # load,send,parse,build utilities
from PP3E.Gui.TextEditor import textEditor   # component and pop up

# modules defined here
import mailconfig                            # user params: servers, fonts, etc.
import popuputil                             # help, busy, passwd pop-up windows
import wraplines                             # wrap long message lines
import messagecache                          # remember already loaded mail
import PyMailGuiHelp                         # user documentation

def printStack(exc_info):
    # debugging: show exception and stack traceback on stdout
    print exc_info[0]
    print exc_info[1]
    import traceback
    traceback.print_tb(exc_info[2], file=sys.stdout)

# thread busy counters for threads run by this GUI
# sendingBusy shared by all send windows, used by main window quit

loadingHdrsBusy = threadtools.ThreadCounter( )   # only 1
deletingBusy    = threadtools.ThreadCounter( )   # only 1
loadingMsgsBusy = threadtools.ThreadCounter( )   # poss many
sendingBusy     = threadtools.ThreadCounter( )   # poss many
```

ListWindows: Message List Windows

The code in Example 15-3 implements mail index list windows—for the server inbox window and for one or more local save-mail file windows. These two types of windows look and behave largely the same, and in fact share most of their code in common in a superclass. The window subclasses mostly just customize the superclass to map mail Load and Delete calls to the server or a local file.

List windows are created on program startup (the initial server window, and possible save-file windows for command-line options), as well as in response to Open button actions in existing list windows (save-file list windows). See the Open button's callback in this example for initiation code.

Notice that the basic mail processing operations in the `mailtools` package from Chapter 14 are mixed into PyMailGUI in a variety of ways. The list window classes in Example 15-3 inherit from the `mailtools` mail parser class, but the server list window class embeds an instance of the message cache object, which in turn inherits from the `mailtools` mail fetcher. The `mailtools` mail sender class is inherited by message view write windows, not list windows; view windows also inherit from the mail parser.

This is a fairly large file; in principle it could be split into three files, one for each class, but these classes are so closely related that it is handy to have their code in a single file for edits. Really, this is one class, with two minor extensions.

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py

```
################################################################################
# Implementation of mail-server and save-file message list main windows:
# one class per kind.  Code is factored here for reuse: server and file
# list windows are customized versions of the PyMailCommon list window class;
# the server window maps actions to mail transferred from a server, and the
# file window applies actions to a local file.  List windows create View,
# Write, Reply, and Forward windows on user actions.  The server list window
# is the main window opened on program startup by the top-level file;  file
# list windows are opened on demand via server and file list window "Open".
# Msgnums may be temporarily out of sync with server if POP inbox changes.
#
# Changes here in 2.1:
# -now checks on deletes and loads to see if msg nums in sync with server
# -added up to N attachment direct-access buttons on view windows
# -threaded save-mail file loads, to avoid N-second pause for big files
# -also threads save-mail file deletes so file write doesn't pause GUI
# TBD:
# -save-mail file saves still not threaded: may pause GUI briefly, but
#  uncommon - unlike load and delete, save/send only appends the local file.
# -implementation of local save-mail files as text files with separators
#  is mostly a prototype: it loads all full mails into memory, and so limits
#  the practical size of these files; better alternative: use 2 DBM keyed
#  access files for hdrs and fulltext, plus a list to map keys to position;
#  in this scheme save-mail files become directories, no longer readable.
################################################################################

from SharedNames import *     # program-wide global objects
from ViewWindows import ViewWindow, WriteWindow, ReplyWindow, ForwardWindow


################################################################################
# main frame - general structure for both file and server message lists
################################################################################


class PyMailCommon(mailtools.MailParser):
    """
    a widget package, with main mail listbox
    mixed in with a Tk, Toplevel, or Frame
    must be customized with actions() and other
    creates view and write windows: MailSenders
    """
    # class attrs shared by all list windows
    threadLoopStarted = False                    # started by first window

    # all windows use same dialogs: remember last dirs
    openDialog = Open(title=appname + ': Open Mail File')
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
    saveDialog = SaveAs(title=appname + ': Append Mail File')

    def __init__(self):
        self.makeWidgets( )                       # draw my contents: list,tools
        if not PyMailCommon.threadLoopStarted:    # server,file can both thread
            PyMailCommon.threadLoopStarted = True # start thread exit check loop
            threadtools.threadChecker(self)       # just one for all windows

    def makeWidgets(self):
        # add all/none checkbtn at bottom
        tools = Frame(self)
        tools.pack(side=BOTTOM, fill=X)
        self.allModeVar = IntVar( )
        chk = Checkbutton(tools, text="All")
        chk.config(variable=self.allModeVar, command=self.onCheckAll)
        chk.pack(side=RIGHT)

        # add main buttons at bottom
        for (title, callback) in self.actions( ):
            Button(tools, text=title, command=callback).pack(side=LEFT, fill=X)

        # add multiselect listbox with scrollbars
        mails    = Frame(self)
        vscroll  = Scrollbar(mails)
        hscroll  = Scrollbar(mails, orient='horizontal')
        fontsz   = (sys.platform[:3] == 'win' and 8) or 10      # defaults
        listbg   = mailconfig.listbg   or 'white'
        listfg   = mailconfig.listfg   or 'black'
        listfont = mailconfig.listfont or ('courier', fontsz, 'normal')
        listbox  = Listbox(mails, bg=listbg, fg=listfg, font=listfont)
        listbox.config(selectmode=EXTENDED)
        listbox.bind('<Double-1>', (lambda event: self.onViewRawMail( )))

        # crosslink listbox and scrollbars
        vscroll.config(command=listbox.yview, relief=SUNKEN)
        hscroll.config(command=listbox.xview, relief=SUNKEN)
        listbox.config(yscrollcommand=vscroll.set, relief=SUNKEN)
        listbox.config(xscrollcommand=hscroll.set)

        # pack last = clip first
        mails.pack(side=TOP, expand=YES, fill=BOTH)
        vscroll.pack(side=RIGHT,  fill=BOTH)
        hscroll.pack(side=BOTTOM, fill=BOTH)
        listbox.pack(side=LEFT, expand=YES, fill=BOTH)
        self.listBox = listbox

    ##################
    # event handlers
    ##################

    def onCheckAll(self):
        # all or none click
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
        if self.allModeVar.get( ):
            self.listBox.select_set(0, END)
        else:
            self.listBox.select_clear(0, END)

    def onViewRawMail(self):
        # possibly threaded: view selected messages - raw text headers, body
        msgnums = self.verifySelectedMsgs( )
        if msgnums:
            self.getMessages(msgnums, after=lambda: self.contViewRaw(msgnums))

    def contViewRaw(self, msgnums):
        for msgnum in msgnums:                      # could be a nested def
            fulltext = self.getMessage(msgnum)      # put in ScrolledText
            from ScrolledText import ScrolledText   # don't need full TextEditor
            window  = windows.QuietPopupWindow(appname, 'raw message viewer')
            browser = ScrolledText(window)
            browser.insert('0.0', fulltext)
            browser.pack(expand=YES, fill=BOTH)

    def onViewFormatMail(self):
        """
        possibly threaded: view selected messages - pop up formatted display
        not threaded if in savefile list, or messages are already loaded
        the after action runs only if getMessages prefetch allowed and worked
        """
        msgnums = self.verifySelectedMsgs( )
        if msgnums:
            self.getMessages(msgnums, after=lambda: self.contViewFmt(msgnums))

    def contViewFmt(self, msgnums):
        for msgnum in msgnums:
            fulltext = self.getMessage(msgnum)
            message  = self.parseMessage(fulltext)
            type, content = self.findMainText(message)
            content  = wraplines.wrapText1(content, mailconfig.wrapsz)
            ViewWindow(headermap   = message,
                       showtext    = content,
                       origmessage = message)

            # non-multipart, content-type text/HTML (rude but true!)
            # can also be opened manually from Split or part button
            # if non-multipart, other: must open part manually with
            # Split or part button; no verify if mailconfig says so;

            if type == 'text/html':
                if ((not mailconfig.verifyHTMLTextOpen) or
                    askyesno(appname, 'Open message text in browser?')):
                    try:
                        from tempfile import gettempdir # or a Tk HTML viewer?
                        tempname = os.path.join(gettempdir( ), 'pymailgui.html')
                        open(tempname, 'w').write(content)
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
                    webbrowser.open_new('file://' + tempname)
                except:
                    show_error(appname, 'Cannot open in browser')

    def onWriteMail(self):
        # compose new email
        starttext = '\n'                        # use auto signature text
        if mailconfig.mysignature:
            starttext += '%s\n' % mailconfig.mysignature
        WriteWindow(starttext = starttext,
                    headermap = {'From': mailconfig.myaddress})

    def onReplyMail(self):
        # possibly threaded: reply to selected emails
        msgnums = self.verifySelectedMsgs( )
        if msgnums:
            self.getMessages(msgnums, after=lambda: self.contReply(msgnums))

    def contReply(self, msgnums):
        for msgnum in msgnums:
            # drop attachments, quote with '>', add signature
            fulltext = self.getMessage(msgnum)
            message  = self.parseMessage(fulltext)         # may fail: error obj
            maintext = self.findMainText(message)[1]
            maintext = wraplines.wrapText1(maintext, mailconfig.wrapsz-2)   # >
            maintext = self.quoteOrigText(maintext, message)
            if mailconfig.mysignature:
                maintext = ('\n%s\n' % mailconfig.mysignature) + maintext

            # preset initial to/from values from mail or config
            # don't use original To for From: may be many or listname
            # To keeps name+<addr> format unless any ';' present: separator
            # ideally, send should fully parse instead of splitting on ';'
            # send changes ';' to ',' required by servers; ',' common in name

            origfrom = message.get('From', '')
            ToPair   = email.Utils.parseaddr(origfrom)     # 1st (name, addr)
            ToStr    = email.Utils.formataddr(ToPair)      # ignore Reply-to
            From     = mailconfig.myaddress                # don't try 'To'
            Subj     = message.get('Subject', '(no subject)')
            if not Subj.startswith('Re:'):
                Subj = 'Re: ' + Subj
            if ';' not in ToStr:                           # uses separator?
                To = ToStr                                 # use name+addr
            else:
                To = ToPair[1]                             # use just addr
            ReplyWindow(starttext = maintext,
                        headermap = {'From': From, 'To': To, 'Subject': Subj})

    def onFwdMail(self):
        # possibly threaded: forward selected emails
        msgnums = self.verifySelectedMsgs( )
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
        if msgnums:
            self.getMessages(msgnums, after=lambda: self.contFwd(msgnums))

    def contFwd(self, msgnums):
        for msgnum in msgnums:
            # drop attachments, quote with '>', add signature
            fulltext = self.getMessage(msgnum)
            message  = self.parseMessage(fulltext)
            maintext = self.findMainText(message)[1]
            maintext = wraplines.wrapText1(maintext, mailconfig.wrapsz-2)
            maintext = self.quoteOrigText(maintext, message)
            if mailconfig.mysignature:
                maintext = ('\n%s\n' % mailconfig.mysignature) + maintext

            # initial from value from config, not mail
            From = mailconfig.myaddress
            Subj = message.get('Subject', '(no subject)')
            if not Subj.startswith('Fwd: '):
                Subj = 'Fwd: ' + Subj
            ForwardWindow(starttext = maintext,
                          headermap = {'From': From, 'Subject': Subj})

    def onSaveMailFile(self):
        """
        save selected emails for offline viewing
        disabled if target file load/delete is in progress
        disabled by getMessages if self is a busy file too
        contSave not threaded: disables all other actions
        """
        msgnums = self.selectedMsgs( )
        if not msgnums:
            showerror(appname, 'No message selected')
        else:
            # caveat: dialog warns about replacing file
            filename = self.saveDialog.show( )             # shared class attr
            if filename:                                   # don't verify num msgs
                filename = os.path.abspath(filename)       # normalize / to \
                self.getMessages(msgnums,
                        after=lambda: self.contSave(msgnums, filename))

    def contSave(self, msgnums, filename):
        # test busy now, after poss srvr msgs load
        if (filename in openSaveFiles.keys( ) and           # viewing this file?
            openSaveFiles[filename].openFileBusy):          # load/del occurring?
            showerror(appname, 'Target file busy - cannot save')
        else:
            try:
                fulltextlist = []
                mailfile = open(filename, 'a')             # caveat:not threaded
                for msgnum in msgnums:                     # < 1sec for N megs
                    fulltext = self.getMessage(msgnum)     # but poss many msgs
                    if fulltext[-1] != '\n': fulltext += '\n'
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
                mailfile.write(saveMailSeparator)
                mailfile.write(fulltext)
                fulltextlist.append(fulltext)
            mailfile.close( )
        except:
            showerror(appname, 'Error during save')
            printStack(sys.exc_info( ))
        else:                                           # why .keys( ): EIBTI
            if filename in openSaveFiles.keys( ):       # viewing this file?
                window = openSaveFiles[filename]        # update list, raise
                window.addSavedMails(fulltextlist)      # avoid file reload
                #window.loadMailFileThread( )           # this was very slow

def onOpenMailFile(self, filename=None):
    # process saved mail offline
    filename = filename or self.openDialog.show( )      # shared class attr
    if filename:
        filename = os.path.abspath(filename)            # match on full name
        if openSaveFiles.has_key(filename):             # only 1 win per file
            openSaveFiles[filename].lift( )             # raise file's window
            showinfo(appname, 'File already open')      # else deletes odd
        else:
            from PyMailGui2 import PyMailFileWindow     # avoid duplicate win
            popup = PyMailFileWindow(filename)          # new list window
            openSaveFiles[filename] = popup             # removed in quit
            popup.loadMailFileThread( )                 # try load in thread

def onDeleteMail(self):
    # delete selected mails from server or file
    msgnums = self.selectedMsgs( )                      # subclass: fillIndex
    if not msgnums:                                     # always verify here
        showerror(appname, 'No message selected')
    else:
        if askyesno(appname, 'Verify delete %d mails?' % len(msgnums)):
            self.doDelete(msgnums)

##################
# utility methods
##################

def selectedMsgs(self):
    # get messages selected in main listbox
    selections = self.listBox.curselection( )  # tuple of digit strs, 0..N-1
    return [int(x)+1 for x in selections]      # convert to ints, make 1..N

warningLimit = 15
def verifySelectedMsgs(self):
    msgnums = self.selectedMsgs( )
    if not msgnums:
        showerror(appname, 'No message selected')
    else:
        numselects = len(msgnums)
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
            if numselects > self.warningLimit:
                if not askyesno(appname, 'Open %d selections?' % numselects):
                    msgnums = []
        return msgnums

    def fillIndex(self, maxhdrsize=25):
        # fill all of main listbox
        hdrmaps  = self.headersMaps( )                  # may be empty
        showhdrs = ('Subject', 'From', 'Date', 'To')    # default hdrs to show
        if hasattr(mailconfig, 'listheaders'):          # mailconfig customizes
            showhdrs = mailconfig.listheaders or showhdrs

        # compute max field sizes <= hdrsize
        maxsize = {}
        for key in showhdrs:
            allLens = [len(msg.get(key, '')) for msg in hdrmaps]
            if not allLens: allLens = [1]
            maxsize[key] = min(maxhdrsize, max(allLens))

        # populate listbox with fixed-width left-justified fields
        self.listBox.delete(0, END)                     # show multiparts with *
        for (ix, msg) in enumerate(hdrmaps):            # via content-type hdr
            msgtype = msg.get_content_maintype( )       # no is_multipart yet
            msgline = (msgtype == 'multipart' and '*') or ' '
            msgline += '%03d' % (ix+1)
            for key in showhdrs:
                mysize  = maxsize[key]
                keytext = msg.get(key, ' ')
                msgline += ' | %-*s' % (mysize, keytext[:mysize])
            msgline += '| %.1fK' % (self.mailSize(ix+1) / 1024.0)
            self.listBox.insert(END, msgline)
        self.listBox.see(END)         # show most recent mail=last line

    def quoteOrigText(self, maintext, message):
        quoted   = '\n-----Original Message-----\n'
        for hdr in ('From', 'To', 'Subject', 'Date'):
            quoted += '%s: %s\n' % (hdr, message.get(hdr, '?'))
        quoted   = quoted + '\n' + maintext
        quoted   = '\n' + quoted.replace('\n', '\n> ')
        return quoted

    ########################
    # subclass requirements
    ########################

    def getMessages(self, msgnums, after):        # used by view,save,reply,fwd
        after( )                                  # redef if cache, thread test

    # plus okayToQuit?, any unique actions
    def getMessage(self, msgnum): assert False    # used by many: full mail text
    def headersMaps(self): assert False           # fillIndex: hdr mappings list
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
    def mailSize(self, msgnum): assert False      # fillIndex: size of msgnum
    def doDelete(self): assert False              # onDeleteMail: delete button


################################################################################
# main window - when viewing messages in local save file (or sent-mail file)
################################################################################


class PyMailFile(PyMailCommon):
    """
    customize for viewing saved-mail file offline
    a Tk, Toplevel, or Frame, with main mail listbox
    opens and deletes run in threads for large files

    save and send not threaded, because only append to
    file; save is disabled if source or target file busy
    with load/delete; save disables load, delete, save
    just because it is not run in a thread (blocks GUI);

    TBD: may need thread and O/S file locks if saves ever
    do run in threads: saves could disable other threads
    with openFileBusy, but file may not be open in GUI;
    file locks not sufficient, because GUI updated too;
    TBD: appends to sent-mail file may require O/S locks:
    as is, user gets error pop up if sent during load/del;
    """
    def actions(self):
        return [ ('View',   self.onViewFormatMail),
                 ('Delete', self.onDeleteMail),
                 ('Write',  self.onWriteMail),
                 ('Reply',  self.onReplyMail),
                 ('Fwd',    self.onFwdMail),
                 ('Save',   self.onSaveMailFile),
                 ('Open',   self.onOpenMailFile),
                 ('Quit',   self.quit) ]

    def __init__(self, filename):
        # caller: do loadMailFileThread next
        PyMailCommon.__init__(self)
        self.filename = filename
        self.openFileBusy = threadtools.ThreadCounter( )      # one per window

    def loadMailFileThread(self):
        """
        Load or reload file and update window index list;
        called on Open, startup, and possibly on Send if
        sent-mail file appended is currently open;  there
        is always a bogus first item after the text split;
        alt: [self.parseHeaders(m) for m in self.msglist];
        could pop up a busy dialog, but quick for small files;
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
        2.1: this is now threaded--else runs < 1sec for N meg
        files, but can pause GUI N seconds if very large file;
        Save now uses addSavedMails to append msg lists for
        speed, not this reload;  still called from Send just
        because msg text unavailable - requires refactoring;
        delete threaded too: prevent open and delete overlap;
        """
        if self.openFileBusy:
            # don't allow parallel open/delete changes
            errmsg = 'Cannot load, file is busy:\n"%s"' % self.filename
            showerror(appname, errmsg)
        else:
            #self.listBox.insert(END, 'loading...')      # error if user clicks
            savetitle = self.title( )                    # set by window class
            self.title(appname + ' - ' + 'Loading...')
            self.openFileBusy.incr( )
            threadtools.startThread(
                action     = self.loadMailFile,
                args       = ( ),
                context    = (savetitle,),
                onExit     = self.onLoadMailFileExit,
                onFail     = self.onLoadMailFileFail)

    def loadMailFile(self):
        # run in a thread while GUI is active
        # open, read, parser may all raise excs
        allmsgs = open(self.filename).read( )
        self.msglist  = allmsgs.split(saveMailSeparator)[1:]      # full text
        self.hdrlist  = map(self.parseHeaders, self.msglist)      # msg objects

    def onLoadMailFileExit(self, savetitle):
        # on thread success
        self.title(savetitle)         # reset window title to filename
        self.fillIndex( )             # updates GUI: do in main thread
        self.lift( )                  # raise my window
        self.openFileBusy.decr( )

    def onLoadMailFileFail(self, exc_info, savetitle):
        # on thread exception
        showerror(appname, 'Error opening "%s"\n%s\n%s' %
                           ((self.filename,) +  exc_info[:2]))
        printStack(exc_info)
        self.destroy( )               # always close my window?
        self.openFileBusy.decr( )     # not needed if destroy

    def addSavedMails(self, fulltextlist):
        """
        optimization: extend loaded file lists for mails
        newly saved to this window's file; in past called
        loadMailThread to reload entire file on save - slow;
        must be called in main GUI thread only: updates GUI;
        sends still reloads sent file if open: no msg text;
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
        """
        self.msglist.extend(fulltextlist)
        self.hdrlist.extend(map(self.parseHeaders, fulltextlist))
        self.fillIndex( )
        self.lift( )

    def doDelete(self, msgnums):
        """
        simple-minded, but sufficient: rewrite all
        nondeleted mails to file; can't just delete
        from self.msglist in-place: changes item indexes;
        Py2.3 enumerate(L) same as zip(range(len(L)), L)
        2.1: now threaded, else N sec pause for large files
        """
        if self.openFileBusy:
            # dont allow parallel open/delete changes
            errmsg = 'Cannot delete, file is busy:\n"%s"' % self.filename
            showerror(appname, errmsg)
        else:
            savetitle = self.title( )
            self.title(appname + ' - ' + 'Deleting...')
            self.openFileBusy.incr( )
            threadtools.startThread(
                action     = self.deleteMailFile,
                args       = (msgnums,),
                context    = (savetitle,),
                onExit     = self.onDeleteMailFileExit,
                onFail     = self.onDeleteMailFileFail)

    def deleteMailFile(self, msgnums):
        # run in a thread while GUI active
        indexed = enumerate(self.msglist)
        keepers = [msg for (ix, msg) in indexed if ix+1 not in msgnums]
        allmsgs = saveMailSeparator.join(['']  + keepers)
        open(self.filename, 'w').write(allmsgs)
        self.msglist = keepers
        self.hdrlist = map(self.parseHeaders, self.msglist)

    def onDeleteMailFileExit(self, savetitle):
        self.title(savetitle)
        self.fillIndex( )              # updates GUI: do in main thread
        self.lift( )                   # reset my title, raise my window
        self.openFileBusy.decr( )

    def onDeleteMailFileFail(self, exc_info, savetitle):
        showerror(appname, 'Error deleting "%s"\n%s\n%s' %
                           ((self.filename,) +  exc_info[:2]))
        printStack(exc_info)
        self.destroy( )                # always close my window?
        self.openFileBusy.decr( )      # not needed if destroy
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
    def getMessages(self, msgnums, after):
        """
        used by view,save,reply,fwd: file load and delete
        threads may change the msg and hdr lists, so disable
        all other operations that depend on them to be safe;
        this test is for self: saves also test target file;
        """
        if self.openFileBusy:
            errmsg = 'Cannot fetch, file is busy:\n"%s"' % self.filename
            showerror(appname, errmsg)
        else:
            after( )                      # mail already loaded

    def getMessage(self, msgnum):
        return self.msglist[msgnum-1]    # full text of 1 mail

    def headersMaps(self):
        return self.hdrlist              # email.Message objects

    def mailSize(self, msgnum):
        return len(self.msglist[msgnum-1])

    def quit(self):
        # don't destroy during update: fillIndex next
        if self.openFileBusy:
            showerror(appname, 'Cannot quit during load or delete')
        else:
            if askyesno(appname, 'Verify Quit Window?'):
                # delete file from open list
                del openSaveFiles[self.filename]
                Toplevel.destroy(self)


################################################################################
# main window - when viewing messages on the mail server
################################################################################


class PyMailServer(PyMailCommon):
    """
    customize for viewing mail still on server
    a Tk, Toplevel, or Frame, with main mail listbox
    maps load, fetch, delete actions to server inbox
    embeds a MessageCache, which is a MailFetcher
    """
    def actions(self):
        return [ ('Load',   self.onLoadServer),
                 ('View',   self.onViewFormatMail),
                 ('Delete', self.onDeleteMail),
                 ('Write',  self.onWriteMail),
                 ('Reply',  self.onReplyMail),
                 ('Fwd',    self.onFwdMail),
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
                ('Save',   self.onSaveMailFile),
                ('Open',   self.onOpenMailFile),
                ('Quit',   self.quit) ]

    def __init__(self):
        PyMailCommon.__init__(self)
        self.cache = messagecache.GuiMessageCache()    # embedded, not inherited
       #self.listBox.insert(END, 'Press Load to fetch mail')

    def makeWidgets(self):                             # help bar: main win only
        self.addHelp()
        PyMailCommon.makeWidgets(self)

    def addHelp(self):
        msg = 'PyMailGUI - a Python/Tkinter email client  (help)'
        title = Button(self, text=msg)
        title.config(bg='steelblue', fg='white', relief=RIDGE)
        title.config(command=self.onShowHelp)
        title.pack(fill=X)

    def onShowHelp(self):
        """
        load,show text block string
        could use HTML and web browser module here too
        but that adds an external dependency
        """
        from PyMailGuiHelp import helptext
        popuputil.HelpPopup(appname, helptext, showsource=self.onShowMySource)

    def onShowMySource(self, showAsMail=False):
        # display my sourcecode file, plus imported modules here & elsewhere
        import PyMailGui2, ListWindows, ViewWindows, SharedNames
        from PP3E.Internet.Email.mailtools import (    # mailtools now a pkg
             mailSender, mailFetcher, mailParser)      # can't use * in def
        mymods = (
            PyMailGui2, ListWindows, ViewWindows, SharedNames,
            PyMailGuiHelp, popuputil, messagecache, wraplines,
            mailtools, mailFetcher, mailSender, mailParser,
            mailconfig, threadtools, windows, textEditor)
        for mod in mymods:
            source = mod.__file__
            if source.endswith('.pyc'):
                source = source[:-4] + '.py'       # assume in same dir, .py
            if showAsMail:
                # this is a bit cheesey...
                code   = open(source).read()
                user   = mailconfig.myaddress
                hdrmap = {'From': appname, 'To': user, 'Subject': mod.__name__}
                ViewWindow(showtext=code,
                           headermap=hdrmap,
                           origmessage=email.Message.Message())
            else:
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
            # more useful text editor
            wintitle = ' - ' + mod.__name__
            textEditor.TextEditorMainPopup(self, source, wintitle)

    def onLoadServer(self, forceReload=False):
        """
        threaded: load or reload mail headers list on request
        Exit,Fail,Progress run by threadChecker after callback via queue
        may overlap with sends, disables all but send
        could overlap with loadingmsgs, but may change msg cache list
        forceReload on delete/synch fail, else loads recent arrivals only;
        2.1: cache.loadHeaders may do quick check to see if msgnums
        in synch with server, if we are loading just newly arrived hdrs;
        """
        if loadingHdrsBusy or deletingBusy or loadingMsgsBusy:
            showerror(appname, 'Cannot load headers during load or delete')
        else:
            loadingHdrsBusy.incr( )
            self.cache.setPopPassword(appname) # don't update GUI in the thread!
            popup = popuputil.BusyBoxNowait(appname, 'Loading message headers')
            threadtools.startThread(
                action     = self.cache.loadHeaders,
                args       = (forceReload,),
                context    = (popup,),
                onExit     = self.onLoadHdrsExit,
                onFail     = self.onLoadHdrsFail,
                onProgress = self.onLoadHdrsProgress)

    def onLoadHdrsExit(self, popup):
        self.fillIndex( )
        popup.quit( )
        self.lift( )
        loadingHdrsBusy.decr( )

    def onLoadHdrsFail(self, exc_info, popup):
        popup.quit( )
        showerror(appname, 'Load failed: \n%s\n%s' % exc_info[:2])
        printStack(exc_info)                       # send stack trace to stdout
        loadingHdrsBusy.decr( )
        if exc_info[0] == mailtools.MessageSynchError:    # synch inbox/index
            self.onLoadServer(forceReload=True)           # new thread: reload
        else:
            self.cache.popPassword = None          # force re-input next time

    def onLoadHdrsProgress(self, i, n, popup):
        popup.changeText('%d of %d' % (i, n))

    def doDelete(self, msgnumlist):
        """
        threaded: delete from server now - changes msg nums;
        may overlap with sends only, disables all except sends;
        2.1: cache.deleteMessages now checks TOP result to see
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
        if headers match selected mails, in case msgnums out of
        synch with mail server: poss if mail deleted by other client,
        or server deletes inbox mail automatically - some ISPs may
        move a mail from inbox to undeliverable on load failure;
        """
        if loadingHdrsBusy or deletingBusy or loadingMsgsBusy:
            showerror(appname, 'Cannot delete during load or delete')
        else:
            deletingBusy.incr( )
            popup = popuputil.BusyBoxNowait(appname, 'Deleting selected mails')
            threadtools.startThread(
                action     = self.cache.deleteMessages,
                args       = (msgnumlist,),
                context    = (popup,),
                onExit     = self.onDeleteExit,
                onFail     = self.onDeleteFail,
                onProgress = self.onDeleteProgress)

    def onDeleteExit(self, popup):
        self.fillIndex( )                    # no need to reload from server
        popup.quit( )                        # refill index with updated cache
        self.lift( )                         # raise index window, release lock
        deletingBusy.decr( )

    def onDeleteFail(self, exc_info, popup):
        popup.quit( )
        showerror(appname, 'Delete failed: \n%s\n%s' % exc_info[:2])
        printStack(exc_info)
        deletingBusy.decr( )                 # delete or synch check failure
        self.onLoadServer(forceReload=True)  # new thread: some msgnums changed

    def onDeleteProgress(self, i, n, popup):
        popup.changeText('%d of %d' % (i, n))

    def getMessages(self, msgnums, after):
        """
        threaded: prefetch all selected messages into cache now
        used by save, view, reply, and forward to prefill cache
        may overlap with other loadmsgs and sends, disables delete
        only runs "after" action if the fetch allowed and successful;
        2.1: cache.getMessages tests if index in synch with server,
        but we only test if we have to go to server, not if cached;
        """
        if loadingHdrsBusy or deletingBusy:
            showerror(appname, 'Cannot fetch message during load or delete')
        else:
            toLoad = [num for num in msgnums if not self.cache.isLoaded(num)]
            if not toLoad:
                after( )         # all already loaded
                return           # process now, no wait pop up
            else:
                loadingMsgsBusy.incr( )
```

Example 15-3. PP3E\Internet\Email\PyMailGui\ListWindows.py (continued)

```
                from popuputil import BusyBoxNowait
                popup = BusyBoxNowait(appname, 'Fetching message contents')
                threadtools.startThread(
                    action     = self.cache.getMessages,
                    args       = (toLoad,),
                    context    = (after, popup),
                    onExit     = self.onLoadMsgsExit,
                    onFail     = self.onLoadMsgsFail,
                    onProgress = self.onLoadMsgsProgress)

    def onLoadMsgsExit(self, after, popup):
        popup.quit( )
        after( )
        loadingMsgsBusy.decr( )    # allow others after afterExit done

    def onLoadMsgsFail(self, exc_info, after, popup):
        popup.quit( )
        showerror(appname, 'Fetch failed: \n%s\n%s' % exc_info[:2])
        printStack(exc_info)
        loadingMsgsBusy.decr( )
        if exc_info[0] == mailtools.MessageSynchError:      # synch inbox/index
            self.onLoadServer(forceReload=True)             # new thread: reload

    def onLoadMsgsProgress(self, i, n, after, popup):
        popup.changeText('%d of %d' % (i, n))

    def getMessage(self, msgnum):
        return self.cache.getMessage(msgnum)                # full mail text

    def headersMaps(self):
        return map(self.parseHeaders, self.cache.allHdrs( )) # email.Message objs

    def mailSize(self, msgnum):
        return self.cache.getSize(msgnum)

    def okayToQuit(self):
        # any threads still running?
        filesbusy = [win for win in openSaveFiles.values( ) if win.openFileBusy]
        busy = loadingHdrsBusy or deletingBusy or sendingBusy or loadingMsgsBusy
        busy = busy or filesbusy
        return not busy
```

ViewWindows: Message View Windows

Example 15-4 lists the implementation of mail view and edit windows. These windows are created in response to actions in list windows—View, Write, Reply, and Forward buttons. See the callbacks for these actions in the list window module of Example 15-3 for view window initiation calls.

As in the prior module (Example 15-3), this file is really one common class and a handful of customizations. The mail view window is nearly identical to the mail edit

window, used for Write, Reply, and Forward requests. Consequently, this example defines the common appearance and behavior in the view window superclass, and extends it by subclassing for edit windows.

Replies and forwards are hardly different from the write window here, because their details (e.g., From: and To: addresses, and quoted message text) are worked out in the list window implementation before an edit window is created.

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py

```
################################################################################
# Implementation of View, Write, Reply, Forward windows: one class per kind.
# Code is factored here for reuse: a Write window is a customized View window,
# and Reply and Forward are custom Write windows.  Windows defined in this
# file are created by the list windows, in response to user actions.  Caveat:
# the 'split' pop ups for opening parts/attachments feel a bit nonintuitive.
# 2.1: this caveat was addressed, by adding quick-access attachment buttons.
# TBD: could avoid verifying quits unless text area modified (like PyEdit2.0),
# but these windows are larger, and would not catch headers already changed.
# TBD: should Open dialog in write windows be program-wide? (per-window now).
################################################################################

from SharedNames import *     # program-wide global objects


################################################################################
# message view window - also a superclass of write, reply, forward
################################################################################


class ViewWindow(windows.PopupWindow, mailtools.MailParser):
    """
    a custom Toplevel, with embedded TextEditor
    inherits saveParts,partsList from mailtools.MailParser
    """
    # class attributes
    modelabel       = 'View'                    # used in window titles
    from mailconfig import okayToOpenParts      # open any attachments at all?
    from mailconfig import verifyPartOpens      # ask before open each part?
    from mailconfig import maxPartButtons       # show up to this many + '...'
    tempPartDir     = 'TempParts'               # where 1 selected part saved

    # all view windows use same dialog: remembers last dir
    partsDialog = Directory(title=appname + ': Select parts save directory')

    def __init__(self, headermap, showtext, origmessage=None):
        """
        header map is origmessage, or custom hdr dict for writing;
        showtext is main text part of the message: parsed or custom;
        origmessage is parsed email.Message for view mail windows
        """
        windows.PopupWindow.__init__(self, appname, self.modelabel)
        self.origMessage = origmessage
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
        self.makeWidgets(headermap, showtext)

    def makeWidgets(self, headermap, showtext):
        """
        add headers, actions, attachments, text editor
        """
        actionsframe = self.makeHeaders(headermap)
        if self.origMessage and self.okayToOpenParts:
            self.makePartButtons( )
        self.editor  = textEditor.TextEditorComponentMinimal(self)
        myactions    = self.actionButtons( )
        for (label, callback) in myactions:
            b = Button(actionsframe, text=label, command=callback)
            b.config(bg='beige', relief=RIDGE, bd=2)
            b.pack(side=TOP, expand=YES, fill=BOTH)

        # body text, pack last=clip first
        self.editor.pack(side=BOTTOM)               # may be multiple editors
        self.editor.setAllText(showtext)            # each has own content
        lines = len(showtext.splitlines( ))
        lines = min(lines + 3, mailconfig.viewheight or 20)
        self.editor.setHeight(lines)                # else height=24, width=80
        self.editor.setWidth(80)                    # or from PyEdit textConfig
        if mailconfig.viewbg:
            self.editor.setBg(mailconfig.viewbg)    # colors, font in mailconfig
        if mailconfig.viewfg:
            self.editor.setFg(mailconfig.viewfg)
        if mailconfig.viewfont:                     # also via editor Tools menu
            self.editor.setFont(mailconfig.viewfont)

    def makeHeaders(self, headermap):
        """
        add header entry fields, return action buttons frame
        """
        top    = Frame(self); top.pack   (side=TOP,   fill=X)
        left   = Frame(top);  left.pack  (side=LEFT,  expand=NO,  fill=BOTH)
        middle = Frame(top);  middle.pack(side=LEFT,  expand=NO,  fill=NONE)
        right  = Frame(top);  right.pack (side=RIGHT, expand=YES, fill=BOTH)

        # headers set may be extended in mailconfig (Bcc)
        self.userHdrs = ( )
        showhdrs = ('From', 'To', 'Cc', 'Subject')
        if hasattr(mailconfig, 'viewheaders') and mailconfig.viewheaders:
            self.userHdrs = mailconfig.viewheaders
            showhdrs += self.userHdrs

        self.hdrFields = []
        for header in showhdrs:
            lab = Label(middle, text=header+':', justify=LEFT)
            ent = Entry(right)
            lab.pack(side=TOP, expand=YES, fill=X)
            ent.pack(side=TOP, expand=YES, fill=X)
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
            ent.insert('0', headermap.get(header, '?'))
            self.hdrFields.append(ent)              # order matters in onSend
        return left

    def actionButtons(self):                        # must be method for self
        return [('Cancel', self.destroy),           # close view window silently
                ('Parts',  self.onParts),           # multiparts list or the body
                ('Split',  self.onSplit)]

    def makePartButtons(self):
        """
        add up to N buttons that open attachments/parts
        when clicked; alternative to Parts/Split (2.1);
        okay that temp dir is shared by all open messages:
        part file not saved till later selected and opened;
        partname=partname is required in lambda in Py2.4;
        caveat: we could try to skip the main text part;
        """
        def makeButton(parent, text, callback):
            link = Button(parent, text=text, command=callback, relief=SUNKEN)
            if mailconfig.partfg: link.config(fg=mailconfig.partfg)
            if mailconfig.partbg: link.config(bg=mailconfig.partbg)
            link.pack(side=LEFT, fill=X, expand=YES)

        parts = Frame(self)
        parts.pack(side=TOP, expand=NO, fill=X)
        for (count, partname) in enumerate(self.partsList(self.origMessage)):
            if count == self.maxPartButtons:
                makeButton(parts, '...', self.onSplit)
                break
            openpart = (lambda partname=partname: self.onOnePart(partname))
            makeButton(parts, partname, openpart)

    def onOnePart(self, partname):
        """
        locate selected part for button and save and open
        okay if multiple mails open: resaves each time selected
        we could probably just use web browser directly here
        caveat: tempPartDir is relative to cwd - poss anywhere
        caveat: tempPartDir is never cleaned up: might be large,
        could use tempfile module like HTML main text part code;
        """
        try:
            savedir  = self.tempPartDir
            message  = self.origMessage
            (contype, savepath) = self.saveOnePart(savedir, partname, message)
        except:
            showerror(appname, 'Error while writing part file')
            printStack(sys.exc_info( ))
        else:
            self.openParts([(contype, os.path.abspath(savepath))])
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
    def onParts(self):
        """
        show message part/attachments in pop-up window;
        uses same file naming scheme as save on Split;
        if non-multipart, single part = full body text
        """
        partnames = self.partsList(self.origMessage)
        msg = '\n'.join(['Message parts:\n'] + partnames)
        showinfo(appname, msg)

    def onSplit(self):
        """
        pop up save dir dialog and save all parts/attachments there;
        if desired, pop up HTML and multimedia parts in web browser,
        text in TextEditor, and well-known doc types on windows;
        could show parts in View windows where embedded text editor
        would provide a save button, but most are not readable text;
        """
        savedir = self.partsDialog.show( )          # class attr: at prior dir
        if savedir:                                 # tk dir chooser, not file
            try:
                partfiles = self.saveParts(savedir, self.origMessage)
            except:
                showerror(appname, 'Error while writing part files')
                printStack(sys.exc_info( ))
            else:
                if self.okayToOpenParts: self.openParts(partfiles)

    def askOpen(self, appname, prompt):
        if not self.verifyPartOpens:
            return True
        else:
            return askyesno(appname, prompt)   # pop-up dialog

    def openParts(self, partfiles):
        """
        auto-open well known and safe file types, but
        only if verified by the user in a pop up; other
        types must be opened manually from save dir;
        caveat: punts for type application/octet-stream
        even if safe filename extension such as .html;
        caveat: image/audio/video could use playfile.py;
        """
        for (contype, fullfilename) in partfiles:
            maintype  = contype.split('/')[0]                     # left side
            extension = os.path.splitext(fullfilename)[1]         # or [-4:]
            basename  = os.path.basename(fullfilename)            # strip dir

            # HTML and XML text, web pages, some media
            if contype  in ['text/html', 'text/xml']:
                if self.askOpen(appname, 'Open "%s" in browser?' % basename):
                    try:
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
                    webbrowser.open_new('file://' + fullfilename)
                except:
                    showerror(appname, 'Browser failed: using editor')
                    textEditor.TextEditorMainPopup(self, fullfilename)

        # text/plain, text/x-python, etc.
        elif maintype == 'text':
            if self.askOpen(appname, 'Open text part "%s"?' % basename):
                textEditor.TextEditorMainPopup(self, fullfilename)

        # multimedia types: Windows opens mediaplayer, imageviewer, etc.
        elif maintype in ['image', 'audio', 'video']:
            if self.askOpen(appname, 'Open media part "%s"?' % basename):
                try:
                    webbrowser.open_new('file://' + fullfilename)
                except:
                    showerror(appname, 'Error opening browser')

        # common Windows documents: Word, Adobe, Excel, archives, etc.
        elif (sys.platform[:3] == 'win' and
              maintype == 'application' and
              extension in ['.doc', '.pdf', '.xls', '.zip','.tar', '.wmv']):
                if self.askOpen(appname, 'Open part "%s"?' % basename):
                    os.startfile(fullfilename)

        else:  # punt!
            msg = 'Cannot open part: "%s"\nOpen manually in: "%s"'
            msg = msg % (basename, os.path.dirname(fullfilename))
            showinfo(appname, msg)


################################################################################
# message edit windows - write, reply, forward
################################################################################


if mailconfig.smtpuser:                             # user set in mailconfig?
    MailSenderClass = mailtools.MailSenderAuth      # login/password required
else:
    MailSenderClass = mailtools.MailSender


class WriteWindow(ViewWindow, MailSenderClass):
    """
    customize view display for composing new mail
    inherits sendMessage from mailtools.MailSender
    """
    modelabel = 'Write'

    def __init__(self, headermap, starttext):
        ViewWindow.__init__(self, headermap, starttext)
        MailSenderClass.__init__(self)
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
        self.attaches   = []                    # each win has own open dialog
        self.openDialog = None                  # dialog remembers last dir

    def actionButtons(self):
        return [('Cancel', self.quit),          # need method to use self
                ('Parts',  self.onParts),       # PopupWindow verifies cancel
                ('Attach', self.onAttach),
                ('Send  ', self.onSend)]

    def onParts(self):
        # caveat: deletes not currently supported
        if not self.attaches:
            showinfo(appname, 'Nothing attached')
        else:
            msg = '\n'.join(['Already attached:\n'] + self.attaches)
            showinfo(appname, msg)

    def onAttach(self):
        """
        attach a file to the mail: name added
        here will be added as a part on Send;
        """
        if not self.openDialog:
            self.openDialog = Open(title=appname + ': Select Attachment File')
        filename = self.openDialog.show( )        # remember prior dir
        if filename:
            self.attaches.append(filename)       # to be opened in send method

    def onSend(self):
        """
        threaded: mail edit window send button press
        may overlap with any other thread, disables none but quit
        Exit,Fail run by threadChecker via queue in after callback
        caveat: no progress here, because send mail call is atomic
        assumes multiple recipient addrs are separated with ';'

        caveat: should parse To,Cc,Bcc instead of splitting on ';',
        or use a multiline input widgets instead of simple entry;
        as is, reply logic and GUI user must avoid embedded ';'
        characters in addresses - very unlikely but not impossible;
        mailtools module saves sent message text in a local file
        """
        fieldvalues = [entry.get( ) for entry in self.hdrFields]
        From, To, Cc, Subj = fieldvalues[:4]
        extraHdrs  = [('Cc', Cc), ('X-Mailer', appname + ' (Python)')]
        extraHdrs += zip(self.userHdrs, fieldvalues[4:])
        bodytext = self.editor.getAllText( )

        # split multiple recipient lists, fix empty fields
        Tos = To.split(';')                                  # split to list
        Tos = [addr.strip( ) for addr in Tos]                # spaces around
        for (ix, (name, value)) in enumerate(extraHdrs):
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
            if value:                                           # ignored if ''
                if value == '?':                                # ? not replaced
                    extraHdrs[ix] = (name, '')
                elif name.lower( ) in ['cc', 'bcc']:
                    values = value.split(';')
                    extraHdrs[ix] = (name, [addr.strip( ) for addr in values])

        # withdraw to disallow send during send
        # caveat: withdraw not foolproof- user may deiconify
        self.withdraw( )
        self.getPassword( )      # if needed; don't run pop up in send thread!
        popup = popuputil.BusyBoxNowait(appname, 'Sending message')
        sendingBusy.incr( )
        threadtools.startThread(
            action  = self.sendMessage,
            args    = (From, Tos, Subj, extraHdrs, bodytext, self.attaches,
                                                  saveMailSeparator),
            context = (popup,),
            onExit  = self.onSendExit,
            onFail  = self.onSendFail)

    def onSendExit(self, popup):
        # erase wait window, erase view window, decr send count
        # sendMessage call auto saves sent message in local file
        # can't use window.addSavedMails: mail text unavailable
        popup.quit( )
        self.destroy( )
        sendingBusy.decr( )

        # poss \ when opened, / in mailconfig
        sentname = os.path.abspath(mailconfig.sentmailfile)  # also expands '.'
        if sentname in openSaveFiles.keys( ):                # sent file open?
            window = openSaveFiles[sentname]                 # update list,raise
            window.loadMailFileThread( )

    def onSendFail(self, exc_info, popup):
        # pop-up error, keep msg window to save or retry, redraw actions frame
        popup.quit( )
        self.deiconify( )
        self.lift( )
        showerror(appname, 'Send failed: \n%s\n%s' % exc_info[:2])
        printStack(exc_info)
        self.smtpPassword = None        # try again
        sendingBusy.decr( )

    def askSmtpPassword(self):
        """
        get password if needed from GUI here, in main thread
        caveat: may try this again in thread if no input first
        time, so goes into a loop until input is provided; see
        pop paswd input logic for a nonlooping alternative
        """
```

Example 15-4. PP3E\Internet\Email\PyMailGui\ViewWindows.py (continued)

```
        password = ''
        while not password:
            prompt = ('Password for %s on %s?' %
                      (self.smtpUser, self.smtpServerName))
            password = popuputil.askPasswordWindow(appname, prompt)
        return password


class ReplyWindow(WriteWindow):
    """
    customize write display for replying
    text and headers set up by list window
    """
    modelabel = 'Reply'


class ForwardWindow(WriteWindow):
    """
    customize reply display for forwarding
    text and headers set up by list window
    """
    modelabel = 'Forward'
```

messagecache: Message Cache Manager

The class in Example 15-5 implements a cache for already loaded messages. Its logic is split off into this file in order to avoid complicating list window implementations. The server list window creates and embeds an instance of this class to interface with the mail server, and to keep track of already loaded mail headers and full text.

Example 15-5. PP3E\Internet\Email\PyMailGui\messagecache.py

```
################################################################################
# manage message and header loads and context, but not GUI
# a MailFetcher, with a list of already loaded headers and messages
# the caller must handle any required threading or GUI interfaces
################################################################################

from PP3E.Internet.Email import mailtools
from popuputil import askPasswordWindow


class MessageInfo:
    """
    an item in the mail cache list
    """
    def __init__(self, hdrtext, size):
        self.hdrtext  = hdrtext           # fulltext is cached msg
        self.fullsize = size              # hdrtext is just the hdrs
        self.fulltext = None              # fulltext=hdrtext if no TOP
```

Example 15-5. PP3E\Internet\Email\PyMailGui\messagecache.py (continued)

```
class MessageCache(mailtools.MailFetcher):
    """
    keep track of already loaded headers and messages
    inherits server transfer methods from MailFetcher
    useful in other apps: no GUI or thread assumptions
    """
    def __init__(self):
        mailtools.MailFetcher.__init__(self)
        self.msglist = []

    def loadHeaders(self, forceReloads, progress=None):
        """
        three cases to handle here: the initial full load,
        load newly arrived, and forced reload after delete;
        don't refetch viewed msgs if hdrs list same or extended;
        retains cached msgs after a delete unless delete fails;
        2.1: does quick check to see if msgnums still in sync
        """
        if forceReloads:
            loadfrom = 1
            self.msglist = []                          # msg nums have changed
        else:
            loadfrom = len(self.msglist)+1             # continue from last load

        # only if loading newly arrived
        if loadfrom != 1:
            self.checkSynchError(self.allHdrs())       # raises except if bad

        # get all or newly arrived msgs
        reply = self.downloadAllHeaders(progress, loadfrom)
        headersList, msgSizes, loadedFull = reply

        for (hdrs, size) in zip(headersList, msgSizes):
            newmsg = MessageInfo(hdrs, size)
            if loadedFull:                             # zip result may be empty
                newmsg.fulltext = hdrs                   # got full msg if no 'top'
            self.msglist.append(newmsg)

    def getMessage(self, msgnum):                      # get raw msg text
        if not self.msglist[msgnum-1].fulltext:        # add to cache if fetched
            fulltext = self.downloadMessage(msgnum)    # harmless if threaded
            self.msglist[msgnum-1].fulltext = fulltext
        return self.msglist[msgnum-1].fulltext

    def getMessages(self, msgnums, progress=None):
        """
        prefetch full raw text of multiple messages, in thread;
        2.1: does quick check to see if msgnums still in sync;
        we can't get here unless the index list already loaded;
        """
        self.checkSynchError(self.allHdrs())           # raises except if bad
```

Example 15-5. PP3E\Internet\Email\PyMailGui\messagecache.py (continued)

```
        nummsgs = len(msgnums)                      # adds messages to cache
        for (ix, msgnum) in enumerate(msgnums):     # some poss already there
             if progress: progress(ix+1, nummsgs)   # only connects if needed
             self.getMessage(msgnum)                # but may connect > once

    def getSize(self, msgnum):                      # encapsulate cache struct
        return self.msglist[msgnum-1].fullsize      # it changed once already!

    def isLoaded(self, msgnum):
        return self.msglist[msgnum-1].fulltext

    def allHdrs(self):
        return [msg.hdrtext for msg in self.msglist]

    def deleteMessages(self, msgnums, progress=None):
        """
        if delete of all msgnums works, remove deleted entries
        from mail cache, but don't reload either the headers list
        or already viewed mails text: cache list will reflect the
        changed msg nums on server;  if delete fails for any reason,
        caller should forceably reload all hdrs next, because _some_
        server msg nums may have changed, in unpredictable ways;
        2.1: this now checks msg hdrs to detect out of synch msg
        numbers, if TOP supported by mail server; runs in thread
        """
        try:
            self.deleteMessagesSafely(msgnums, self.allHdrs( ), progress)
        except mailtools.TopNotSupported:
            mailtools.MailFetcher.deleteMessages(self, msgnums, progress)

        # no errors: update index list
        indexed = enumerate(self.msglist)
        self.msglist = [msg for (ix, msg) in indexed if ix+1 not in msgnums]


class GuiMessageCache(MessageCache):
    """
    add any GUI-specific calls here so cache usable in non-GUI apps
    """

    def setPopPassword(self, appname):
        """
        get password from GUI here, in main thread
        forceably called from GUI to avoid pop ups in threads
        """
        if not self.popPassword:
            prompt = 'Password for %s on %s?' % (self.popUser, self.popServer)
            self.popPassword = askPasswordWindow(appname, prompt)

    def askPopPassword(self):
        """
        but don't use GUI pop up here: I am run in a thread!
```

Example 15-5. PP3E\Internet\Email\PyMailGui\messagecache.py (continued)

```
        when tried pop up in thread, caused GUI to hang;
        may be called by MailFetcher superclass, but only
        if passwd is still empty string due to dialog close
        """
        return self.popPassword
```

popuputil: General-Purpose GUI Pop Ups

Example 15-6 implements a handful of utility pop-up windows in a module, in case they ever prove useful in other programs. Note that the same windows utility module is imported here, to give a common look-and-feel to the popups (icons, titles, and so on).

Example 15-6. PP3E\Internet\Email\PyMailGui\popuputil.py

```
##############################################################################
# utility windows - may be useful in other programs
##############################################################################

from Tkinter import *
from PP3E.Gui.Tools.windows import PopupWindow


class HelpPopup(PopupWindow):
    """
    custom Toplevel that shows help text as scrolled text
    source button runs a passed-in callback handler
    alternative: use HTML file and webbrowser module
    """
    myfont = 'system'  # customizable

    def __init__(self, appname, helptext, iconfile=None, showsource=lambda:0):
        PopupWindow.__init__(self, appname, 'Help', iconfile)
        from ScrolledText import ScrolledText       # a nonmodal dialog
        bar  = Frame(self)                          # pack first=clip last
        bar.pack(side=BOTTOM, fill=X)
        code = Button(bar, bg='beige', text="Source", command=showsource)
        quit = Button(bar, bg='beige', text="Cancel", command=self.destroy)
        code.pack(pady=1, side=LEFT)
        quit.pack(pady=1, side=LEFT)
        text = ScrolledText(self)                   # add Text + scrollbar
        text.config(font=self.myfont,  width=70)    # too big for showinfo
        text.config(bg='steelblue', fg='white')     # erase on btn or return
        text.insert('0.0', helptext)
        text.pack(expand=YES, fill=BOTH)
        self.bind("<Return>", (lambda event: self.destroy()))


def askPasswordWindow(appname, prompt):
    """
    modal dialog to input password string
```

Example 15-6. PP3E\Internet\Email\PyMailGui\popuputil.py (continued)

```
    getpass.getpass uses stdin, not GUI
    tkSimpleDialog.askstring echos input
    """
    win = PopupWindow(appname, 'Prompt')               # a configured Toplevel
    Label(win, text=prompt).pack(side=LEFT)
    entvar = StringVar(win)
    ent = Entry(win, textvariable=entvar, show='*')    # display * for input
    ent.pack(side=RIGHT, expand=YES, fill=X)
    ent.bind('<Return>', lambda event: win.destroy())
    ent.focus_set(); win.grab_set(); win.wait_window()
    win.update()                                        # update forces redraw
    return entvar.get()                                 # ent widget is now gone


class BusyBoxWait(PopupWindow):
    """
    pop up blocking wait message box: thread waits
    main GUI event thread stays alive during wait
    but GUI is inoperable during this wait state;
    uses quit redef here because lower, not leftmost;
    """
    def __init__(self, appname, message):
        PopupWindow.__init__(self, appname, 'Busy')
        self.protocol('WM_DELETE_WINDOW', lambda:0)        # ignore deletes
        label = Label(self, text=message + '...')          # win.quit() to erase
        label.config(height=10, width=40, cursor='watch')  # busy cursor
        label.pack()
        self.makeModal()
        self.message, self.label = message, label
    def makeModal(self):
        self.focus_set()                                   # grab application
        self.grab_set()                                    # wait for threadexit
    def changeText(self, newtext):
        self.label.config(text=self.message + ': ' + newtext)
    def quit(self):
        self.destroy()                                     # don't verify quit

class BusyBoxNowait(BusyBoxWait):
    """
    pop up nonblocking wait window
    call changeText to show progress, quit to close
    """
    def makeModal(self):
        pass

if __name__ == '__main__':
    HelpPopup('spam', 'See figure 1...\n')
    print askPasswordWindow('spam', 'enter password')
    raw_input('Enter to exit')
```

wraplines: Line Split Tools

The module in Example 15-7 implements general tools for wrapping long lines, at either a fixed column or the first delimiter at or before a fixed column. PyMailGUI uses this file's `wrapText1` function for text in view, reply, and forward windows, but this code is potentially useful in other programs. Run the file as a script to watch its self-test code at work, and study its functions to see its text-processing logic.

Example 15-7. PP3E\Internet\Email\PyMailGui\wraplines.py

```
################################################################################
# split lines on fixed columns or at delimiters before a column
# see also: related but different textwrap standard library module (2.3+)
################################################################################

defaultsize = 80

def wrapLinesSimple(lineslist, size=defaultsize):
    "split at fixed position size"
    wraplines = []
    for line in lineslist:
        while True:
            wraplines.append(line[:size])         # OK if len < size
            line = line[size:]                    # split without analysis
            if not line: break
    return wraplines

def wrapLinesSmart(lineslist, size=defaultsize, delimiters='.,:\t '):
    "wrap at first delimiter left of size"
    wraplines = []
    for line in lineslist:
        while True:
            if len(line) <= size:
                wraplines += [line]
                break
            else:
                for look in range(size-1, size/2, -1):
                    if line[look] in delimiters:
                        front, line = line[:look+1], line[look+1:]
                        break
                else:
                    front, line = line[:size], line[size:]
                wraplines += [front]
    return wraplines

################################################################################
# common use case utilities
################################################################################

def wrapText1(text, size=defaultsize):         # better for line-based txt: mail
    "when text read all at once"               # keeps original line brks struct
    lines = text.split('\n')                   # split on newlines
```

Example 15-7. PP3E\Internet\Email\PyMailGui\wraplines.py (continued)

```
    lines = wrapLinesSmart(lines, size)        # wrap lines on delimiters
    return '\n'.join(lines)                    # put back together

def wrapText2(text, size=defaultsize):         # more uniform across lines
    "same, but treat as one long line"         # but loses original line struct
    text  = text.replace('\n', ' ')            # drop newlines if any
    lines = wrapLinesSmart([text], size)       # wrap single line on delimiters
    return lines                               # caller puts back together

def wrapText3(text, size=defaultsize):
    "same, but put back together"
    lines = wrapText2(text, size)              # wrap as single long line
    return '\n'.join(lines) + '\n'             # make one string with newlines

def wrapLines1(lines, size=defaultsize):
    "when newline included at end"
    lines = [line[:-1] for line in lines]      # strip off newlines (or .rstrip)
    lines = wrapLinesSmart(lines, size)        # wrap on delimiters
    return [(line + '\n') for line in lines]   # put them back

def wrapLines2(lines, size=defaultsize):       # more uniform across lines
    "same, but concat as one long line"        # but loses original structure
    text  = ''.join(lines)                     # put together as 1 line
    lines = wrapText2(text)                    # wrap on delimiters
    return [(line + '\n') for line in lines]   # put newlines on ends

################################################################################
# self-test
################################################################################

if __name__ == '__main__':
    lines = ['spam ham ' * 20 + 'spam,ni' * 20,
             'spam ham ' * 20,
             'spam,ni'   * 20,
             'spam ham.ni' * 20,
             '',
             'spam'*80,
             ' ',
             'spam ham eggs']
    print 'all', '-'*30
    for line in lines: print repr(line)
    print 'simple', '-'*30
    for line in wrapLinesSimple(lines): print repr(line)
    print 'smart', '-'*30
    for line in wrapLinesSmart(lines): print repr(line)

    print 'single1', '-'*30
    for line in wrapLinesSimple([lines[0]], 60): print repr(line)
    print 'single2', '-'*30
    for line in wrapLinesSmart([lines[0]], 60): print repr(line)
    print 'combined text', '-'*30
    for line in wrapLines2(lines): print repr(line)
```

Example 15-7. PP3E\Internet\Email\PyMailGui\wraplines.py (continued)

```
print 'combined lines', '-'*30
print wrapText1('\n'.join(lines))

assert ''.join(lines) == ''.join(wrapLinesSimple(lines, 60))
assert ''.join(lines) == ''.join(wrapLinesSmart(lines, 60))
print len(''.join(lines)),
print len(''.join(wrapLinesSimple(lines))),
print len(''.join(wrapLinesSmart(lines))),
print len(''.join(wrapLinesSmart(lines, 60))),
raw_input('Press enter')
```

mailconfig: User Configurations

In Example 15-8, PyMailGUI's `mailconfig` user settings module is listed. This program has its own version of this module because many of its settings are unique for PyMailGUI. To use the program for reading your own email, set its initial variables to reflect your POP and SMTP server names and login parameters. The variables in this module also allow the user to tailor the appearance and operation of the program without finding and editing actual program logic.

Example 15-8. PP3E\Internet\Email\PyMailGui\mailconfig.py

```
##############################################################################
# PyMailGUI user configuration settings.
# Email scripts get their server names and other email config options from
# this module: change me to reflect your machine names, sig, and preferences.
# Warning: PyMailGUI won't run without most variables here: make a backup copy!
# Notes: could get some settings from the command line too, and a configure
# dialog would be better, but this common module file suffices for now.
##############################################################################

#-----------------------------------------------------------------------------
# (required for load, delete) POP3 email server machine, user
#-----------------------------------------------------------------------------

# popservername = '?Please set your mailconfig.py attributes?'

#popservername  = 'pop.rmi.net'            # or starship.python.net, 'localhost'
#popusername    = 'lutz'                   # password fetched or asked when run

#popservername  = 'pop.mindspring.com'     # yes, I have a few email accounts
#popusername    = 'lutz4'

#popservername  = 'pop.yahoo.com'
#popusername    = 'for_personal_mail'

popservername  = 'pop.earthlink.net'       # pp3e@earthlink.net
popusername    = 'pp3e'
```

Example 15-8. PP3E\Internet\Email\PyMailGui\mailconfig.py (continued)

```
#-------------------------------------------------------------------------------
# (required for send) SMTP email server machine name
# see Python smtpd module for a SMTP server class to run locally;
# note: your ISP may require that you be directly connected to their system:
# I can email through Earthlink on dial up, but cannot via Comcast cable
#-------------------------------------------------------------------------------

smtpservername = 'smtp.comcast.net'     # or 'smtp.mindspring.com', 'localhost'

#-------------------------------------------------------------------------------
# (may be required for send) SMTP user/password if authenticated
# set user to None or '' if no login/authentication is required
# set pswd to name of a file holding your SMTP password, or an
# empty string to force programs to ask (in a console, or GUI)
#-------------------------------------------------------------------------------

smtpuser  = None                          # per your ISP
smtppasswdfile  = ''                      # set to '' to be asked

#-------------------------------------------------------------------------------
# (optional) PyMailGUI: name of local one-line text file with your POP
# password; if empty or file cannot be read, pswd is requested when first
# connecting; pswd not encrypted: leave this empty on shared machines;
# PyMailCGI always asks for pswd (runs on a possibly remote server);
#-------------------------------------------------------------------------------

poppasswdfile  = r'c:\temp\pymailgui.txt'      # set to '' to be asked

#-------------------------------------------------------------------------------
# (optional) personal information used by PyMailGUI to fill in edit forms;
# if not set, does not fill in initial form values;
# sig  -- can be a triple-quoted block, ignored if empty string;
# addr -- used for initial value of "From" field if not empty,
# no longer tries to guess From for replies--varying success;
#-------------------------------------------------------------------------------

myaddress   = 'pp3e@earthlink.net'                          # lutz@rmi.net
mysignature = '--Mark Lutz  (http://www.rmi.net/~lutz)'

#-------------------------------------------------------------------------------
# (optional) local file where sent messages are saved;
# PyMailGUI 'Open' button allows this file to be opened and viewed
# don't use '.' form if may be run from another dir: e.g., pp3e demos
#-------------------------------------------------------------------------------

#sentmailfile   = r'.\sentmail.txt'             # . means in current working dir

#sourcedir    = r'C:\Mark\PP3E-cd\Examples\PP3E\Internet\Email\PyMailGui\'
#sentmailfile = sourcedir + 'sentmail.txt'

# determine auto from one of my source files
import wraplines, os
mysourcedir   = os.path.dirname(os.path.abspath(wraplines.__file__))
```

Example 15-8. PP3E\Internet\Email\PyMailGui\mailconfig.py (continued)

```
sentmailfile  = os.path.join(mysourcedir, 'sentmail.txt')

#-----------------------------------------------------------------------------
# (optional) local file where pymail saves POP mail;
# PyMailGUI instead asks for a name with a pop-up dialog
#-----------------------------------------------------------------------------

savemailfile   = r'c:\temp\savemail.txt'       # not used in PyMailGUI: dialog

#-----------------------------------------------------------------------------
# (optional) customize headers displayed in PyMailGUI list and view windows;
# listheaders replaces default, viewheaders extends it; both must be tuple of
# strings, or None to use default hdrs;
#-----------------------------------------------------------------------------

listheaders = ('Subject', 'From', 'Date', 'To', 'X-Mailer')
viewheaders = ('Bcc',)

#-----------------------------------------------------------------------------
# (optional) PyMailGUI fonts and colors for text server/file message list
# windows, message content view windows, and view window attachment buttons;
# use ('family', size, 'style') for font; 'colorname' or hexstr '#RRGGBB' for
# color (background, foreground);  None means use defaults;  font/color of
# view windows can also be set interactively with texteditor's Tools menu;
#-----------------------------------------------------------------------------

listbg   = 'indianred'         # None, 'white', '#RRGGBB' (see setcolor example)
listfg   = 'black'
listfont = ('courier', 9, 'bold')       # None, ('courier', 12, 'bold italic')
                                        # use fixed-width font for list columns
viewbg     = '#dbbedc'
viewfg     = 'black'
viewfont   = ('courier', 10, 'bold')
viewheight = 24                         # max lines for height when opened

partfg   = None
partbg   = None

# see Tk color names: aquamarine paleturqoise powderblue goldenrod burgundy ....

#listbg = listfg = listfont = None
#viewbg = viewfg = viewfont = viewheight = None      # to use defaults
#partbg = partfg = None

#-----------------------------------------------------------------------------
# (optional) column at which mail's original text should be wrapped for view,
# reply, and forward;  wraps at first delimiter to left of this position;
# composed text is not auto-wrapped: user or recipient's mail tool must wrap
# new text if desired; to disable wrapping, set this to a high value (1024?);
#-----------------------------------------------------------------------------

wrapsz = 100
```

Example 15-8. PP3E\Internet\Email\PyMailGui\mailconfig.py (continued)

```
#-----------------------------------------------------------------------------
# (optional) control how PyMailGUI opens mail parts in the GUI;
# for view window Split actions and attachment quick-access buttons;
# if not okayToOpenParts, quick-access part buttons will not appear in
# the GUI, and Split saves parts in a directory but does not open them;
# verifyPartOpens used by both Split action and quick-access buttons:
# all known-type parts open automatically on Split if this set to False;
# verifyHTMLTextOpen used by web browser open of HTML main text part:
#-----------------------------------------------------------------------------

okayToOpenParts    = True      # open any parts/attachments at all?
verifyPartOpens    = False     # ask permission before opening each part?
verifyHTMLTextOpen = False     # if main text part is HTML, ask before open?

#-----------------------------------------------------------------------------
# (optional) the maximum number of quick-access mail part buttons to show
# in the middle of view windows; after this many, a "..." button will be
# displayed, which runs the "Split" action to extract additional parts;
#-----------------------------------------------------------------------------

maxPartButtons = 8             # how many part buttons in view windows

#end
```

PyMailGuiHelp: User Help Text

Finally, Example 15-9 lists the module that defines the text displayed in PyMailGUI's help popup as one triple-quoted string. Read this here or live within the program to learn more about how PyMailGUI's interface operates (click the help bar at the top of the server list window to open the help display). It is included because it explains some properties of PyMailGUI not introduced by the demo earlier in this chapter.

This text may be more usefully formatted as HTML with section links and popped up in a web browser, but we take a lowest-common–denominator approach here to minimize external dependencies—we don't want help to fail if no browser can be located, and we don't want to maintain both text and HTML versions. Other schemes are possible (e.g., converting HTML to text as a fallback by parsing), but they are left as suggested improvements.

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py

```
########################################################################
# PyMailGUI help text string, in this separate module only to avoid
# distracting from executable code.  As coded, we throw up this text
# in a simple scrollable text box; in the future, we might instead
# use an HTML file opened with a browser (use webbrowser module, or
# run a "netscape help.html" or DOS "start help.html" with os.system);
# that would add an external dependency, unless text on browser fail;
########################################################################
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
# used to have to be narrow for Linux info box pop ups;
# now uses scrolledtext with buttons instead;

helptext = """PyMailGUI, version 2.1
January, 2006
Programming Python, 3rd Edition
O'Reilly Media, Inc.

PyMailGUI is a multiwindow interface for processing email, both online and
offline.  Its main interfaces include one list window for the mail server,
zero or more list windows for mail save files, and multiple view windows for
composing or viewing emails selected in a list window.  On startup, the main
(server) list window appears first, but no mail server connection is attempted
until a Load or message send request.  All PyMailGUI windows may be resized,
which is especially useful in list windows to see additional columns.


Major enhancements in this version:

* MIME multipart mails with attachments may be both viewed and composed.
* Mail transfers are no longer blocking, and may overlap in time.
* Mail may be saved and processed offline from a local file.
* Message parts may now be opened automatically within the GUI.
* Multiple messages may be selected for processing in list windows.
* Initial downloads fetch mail headers only; full mails are fetched on request.
* View window headers and list window columns are configurable.
* Deletions are performed immediately, not delayed until program exit.
* Most server transfers report their progress in the GUI.
* Long lines are intelligently wrapped in viewed and quoted text.
* Fonts and colors in list and view windows may be configured by the user.
* Authenticating SMTP mail-send servers that require login are supported.
* Sent messages are saved in a local file, which may be opened in the GUI.
* View windows intelligently pick a main text part to be displayed.
* Already fetched mail headers and full mails are cached for speed.
* Date strings and addresses in composed mails are formatted properly.
* (2.1) View windows now have quick-access buttons for attachments/parts.
* (2.1) Inbox out-of-synch errors detected on deletes, and index and mail loads.
* (2.1) Save-file loads and deletes threaded, to avoid pauses for large files.


Note: To use PyMailGUI to read and write email of your own, you must change
the POP and SMTP server names and login details in the file mailconfig.py,
located in PyMailGUI's source-code directory.  See section 10 for details.

Contents:
1)  LIST WINDOW ACTIONS
2)  VIEW WINDOW ACTIONS
3)  OFFLINE PROCESSING
4)  VIEWING TEXT AND ATTACHMENTS
5)  SENDING TEXT AND ATTACHMENTS
6)  MAIL TRANSFER OVERLAP
7)  MAIL DELETION
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
8)  INBOX MESSAGE NUMBER SYNCHRONIZATION
9)  LOADING EMAIL
10) THE mailconfig CONFIGURATION MODULE
11) DEPENDENCIES
12) MISCELLANEOUS HINTS



1) LIST WINDOW ACTIONS

Click list window buttons to process email:

- Load:\t fetch all (or new) mail headers from POP server inbox
- View:\t display selected emails nicely formatted
- Delete:\t delete selected emails from server or save file
- Write:\t compose a new email message, send by SMTP
- Reply:\t compose replies to selected emails, send by SMTP
- Fwd:\t compose forwards of selected emails, send by SMTP
- Save:\t write all selected emails to a chosen save file
- Open:\t load emails from an offline save file into new list window
- Quit:\t exit program (server list), close window (file list)

Double-click on an email in a list window's listbox to view the mail's raw text,
including any mail headers not shown by the View button.  List windows opened
for mail save files support all of the above except Load.  After the initial
Load, Load only fetches newly arrived message headers.  To forceably reload all
mails from the server, restart PyMailGUI.  There is reload button, because full
reloads are only required on rare deletion and inbox synchronization errors
(described ahead), and reloads are initiated automatically in these cases.

Click on emails in the main window's listbox to select them.  Click the "All"
checkbox to select all or no emails at once.  More than one email may be
selected at the same time: View, Delete, Reply, Fwd, and Save buttons are
applied to all currently selected emails, in both server and save-file list
windows.  Use Ctrl+click to select multiple mails, Shift+click to select all
from prior selecion, or click+move to drag the selection out.

In 2.1, most of the actions in the server List window automatically run a
quick-check to detect inbox out-of-synch errors with the server.  If a synch
error pop up appears, a full index reload will be automatically run; there
is no need to stop and restart PyMailGUI (see ahead in this help).



2) VIEW WINDOW ACTIONS

Action buttons in message view windows (View):

- Cancel:\t closes the message view window
- Parts:\t lists all message parts, including attachments
- Split:\t extracts, saves, and possibly opens message parts
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
Actions in message compose windows (Write, Reply, Fwd):

- Cancel:\t closes the message window, discarding its content
- Parts:\t lists files already attached to mail being edited
- Attach:\t adds a file as an attachment to a mail being edited
- Send:\t sends the message to all its recipients

Parts and Split buttons appear in all View windows; for simple messages, the
sole part is the message body.  Message reply, forward, and delete requests are
made in the list windows, not message view windows.  Deletions do not erase
open view windows.

New in 2.1: View windows also have up to a fixed maximum number of quick
access buttons for attached message parts.  They are alternatives to Split.
After the maximum number, a '...' button is added, which simply runs Split.
The maximum number of part buttons to display per view window can be set in
the mailconfig.py user settings module (described ahead).


3) OFFLINE PROCESSING

To process email offline: Load from the server, Save to a local file, and
later select Open to open a save file's list window in either the server List
window or another save file's List window.  Open creates a new List window for
the file, or raises its window if the file is already open.

A save file's list window allows all main window actions listed above, except
for Load.  For example, saved messages can be viewed, deleted, replied to, or
forwarded, from the file's list window.  Operations are mapped to the local
mail save file, instead of the server's inbox.   Saved messages may also be
saved: to move mails from one save file to another, Save and then Delete from
the source file's window.

You do not need to connect to a server to process save files offline: click
the Open button in the main list window.  In a save-file list window, a Quit
erases that window only; a Delete removes the message from the local save file,
not from a server. Save-file list windows are automatically updated when new
mails are saved to the corresponding file anywhere in the GUI.  The sent-mail
file may also be opened and processed as a normal save-mail file, with Open.

Save buttons in list windows save the full message text (including its headers,
and a message separator).  To save just the main text part of a message, either
use the Save button in the TextEditor at the bottom of a view or edit window,
or select the "Split" action button.  To save attachments, see the next section.

New in 2.1: local save-file Open and Delete requests are threaded to avoid
blocking the GUI during loads and deletes of large files.  Because of this, a
loaded file's index may not appear in its List window immediately.  Similarly,
when new mails are saved or messages are sent, there may be a delay before
the corresponding local file's List window is updated, if it is currently open.
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
As a status indication, the window's title changes to "Loading..." on loads
and "Deleting..." during deletes, and is reset to the file's name after the thread
exits (the server window uses pop ups for status indication, because the delay
is longer, and there is progress to display).  Eventually, either the index will
appear and its window raised, or an error message will pop up.  Save-file loads
and deletes are not allowed to overlap with each other for a given file, but
may overlap with server transfers and operations on other open files.

Note: save-file Save operations are still not threaded, and may pause the GUI
momentarily when saving very many large mails.  This is normaly not noticeable,
because unlike Open and Delete, saves simply append to the save-file, and do
not reload its content.  To avoid pauses completely, though, don't save very
many large mails in a single operation.

Also note: the current implementation loads the entire save-mail file into
memory when opened.  Because of this, save-mail files are limited in size,
depending upon your computer.  To avoid consuming too much memory, you
should try to keep your save files relatively small (at the least, smaller
than your computer's available memory).  As a rule of thumb, organize your
saved mails by categories into many small files, instead of a few large files.


4) VIEWING TEXT AND ATTACHMENTS

PyMailGUI's view windows use a text-oriented display.  When a mail is viewed,
its main text is displayed in the View window.  This text is taken from the
entire body of a simple message, or the first text part of a multipart MIME
message.  To extract the main message text, PyMailGUI looks for plain text,
then HTML, and then text of any other kind.  If no such text content is found,
nothing is displayed in the view window, but parts may be opened manually with
the "Split" button (and quick-access part buttons in 2.1, described below).

If the body of a simple message is HTML type, or a HTML part is used as the
main message text, a web browser is popped up as an alternative display for
the main message text, if verified by the user (the mailconfig module can be
used to bypass the verification; see ahead).  This is equivalent to opening
the HTML part with the "Split" button, but is initiated automatically for the
main message text's HTML.  If a simple message is something other than text
or HTML, its content must be openened manually with Split.

When viewing mails, messages with multipart attachments are prefixed with
a "*" in list windows.  "Parts" and "Split" buttons appear in all View windows.
Message parts are defined as follows:

- For simple messages, the message body is considered to be the sole
  part of the message.

- For multipart messages, the message parts list includes the main
  message text, as well as all attachments.

In both cases, message parts may be saved and opened with the "Split" button.
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
For simple messages, the message body may be saved with Split, as well as the
Save button in the view window's text editor.  To process multipart messages:

- Use "Parts" to display the names of all message parts, including any
  attachments, in a pop-up dialog.

- Use "Split" to view message parts: all mail parts are first saved to a
  selected directory, and any well-known and generally safe part files are
  opened automatically, but only if verified by the user.

- See also the note below about 2.1 quick access buttons, for an alternative
  to the Parts/Split interface on View windows.

For "Split", select a local directory to save parts to.  After the save, text
parts open in the TextEditor GUI, HTML and multimedia types open in a web
browser, and common Windows document types (e.g., .doc and .xls files) open via
the Windows registry entry for the filename extension.  For safety, unknown
types and executable program parts are never run automatically; even Python
programs are displayed as source text only (save the code to run manually).

Web browsers on some platforms may open multimedia types (image, audio, video)
in specific content handler programs (e.g., MediaPlayer, image viewers).  No
other types of attachments are ever opened, and attachments are never opened
without user verification (or mailconfig.py authorization in 2.1, described
below).  Browse the parts save directory to open other parts manually.

To avoid scrolling for very long lines (sometimes sent by HTML-based mailers),
the main text part of a message is automatically wrapped for easy viewing.
Long lines are split up at the first delimiter found before a fixed column,
when viewed, replied, or forwarded.  The wrapping column may be configured or
disabled in the mailconfig module (see ahead).  Text lines are never
automatically wrapped when sent; users or recipients should manage line length
in composed mails.

New in 2.1: View windows also have up to a fixed maximum number of quick-access
buttons for attached message parts.  They are alternatives to Split: selecting
an attachment's button automatically extracts, saves, and opens that single
attachment directly, without Split directory and pop-up dialogs (a temporary
directory is used).  The maximum number of part buttons to display per view
window can be set in the mailconfig.py user settings module (described ahead).
For mails with more than the maximum number of attachments, a '...' button is
added which simply runs Split to save and open any additional attachments.

Also in 2.1, two settings in the mailconfig.py module (see section 10) can be
used to control how PyMailGUI opens parts in the GUI:

- okayToOpenParts: controls whether part opens are allowed at all
- verifyPartOpens: controls whether to ask before each part is opened.

Both are used for View window Split actions and part quick-access buttons.  If
okayToOpenParts is False, quick-access part buttons will not appear in the GUI,
and Split saves parts in a directory but does not open them.  verifyPartOpens
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
is used by both Split and quick-access part buttons: if False, part buttons
open parts immediately, and Split opens all known-type parts automatically
after they are saved (unknown types and executables are never opened).

An additional setting in this module, verifyHTMLTextOpen, controls verification
of opening a web browser on a HTML main text part of a message; if False, the
web browser is opened without prompts.  This is a separate setting from
verifyPartOpens, because this is more automatic than part opens, and some
HTML main text parts may have dubious content (e.g., images, ads).



5) SENDING TEXT AND ATTACHMENTS

When composing new mails, the view window's "Attach" button adds selected files
as attachments, to be sent with the email's main text when the View window's
"Send" is clicked.  Attachment files may be of any type; they are selected in
a pop-up dialog, but are not loaded until Send.  The view window's "Parts"
button displays attachments already added.

The main text of the message (in the view window editor) is sent as a simple
message if there are no attachments, or as the first part of a multipart MIME
message if there are.  In both cases, the main message text is always sent as
plain text.  HTML files may be attached to the message, but there is no support
for text-or-HTML multipart alternative format for the main text, nor for
sending the main text as HTML only.  Not all clients can handle HTML, and
PyMailGUI's text-based view windows have no HTML editing tools.

Multipart nesting is never applied: composed mails are always either a simple
body, or a linear list of parts containing the main message text and attachment
files.

For mail replies and forwards, headers are given intial values, the main
message text (described in the prior section) is wrapped and quoted with '>'
prefixes, and any attachments in the original message are stripped.  Only
new attachments added are sent with the message.

To send to multiple addresses, separate each recipient's address in To and Cc
fields with semicolons.  For instance:
        tim@oreily.com; "Smith, Bob" <bob@bob.com>; john@nasa.gov
Note that because these fields are split on semicolons without full parsing,
a recipient's address should not embed a ';'.  For replies, this is handled
automatically: the To field is prefilled with the original message's From,
using either the original name and address, or just the address if the name
contains a ';' (a rare case).  Cc and Bcc headers fields are ignored if they
contain just the initial "?" when sent.

Successfully sent messages are saved in a local file whose name you list in the
mailconfig.py module.  Sent mails are saved if the variable "sentmailfile" is
set to a valid filename; set to an empty string to disable saves.  This file
may be opened using the Open button of the GUI's list windows, and its content
may be viewed, processed, deleted, saved, and so on within the GUI, just like a
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
manually saved mail file.  Also like manually saved mail files, the sent-file
list window is automatically updated whenever a new message is sent, if it is
open (there is no need to close and reopen to see new sends).  If this file
grows too large to open, you can delete its content with Delete, after possibly
saving sent mails you wish to keep to another file with Save.

Note that some ISPs may require that you be connected to their systems in order
to use their SMTP servers (sending through your dial-up ISP's server while
connected to a broadband provider may not work--try the SMTP server at your
broadband provider instead), and some SMTP servers may require authentication
(set the "smtpuser" variable in the mailconfig.py module to force authentication
logins on sends).  See also the Python library module smptd for SMTP server
tools; in principle, you could run your own SMTP server locally on 'localhost'.


6) MAIL TRANSFER OVERLAP

PyMailGUI runs mail server transfers (loads, sends, and deletes) in threads, to
avoid blocking the GUI.  Transfers never block the GUI's windows, and windows
do not generally block other windows.  Users can view, create, and process mails
while server transfers are in progress.  The transfers run in the background,
while the GUI remains active.

PyMailGUI also allows mail transfer threads to overlap in time.  In particular,
new emails may be written and sent while a load or send is in progress, and mail
loads may overlap with sends and other mail loads already in progress.  For
example, while waiting for a download of mail headers or a large message, you
can open a new Write window, compose a message, and send it; the send will
overlap with the load currently in progress.  You may also load another mail,
while the load of a large mail is in progress.

While mail transfers are in progress, pop-up windows display their current
progress as a message counter.  When sending a message, the original edit
View window is popped back up automatically on Send failures, to save or retry.
Because delete operations may change POP message numbers on the server, this
operation disables other deletions and loads while in progress.

Offline mail save-file loads and deletes are also threaded: these threads may
overlap in time with server transfers, and with operations on other open save
files.  Saves are disabled if the source or target file is busy with a load
or save operation.  Quit is never allowed while any thread is busy.


7) MAIL DELETION

Mail is not removed from POP servers on Load requests, but only on explicit
"Delete" button deletion requests, if verified by the user.  Delete requests
are run immediately, upon user verification.
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
To delete your mail from a server and process offline: in the server list
window select the All checkbutton, Save to a local file, and then Delete to
delete all mails from the server; use Open to open the save file later to view
and process saved mail.

When deleting from the server window, the mail list (and any already viewed
message text) is not reloaded from server, if the delete was successful.  If
the delete fails, all email must be reloaded, because some POP message numbers
may have changed; the reload occurs automatically.  Delete in a file list
window deletes from the loal file only.

As of version 2.1, PyMailGUI automatically matches messages selected for
deletion with their headers on the mail server, to ensure that the correct
mail is deleted.  If the mail index is out of synch with the server, mails
that do not match the server are not deleted, since their POP message numbers
are no longer accurate.  In this event, an error is displayed, and a full reload
of the mail index list is automatically performed; you do not need to stop and
restart PyMailGUI to reload the index list.  This can slow deletions (it adds
roughly one second per deleted mail on the test machine used), but prevents
the wrong mail from being deleted.  See the POP message number synchronization
errors description in the next section.


8) INBOX MESSAGE NUMBER SYNCHRONIZATION

PyMailGUI does header matching in order to ensure that deletions only delete
the correct messages, and periodically detect synchronization errors with the
server.  If a synchronization error message appears, the operation is cancelled,
and a full index reload from the server is automatically performed.  You need
not stop and restart PyMailGUI and reload the index, but must reattempt the
operation after the reload.

The POP email protocol assigns emails a relative message number, reflecting
their position in your inbox.  In the server List window, PyMailGUI loads its
mail index list on demand from the server, and assumes it reflects the content
of your inbox from that point on.  A message's position in the index list is
used as its POP relative message number for later loads and deletes.

This normally works well, since newly arrived emails are added to the end
of the inbox.  However, the message numbers of the index list can become out
of synch with the server in two ways:

A) Because delete operations change POP relative message numbers in the inbox,
deleting messages in another email client (even another PyMailGUI instance)
while the PyMailGUI server list window is open can invalidate PyMailGUI's
message index numbers.  In this case, the index list window may be arbitrarily
out of synch with the inbox on the server.

B) It is also possible that your ISP may automatically delete emails from
your inbox at any time, making PyMailGUI's email list out of synch with message
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
numbers on the mail server.  For example, some ISPs may automatically move an
email from the inbox to the undeliverable box, in response to a fetch failure.
If this happens, PyMailGUI's message numbers will be off by one, according to
the server's inbox.

To accommodate such cases, PyMailGUI 2.1 always matches messages to be deleted
against the server's inbox, by comparing already fetched headers text with the
headers text returned for the same message number; the delete only occurs if
the two match.  In addition, PyMailGUI runs a quick check for out-of-synch
errors by comparing headers for just the last message in the index, whenever
the index list is updated, and whenever full messages are fetched.

This header matching adds a slight overhead to deletes, index loads, and mail
fetches, but guarantees that deletes will not remove the wrong message, and
ensures that the message you receive corresponds to the item selected in the
server index List window.  The synch test overhead is one second or less on
test machines used - it requires 1 POP server connect and an inbox size and
(possibly) header text fetch.

In general, you still shouldn't delete messages in PyMailGUI while running a
different email client, or that client's message numbers may become confused
unless it has simlar synchronization tests.  If you receive a synch error
pop up on deletes or loads, PyMailGUI automatically begins a full reload of
the mail index list displayed in the server List window.



9) LOADING EMAIL

To save time, Load requests only fetch mail headers, not entire messages.
View operations fetch the entire message, unless it has been previously viewed
(already loaded messages are cached).  Multiple message downloads may overlap
in time, and may overlap with message editing and sends.

In addition, after the initial load, new Load requests only fetch headers of
newly arrived messages.  All headers must be refetched after a delete failure,
however, due to possibly changed POP message numbers.

PyMailGUI only is connected to a mail server while a load, send, or delete
operation is in progress.  It does not connect at all unless one of these
operations is attempted, and disconnects as soon as the operation finishes.
You do not need any Internet connectivity to run PyMailGUI unless you attempt
one of these operations.  In addition, you may disconnect from the Internet
when they are not in progress, without having to stop the GUI--the program
will reconnect on the next transfer operation.

Note: if your POP mail server does support the TOP command for fetching mail
headers (most do), see variable "srvrHasTop" in the mailtools.py module to
force full message downloads.

Also note that, although PyMailGUI only fetches message headers initially if
your email server supports TOP, this can still take some time for very large
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
inboxes; as a rule of thumb, use save-mail files and deletions to keep your
inbox small.


10) THE mailconfig CONFIGURATION MODULE

Change the mailconfig.py module file in PyMailGUI's home directory on your
own machine to reflect your email server names, username, email address, and
optional mail signature line added to all composed mails.

Most settings in this module are optional, or have reasonable preset defaults.
However, you must minimally set this module's "smtpservername" variable to send
mail, and its "popservername" and "popusername" to load mail from a server.
These are simple Python variables assigned to strings in this file.  See the
module file and its embedded comments for details.

The mailconfig module's "listheaders" attribute can also be set to a tuple of
string header field name, to customize the set of headers displayed in list
windows; mail size is always displayed last.  Similarly mailconfig's
"viewheaders" attribute can extend the set of headers shown in a View window
(though From, To, Cc, and Subject fields are always shown).  List windows
display message headers in fixed-width columns.

Variables in the mailconfig module also can be used to tailor the font used in
list windows ("fontsz"), the column at which viewed and quoted text is
automatically wrapped ("wrapsz"), colors and fonts in various windows, the
local file where sent messages are saved, the opening of mail parts, and more;
see the file's source code for more details.

Note: use caution when changing this file, as PyMailGUI may not run at all if
some of its variables are missing.  You may wish to make a backup copy before
editing it in case you need to restore its defaults.  A future version of this
system may have a configuration dialog which generates this module's code.


11) DEPENDENCIES

This client-side program currently requires Python and Tkinter.  It uses Python
threads, if installed, to avoid blocking the GUI.  Sending and loading email
from a server requires an Internet connection.  Requires Python 2.3 or later,
and uses the Python "email" standard library module to parse and compose mail
text.  Reuses a number of modules located in the PP3E examples tree.


12) MISCELLANEOUS HINTS

- Use ';' between multiple addresses in "To" and "Cc" headers.
- Passwords are requested if needed, and not stored by PyMailGUI.
- You may list your password in a file named in mailconfig.py.
```

Example 15-9. PP3E\Internet\PyMailGui2\PyMailGuiHelp.py (continued)

```
- Reply and Fwd automatically quote the original mail text.
- Save pops up a dialog for selecting a file to hold saved mails.
- Save always appends to the save file, rather than erasing it.
- Delete does not reload message headers, unless it fails.
- Delete checks your inbox to make sure the correct mail is deleted.
- Fetches detect inbox changes and automatically reload index list.
- Open and save dialogs always remember the prior directory.
- To print emails, "Save" to a text file and print with other tools.
- Click this window's Source button to view its source-code files.
- Watch http://www.rmi.net/~lutz for updates and patches
- This is an Open Source system: change its code as you like.
- See the SpamBayes system for a spam filter for incoming email.
"""

if __name__ == '__main__':
    print helptext                    # to stdout if run alone
    raw_input('Press Enter key')      # pause in DOS console pop ups
```

Ideas for Improvement

Although I use PyMailGUI on a regular basis as is, there is always room for improvement to software, and this system is no exception. If you wish to experiment with its code, here are a few suggested projects:

- Mail list windows could be sorted by columns on demand. This may require a more sophisticated list window structure.
- The implementation of save-mail files limits their size by loading them into memory all at once; a DBM keyed-access implementation may work around this constraint. See the list windows module comments for ideas.
- Hyperlink URLs within messages could be highlighted visually and made to spawn a web browser automatically when clicked by using the launcher tools we met in the GUI and system parts of this book (Tkinter's text widget supports links directly).
- Because Internet newsgroup posts are similar in structure to emails (header lines plus body text; see the `nntplib` example in Chapter 14), this script could in principle be extended to display both email messages and news articles. Classifying such a mutation as clever generalization or diabolical hack is left as an exercise in itself.
- The help text has grown large in this version: this might be better implemented as HTML, and displayed in a web browser, with simple text as a fallback option. In fact, we might extract the simple text from the HTML, to avoid redundant copies.
- Saves and Split writes could also be threaded for worst-case scenarios. For pointers on making Saves parallel, see the comments in the file class of `ListWindows.py`; there may be some subtle issues that require both thread locks and general file locking for potentially concurrent updates.

- There is currently no way to delete an attachment once it has been added in compose windows. This might be supported by adding quick-access part buttons to compose windows, too, which could verify and delete the part when clicked.
- We could add an automatic spam filter for mails fetched, in addition to any provided at the email server or ISP.
- Mail text is editable in message view windows, even though a new mail is not being composed. This is deliberate—users can annotate the message's text and save it in a text file with the Save button at the bottom of the window. This might be confusing, though, and is redundant (we can also edit and save by clicking on the main text's quick-access part button). Removing edit tools would require extending PyEdit.
- We could also add support for mail lists, allowing users to associate multiple email addresses with a saved list name. On sends to a list name, the mail would be sent to all on the list (the "To:" addresses passed to `smtplib`), but the email list could be used for the email's "To:" header line).

And so on—because this software is open source, it is also necessarily open-ended. Suggested exercises in this category are delegated to your imagination.

This concludes our tour of Python client-side programming. In the next chapter, we'll hop the fence to the other side of the Internet world and explore scripts that run on server machines. Such programs give rise to the grander notion of applications that live entirely on the Web and are launched by web browsers. As we take this leap in structure, keep in mind that the tools we met in this and the previous chapter are often sufficient to implement all the distributed processing that many applications require, and they can work in harmony with scripts that run on a server. To completely understand the Web world view, though, we need to explore the server realm, too.