Network Protocols

"Increasingly, people seem to misinterpret complexity as sophistication, which is baffling — the incomprehensible should cause suspicion rather than admiration. Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user"

Niklaus Wirth

Overview

This chapter describes Python's socket protocol support, and the networking modules built on top of the socket module. This includes client handlers for most popular Internet protocols, as well as several frameworks that can be used to implement Internet servers.

For the low-level examples in this chapter I'll use two protocols for illustration; the *Internet Time Protocol*, and the *Hypertext Transfer Protocol*.

Internet Time Protocol

The Internet Time Protocol (RFC 868, Postel and Harrenstien 1983) is a simple protocol which allows a network client to get the current time from a server.

Since this protocol is relatively light weight, many (but far from all) Unix systems provide this service. It's also about as easy to implement as a network protocol can possibly be. The server simply waits for a connection request, and immediately returns the current time as a 4-byte integer, containing the number of seconds since January 1st, 1900.

In fact, the protocol is so simple that I can include the entire specification:

```
Network Working Group                                  J. Postel - ISI
Request for Comments: 868                           K. Harrenstien - SRI
                                                               May 1983

                                Time Protocol

This RFC specifies a standard for the ARPA Internet community.  Hosts on
the ARPA Internet that choose to implement a Time Protocol are expected
to adopt and implement this standard.

This protocol provides a site-independent, machine readable date and
time.  The Time service sends back to the originating source the time in
seconds since midnight on January first 1900.

One motivation arises from the fact that not all systems have a
date/time clock, and all are subject to occasional human or machine
error.  The use of time-servers makes it possible to quickly confirm or
correct a system's idea of the time, by making a brief poll of several
independent sites on the network.

This protocol may be used either above the Transmission Control Protocol
(TCP) or above the User Datagram Protocol (UDP).

When used via TCP the time service works as follows:

   S: Listen on port 37 (45 octal).

   U: Connect to port 37.

   S: Send the time as a 32 bit binary number.

   U: Receive the time.

   U: Close the connection.

   S: Close the connection.

   The server listens for a connection on port 37.  When the connection
   is established, the server returns a 32-bit time value and closes the
   connection.  If the server is unable to determine the time at its
   site, it should either refuse the connection or close it without
   sending anything.
```

When used via UDP the time service works as follows:

S: Listen on port 37 (45 octal).

U: Send an empty datagram to port 37.

S: Receive the empty datagram.

S: Send a datagram containing the time as a 32 bit binary number.

U: Receive the time datagram.

The server listens for a datagram on port 37. When a datagram arrives, the server returns a datagram containing the 32-bit time value. If the server is unable to determine the time at its site, it should discard the arriving datagram and make no reply.

The Time

The time is the number of seconds since 00:00 (midnight) 1 January 1900 GMT, such that the time 1 is 12:00:01 am on 1 January 1900 GMT; this base will serve until the year 2036.

For example:

the time 2,208,988,800 corresponds to 00:00 1 Jan 1970 GMT,

2,398,291,200 corresponds to 00:00 1 Jan 1976 GMT,

2,524,521,600 corresponds to 00:00 1 Jan 1980 GMT,

2,629,584,000 corresponds to 00:00 1 May 1983 GMT,

and -1,297,728,000 corresponds to 00:00 17 Nov 1858 GMT.

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP, Fielding et al., RFC 2616) is something completely different. The most recent specification (version 1.1), is over 100 pages.

However, in its simplest form, this protocol is very straightforward. To fetch a document, the client connects to the server, and sends a request like:

```
GET /hello.txt HTTP/1.0
Host: hostname
User-Agent: name

[optional request body]
```

In return, the server returns a response like this:

```
HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 7

Hello
```

Both the request and response headers usually contains more fields, but the **Host** field in the request header is the only one that must always be present.

The header lines are separated by "**\r\n**", and the header must be followed by an empty line, even if there is no body (this applies to both the request and the response).

The rest of the HTTP specification deals with stuff like content negotiation, cache mechanics, persistent connections, and much more. For the full story, see *Hypertext Transfer Protocol — HTTP/1.1*.

The socket module

This module implements an interface to the socket communication layer. You can create both client and server sockets using this module.

Let's start with a client example. The following client connects to a time protocol server, reads the 4-byte response, and converts it to a time value.

Example: Using the socket module to implement a time client

```
# File:socket-example-1.py

import socket
import struct, time

# server
HOST = "www.python.org"
PORT = 37

# reference time (in seconds since 1900-01-01 00:00:00)
TIME1970 = 2208988800L # 1970-01-01 00:00:00

# connect to server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

# read 4 bytes, and convert to time value
t = s.recv(4)
t = struct.unpack("!I", t)[0]
t = int(t - TIME1970)

s.close()

# print results
print "server time is", time.ctime(t)
print "local clock is", int(time.time()) - t, "seconds off"

server time is Sat Oct 09 16:42:36 1999
local clock is 8 seconds off
```

The **socket** factory function creates a new socket of the given type (in this case, an Internet stream socket, also known as a TCP socket). The **connect** method attempts to connect this socket to the given server. Once that has succeeded, the **recv** method is used to read data.

Creating a server socket is done in a similar fashion. But instead of connecting to a server, you **bind** the socket to a port on the local machine, tell it to **listen** for incoming connection requests, and process each request as fast as possible.

The following example creates a time server, bound to port 8037 on the local machine (port numbers up to 1024 are reserved for system services, and you have to have root privileges to use them to implement services on a Unix system):

Example: Using the socket module to implement a time server

```
# File:socket-example-2.py

import socket
import struct, time

# user-accessible port
PORT = 8037

# reference time
TIME1970 = 2208988800L

# establish server
service = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
service.bind(("", PORT))
service.listen(1)

print "listening on port", PORT

while 1:
    # serve forever
    channel, info = service.accept()
    print "connection from", info
    t = int(time.time()) + TIME1970
    t = struct.pack("!I", t)
    channel.send(t) # send timestamp
    channel.close() # disconnect
```

```
listening on port 8037
connection from ('127.0.0.1', 1469)
connection from ('127.0.0.1', 1470)
...
```

The **listen** call tells the socket that we're willing to accept incoming connections. The argument gives the size of the connection queue (which holds connection requests that our program hasn't gotten around to processing yet). Finally, the **accept** loop returns the current time to any client bold enough to connect.

Note that the **accept** function returns a new socket object, which is directly connected to the client. The original socket is only used to establish the connection; all further traffic goes via the new socket.

To test this server, we can use the following generalized version of our first example:

Example: A time protocol client

```
# File:timeclient.py

import socket
import struct, sys, time

# default server
host = "localhost"
port = 8037

# reference time (in seconds since 1900-01-01 00:00:00)
TIME1970 = 2208988800L # 1970-01-01 00:00:00

def gettime(host, port):
    # fetch time buffer from stream server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    t = s.recv(4)
    s.close()
    t = struct.unpack("!I", t)[0]
    return int(t - TIME1970)

if __name__ == "__main__":
    # command line utility
    if sys.argv[1:]:
        host = sys.argv[1]
        if sys.argv[2:]:
            port = int(sys.argv[2])
        else:
            port = 37 # default for public servers

    t = gettime(host, port)
    print "server time is", time.ctime(t)
    print "local clock is", int(time.time()) - t, "seconds off"
```

```
server time is Sat Oct 09 16:58:50 1999
local clock is 0 seconds off
```

This sample script can also be used as a module; to get the current time from a server, import the **timeclient** module, and call the **gettime** function.

This far, we've used stream (or TCP) sockets. The time protocol specification also mentions UDP sockets, or datagrams. Stream sockets work pretty much like a phone line; you'll know if someone at the remote end picks up the receiver, and you'll notice when she hangs up. In contrast, sending datagrams is more like shouting into a dark room. There might be someone there, but you won't know unless she replies.

You don't need to connect to send data over a datagram socket. Instead, you use the **sendto** method, which takes both the data and the address of the receiver. To read incoming datagrams, use the **recvfrom** method.

Example: Using the socket module to implement a datagram time client

```
# File:socket-example-4.py

import socket
import struct, time

# server
HOST = "localhost"
PORT = 8037

# reference time (in seconds since 1900-01-01 00:00:00)
TIME1970 = 2208988800L # 1970-01-01 00:00:00

# connect to server
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# send empty packet
s.sendto("", (HOST, PORT))

# read 4 bytes from server, and convert to time value
t, server = s.recvfrom(4)
t = struct.unpack("!I", t)[0]
t = int(t - TIME1970)

s.close()

print "server time is", time.ctime(t)
print "local clock is", int(time.time()) - t, "seconds off"
```

```
server time is Sat Oct 09 16:42:36 1999
local clock is 8 seconds off
```

Note that **recvfrom** returns two values; the actual data, and the address of the sender. Use the latter if you need to reply.

Here's the corresponding server:

Example: Using the socket module to implement a datagram time server

```
# File:socket-example-5.py

import socket
import struct, time

# user-accessible port
PORT = 8037

# reference time
TIME1970 = 2208988800L

# establish server
service = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
service.bind(("", PORT))
```

```
print "listening on port", PORT

while 1:
    # serve forever
    data, client = service.recvfrom(0)
    print "connection from", client
    t = int(time.time()) + TIME1970
    t = struct.pack("!I", t)
    service.sendto(t, client) # send timestamp
```

```
listening on port 8037
connection from ('127.0.0.1', 1469)
connection from ('127.0.0.1', 1470)
...
```

The main difference is that the server uses **bind** to assign a known port number to the socket, and sends data back to the client address returned by **recvfrom**.

The select module

This module allows you to check for incoming data on one or more sockets, pipes, or other compatible stream objects.

You can pass one or more sockets to the **select** function, to wait for them to become readable, writable, or signal an error.

- A socket becomes *ready for reading* when 1) someone connects after a call to **listen** (which means that **accept** won't block), or 2) data arrives from the remote end, or 3) the socket is closed or reset (in this case, **recv** will return an empty string).
- A socket becomes *ready for writing* when 1) the connection is established after a non-blocking call to **connect**, or 2) data can be written to the socket.
- A socket signals an *error condition* when the connection fails after a non-blocking call to **connect**.

Example: Using the select module to wait for data arriving over sockets

```
# File:select-example-1.py

import select
import socket
import time

PORT = 8037

TIME1970 = 2208988800L

service = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
service.bind(("", PORT))
service.listen(1)

print "listening on port", PORT

while 1:
    is_readable = [service]
    is_writable = []
    is_error = []
    r, w, e = select.select(is_readable, is_writable, is_error, 1.0)
    if r:
        channel, info = service.accept()
        print "connection from", info
        t = int(time.time()) + TIME1970
        t = chr(t>>24&255) + chr(t>>16&255) + chr(t>>8&255) + chr(t&255)
        channel.send(t) # send timestamp
        channel.close() # disconnect
    else:
        print "still waiting"
```

```
listening on port 8037
still waiting
still waiting
connection from ('127.0.0.1', 1469)
still waiting
connection from ('127.0.0.1', 1470)
...
```

In this example, we wait for the listening socket to become readable, which indicates that a connection request has arrived. We treat the channel socket as usual, since it's not very likely that writing the four bytes will fill the network buffers. If you need to send larger amounts of data to the client, you should add it to the **is_writable** list at the top of the loop, and write only when **select** tells you to.

If you set the socket in *non-blocking mode* (by calling the **setblocking** method), you can use **select** also to wait for a socket to become connected. But the **asyncore** module (see the next section) provides a powerful framework which handles all this for you, so I won't go into further details here.

The asyncore module

This module provides a "reactive" socket implementation. Instead of creating socket objects, and calling methods on them to do things, this module lets you write code that is called when something can be done. To implement an asynchronous socket handler, subclass the **dispatcher** class, and override one or more of the following methods:

- **handle_connect** is called when a connection is successfully established.
- **handle_expt** is called when a connection fails.
- **handle_accept** is called when a connection request is made to a listening socket. The callback should call the **accept** method to get the client socket.
- **handle_read** is called when there is data waiting to be read from the socket. The callback should call the **recv** method to get the data.
- **handle_write** is called when data can be written to the socket. Use the **send** method to write data.
- **handle_close** is called when the socket is closed or reset.
- **handle_error(type, value, traceback)** is called if a Python error occurs in any of the other callbacks. The default implementation prints an abbreviated traceback to **sys.stdout**.

The first example shows a time client, similar to the one for the **socket** module:

Example: Using the asyncore module to get the time from a time server

```
# File:asyncore-example-1.py

import asyncore
import socket, time

# reference time (in seconds since 1900-01-01 00:00:00)
TIME1970 = 2208988800L # 1970-01-01 00:00:00

class TimeRequest(asyncore.dispatcher):
    # time requestor (as defined in RFC 868)

    def __init__(self, host, port=37):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect((host, port))

    def writable(self):
        return 0 # don't have anything to write

    def handle_connect(self):
        pass # connection succeeded

    def handle_expt(self):
        self.close() # connection failed, shutdown
```

```
    def handle_read(self):
        # get local time
        here = int(time.time()) + TIME1970

        # get and unpack server time
        s = self.recv(4)
        there = ord(s[3]) + (ord(s[2])<<8) + (ord(s[1])<<16) + (ord(s[0])<<24L)

        self.adjust_time(int(here - there))

        self.handle_close() # we don't expect more data

    def handle_close(self):
        self.close()

    def adjust_time(self, delta):
        # override this method!
        print "time difference is", delta

#
# try it out

request = TimeRequest("www.python.org")

asyncore.loop()
```

```
log: adding channel <TimeRequest  at 8cbe90>
time difference is 28
log: closing channel 192:<TimeRequest connected at 8cbe90>
```

If you don't want the log messages, override the **log** method in your **dispatcher** subclass.

Here's the corresponding time server. Note that it uses two **dispatcher** subclasses, one for the listening socket, and one for the client channel.

Example: Using the asyncore module to implement a time server

```
# File:asyncore-example-2.py

import asyncore
import socket, time

# reference time
TIME1970 = 2208988800L

class TimeChannel(asyncore.dispatcher):

    def handle_write(self):
        t = int(time.time()) + TIME1970
        t = chr(t>>24&255) + chr(t>>16&255) + chr(t>>8&255) + chr(t&255)
        self.send(t)
        self.close()
```

```
class TimeServer(asyncore.dispatcher):

    def __init__(self, port=37):
        self.port = port
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(("", port))
        self.listen(5)
        print "listening on port", self.port

    def handle_accept(self):
        channel, addr = self.accept()
        TimeChannel(channel)

server = TimeServer(8037)
asyncore.loop()
```

```
log: adding channel <TimeServer  at 8cb940>
listening on port 8037
log: adding channel <TimeChannel  at 8b2fd0>
log: closing channel 52:<TimeChannel connected at 8b2fd0>
```

In addition to the plain **dispatcher**, this module also includes a **dispatcher_with_send** class. This class allows you send larger amounts of data, without clogging up the network transport buffers.

The following module defines an **AsyncHTTP** class based on the **dispatcher_with_send** class. When you create an instance of this class, it issues an HTTP GET request, and sends the incoming data to a "consumer" target object.

Example: Using the asyncore module to do HTTP requests

```
# File:SimpleAsyncHTTP.py

import asyncore
import string, socket
import StringIO
import mimetools, urlparse

class AsyncHTTP(asyncore.dispatcher_with_send):
    # HTTP requestor

    def __init__(self, uri, consumer):
        asyncore.dispatcher_with_send.__init__(self)

        self.uri = uri
        self.consumer = consumer

        # turn the uri into a valid request
        scheme, host, path, params, query, fragment = urlparse.urlparse(uri)
        assert scheme == "http", "only supports HTTP requests"
        try:
            host, port = string.split(host, ":", 1)
            port = int(port)
        except (TypeError, ValueError):
            port = 80 # default port
        if not path:
            path = "/"
        if params:
            path = path + ";" + params
        if query:
            path = path + "?" + query

        self.request = "GET %s HTTP/1.0\r\nHost: %s\r\n\r\n" % (path, host)

        self.host = host
        self.port = port

        self.status = None
        self.header = None

        self.data = ""

        # get things going!
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect((host, port))

    def handle_connect(self):
        # connection succeeded
        self.send(self.request)
```

```
    def handle_expt(self):
        # connection failed; notify consumer (status is None)
        self.close()
        try:
            http_header = self.consumer.http_header
        except AttributeError:
            pass
        else:
            http_header(self)

    def handle_read(self):
        data = self.recv(2048)
        if not self.header:
            self.data = self.data + data
            try:
                i = string.index(self.data, "\r\n\r\n")
            except ValueError:
                return # continue
            else:
                # parse header
                fp = StringIO.StringIO(self.data[:i+4])
                # status line is "HTTP/version status message"
                status = fp.readline()
                self.status = string.split(status, " ", 2)
                # followed by a rfc822-style message header
                self.header = mimetools.Message(fp)
                # followed by a newline, and the payload (if any)
                data = self.data[i+4:]
                self.data = ""
                # notify consumer (status is non-zero)
                try:
                    http_header = self.consumer.http_header
                except AttributeError:
                    pass
                else:
                    http_header(self)
                if not self.connected:
                    return # channel was closed by consumer

        self.consumer.feed(data)

    def handle_close(self):
        self.consumer.close()
        self.close()
```

And here's a simple script using that class:

Example: Using the SimpleAsyncHTTP class

```
# File:asyncore-example-3.py

import SimpleAsyncHTTP
import asyncore

class DummyConsumer:
    size = 0

    def http_header(self, request):
        # handle header
        if request.status is None:
            print "connection failed"
        else:
            print "status", "=>", request.status
            for key, value in request.header.items():
                print key, "=", value

    def feed(self, data):
        # handle incoming data
        self.size = self.size + len(data)

    def close(self):
        # end of data
        print self.size, "bytes in body"

#
# try it out

consumer = DummyConsumer()

request = SimpleAsyncHTTP.AsyncHTTP(
    "http://www.pythonware.com",
    consumer
    )

asyncore.loop()
```

```
log: adding channel <AsyncHTTP  at 8e2850>
status => ['HTTP/1.1', '200', 'OK\015\012']
server = Apache/Unix (Unix)
content-type = text/html
content-length = 3730
...
3730 bytes in body
log: closing channel 156:<AsyncHTTP connected at 8e2850>
```

Note that the consumer interface is designed to be compatible with the **htmllib** and **xmllib** parsers. This allows you to parse HTML or XML data on the fly. Note that the **http_header** method is optional; if it isn't defined, it's simply ignored.

A problem with the above example is that it doesn't work for redirected resources. The following example adds an extra consumer layer, which handles the redirection:

Example: Using the SimpleAsyncHTTP class with redirection

```
# File:asyncore-example-4.py

import SimpleAsyncHTTP
import asyncore

class DummyConsumer:
    size = 0

    def http_header(self, request):
        # handle header
        if request.status is None:
            print "connection failed"
        else:
            print "status", "=>", request.status
            for key, value in request.header.items():
                print key, "=", value

    def feed(self, data):
        # handle incoming data
        self.size = self.size + len(data)

    def close(self):
        # end of data
        print self.size, "bytes in body"

class RedirectingConsumer:

    def __init__(self, consumer):
        self.consumer = consumer

    def http_header(self, request):
        # handle header
        if request.status is None or\
           request.status[1] not in ("301", "302"):
            try:
                http_header = self.consumer.http_header
            except AttributeError:
                pass
            else:
                return http_header(request)
        else:
            # redirect!
            uri = request.header["location"]
            print "redirecting to", uri, "..."
            request.close()
            SimpleAsyncHTTP.AsyncHTTP(uri, self)
```

```
    def feed(self, data):
        self.consumer.feed(data)

    def close(self):
        self.consumer.close()

#
# try it out

consumer = RedirectingConsumer(DummyConsumer())

request = SimpleAsyncHTTP.AsyncHTTP(
    "http://www.pythonware.com/library",
    consumer
    )

asyncore.loop()
```

```
log: adding channel <AsyncHTTP  at 8e64b0>
redirecting to http://www.pythonware.com/library/ ...
log: closing channel 48:<AsyncHTTP connected at 8e64b0>
log: adding channel <AsyncHTTP  at 8ea790>
status => ['HTTP/1.1', '200', 'OK\015\012']
server = Apache/Unix (Unix)
content-type = text/html
content-length = 387
...
387 bytes in body
log: closing channel 236:<AsyncHTTP connected at 8ea790>
```

If the server returns status 301 (permanent redirection) or 302 (temporary redirection), the redirecting consumer closes the current request, and issues a new one for the new address. All other calls to the consumer are delegated to the original consumer.

The asynchat module

This module is an extension to **asyncore**. It provides additional support for line oriented protocols. It also provides improved buffering support, via the **push** methods and the "producer" mechanism.

The following example implements a very minimal HTTP responder. It simply returns a HTML document containing information from HTTP request (the output appears in the browser window):

Example: Using the asynchat module to implement a minimal HTTP server

```
# File:asynchat-example-1.py

import asyncore, asynchat
import os, socket, string

PORT = 8000

class HTTPChannel(asynchat.async_chat):

    def __init__(self, server, sock, addr):
        asynchat.async_chat.__init__(self, sock)
        self.set_terminator("\r\n")
        self.request = None
        self.data = ""
        self.shutdown = 0

    def collect_incoming_data(self, data):
        self.data = self.data + data

    def found_terminator(self):
        if not self.request:
            # got the request line
            self.request = string.split(self.data, None, 2)
            if len(self.request) != 3:
                self.shutdown = 1
            else:
                self.push("HTTP/1.0 200 OK\r\n")
                self.push("Content-type: text/html\r\n")
                self.push("\r\n")
            self.data = self.data + "\r\n"
            self.set_terminator("\r\n\r\n") # look for end of headers
        else:
            # return payload.
            self.push("<html><body><pre>\r\n")
            self.push(self.data)
            self.push("</pre></body></html>\r\n")
            self.close_when_done()
```

```
class HTTPServer(asyncore.dispatcher):

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

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

#
# try it out

s = HTTPServer(PORT)
print "serving at port", PORT, "..."
asyncore.loop()
```

```
GET / HTTP/1.1
Accept: */*
Accept-Language: en, sv
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; Bruce/1.0)
Host: localhost:8000
Connection: Keep-Alive
```

The producer interface allows you to "push" objects that are too large to store in memory. **asyncore** calls the producer's **more** method whenever it needs more data. To signal end of file, just return an empty string.

The following example implements a very simple file-based HTTP server, using a simple **FileProducer** class that reads data from a file, a few kilobytes at the time.

Example: Using the asynchat module to implement a simple HTTP server

```
# File:asynchat-example-2.py

import asyncore, asynchat
import os, socket, string, sys
import StringIO, mimetools

ROOT = "."

PORT = 8000
```

```
class HTTPChannel(asynchat.async_chat):

    def __init__(self, server, sock, addr):
        asynchat.async_chat.__init__(self, sock)
        self.server = server
        self.set_terminator("\r\n\r\n")
        self.header = None
        self.data = ""
        self.shutdown = 0

    def collect_incoming_data(self, data):
        self.data = self.data + data
        if len(self.data) > 16384:
            # limit the header size to prevent attacks
            self.shutdown = 1

    def found_terminator(self):
        if not self.header:
            # parse http header
            fp = StringIO.StringIO(self.data)
            request = string.split(fp.readline(), None, 2)
            if len(request) != 3:
                # badly formed request; just shut down
                self.shutdown = 1
            else:
                # parse message header
                self.header = mimetools.Message(fp)
                self.set_terminator("\r\n")
                self.server.handle_request(
                    self, request[0], request[1], self.header
                    )
                self.close_when_done()
            self.data = ""
        else:
            pass # ignore body data, for now

    def pushstatus(self, status, explanation="OK"):
        self.push("HTTP/1.0 %d %s\r\n" % (status, explanation))


class FileProducer:
    # a producer which reads data from a file object

    def __init__(self, file):
        self.file = file

    def more(self):
        if self.file:
            data = self.file.read(2048)
            if data:
                return data
            self.file = None
        return ""
```

```
class HTTPServer(asyncore.dispatcher):

    def __init__(self, port=None, request=None):
        if not port:
            port = 80
        self.port = port
        if request:
            self.handle_request = request # external request handler
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind(("", port))
        self.listen(5)

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

    def handle_request(self, channel, method, path, header):
        try:
            # this is not safe!
            while path[:1] == "/":
                path = path[1:]
            filename = os.path.join(ROOT, path)
            print path, "=>", filename
            file = open(filename, "r")
        except IOError:
            channel.pushstatus(404, "Not found")
            channel.push("Content-type: text/html\r\n")
            channel.push("\r\n")
            channel.push("<html><body>File not found.</body></html>\r\n")
        else:
            channel.pushstatus(200, "OK")
            channel.push("Content-type: text/html\r\n")
            channel.push("\r\n")
            channel.push_with_producer(FileProducer(file))

#
# try it out

s = HTTPServer(PORT)
print "serving at port", PORT
asyncore.loop()
```

```
serving at port 8000
log: adding channel <HTTPServer  at 8e54d0>
log: adding channel <HTTPChannel  at 8e64a0>
samples/sample.htm => .\samples/sample.htm
log: closing channel 96:<HTTPChannel connected at 8e64a0>
```

The urllib module

This module provides a unified client interface for HTTP, FTP, and gopher. It automatically picks the right protocol handler based on the uniform resource locator (URL) passed to the library.

Fetching data from an URL is extremely easy. Just call the **urlopen** method, and read from the returned stream object.

Example: Using the urllib module to fetch a remote resource

```
# File:urllib-example-1.py

import urllib

fp = urllib.urlopen("http://www.python.org")

op = open("out.html", "wb")

n = 0

while 1:
    s = fp.read(8192)
    if not s:
        break
    op.write(s)
    n = n + len(s)

fp.close()
op.close()

for k, v in fp.headers.items():
    print k, "=", v

print "copied", n, "bytes from", fp.url
```

```
server = Apache/1.3.6 (Unix)
content-type = text/html
accept-ranges = bytes
date = Mon, 11 Oct 1999 20:11:40 GMT
connection = close
etag = "741e9-7870-37f356bf"
content-length = 30832
last-modified = Thu, 30 Sep 1999 12:25:35 GMT
copied 30832 bytes from http://www.python.org
```

Note that stream object provides some non-standard attributes. **headers** is a **Message** object (as defined by the **mimetools** module), and **url** contains the actual URL. The latter is updated if the server redirects the client to a new URL.

The **urlopen** function is actually a helper function, which creates an instance of the **FancyURLopener** class, and calls its **open** method. To get special behavior, you can subclass that class. For example, the following class automatically logs in to the server, when necessary:

Example: Using the urllib module with automatic authentication

```
# File:urllib-example-3.py

import urllib

class myURLOpener(urllib.FancyURLopener):
    # read an URL, with automatic HTTP authentication

    def setpasswd(self, user, passwd):
        self.__user = user
        self.__passwd = passwd

    def prompt_user_passwd(self, host, realm):
        return self.__user, self.__passwd

urlopener = myURLOpener()
urlopener.setpasswd("mulder", "trustno1")

fp = urlopener.open("http://www.secretlabs.com")
print fp.read()
```

The urlparse module

This module contains functions to process uniform resource locators (URLs), and to convert between URLs and platform-specific filenames.

Example: Using the urlparse module

```
# File:urlparse-example-1.py

import urlparse

print urlparse.urlparse("http://host/path;params?query#fragment")
```

```
('http', 'host', '/path', 'params', 'query', 'fragment')
```

A common use is to split an HTTP URLs into host and path components (an HTTP request involves asking the host to return data identified by the path):

Example: Using the urlparse module to parse HTTP locators

```
# File:urlparse-example-2.py

import urlparse

scheme, host, path, params, query, fragment =\
        urlparse.urlparse("http://host/path;params?query#fragment")

if scheme == "http":
    print "host", "=>", host
    if params:
        path = path + ";" + params
    if query:
        path = path + "?" + query
    print "path", "=>", path
```

```
host => host
path => /path;params?query
```

Alternatively, you can use the **urlunparse** function to put the URL back together again:

Example: Using the urlparse module to parse HTTP locators

```
# File:urlparse-example-3.py

import urlparse

scheme, host, path, params, query, fragment =\
        urlparse.urlparse("http://host/path;params?query#fragment")

if scheme == "http":
    print "host", "=>", host
    print "path", "=>", urlparse.urlunparse((None, None, path, params, query, None))
```

```
host => host
path => /path;params?query
```

The **urljoin** function is used to combine an absolute URL with a second, possibly relative URL:

Example: Using the urlparse module to combine relative locators

```
# File:urlparse-example-4.py

import urlparse

base = "http://spam.egg/my/little/pony"

for path in "/index", "goldfish", "../black/cat":
    print path, "=>", urlparse.urljoin(base, path)

/index => http://spam.egg/index
goldfish => http://spam.egg/my/little/goldfish
../black/cat => http://spam.egg/my/black/cat
```

The Cookie module

(New in 2.0). This module provides basic cookie support for HTTP clients and servers.

Example: Using the cookie module

```
# File:cookie-example-1.py

import Cookie
import os, time

cookie = Cookie.SimpleCookie()
cookie["user"] = "Mimi"
cookie["timestamp"] = time.time()

print cookie

# simulate CGI roundtrip
os.environ["HTTP_COOKIE"] = str(cookie)

print

cookie = Cookie.SmartCookie()
cookie.load(os.environ["HTTP_COOKIE"])

for key, item in cookie.items():
    # dictionary items are "Morsel" instances
    # use value attribute to get actual value
    print key, repr(item.value)
```

```
Set-Cookie: timestamp=736513200;
Set-Cookie: user=Mimi;

user 'Mimi'
timestamp '736513200'
```

The robotparser module

(New in 2.0). This module reads **robots.txt** files, which are used to implement the *Robot Exclusion Protocol*.

If you're implementing an HTTP robot that will visit arbitrary sites on the net (not just your own sites), it's a good idea to use this module to check that you really are welcome.

Example: Using the robotparser module

```
# File:robotparser-example-1.py

import robotparser

r = robotparser.RobotFileParser()
r.set_url("http://www.python.org/robots.txt")
r.read()

if r.can_fetch("*", "/index.html"):
    print "may fetch the home page"

if r.can_fetch("*", "/tim_one/index.html"):
    print "may fetch the tim peters archive"
```

```
may fetch the home page
```

The ftplib module

This module contains a *File Transfer Protocol* (FTP) client implementation.

The first example shows how to log in and get a directory listing of the login directory. Note that the format of the directory listing is server dependent (it's usually the same as the format used by the directory listing utility on the server host platform).

Example: Using the ftplib module to get a directory listing

```
# File:ftplib-example-1.py

import ftplib

ftp = ftplib.FTP("www.python.org")
ftp.login("anonymous", "ftplib-example-1")

print ftp.dir()

ftp.quit()
```

```
total 34
drwxrwxr-x  11 root    4127       512 Sep 14 14:18 .
drwxrwxr-x  11 root    4127       512 Sep 14 14:18 ..
drwxrwxr-x   2 root    4127       512 Sep 13 15:18 RCS
lrwxrwxrwx   1 root    bin         11 Jun 29 14:34 README -> welcome.msg
drwxr-xr-x   3 root    wheel      512 May 19  1998 bin
drwxr-sr-x   3 root    1400       512 Jun  9  1997 dev
drwxrwxr--   2 root    4127       512 Feb  8  1998 dup
drwxr-xr-x   3 root    wheel      512 May 19  1998 etc
...
```

Downloading files is easy; just use the appropriate **retr** function. Note that when you download a text file, you have to add line endings yourself. The following function uses a **lambda** expression to do that on the fly.

Example: Using the ftplib module to retrieve files

```
# File:ftplib-example-2.py

import ftplib
import sys

def gettext(ftp, filename, outfile=None):
    # fetch a text file
    if outfile is None:
        outfile = sys.stdout
    # use a lambda to add newlines to the lines read from the server
    ftp.retrlines("RETR " + filename, lambda s, w=outfile.write: w(s+"\n"))

def getbinary(ftp, filename, outfile=None):
    # fetch a binary file
    if outfile is None:
        outfile = sys.stdout
    ftp.retrbinary("RETR " + filename, outfile.write)
```

```
ftp = ftplib.FTP("www.python.org")
ftp.login("anonymous", "ftplib-example-2")

gettext(ftp, "README")
getbinary(ftp, "welcome.msg")
```

```
WELCOME to python.org, the Python programming language home site.

You are number %N of %M allowed users.  Ni!

Python Web site: http://www.python.org/

CONFUSED FTP CLIENT?  Try begining your login password with '-' dash.
This turns off continuation messages that may be confusing your client.
...
```

Finally, here's a simple example that copies files to the FTP server. This script uses the file extension to figure out if the file is a text file or a binary file:

Example: Using the ftplib module to store files

```
# File:ftplib-example-3.py

import ftplib
import os

def upload(ftp, file):
    ext = os.path.splitext(file)[1]
    if ext in (".txt", ".htm", ".html"):
        ftp.storlines("STOR " + file, open(file))
    else:
        ftp.storbinary("STOR " + file, open(file, "rb"), 1024)

ftp = ftplib.FTP("ftp.fbi.gov")
ftp.login("mulder", "trustno1")

upload(ftp, "trixie.zip")
upload(ftp, "file.txt")
upload(ftp, "sightings.jpg")
```

The gopherlib module

This module contains a gopher client implementation.

Example: Using the gopherlib module

```
# File:gopherlib-example-1.py

import gopherlib

host = "gopher.spam.egg"

f = gopherlib.send_selector("1/", host)
for item in gopherlib.get_directory(f):
    print item
```

```
['0', "About Spam.Egg's Gopher Server", "0/About's Spam.Egg's
Gopher Server", 'gopher.spam.egg', '70', '+']
['1', 'About Spam.Egg', '1/Spam.Egg', 'gopher.spam.egg', '70', '+']
['1', 'Misc', '1/Misc', 'gopher.spam.egg', '70', '+']
...
```

The httplib module

This module provides a low-level HTTP client interface.

Example: Using the httplib module

```
# File:httplib-example-1.py

import httplib

USER_AGENT = "httplib-example-1.py"

class Error:
    # indicates an HTTP error
    def __init__(self, url, errcode, errmsg, headers):
        self.url = url
        self.errcode = errcode
        self.errmsg = errmsg
        self.headers = headers
    def __repr__(self):
        return (
            "<Error for %s: %s %s>" %
            (self.url, self.errcode, self.errmsg)
            )

class Server:

    def __init__(self, host):
        self.host = host

    def fetch(self, path):
        http = httplib.HTTP(self.host)

        # write header
        http.putrequest("GET", path)
        http.putheader("User-Agent", USER_AGENT)
        http.putheader("Host", self.host)
        http.putheader("Accept", "*/*")
        http.endheaders()

        # get response
        errcode, errmsg, headers = http.getreply()

        if errcode != 200:
            raise Error(errcode, errmsg, headers)

        file = http.getfile()
        return file.read()
```

```
if __name__ == "__main__":

    server = Server("www.pythonware.com")
    print server.fetch("/index.htm")
```

Note that the HTTP client provided by this module blocks while waiting for the server to respond. For an asynchronous solution, which among other things allows you to issue multiple requests in parallel, see the examples for the **asyncore** module.

Posting data to an HTTP server

The **httplib** module also allows you to send other HTTP commands, such as **POST**.

Example: Using the httplib module to post data

```
# File:httplib-example-2.py

import httplib

USER_AGENT = "httplib-example-2.py"

def post(host, path, data, type=None):

    http = httplib.HTTP(host)

    # write header
    http.putrequest("PUT", path)
    http.putheader("User-Agent", USER_AGENT)
    http.putheader("Host", host)
    if type:
        http.putheader("Content-Type", type)
    http.putheader("Content-Length", str(len(data)))
    http.endheaders()

    # write body
    http.send(data)

    # get response
    errcode, errmsg, headers = http.getreply()

    if errcode != 200:
        raise Error(errcode, errmsg, headers)

    file = http.getfile()
    return file.read()

if __name__ == "__main__":

    post("www.spam.egg", "/bacon.htm", "a piece of data", "text/plain")
```

The poplib module

This module provides a *Post Office Protocol* (POP3) client implementation. This protocol is used to "pop" (copy) messages from a central mail server to your local computer.

Example: Using the poplib module

```
# File:poplib-example-1.py

import poplib
import string, random
import StringIO, rfc822

SERVER = "pop.spam.egg"

USER  = "mulder"
PASSWORD = "trustno1"

# connect to server
server = poplib.POP3(SERVER)

# login
server.user(USER)
server.pass_(PASSWORD)

# list items on server
resp, items, octets = server.list()

# download a random message
id, size = string.split(random.choice(items))
resp, text, octets = server.retr(id)

text = string.join(text, "\n")
file = StringIO.StringIO(text)

message = rfc822.Message(file)

for k, v in message.items():
    print k, "=", v

print message.fp.read()
```

```
subject = ANN: (the eff-bot guide to) The Standard Python Library
message-id = <199910120808.KAA09206@spam.egg>
received = (from fredrik@spam.egg)
 by spam.egg (8.8.7/8.8.5) id KAA09206
 for mulder; Tue, 12 Oct 1999 10:08:47 +0200
from = Fredrik Lundh <fredrik@spam.egg>
date = Tue, 12 Oct 1999 10:08:47 +0200
to = mulder@spam.egg

...
```

The imaplib module

This module provides an *Internet Message Access Protocol* (IMAP) client implementation. This protocol lets you access mail folders stored on a central mail server, as if they were local.

Example: Using the imaplib module

```
# File:imaplib-example-1.py

import imaplib
import string, random
import StringIO, rfc822

SERVER = "imap.spam.egg"

USER  = "mulder"
PASSWORD = "trustno1"

# connect to server
server = imaplib.IMAP4(SERVER)

# login
server.login(USER, PASSWORD)
server.select()

# list items on server
resp, items = server.search(None, "ALL")
items = string.split(items[0])

# fetch a random item
id = random.choice(items)
resp, data = server.fetch(id, "(RFC822)")
text = data[0][1]

file = StringIO.StringIO(text)

message = rfc822.Message(file)

for k, v in message.items():
    print k, "=", v

print message.fp.read()

server.logout()
```

```
subject = ANN: (the eff-bot guide to) The Standard Python Library
message-id = <199910120816.KAA12177@larch.spam.egg>
to = mulder@spam.egg
date = Tue, 12 Oct 1999 10:16:19 +0200 (MET DST)
from = <effbot@spam.egg>
received = (effbot@spam.egg) by imap.algonet.se (8.8.8+Sun/8.6.12)
id KAA12177 for effbot@spam.egg; Tue, 12 Oct 1999 10:16:19 +0200
(MET DST)

body text for test 5
```

The smtplib module

This module provides a *Simple Mail Transfer Protocol* (SMTP) client implementation. This protocol is used to send mail through Unix mailservers.

To read mail, use the **poplib** or **imaplib** modules.

Example: Using the smtplib module

```
# File:smtplib-example-1.py

import smtplib
import string, sys

HOST = "localhost"

FROM = "effbot@spam.egg"
TO = "fredrik@spam.egg"

SUBJECT = "for your information!"

BODY = "next week: how to fling an otter"

body = string.join((
    "From: %s" % FROM,
    "To: %s" % TO,
    "Subject: %s" % SUBJECT,
    "",
    BODY), "\r\n")

print body

server = smtplib.SMTP(HOST)
server.sendmail(FROM, [TO], body)
server.quit()
```

```
From: effbot@spam.egg
To: fredrik@spam.egg
Subject: for your information!

next week: how to fling an otter
```

The telnetlib module

This module provides a TELNET client implementation.

The following example connects to a Unix computer, logs in, and retrieves a directory listing.

Example: Using the telnetlib module to login on a remote server

```
# File:telnetlib-example-1.py

import telnetlib
import sys

HOST = "spam.egg"

USER = "mulder"
PASSWORD = "trustno1"

telnet = telnetlib.Telnet(HOST)

telnet.read_until("login: ")
telnet.write(USER + "\n")

telnet.read_until("Password: ")
telnet.write(PASSWORD + "\n")

telnet.write("ls librarybook\n")
telnet.write("exit\n")

print telnet.read_all()
```

```
[spam.egg mulder]$ ls
README                          os-path-isabs-example-1.py
SimpleAsyncHTTP.py                os-path-isdir-example-1.py
aifc-example-1.py               os-path-isfile-example-1.py
anydbm-example-1.py               os-path-islink-example-1.py
array-example-1.py              os-path-ismount-example-1.py
...
```

The nntplib module

This module provides a *Network News Transfer Protocol* (NNTP) client implementation.

Listing messages

Prior to reading messages from a news server, you have to connect to the server, and then select a newsgroup. The following script also downloads a complete list of all messages on the server, and extracts some more or less interesting statistics from that list:

Example: Using the nntplib module to list messages

```
# File:nntplib-example-1.py

import nntplib
import string

SERVER = "news.spam.egg"
GROUP  = "comp.lang.python"
AUTHOR = "fredrik@pythonware.com" # eff-bots human alias

# connect to server
server = nntplib.NNTP(SERVER)

# choose a newsgroup
resp, count, first, last, name = server.group(GROUP)
print "count", "=>", count
print "range", "=>", first, last

# list all items on the server
resp, items = server.xover(first, last)

# extract some statistics
authors = {}
subjects = {}
for id, subject, author, date, message_id, references, size, lines in items:
    authors[author] = None
    if subject[:4] == "Re: ":
        subject = subject[4:]
    subjects[subject] = None
    if string.find(author, AUTHOR) >= 0:
        print id, subject

print "authors", "=>", len(authors)
print "subjects", "=>", len(subjects)
```

```
count => 607
range => 57179 57971
57474 Three decades of Python!
...
57477 More Python books coming...
authors => 257
subjects => 200
```

Downloading messages

Downloading a message is easy. Just call the **article** method, as shown in this script:

Example: Using the nntplib module to download messages

```
# File:nntplib-example-2.py

import nntplib
import string

SERVER = "news.spam.egg"
GROUP  = "comp.lang.python"
KEYWORD = "tkinter"

# connect to server
server = nntplib.NNTP(SERVER)

resp, count, first, last, name = server.group(GROUP)
resp, items = server.xover(first, last)
for id, subject, author, date, message_id, references, size, lines in items:
    if string.find(string.lower(subject), KEYWORD) >= 0:
        resp, id, message_id, text = server.article(id)
        print author
        print subject
        print len(text), "lines in article"
```

```
"Fredrik Lundh" <fredrik@pythonware.com>
Re: Programming Tkinter (In Python)
110 lines in article
...
```

To further manipulate the messages, you can wrap it up in a **Message** object (using the **rfc822** module):

Example: Using the nntplib and rfc822 modules to process messages

```
# File:nntplib-example-3.py

import nntplib
import string, random
import StringIO, rfc822

SERVER = "news.spam.egg"
GROUP  = "comp.lang.python"

# connect to server
server = nntplib.NNTP(SERVER)

resp, count, first, last, name = server.group(GROUP)
for i in range(10):
    try:
        id = random.randint(int(first), int(last))
        resp, id, message_id, text = server.article(str(id))
    except (nntplib.error_temp, nntplib.error_perm):
        pass # no such message (maybe it was deleted?)
    else:
        break # found a message!
else:
    raise SystemExit

text = string.join(text, "\n")
file = StringIO.StringIO(text)

message = rfc822.Message(file)

for k, v in message.items():
    print k, "=", v

print message.fp.read()
```

```
mime-version = 1.0
content-type = text/plain; charset="iso-8859-1"
message-id = <008501bf1417$1cf90b70$f29b12c2@sausage.spam.egg>
lines = 22
...
from = "Fredrik Lundh" <fredrik@pythonware.com>
nntp-posting-host = parrot.python.org
subject = ANN: (the eff-bot guide to) The Standard Python Library
...
</F>
```

Once you've gotten this far, you can use modules like **htmllib**, **uu**, and **base64** to further process the messages.

The SocketServer module

This module provides a framework for various kinds of socket-based servers. The module provides a number of classes that can be mixed and matched to create servers for different purposes.

The following example implements an Internet Time Protocol server, using this module. Use the **timeclient** script to try it out:

Example: Using the SocketServer module

```
# File:socketserver-example-1.py

import SocketServer
import time

# user-accessible port
PORT = 8037

# reference time
TIME1970 = 2208988800L

class TimeRequestHandler(SocketServer.StreamRequestHandler):
    def handle(self):
        print "connection from", self.client_address
        t = int(time.time()) + TIME1970
        b = chr(t>>24&255) + chr(t>>16&255) + chr(t>>8&255) + chr(t&255)
        self.wfile.write(b)

server = SocketServer.TCPServer(("", PORT), TimeRequestHandler)
print "listening on port", PORT
server.serve_forever()
```

```
connection from ('127.0.0.1', 1488)
connection from ('127.0.0.1', 1489)
...
```

The BaseHTTPServer module

This is a basic framework for HTTP servers, built on top of the **SocketServer** framework.

The following example generates a random message each time you reload the page. The **path** variable contains the current URL, which you can use to generate different contents for different URLs (as it stands, the script returns an error page for anything but the root path).

Example: Using the BaseHTTPServer module

```
# File:basehttpserver-example-1.py

import BaseHTTPServer
import cgi, random, sys

MESSAGES = [
    "That's as maybe, it's still a frog.",
    "Albatross! Albatross! Albatross!",
    "A pink form from Reading.",
    "Hello people, and welcome to 'It's a Tree'"
    "I simply stare at the brick and it goes to sleep.",
]

class Handler(BaseHTTPServer.BaseHTTPRequestHandler):

    def do_GET(self):
        if self.path != "/":
            self.send_error(404, "File not found")
            return
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        try:
            # redirect stdout to client
            stdout = sys.stdout
            sys.stdout = self.wfile
            self.makepage()
        finally:
            sys.stdout = stdout # restore

    def makepage(self):
        # generate a random message
        tagline = random.choice(MESSAGES)
        print "<html>"
        print "<body>"
        print "<p>Today's quote: "
        print "<i>%s</i>" % cgi.escape(tagline)
        print "</body>"
        print "</html>"

PORT = 8000

httpd = BaseHTTPServer.HTTPServer(("", PORT), Handler)
print "serving at port", PORT
httpd.serve_forever()
```

See the **SimpleHTTPServer** and **CGIHTTPServer** modules for more extensive HTTP frameworks.

The SimpleHTTPServer module

This is a simple HTTP server that provides standard GET and HEAD request handlers. The path name given by the client is interpreted as a relative file name (relative to the current directory when the server was started, that is).

Example: Using the SimpleHTTPServer module

```
# File:simplehttpserver-example-1.py

import SimpleHTTPServer
import SocketServer

# minimal web server.  serves files relative to the
# current directory.

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```

```
serving at port 8000
localhost - - [11/Oct/1999 15:07:44] code 403, message Directory listing not sup
ported
localhost - - [11/Oct/1999 15:07:44] "GET / HTTP/1.1" 403 -
localhost - - [11/Oct/1999 15:07:56] "GET /samples/sample.htm HTTP/1.1" 200 -
```

The server ignores drive letters and relative path names (such as '..'). However, it does not implement any other access control mechanisms, so be careful how you use it.

The second example implements a truly minimal web proxy. When sent to a proxy, the HTTP requests should include the full URI for the target server. This server uses **urllib** to fetch data from the target.

Example: Using the SimpleHTTPServer module as a proxy

```
# File:simplehttpserver-example-2.py

# a truly minimal HTTP proxy

import SocketServer
import SimpleHTTPServer
import urllib

PORT = 1234

class Proxy(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):
        self.copyfile(urllib.urlopen(self.path), self.wfile)

httpd = SocketServer.ForkingTCPServer(('', PORT), Proxy)
print "serving at port", PORT
httpd.serve_forever()
```

The CGIHTTPServer module

This is a simple HTTP server that can call external scripts through the common gateway interface (CGI).

Example: Using the CGIHTTPServer module

```
# File:cgihttpserver-example-1.py

import CGIHTTPServer
import BaseHTTPServer

class Handler(CGIHTTPServer.CGIHTTPRequestHandler):
    cgi_directories = ["/cgi"]

PORT = 8000

httpd = BaseHTTPServer.HTTPServer(("", PORT), Handler)
print "serving at port", PORT
httpd.serve_forever()
```

The cgi module

This module provides a number of support functions and classes for common gateway interface (CGI) scripts. Among other things, it can parse CGI form data.

Here's a simple CGI script that returns a list of a files in a given directory (relative to the root directory specified in the script).

Example: Using the cgi module

```
# File:cgi-example-1.py

import cgi
import os, urllib

ROOT = "samples"

# header
print "text/html"
print

query = os.environ.get("QUERY_STRING")
if not query:
    query = "."

script = os.environ.get("SCRIPT_NAME", "")
if not script:
    script = "cgi-example-1.py"

print "<html>"
print "<head>"
print "<title>file listing</title>"
print "</head>"
print "</html>"

print "<body>"

try:
    files = os.listdir(os.path.join(ROOT, query))
except os.error:
    files = []

for file in files:
    link = cgi.escape(file)
    if os.path.isdir(os.path.join(ROOT, query, file)):
        href = script + "?" + os.path.join(query, file)
        print "<p><a href='%s'>%s</a>" % (href, cgi.escape(link))
    else:
        print "<p>%s" % link

print "</body>"
print "</html>"
```

```
text/html

<html>
<head>
<title>file listing</title>
</head>
</html>
<body>
<p>sample.gif
<p>sample.gz
<p>sample.netrc
...
<p>sample.txt
<p>sample.xml
<p>sample~
<p><a href='cgi-example-1.py?web'>web</a>
</body>
</html>
```

The webbrowser module

(New in 2.0) This module provides a basic interface to the system's standard webbrowser. It provides a **open** function, which takes a file name or an URL, and displays it in the browser. If you call **open** again, it attempts to display the new page in the same browser window.

Example: Using the webbrowser module

```
# File:webbrowser-example-1.py

import webbrowser
import time

webbrowser.open("http://www.pythonware.com")

# wait a while, and then go to another page
time.sleep(5)
webbrowser.open(
    "http://www.pythonware.com/people/fredrik/librarybook.htm"
    )
```

On Unix, this module supports lynx, Netscape, Mosaic, Konquerer, and Grail. On Windows and Macintosh, it uses the standard browser (as defined in the registry or the Internet configuration panel).