CHAPTER 24

Project 5: A Virtual Tea Party

In this project, you do some serious network programming. You'll write a chat server—a program that lets several people connect via the Internet and chat with each other in real time. There are many ways to create such a beast in Python. A simple and natural approach might be to use the Twisted framework (discussed in Chapter 14), for example, with the `LineReceiver` class taking center stage. In this chapter, I stick to the standard libraries, basing the program on the modules `asyncore` and `asynchat`. If you like, you could try out some of the alternative methods (such as forking or threading) discussed in Chapter 14.

What's the Problem?

Online chatting is quite common. Many chat services of various kinds (IRC, instant messaging services, and so forth) are available all over the Internet. Some of these are even full-fledged text-based virtual worlds (see `http://www.mudconnect.com` for a long list). If you want to set up a chat server, you can just download and install one of the many free server programs. However, writing a chat server yourself is useful for two reasons:

- You learn about network programming.
- You can customize it as much as you want.

The second point suggests that you can start with a simple chat server and develop it into basically any kind of server (including a virtual world), with all the power of Python at your fingertips. Pretty awesome, isn't it?

For now, the chat server project has the following requirements:

- The server should be able to receive multiple connections from different users.
- It should let the users act *in parallel*.
- It should be able to interpret commands such as `say` or `logout`.
- The server should be easily extensible.

The two things that will require special tools are the network connections and the asynchronous nature of the program.

Useful Tools

The only new tools you need in this project are the `asyncore` module from the standard library and its relative `asynchat`. I'll describe the basics of how these work. You can find more details about them in the Python Library Reference (`http://python.org/doc/lib/module-asyncore.html` and `http://python.org/doc/lib/module-asynchat.html`).

As discussed in Chapter 14, the basic component in a network program is the *socket*. Sockets can be created directly by importing the `socket` module and using the functions there. So what do you need `asyncore` for?

The `asyncore` framework enables you to juggle several users who are connected simultaneously. Imagine a scenario in which you have no special tools for handling this. When you start up the server, it waits for users to connect. When one user is connected, it starts reading data from that user and supplying results through a socket. But what happens if another user is already connected? The second user to connect must wait until the first one has finished. In some cases, that will work just fine, but when you're writing a chat server, the whole point is that more than one user can be connected—how else could users chat with one another?

The `asyncore` framework is based on an underlying mechanism (the `select` function from the `select` module, as discussed in Chapter 14) that allows the server to serve all the connected users in a piecemeal fashion. Instead of reading *all* the available data from one user before going on to the next, only *some* data is read. Also, the server reads only from the sockets where there *is* data to be read. This is done again and again, in a loop. Writing is handled similarly. You could implement this yourself using just the modules `socket` and `select`, but `asyncore` and `asynchat` provide a very useful framework that takes care of the details for you. (For alternative ways of implementing parallel user connections, see the section "Multiple Connections" in Chapter 14.)

Preparations

The first thing you need is a computer that's connected to a network (such as the Internet); otherwise, others won't be able to connect to your chat server. (It is possible to connect to the chat server from your own machine, but that's not much fun in the long run, is it?) To be able to connect, the user must know the address of your machine (a machine name such as `foo.bar.baz.com` or an IP address). In addition, the user must know the *port number* used by your server. You can set this in your program; in the code in this chapter, I use the (rather arbitrary) port number 5005.

■ **Note** As mentioned in Chapter 14, certain port numbers are restricted and require administrator privileges. In general, numbers greater than 1023 are okay.

To test your server, you need a *client*—the program on the user side of the interaction. A simple program for this sort of thing is `telnet` (which basically lets you connect to any socket server). In UNIX, you probably have this program available on the command line:

```
$ telnet some.host.name 5005
```

The preceding command connects to the machine `some.host.name` on port 5005. To connect to the same machine on which you're running the `telnet` command, simply use the machine name localhost. (You might want to supply an escape character through the `-e` switch to make sure you can quit `telnet` easily. See the `man` page for more details.)

In Windows, you can use either the standard `telnet` command (in a command-prompt window) or a terminal emulator with `telnet` functionality, such as PuTTY (software and more information available at `http://www.chiark.greenend.org.uk/~sgtatham/putty`). However, if you are installing new software, you might as well get a client program tailored to chatting. MUD (or MUSH or MOO or some other related acronym) clients[1] are quite suitable for this sort of thing. My client of choice is TinyFugue (software and more information available at `http://tinyfugue.sf.net`). It is mainly designed for use in UNIX. (Several clients are available for Windows as well; just do a web search for "mud client" or something similar.)

First Implementation

Let's break things down a bit. We need to create two main classes: one representing the chat server and one representing each of the chat sessions (the connected users).

The ChatServer Class

To create the basic `ChatServer`, you subclass the `dispatcher` class from `asyncore`. The `dispatcher` is basically just a socket object, but with some extra event-handling features, which you'll be using in a minute.

See Listing 24-1 for a basic chat server program (that does very little).

Listing 24-1. *A Minimal Server Program*

```
from asyncore import dispatcher
import asyncore

class ChatServer(dispatcher): pass

s = ChatServer()
asyncore.loop()
```

If you run this program, nothing happens. To make the server do anything interesting, you should call its `create_socket` method to create a socket, and its `bind` and `listen` methods to bind the socket to a specific port number and to tell it to listen for incoming connections. (That is what servers do, after all.) In addition, you'll override the `handle_accept` event-handling method to actually do something when the server accepts a client connection. The resulting program is shown in Listing 24-2.

1. MUD stands for Multi-User Dungeon/Domain/Dimension. MUSH stands for Multi-User Shared Hallucination. MOO means MUD, object-oriented. See, for example, Wikipedia (`http://en.wikipedia.org/wiki/MUD`) for more information.

Listing 24-2. *A Server That Accepts Connections*

```
from asyncore import dispatcher
import socket, asyncore

class ChatServer(dispatcher):

    def handle_accept(self):
        conn, addr = self.accept()
        print 'Connection attempt from', addr[0]

s = ChatServer()
s.create_socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 5005))
s.listen(5)
asyncore.loop()
```

The `handle_accept` method calls `self.accept`, which lets the client connect. This returns a connection (a socket that is specific for this client) and an address (information about which machine is connecting). Instead of doing anything useful with this connection, the `handle_accept` method simply prints that a connection attempt was made. `addr[0]` is the IP address of the client.

The server initialization calls `create_socket` with two arguments that specify the type of socket you want. You could use different types, but those shown here are what you usually want. The call to the `bind` method simply binds the server to a specific address (host name and port). The host name is empty (an empty string, essentially meaning localhost, or, more technically, "all interfaces on this machine") and the port number is 5005. The call to `listen` tells the server to listen for connections; it also specifies a backlog of five connections. The final call to `asyncore.loop` starts the server's listening loop as before.

This server actually works. Try to run it and then connect to it with your client. The client should immediately be disconnected, and the server should print out the following:

```
Connection attempt from 127.0.0.1
```

The IP address will be different if you don't connect from the same machine as your server.

To stop the server, simply use a keyboard interrupt: Ctrl+C in UNIX or Ctrl+Break in Windows.

Shutting down the server with a keyboard interrupt results in a stack trace. To avoid that, you can wrap the `loop` in a `try/except` statement. With some other cleanups, the basic server ends up as shown in Listing 24-3.

Listing 24-3. *The Basic Server with Some Cleanups*

```
from asyncore import dispatcher
import socket, asyncore

PORT = 5005
```

```
class ChatServer(dispatcher):

    def __init__(self, port):
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)

    def handle_accept(self):
        conn, addr = self.accept()
        print 'Connection attempt from', addr[0]

if __name__ == '__main__':
    s = ChatServer(PORT)
    try: asyncore.loop()
    except KeyboardInterrupt: pass
```

The added call to `set_reuse_addr` lets you reuse the same address (specifically, the port number) even if the server isn't shut down properly. (Without this call, you may need to wait for a while before the server can be started again, or change the port number each time the server crashes, because your program may not be able to properly notify your operating system that it's finished with the port.)

The ChatSession Class

The basic `ChatServer` isn't very useful. Instead of ignoring the connection attempts, a new `dispatcher` object should be created for each connection. However, these objects will behave differently from the one used as the main server. They won't be listening on a port for incoming connections; they already *are* connected to a client. Their main task is collecting data (text) coming from the client and responding to it. You could implement this functionality yourself by subclassing `dispatcher` and overriding various methods, but, luckily, there is a module that already does most of the work: `asynchat`.

Despite the name, `asynchat` isn't specifically designed for the type of streaming (continuous) chat application that we're working on. (The `chat` in the name refers to "chat-style" or command-response protocols.) The good thing about the `async_chat` class (found in the `asynchat` module) is that it hides the most basic socket reading and writing operations, which can be a bit difficult to get right. All that's needed to make it work is to override two methods: `collect_incoming_data` and `found_terminator`. The former is called each time a bit of text has been read from the socket, and the latter is called when a *terminator* is read. The terminator (in this case) is just a line break. (You'll need to tell the `async_chat` object about that by calling `set_terminator` as part of the initialization.)

An updated program, now with a `ChatSession` class, is shown in Listing 24-4.

Listing 24-4. *Server Program with ChatSession Class*

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005

class ChatSession(async_chat):

    def __init__(self, sock):
        async_chat.__init__(self, sock)
        self.set_terminator("\r\n")
        self.data = []

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        line = ''.join(self.data)
        self.data = []
        # Do something with the line...
        print line

class ChatServer(dispatcher):

    def __init__(self, port):
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)
        self.sessions = []


    def handle_accept(self):
        conn, addr = self.accept()
        self.sessions.append(ChatSession(conn))

if __name__ == '__main__':
    s = ChatServer(PORT)
    try: asyncore.loop()
    except KeyboardInterrupt: print
```

Several things are worth noting in this new version:

- The `set_terminator` method is used to set the line terminator to `"\r\n"`, which is the commonly used line terminator in network protocols.
- The `ChatSession` object keeps the data it has read so far as a list of strings called `data`. When more data is read, `collect_incoming_data` is called automatically, and it simply appends the data to the list. Using a list of strings and later joining them (with the `join` string method) is a common idiom (and historically more efficient than incrementally adding strings). Feel free to use `+=` with strings instead.
- The `found_terminator` method is called when a terminator is found. The current implementation creates a line by joining the current data items, and resets `self.data` to an empty list. However, because you don't have anything useful to do with the line yet, it is simply printed.
- The `ChatServer` keeps a list of sessions.
- The `handle_accept` method of the `ChatServer` now creates a new `ChatSession` object and appends it to the list of sessions.

Try running the server and connecting with two (or more) clients simultaneously. Every line you type in a client should be printed in the terminal where your server is running. That means the server is now capable of handling several simultaneous connections. Now all that's missing is the capability for the clients to see what the others are saying!

Putting It Together

Before the prototype can be considered a fully functional (albeit simple) chat server, one main piece of functionality is lacking: what the users say (each line they type) should be broadcast to the others. That functionality can be implemented by a simple `for` loop in the server, which loops over the list of sessions and writes the line to each of them. To write data to an `async_chat` object, you use the `push` method.

This broadcasting behavior also adds another problem: you must make sure that connections are removed from the list when the clients disconnect. You can do that by overriding the event-handling method `handle_close`. The final version of the first prototype can be seen in Listing 24-5.

Listing 24-5. *A Simple Chat Server (simple_chat.py)*

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005
NAME = 'TestChat'
```

```
class ChatSession(async_chat):
    """
    A class that takes care of a connection between the server
    and a single user.
    """
    def __init__(self, server, sock):
        # Standard setup tasks:
        async_chat.__init__(self, sock)
        self.server = server
        self.set_terminator("\r\n")
        self.data = []
        # Greet the user:
        self.push('Welcome to %s\r\n' % self.server.name)

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        """
        If a terminator is found, that means that a full
        line has been read. Broadcast it to everyone.
        """
        line = ''.join(self.data)
        self.data = []
        self.server.broadcast(line)

    def handle_close(self):
        async_chat.handle_close(self)
        self.server.disconnect(self)

class ChatServer(dispatcher):
    """
    A class that receives connections and spawns individual
    sessions. It also handles broadcasts to these sessions.
    """
    def __init__(self, port, name):
        # Standard setup tasks
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)
        self.name = name
        self.sessions = []
```

```
    def disconnect(self, session):
        self.sessions.remove(session)

    def broadcast(self, line):
        for session in self.sessions:
            session.push(line + '\r\n')

    def handle_accept(self):
        conn, addr = self.accept()
        self.sessions.append(ChatSession(self, conn))

if __name__ == '__main__':
    s = ChatServer(PORT, NAME)
    try: asyncore.loop()
    except KeyboardInterrupt: print
```

Second Implementation

The first prototype may be a fully functioning chat server, but its functionality is quite limited. The most obvious limitation is that you can't discern who is saying what. Also, it does not interpret commands (such as `say` or `logout`), which the original specification requires. So, you need to add support for identity (one unique name per user) and command interpretation, and you must make the behavior of each session depend on the state it's in (just connected, logged in, and so on)—all of this in a manner that lends itself easily to extension.

Basic Command Interpretation

I'll show you how to model the command interpretation on the `Cmd` class of the `cmd` module in the standard library. (Unfortunately, you can't use this class directly because it can be used only with `sys.stdin` and `sys.stdout`, and you're working with several streams.) What you need is a function or method that can handle a single line of text (as typed by the user). It should split off the first word (the command) and call an appropriate method based on it. For example, this line:

```
say Hello, world!
```

might result in the following call:

```
do_say('Hello, world!')
```

possibly with the session itself as an added argument (so `do_say` would know who did the talking).

Here is a simple implementation, with an added method to express that a command is unknown:

```
class CommandHandler:
    """
    Simple command handler similar to cmd.Cmd from the standard
    library.
    """
```

```
    def unknown(self, session, cmd):
        session.push('Unknown command: %s\r\n' % cmd)

    def handle(self, session, line):
        if not line.strip(): return
        parts = line.split(' ', 1)
        cmd = parts[0]
        try: line = parts[1].strip()
        except IndexError: line = ''
        meth = getattr(self, 'do_'+cmd, None)
        try:
            meth(session, line)
        except TypeError:
            self.unknown(session, cmd)
```

The use of `getattr` in this class is similar to that in the markup project in Chapter 20.

With the basic command handling out of the way, you need to define some actual commands. And which commands are available (and what they do) should depend on the current state of the session. How do you represent that state?

Rooms

Each state can be represented by a custom command handler. This is easily combined with the standard notion of chat rooms (or locations in a MUD). Each room is a `CommandHandler` with its own specialized commands. In addition, it should keep track of which users (sessions) are currently inside it. Here is a generic superclass for all your rooms:

```
class EndSession(Exception): pass

class Room(CommandHandler):
    """
    A generic environment which may contain one or more users
    (sessions). It takes care of basic command handling and
    broadcasting.
    """

    def __init__(self, server):
        self.server = server
        self.sessions = []

    def add(self, session):
        self.sessions.append(session)
```

```
    def remove(self, session):
        self.sessions.remove(session)

    def broadcast(self, line):
        for session in self.sessions:
            session.push(line)

    def do_logout(self, session, line):
        raise EndSession
```

In addition to the basic `add` and `remove` methods, a `broadcast` method simply calls `push` on all of the users (sessions) in the room. There is also a single command defined—`logout` (in the form of the `do_logout` method). It raises an exception (`EndSession`), which is dealt with at a higher level of the processing (in `found_terminator`).

Login and Logout Rooms

In addition to representing normal chat rooms (this project includes only one such chat room), the `Room` subclasses can represent other states, which was indeed the intention. For example, when a user connects to the server, he is put in a dedicated `LoginRoom` (with no other users in it). The `LoginRoom` prints a welcome message when the user enters (in the `add` method). It also overrides the `unknown` method to tell the user to log in; the only command it responds to is the `login` command, which checks whether the name is acceptable (not an empty string, and not already used by another user).

The `LogoutRoom` is much simpler. Its only job is to delete the user's name from the server (which has a dictionary called `users` where the sessions are stored). If the name isn't there (because the user never logged in), the resulting `KeyError` is ignored.

For the source code of these two classes, see Listing 24-6 later in this chapter.

■ **Note** Even though the server's `users` dictionary keeps references to all the sessions, no session is ever retrieved from it. The `users` dictionary is used only to keep track of which names are in use. However, instead of using some arbitrary value (such as `True`), I decided to let each user name refer to the corresponding session. Even though there is no immediate use for it, it may be useful in some later version of the program (for example, if one user wants to send a message privately to another). An alternative would have been to simply keep a set or list of sessions.

The Main Chat Room

The main chat room also overrides the `add` and `remove` methods. In `add`, it broadcasts a message about the user who is entering, and it adds the user's name to the `users` dictionary in the server. The `remove` method broadcasts a message about the user who is leaving.

In addition to these methods, the ChatRoom class implements three commands:

- The say command (implemented by do_say) broadcasts a single line, prefixed with the name of the user who spoke.
- The look command (implemented by do_look) tells the user which users are currently in the room.
- The who command (implemented by do_who) tells the user which users are currently logged in. In this simple server, look and who are equivalent, but if you extend it to contain more than one room, their functionality will differ.

For the source code, see Listing 24-6 later in this chapter.

The New Server

I've now described most of the functionality. The main additions to ChatSession and ChatServer are as follows:

- ChatSession has a method called enter, which is used to enter a new room.
- The ChatSession constructor uses LoginRoom.
- The handle_close method uses LogoutRoom.
- The ChatServer constructor adds the dictionary users and the ChatRoom called main_room to its attributes.

Notice also how handle_accept no longer adds the new ChatSession to a list of sessions because the sessions are now managed by the rooms.

■ **Note** In general, if you simply instantiate an object, like the ChatSession in handle_accept, without binding a name to it or adding it to a container, it will be lost, and may be garbage-collected (which means that it will disappear completely). Because all dispatchers are handled (referenced) by asyncore (and async_chat is a subclass of dispatcher), this is not a problem here.

The final version of the chat server is shown in Listing 24-6. For your convenience, I've listed the available commands in Table 24-1.

Listing 24-6. *A Slightly More Complicated Chat Server (chatserver.py)*

```
from asyncore import dispatcher
from asynchat import async_chat
import socket, asyncore

PORT = 5005
NAME = 'TestChat'
```

```
class EndSession(Exception): pass

class CommandHandler:
    """
    Simple command handler similar to cmd.Cmd from the standard
    library.
    """


    def unknown(self, session, cmd):
        'Respond to an unknown command'
        session.push('Unknown command: %s\r\n' % cmd)

    def handle(self, session, line):
        'Handle a received line from a given session'
        if not line.strip(): return
        # Split off the command:
        parts = line.split(' ', 1)
        cmd = parts[0]
        try: line = parts[1].strip()
        except IndexError: line = ''
        # Try to find a handler:
        meth = getattr(self, 'do_'+cmd, None)
        try:
            # Assume it's callable:
            meth(session, line)
        except TypeError:
            # If it isn't, respond to the unknown command:
            self.unknown(session, cmd)

class Room(CommandHandler):
    """
    A generic environment that may contain one or more users
    (sessions). It takes care of basic command handling and
    broadcasting.
    """

    def __init__(self, server):
        self.server = server
        self.sessions = []

    def add(self, session):
        'A session (user) has entered the room'
        self.sessions.append(session)
```

```
    def remove(self, session):
        'A session (user) has left the room'
        self.sessions.remove(session)

    def broadcast(self, line):
        'Send a line to all sessions in the room'
        for session in self.sessions:
            session.push(line)


    def do_logout(self, session, line):
        'Respond to the logout command'
        raise EndSession

class LoginRoom(Room):
    """
    A room meant for a single person who has just connected.
    """

    def add(self, session):
        Room.add(self, session)
        # When a user enters, greet him/her:
        self.broadcast('Welcome to %s\r\n' % self.server.name)

    def unknown(self, session, cmd):
        # All unknown commands (anything except login or logout)
        # results in a prodding:
        session.push('Please log in\nUse "login <nick>"\r\n')

    def do_login(self, session, line):
        name = line.strip()
        # Make sure the user has entered a name:
        if not name:
            session.push('Please enter a name\r\n')
        # Make sure that the name isn't in use:
        elif name in self.server.users:
            session.push('The name "%s" is taken.\r\n' % name)
            session.push('Please try again.\r\n')
        else:
            # The name is OK, so it is stored in the session, and
            # the user is moved into the main room.
            session.name = name
            session.enter(self.server.main_room)
```

```
class ChatRoom(Room):
    """
    A room meant for multiple users who can chat with the others in
    the room.
    """

    def add(self, session):
        # Notify everyone that a new user has entered:
        self.broadcast(session.name + ' has entered the room.\r\n')
        self.server.users[session.name] = session
        Room.add(self, session)


    def remove(self, session):
        Room.remove(self, session)
        # Notify everyone that a user has left:
        self.broadcast(session.name + ' has left the room.\r\n')

    def do_say(self, session, line):
        self.broadcast(session.name+': '+line+'\r\n')

    def do_look(self, session, line):
        'Handles the look command, used to see who is in a room'
        session.push('The following are in this room:\r\n')
        for other in self.sessions:
            session.push(other.name + '\r\n')

    def do_who(self, session, line):
        'Handles the who command, used to see who is logged in'
        session.push('The following are logged in:\r\n')
        for name in self.server.users:
            session.push(name + '\r\n')

class LogoutRoom(Room):
    """
    A simple room for a single user. Its sole purpose is to remove
    the user's name from the server.
    """

    def add(self, session):
        # When a session (user) enters the LogoutRoom it is deleted
        try: del self.server.users[session.name]
        except KeyError: pass
```

```
class ChatSession(async_chat):
    """
    A single session, which takes care of the communication with a
    single user.
    """

    def __init__(self, server, sock):
        async_chat.__init__(self, sock)
        self.server = server
        self.set_terminator("\r\n")
        self.data = []
        self.name = None
        # All sessions begin in a separate LoginRoom:
        self.enter(LoginRoom(server))


    def enter(self, room):
        # Remove self from current room and add self to
        # next room...
        try: cur = self.room
        except AttributeError: pass
        else: cur.remove(self)
        self.room = room
        room.add(self)

    def collect_incoming_data(self, data):
        self.data.append(data)

    def found_terminator(self):
        line = ''.join(self.data)
        self.data = []
        try: self.room.handle(self, line)
        except EndSession:
            self.handle_close()

    def handle_close(self):
        async_chat.handle_close(self)
        self.enter(LogoutRoom(self.server))

class ChatServer(dispatcher):
    """
    A chat server with a single room.
    """
```

```
    def __init__(self, port, name):
        dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(('', port))
        self.listen(5)
        self.name = name
        self.users = {}
        self.main_room = ChatRoom(self)

    def handle_accept(self):
        conn, addr = self.accept()
        ChatSession(self, conn)

if __name__ == '__main__':
    s = ChatServer(PORT, NAME)
    try: asyncore.loop()
    except KeyboardInterrupt: print
```

Table 24-1. *Commands Available in the Chat Server*

Command	Available In	Description
`login name`	Login room	Used to log into the server
`logout`	All rooms	Used to log out of the server
`say statement`	Chat room(s)	Used to say something
`look`	Chat room(s)	Used to find out who is in the same room
`who`	Chat room(s)	Used to find out who is logged on to the server

An example of a chat session is shown in Figure 24-1. The server in that example was started with the this command:

```
python chatserver.py
```

and the user `dilbert` connected to the server using this command:

```
telnet localhost 5005
```

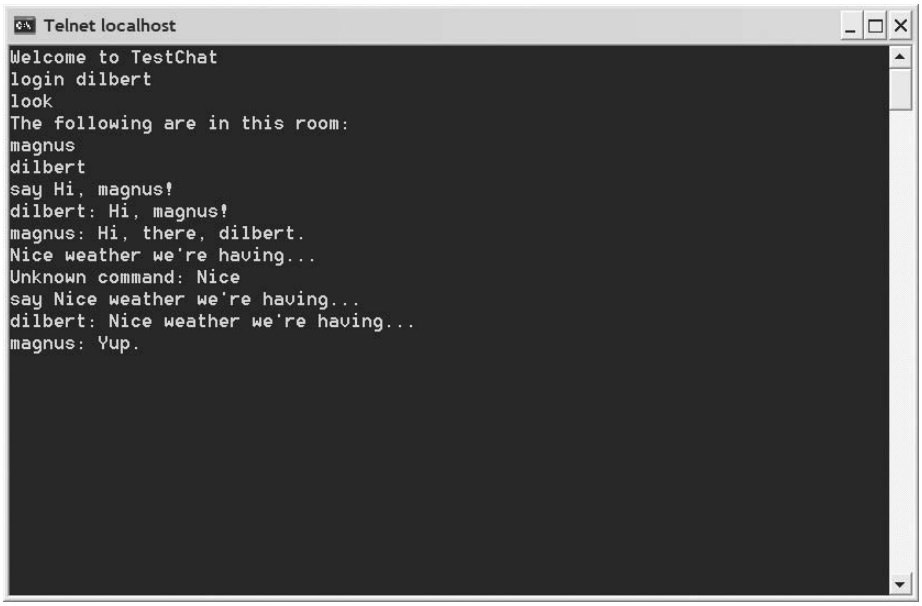

Figure 24-1. *A sample chat session*

Further Exploration

You can do a lot to extend and enhance the basic server presented in this chapter:

- You could make a version with multiple chat rooms, and you could extend the command set to make it behave in any way you want.
- You might want to make the program recognize only certain commands (such as `login` or `logout`) and treat all other text entered as general chatting, thereby avoiding the need for a `say` command.
- You could prefix all commands with a special character (for example, a slash, giving commands like `/login` and `/logout`) and treat everything that doesn't start with the specified character as general chatting.
- You might want to create your own GUI client, but that's a bit trickier than it might seem. The GUI toolkit has one event loop, and the communication with the server may require another. To make them cooperate, you may need to use threading. (For an example of how this can be done in simple cases where the various threads don't directly access each other's data, see Chapter 28.)

What Now?

Now you have your very own chat server. In the next project, you tackle a different type of network programming: CGI, the mechanism underlying most web applications (as discussed in Chapter 15). The specific application of this technology in the next project is *remote editing*, which enables several users to collaborate on developing the same document. You may even use it to edit your own web pages remotely.