15

Python programs as network servers

What you will learn

In this chapter, you'll learn how to create a Python program that will act as a server for network clients. You'll also discover how to make a Python program that responds to posts from users, and you'll create your first web application. This chapter will get you started creating solutions that use the web.

Create a web server in Python

The web works by using socket network connections, just like those we created in Chapter 14. When we use a browser to connect to a web server, the basis of the communication is a socket. A server program listening to a socket connection will send back the page that your browser has requested.

In Chapter 14, when we created a simple program to read webpages from a server, we noted that the appearance of webpages is expressed Hypertext Markup Language (HTML), and the conversation between a browser and a server is managed by a protocol called Hypertext Transfer Protocol (HTTP). In this section, we'll learn a bit more about the communication between a web server and a browser and create some web servers of our own.

A tiny socket-based server

I've created a tiny Python program that provides a socket connection that you can connect to via a browser program on your computer. It serves out a tiny webpage that you can view. Let's look at the code:

```
# EG15-01 Tiny socket web server

import socket                                               Import the socket library

host_ip = 'localhost'                                       Use the localhost name for this server

host_socket = 8080                                          The server will listen on port 8080

full_address = 'http://' + host_ip + ':' + str(host_socket) Build a string that contains
                                                            the server address
print('Open your browser and connect to: ', full_address)   Tell the user what to
                                                            connect to
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) Create the socket
listen_address = (host_ip, host_socket)                     Create the address to listen on

listen_socket.bind(listen_address)                          Bind the socket to the server address
listen_socket.listen()

connection, address = listen_socket.accept()                Wait for a request from a browser
print('Got connection from: ', address)                     Indicate we have a connection
```

```
network_message = connection.recv(1024)
request_string = network_message.decode()
print(request_string)

status_string = 'HTTP/1.1 200 OK'

header_string = '''Content-Type: text/html; charset=UTF-8
Connection: close

'''

content_string = '''<html>
<body>
<p>hello from our tiny server</p>
</body>
</html>

'''

response_string = status_string + header_string + content_string

response_bytes = response_string.encode()

connection.send(response_bytes)

connection.close()
```

Get the network message

Decode the network message into the request string

Print the request string

HTTP status response

HTTP response headers

HTTP content

Build the complete response

Encode the response into bytes

Send the response bytes

Close the connection

MAKE SOMETHING HAPPEN

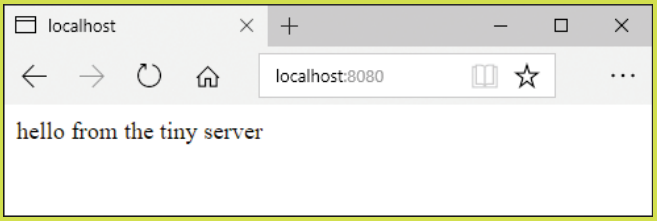
Connect to a simple server

You can use the socket web server on your PC to explore how the web works. Use IDLE to open the example program **EG15-01 Socket web server** and get started.

When you run the program, it will display the address of the web server that has been created and is waiting for a web request. You should see a display like the one below.

```
>>>
 RESTART: C:/Users/Rob/EG14-03 Tiny socket web server.py
Open your browser and connect to:  http:/localhost:8080
```

Now open your browser and connect to the address. The browser will connect to the socket from the server program and will display the webpage that it serves out:



hello from the tiny server

If you now go back to IDLE, you should see the contents of the web request made by the browser that's been printed.

```
>>>
 RESTART: C:/Users/Rob/EG15-01 Tiny socket web server.py
Got connection from:  ('192.168.1.56', 51221)
GET / HTTP/1.1
Host: 192.168.1.56:8080
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
>>>
```

The most important word on the page is the very first word of the message, GET, which is the beginning of the request for a webpage. The GET request is followed by information that the server uses to determine what kind of responses the browser can accept.

CODE ANALYSIS

Web server program

Question: Previous sockets that we have created have used a socket type of socket.SOCK_DGRAM. Why is this program using a socket type of socket.SOCK_STREAM?

Answer: The programs we created in Chapter 14 to send packets between computers sent individual datagrams using the User Datagram protocol (UDP). A datagram is very useful for sending quick messages to another computer. You can think of it as the network equivalent of a text message. When you send a text message, you have no way of knowing whether the message has been received. The browsers and servers on the web don't use datagrams to communicate; instead, they establish a network connection using the Transport Control Protocol (TCP) that allows them to exchange large amounts of data and ensure that the data has arrived. When a Python program creates a socket, it can identify that socket as using datagrams (SOCK_DGRAM) or a connection (SOCK_STREAM).

Question: What are the status_string, header_string, and content_string variables in the program used for?

Answer: The HTTP protocol defines how servers and browsers should interact. The browser will send a GET command to ask the server for a webpage. The server will send three items in its response. The first is a status response. If the page was found successfully, the status returned will be 200, as in the contents of the variable status_string above. If the page is not found, the status returned will be 404, which means "page not found."

The status information is followed directly by a header string that gives the browser information about the response. In the program above, the value assigned to header_string tells the browser that the content is text and that the network connection will be closed once the content has been delivered.

Finally, the server will send the HTML document that describes the webpage to be displayed. The content string is placed in the variable content_string in the program above. If you want to use this program to serve different content to the browser, just change the text in content_string. These three strings are added together to create the complete response string.

Question: What are the encode and decode methods used for?

Answer: The encode method takes a string of text and encodes it as a block of bytes, ready for transmission over the network. The string type provides a method called encode, which will return the contents of a string encoded as a block of bytes. The program uses this method to encode the response string that the server sends to the browser:

```
response_bytes = response_string.encode()
```

The bytes type provides a method called decode that returns the contents of the bytes decoded as a string of text. The program uses this method to decode the command that the server receives from the browser.

```
request_string = network_message.decode()
```

The network_message contains the block of bytes received from the network, which is converted into the request_string. The tiny server always serves out the same message to the browser, but it could use the contents of the request to determine which page was being requested.

Question: Could browser clients connect to this server via the Internet?

Answer: This would only be possible if your computer was directly connected to the Internet, which is not usually the case. As we saw in Chapter 14, a computer is normally connected to a local network, and the local network is connected via a router to the Internet. All the machines connected to a local network (whether it's a home, a school, or a hotel) could potentially connect to a server connected to that network, but you would need to configure the router (which connects a local network to the Internet) to allow messages from the Internet to reach your computer if you want to serve out webpages to the Internet. This is not something that's normally permitted because it opens up a machine to attack from malicious systems on the Internet.

Question: How does the statement that gets the connection work?

Answer: The following statement gets the connection to the socket:

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

This statement uses a form of method calling that we haven't used very often. It's explained at the end of Chapter 8, in the descriptions of tuples. The accept method returns a *tuple* that holds the connection and address values of the system that has connected. We can assign these values directly to variables by using the statement above. The connection object is like the object we use when we open a file. We can call methods on the connection object to read messages sent by the program at the other end of the network connection. We can also call methods on the connection to send messages to the distant machine.

Question: How could I make the sample program above into a proper web server?

Answer: We would have to add a loop so that the web server would return to waiting for connections once it had finished dealing with a request. A "proper" web server would also be able to support multiple web requests at the same time. The socket mechanism can accept more than one connection at the same time, and Python allows the creation of threads that can run simultaneously on a computer. However, we wouldn't want to create our own web server, as the developers of Python have already done this for us. We'll use their server in the next section.

Python web server

We know that a web server is just a program that uses the network to listen for requests from clients. We could create a complete web server by building on the tiny server we've just created, but it turns out that Python provides ready-built classes that we can use to do this. The `HTTPserver` class allows us to create objects that will accept connections on a network socket and dispatch them to a class that will decode and act on them.

The `BaseHTTPRequestHandler` class provides the basis of a handler for incoming web requests that our server receives. We can use the `HTTPserver` and `BaseHTTPRequestHandler` classes to create a web server as shown in the example code below. You can use a browser to connect to this server in the same way as the one we wrote above, but this server does not stop after the first request; it will continue to accept connections and serve out the website until the program is stopped.

```
# EG15-02 Python web server

import http.server

class WebServerHandler(http.server.BaseHTTPRequestHandler):

    def do_GET(self):
        '''
        This method is called when the server receives
        a GET request from the client
        It sends a fixed message back to the client
        '''
        self.send_response(200)
        self.send_header('Content-type','text/html')
        self.end_headers()

        message_text = '''<html>
<body>
<p>hello from the Python server</p>
</body>
</html>
'''
        message_bytes = message_text.encode()

        self.wfile.write(message_bytes)
        return
```

Get the server module

Create a subclass of the `BaseHTTPRequestHandler` class

Add a `do_GET` method into the handler class

Send a 200 response (OK)

Add the content type to the header

Send the header to the browser

Text of the webpage to be sent to the browser

Encode the HTML string into bytes

Write the bytes back to the browser

```
host_socket = 8080                                      Socket number for this server
host_ip = 'localhost'                                   Use localhost as the network address

host_address = (host_ip, host_socket)                   Create the host address

my_server = http.server.HTTPServer(host_address, WebServerHandler)   Create a server
my_server.serve_forever()                                            Start the server
```

CODE ANALYSIS

Python server program

Question: How does this work?

Answer: You can think of the HTTPServer class as the dispatcher for incoming requests, a bit like a receptionist at a large company. An employee of a company could tell the receptionist "If anyone asks for me, I'm in the board room." When we create the HTTP-Server, we tell it "If any web requests come in, create an instance of WebServerHandler to deal with them."

When a request comes in, the HTTPServer creates a WebServerHandler and adds all the attributes that describe the incoming request. The server then looks through the incoming request and calls the method in the WebServerHandler that matches the request that's been made. The handler we created above can only handle GET requests as it only contains a do_GET method.

Question: What does the WebServerHandler class do?

Answer: The WebServerHandler class is a subclass of a superclass called BaseHTTPRe-questHandler. A subclass of a superclass inherits all the attributes of the superclass and can add attributes of its own. The WebServerHandler above contains one attribute, which is the method called do_GET. The do_GET method will run when a browser tries to get a webpage from our server; the do_GET method returns the webpage requested by the browser. We can create different server behavior by changing what the do_GET method does. We can also make a handler that responds to other HTTP messages by adding more methods to the handler class (covered later in this chapter).

Question: How does the server program send the page back to the host?

Answer: The connection to the host takes the form of a file connection. When the WebServerHandler instance is created, it is given an attribute called wfile, which is the write file for this web request. The do_GET method can use the wfile attribute to write back the message to the server.

```
self.wfile.write(message_bytes)
```

The variable message_bytes contains the message the server is returning. Using a file in this way makes it very easy for a server to send back any kind of information, including images.

Question: How is the WebServerHandler class connected to the server?

Answer: When we create the server, we pass the server a reference to the class that it will use to respond to incoming web requests.

```
my_server = http.server.HTTPServer(host_address, WebServerHandler)
```

Above is the statement that constructs the server. Note that the second argument to the call is WebServerHandler. When the server receives a request from a browser, it creates an instance of the WebServerHandler class and then calls methods in that instance to deal with the request.

Serve webpages from files

The web servers we've created so far are not very useful because they just serve out the same information. However, we know that a single web server can serve out may different pages. Browsers and servers on the World Wide Web use a *Uniform Resource Locator,* or *URL,* string to identify destinations, which includes a *path* to the resource that will be provided. **Figure 15-1** shows the anatomy of a URL.

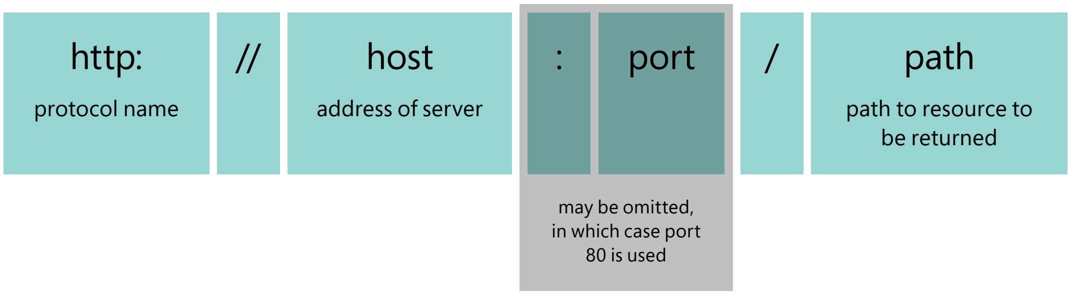

http:
protocol name
//
host
address of server
:
port
may be omitted, in which case port 80 is used
/
path
path to resource to be returned

Figure 15-1 Anatomy of a Uniform Resource Locator (URL)

The URL of a host contains the protocol to be used, the network address of the server, the socket to be used for the connection to the server, and the path to the page on the server. The URL for the webpage that contains a description of how URLs are constructed is shown in **Figure 15-2**.

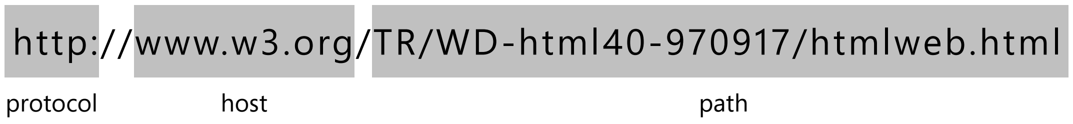


Figure 15-2 URL example

This shows that the path to a resource can include folders. In the path shown, the requested page is in the folder `WD-html40-970917`, which is held in the folder `TR`. This URL does not include a socket because the server is using port 80. If the port address is left out, the browser will use port number 80, which is the Internet port associated with the web. We've been using port 8080 for the web servers on our local machine.

A server can extract the path information from the `GET` request and send back the page that was requested. If the path is left out, the server will send back the "home" page for that location. A server can use the path to determine which file to return to the browser. The very first web servers were used to serve files of text that were stored on them. Below is a web request handler that serves out files.

```python
# EG15-03 Python webpage server
class WebServerHandler(http.server.BaseHTTPRequestHandler):

    def do_GET(self):
        '''
        This method is called when the server receives
        a GET request from the client
        It opens a file with the requested path
        and sends back the contents
        '''
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

        # trim off the leading / character in the path
        file_path = self.path[1:]

        with open(file_path, 'r') as input_file:
            message_text = input_file.read()

        message_bytes = message_text.encode()

        self.wfile.write(message_bytes)

        return
```

Send a 200 response (OK)

Tell the browser the content is text

Finish sending the header

Get the file name from the path supplied in the `GET` request

Open the file

Read the file

Encode the file into a block of bytes

Write the file back to the browser

Extract slices from a collection

The code above uses *slicing,* which is something we haven't seen before. Python programs can extract slices from collections. **Figure 15-3** shows how we would express a slicing action.

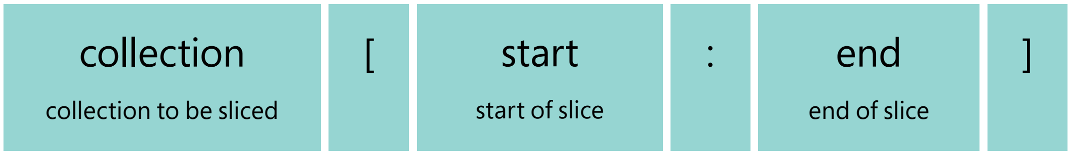


Figure 15-3 Anatomy of a slice

The start and end positions of the slice are given in square brackets, separated by a colon character. We can see how this works by slicing my name, which can be regarded as a collection of individual characters.

```
>>> 'Robert'[0:3]
'Rob'
```

The statement above creates a slice from my full name. It starts at the character at the beginning of my name (with the index 0) and ends at the character "e" (with the index 3). Note that the "terminating" character is not included in the slice. Here's another slice:

```
>>> 'Robert'[1:2]
'o'
```

The statement above just extracts the "o" from my name. It starts at the character with the index of 1 and ends at the character with the index of 2 (but does not include the "b"). Here's another example:

```
>>> 'Robert'[:4]
'Robe'
```

If I leave out the start position, the slice starts at the start of the collection, as shown above. If I leave out the end position, as shown below, the slice continues to the end of the string.

```
>>> 'Robert'[2:]
'bert'
```

I can also use negative numbers in my slices, in which case the number is used as an index from the end of the collection:

```
>>> 'Robert'[-2:-1]
'r'
```

The above slice starts two positions in from the end of the string, and ends one position in from the end of the string, which means that it just slices off the letter "r."

You can use slicing on any Python collection, including a tuple. Note that slicing doesn't affect the item being sliced, it just returns a "slice" of that item.

The program above uses slicing to get rid of a leading `/` character on the `path` attribute in the `WebServerHandler` object. The statement below would convert "\index.html" to "index.html" by creating a slice that contains everything but the first character of the string. The web server can then use this as the name of the file to be opened and returned.

```
file_path = self.path[1:]
```

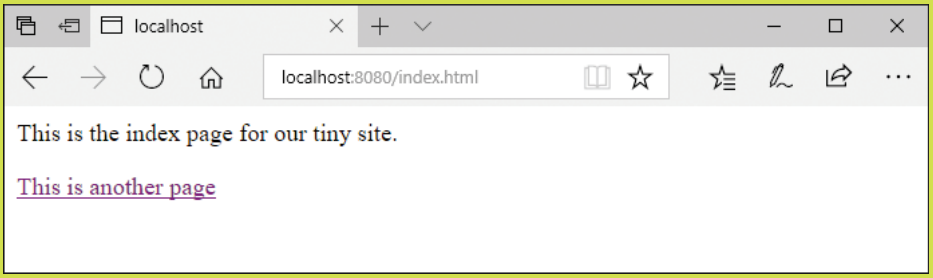
MAKE SOMETHING HAPPEN

Connect to a file server

We can use the web server above to browse a tiny website. Use IDLE to open the example program **EG15-03 Python webpage server** in the folder **EG15-03 Python webpage server** in the sample programs folder for this chapter. The folder also contains two HTML pages that the server will return to the browser. They are called `index.html` and `page.html`.

Start the program and open the following address with your browser:

http:/localhost:8080/index.html



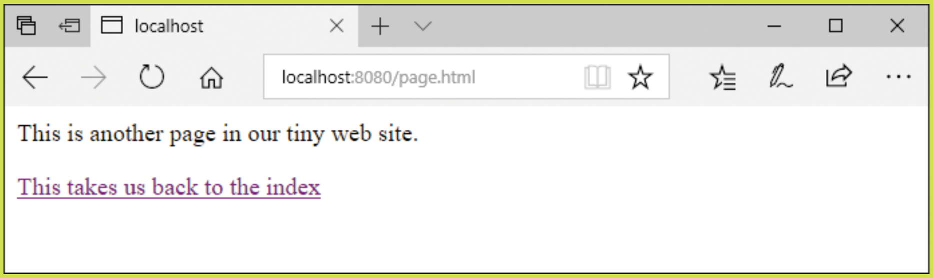
This is the index page for our tiny site.

This is another page

The browser will show the first page of our site.

```
<html>
<body>
<p> This is the index page for our tiny site.</p>
<a href="page.html">This is another page</a>
</body>
</html>
```

This is the HTML file for the index page. It contains the text you can see on the page, along with a link to a second page. When you click the link, the browser will load the next page and display it.



This is another page in our tiny web site.

This takes us back to the index

You can click the link on this page to return to the index.

This is the HTML for the second page of our tiny website:

```
<html>
<body>
<p>This is another page in our tiny website.</p>
<a href="index.html">This takes us back to the index</a> </body>
</html>
```

By now you should have a good understanding of how a web server works and how we can use Python to create them. We could extend our web server above to serve out image files and handle the situation when a browser tries to load a file that doesn't exist, but the Python libraries provide a web server handler called `SimpleHTTPRequestHandler` that can be used to serve out files. Below is a program that uses this handler to create what must be one of the tiniest web servers you can build.

```
# EG15-04 Full Python webpage server

import http.server

host_socket = 8080
host_ip = 'localhost'

host_address = (host_ip, host_socket)

my_server = http.server.HTTPServer(host_address,
                                   http.server.SimpleHTTPRequestHandler)
my_server.serve_forever()
```

Respond to web requests on port 8080

Use the localhost address

Create the host address

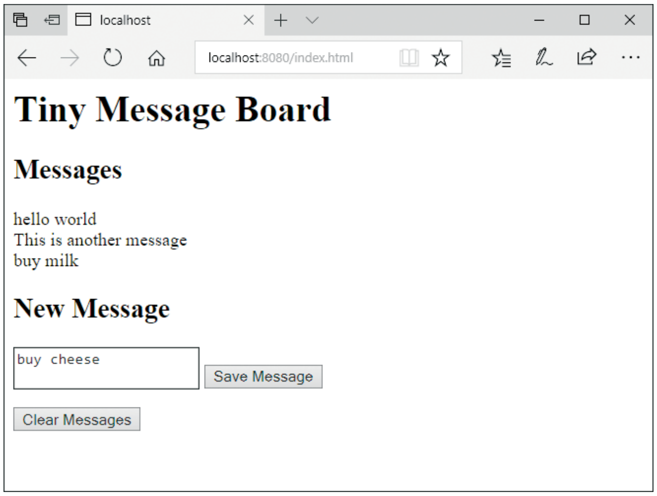
Address for the server

Request handler class

Get information from web users

We can use the Python servers we've created to provide information to users. Next, we'll see how our users can send information back to the Python program. To show how this works, we'll create a Tiny Message program. Anyone can write messages into the program for other readers to see via their browser.

Figure 15-4 shows the user interface for this message board. The user can type in messages and click Save Message to add a message in the list. Also, the user can click Clear Messages to clear all the messages from the board.



Tiny Message Board

Messages

hello world
This is another message
buy milk

New Message

buy cheese

Figure 15-4 Tiny Message Board

MAKE SOMETHING HAPPEN

Use a message board

The best way to learn what this program will do is to try it. Use IDLE to open the example program **EG15-05 Web message board** in the sample programs for this chapter. Start the program and open the following address with your browser:

http:/localhost:8080/index.html

You should see the message board display. Enter a message into the text area underneath the New Message heading and click the **Save Message** button. The page will refresh, and the message will be displayed in the Messages part of the page. If you add a second message, you will see it appear below the first one. If you click **Clear Messages**, all the messages will be removed from the screen. Now that you know what the program does, we can investigate how the program does it.

The HTTP POST request

Hypertext Transport Protocol, or HTTP, describes how the web browser and the web server communicate. It defines a series of browser requests. Until now, the only HTTP request that our server has responded to is the GET request, which is a request to get a webpage. There are several other browser requests, such as the POST request, which allows a browser to post information back to the server.

```
<form method="post">
    <textarea name="message"></textarea>
    <button id="save" type="submit">Save Message</button>
</form>
```

This is the Hypertext Markup Language (HTML) that describes the part of the webpage used to submit a new message. The browser will generate a text input area and a Save button that looks like **Figure 15-5**.

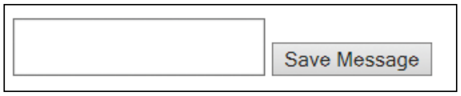


Figure 15-5 Text entry

The HTML tells the browser to perform a POST request when the user clicks the Save Message button. The message sent with the POST request will include the contents of the text area.

We can create a do_POST method in our HTTP request handler class that will deal with a POST request.

```
def do_POST(self):
                                                         Get the length of the reply from the browser
    length = int(self.headers['Content-Length'])
    post_body_bytes = self.rfile.read(length)            Read the reply into a block of bytes
    post_body_text = post_body_bytes.decode()            Convert the block of bytes into a text string
                                                         Convert the text into a dictionary of query items
    query_strings = urllib.parse.parse_qs(post_body_text,
                                          keep_blank_values=True)   Allow blank values
                                                                    in the query string
    message = query_strings['message'][0]                Extract the message from the query string
    messages.append(message)                             Add the message to the existing messages

    self.send_response(200)                              Send the OK response
    self.send_header('Content-type','text/html')         Tell the browser it is getting text back
    self.end_headers()                                   Send the headers

    message_text = self.make_page()                      Call a method to build the webpage to send back

    message_bytes = message_text.encode()                Encode the webpage into a block of bytes

    self.wfile.write(message_bytes)                      Send the bytes to the browser
```

CODE ANALYSIS

POST handler

The POST handler method is quite complicated, although it is not very long. You might have a few questions about how it works. When trying to work out what is happening, remember what the method has been written to do. The user has filled in a form on the webpage and pressed the Save Message button. The browser has assembled a response that includes the text the user entered and sent this back to the server as a POST request.

The POST request has arrived at the server, which has created an instance of the webServerHandler class to deal with this POST and then called the do_POST method in this class to deal with the POST.

Question: How does do_POST read the information sent by the browser?

Answer: The message being posted by the browser can be read via a file connection. The first statement of the do_POST method determines the length of the file by reading the Content-Length item from the message header sent by the browser.

```
length = int(self.headers['Content-Length'])
```

The headers are provided in the webServerHandler as a dictionary (called headers), from which a program can load header items by name. The statement above gets the Content-Length header and then converts it into an integer, which is then used to read in the response:

```
post_body_bytes = self.rfile.read(length)
```

The variable post_body_bytes refers to a block of bytes that contain the response from the browser. Next, the method converts these bytes into a string using the decode method:

```
post_body_text = post_body_bytes.decode()
```

Now we have the text that the browser is sending back to the server. This text is presented by the browser in the form of a *query string*, which is a way that HTTP encodes named items. Items in a query string are given in the form:

```
name=item
```

The name of the item will be the name of the textarea being sent back; in this case, the name is "message," which you can see in the HTML for the page above. Python provides a method that converts query strings into a dictionary, which saves us from having to write our own code to process query strings.

```
query_strings = urllib.parse.parse_qs(post_body_text,
                                      keep_blank_values=True)
```

The parse_qs method creates a dictionary that contains a key for each named item in the query string. It has been given an extra argument to tell it to add blank query string values to the dictionary; we will use this when we add the clear command later.

Now that we have our query strings, we can extract the content of the textarea from the response:

```
message = query_strings['message'][0]
```

The parse_qs method creates a list of items for each key, so the statement above takes the item at the start of this list (which is the text we want) and sets the variable message to this. So, at this point, the variable message contains the text that the webpage user has entered. Now we just need to add the text to the messages that the program is storing.

```
messages.append(message)
```

The variable messages is declared as a global variable, and it is a list that holds each of the entered messages. The make_page method uses the list of messages to create a webpage, which is returned to the browser.

Question: How does the get_POST method generate the webpage that contains the messages the user entered?

Answer: The get_POST method above extracts the message from the POST from the browser and adds it to a list of messages. It then calls the make_page method to create a webpage that includes these messages. Next, we'll investigate this method.

A server must send a webpage in response to a POST request from a browser. Sometimes this webpage contains the message, "Thank you for submitting the information," but our message program will just redraw the webpage with the new message included. The webPageHandler class contains a method, make_page, that does this. The make_page method is called in the do_GET and do_POST methods.

```
def make_page(self):
    all_messages = '<br>'.join(messages)      Create a list of strings separated by the <br>
    page = '''<html>
<body>
<h1>Tiny Message Board</h1>
<h2>Messages</h2>
<p> {0} </p>                                   Placeholder for the list of messages
<h2>New Message</h2>
<form method="post">
    <textarea name="message"></textarea>
    <button id="save" type="submit">Save Message</button>
</form>
<form method="post">
    <button name="clear" type="submit">Clear Messages</button>
</form>
</body>
</html>'''
    return page.format(all_messages)
```

CODE ANALYSIS

Make a webpage from Python code

We've seen that a web server can send the contents of a file back to the browser client. It can also create HTML (HyperText Markup Language) text and send this back. The `make_page` method constructs a page of HTML that contains the input text area as well as the buttons. It also contains all the messages that have been entered. You might have some questions.

Question: How does this method create a list of messages?

> **Answer:** The HTML format needs to be told when to end a line of text displayed on a webpage. The HTML command to do this is `<br>` (which is short for "line-break"). The `make_page` method uses `join` (which we first saw in Chapter 10 when we used it to make a string containing a list of Time Tracker sessions) to create a string containing a list of messages separated by the `<br>` command.

Question: How does this method insert the message list into the HTML that describes the page?

> **Answer:** The method uses Python string formatting. It contains the placeholder `{0}` for a value to be inserted into the page. The string containing the messages, which was created using `join`, is entered as the value.

The final element of the application that we need to implement is the Clear button, which can be used to clear all the elements in the message list. We can add a clear behavior to the `do_POST` method by checking for certain elements in the query string returned by the browser.

```
if 'clear' in query_strings:
    messages.clear()
elif 'message' in query_strings:
    message = query_strings['message'][0]
    messages.append(message)
```

Has the user clicked on Clear?

If Clear clicked, clear the messages

Has the user clicked on Save Message?

If Save Message clicked, save the message

The `in` operator returns `True` if a given dictionary contains a particular key. The code above checks to see if the clear entry is in the dictionary. If you look in the HTML returned by the `make_page` method above, you'll see that the "Clear Messages" button has been given the name `clear`.

Host Python applications on the web

The web applications we've created in this chapter have been hosted on our own computers, and we've used the special port number 8080. In theory, we could host these programs on a machine connected to the Internet and make them available for anyone to use. However, while writing our own client and server applications has given us a good understanding of how the web works, it turns out that there are much better ways to create web applications using Python than by writing them from scratch as we've been doing. Some existing Python frameworks give you a head start in creating web applications. I strongly recommend that you look at Flask (flask.pocoo.org) and Django (djangoproject.com) These frameworks hide a lot of the low-level network access and provide access to databases and components that make it very easy to produce good-looking websites underpinned with Python code.

Once you've created your Python web application, you will need to find a place on the Internet to host it so that it's available to your users. Find out more about how to use Azure to host your applications at https://azure.microsoft.com/en-us/develop/python/.

What you have learned

In this chapter, you discovered how to create Python programs that serve out webpages in response to requests from web browsers. You looked at the HTTP protocol used to manage web requests and saw that there are numerous web requests, including the `GET` request, to load a page. You saw that the `POST` request is used to post data back to a server. You saw that the server response contains a status line, a header element, and a content element. You discovered that Python provides a helper class called `HTTPServer` that can manage a web server and also a class `BaseHTTPRequestHandler` that can be used as the starting point for making programs that respond to web requests.

You created a simple message board application that responds to `GET` and `POST` requests and learned that the basis of web applications is creating programs that respond to these and other requests from the browser.

Here are some points to ponder about Python and web servers.

Is this how webpages work?

The original world wide web worked in the same manner as the programs we created in this chapter. A web server delivered pages of data (which were loaded from files of text) in response to requests from a browser. However, the web today is slightly more complicated. Modern webpages contain program code, usually written in a language called JavaScript. The program code in the webpages interacts with the user and sends requests to programs running on the server. The actual layout and appearance of webpages that the user sees are expressed using "style sheets" that are acted on by the browser when a page is displayed. However, a solid understanding of the concepts described in this chapter and Chapter 14 will serve as a very good starting point for web development.

Can a web server determine what kind of client program is reading the webpage?

Yes. The header sent by the browser contains details of the browser type and even the kind of computer and operating system being used.

Can a web server have a conversation with a user?

You can think of a request from a web browser as a question. The server then provides the answer; however, this is not a conversation. Each question and answer is an individual transaction. When two people are talking, they will establish a context for their conversation. If you and I were talking about a particular type of computer and you asked me, "How fast is it?" I'd remember that we were talking about computers and give the appropriate answer. HTTP does not work on the basis of a conversation like this.

However, websites can use "cookies" to establish a conversation with a user. A cookie is a tiny piece of data that the web server gives the browser. The cookie is stored on the client computer, and at a later time the server can request the cookie so that it can retrieve context information. Cookies are used to implement things like shopping carts, and to allow a website to discover the identity of a user. However, they are also somewhat contentious in that they allow websites to track users in ways that the user might not be aware of.

How can I make my website secure?

The webpages we've created so far have been insecure. The messages exchanged between the browser and the server are sent as plain text. The free program Wireshark, which you can download from www.wireshark.org, can be used to capture and view network messages.

To counter against network eavesdroppers, modern browsers and servers *encrypt* the data they're transferring. Encryption is the process of converting the plain text messages into data that only makes sense when it has been decrypted by the receiver.

Encrypted websites use the protocol name https (rather than http) and they also connect via port 443 rather than port 80. If you want to create a secure, web-based application, you should look at the two previously suggested frameworks, Flask and Django, as they provide support for these kinds of sites. These also provide support for user authentication.