Chapter No.4
"Programming with HTTP for the Internet"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Programming with HTTP for the Internet"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Dr. M. O. Faruque Sarker** is a software architect, and DevOps engineer who's currently working at University College London (UCL), United Kingdom. In recent years, he has been leading a number of Python software development projects, including the implementation of an interactive web-based scientific computing framework using the IPython Notebook service at UCL. He is a specialist and an expert in open source technologies, for example, e-learning and web application platforms, agile software development, and IT service management methodologies such as DSDM Atern and ITIL Service management frameworks.

Dr. Sarker received his PhD in multirobot systems from University of South Wales where he adopted various Python open source projects for integrating the complex software infrastructure of one of the biggest multirobot experiment testbeds in UK. To drive his multirobot fleet, he designed and implemented a decoupled software architecture called hybrid event-driven architecture on D-Bus. Since 1999, he has been deploying Linux and open source software in commercial companies, educational institutions, and multinational consultancies. He was invited to work on the Google Summer of Code 2009/2010 programs for contributing to the BlueZ and Tahoe-LAFS open source projects.

For More Information:

Currently, Dr. Sarker has a research interest in self-organized cloud architecture. In his spare time, he likes to play with his little daughter, Ayesha, and is keen to learn about child-centric educational methods that can empower children with self-confidence by engaging with their environment.

I would like to thank everyone who has contributed to the publication of this book, including the publisher, technical reviewers, editors, friends, and my family members, specially my wife Shahinur Rijuani for her love and support in my work. I also thank the readers who have patiently been waiting for this book and who have given me lots of valuable feedback.

For More Information:
Python Network Programming Cookbook

All praises be to God! I am glad that this book is now published, and I would like to thank everyone behind the publication of this book. This book is an exploratory guide to network programming in Python. It has touched a wide range of networking protocols such as TCP/UDP, HTTP/HTTPS, FTP, SMTP, POP3, IMAP, CGI, and so forth. With the power and interactivity of Python, it brings joy and fun to develop various scripts for performing real-world tasks on network and system administration, web application development, interacting with your local and remote network, low-level network packet capture and analysis, and so on. The primary focus of this book is to give you a hands-on experience on the topics covered. So, this book covers less theory, but it's packed with practical materials.

This book is written with a "devops" mindset where a developer is also more or less in charge of operation, that is, deploying the application and managing various aspects of it, such as remote server administration, monitoring, scaling-up, and optimizing for better performance. This book introduces you to a bunch of open-source, third-party Python libraries, which are awesome to use in various use cases. I use many of these libraries on a daily basis to enjoy automating my devops tasks. For example, I use Fabric for automating software deployment tasks and other libraries for other purposes, such as, searching things on the Internet, screenscraping, or sending an e-mail from a Python script.

I hope you'll enjoy the recipes presented in this book and extend them to make them even more powerful and enjoyable.

What This Book Covers

Chapter 1, Sockets, IPv4, and Simple Client/Server Programming, introduces you to Python’s core networking library with various small tasks and enables you to create your first client/server application.

Chapter 2, Multiplexing Socket I/O for Better Performance, discusses various useful techniques for scaling your client/server applications with default and third-party libraries.

Chapter 3, IPv6, Unix Domain Sockets, and Network Interfaces, focuses more on administering your local machine and looking after your local area network.

Chapter 4, *Programming with HTTP for the Internet*, enables you to create a mini commandline browser with various features such as submitting web forms, handling cookies, managing partial downloads, compressing data, and serving secure contents over HTTPS.

Chapter 5, *E-mail Protocols, FTP, and CGI Programming*, brings you the joy of automating your FTP and e-mail tasks such as manipulating your Gmail account, and reading or sending e-mails from a script or creating a guest book for your web application.

Chapter 6, *Screen-scraping and Other Practical Applications*, introduces you to various third-party Python libraries that do some practical tasks, for example, locating companies on Google maps, grabbing information from Wikipedia, searching code repository on GitHub, or reading news from the BBC.

Chapter 7, *Programming Across Machine Boundaries*, gives you a taste of automating your system administration and deployment tasks over SSH. You can run commands, install packages, or set up new websites remotely from your laptop.

Chapter 8, *Working with Web Services – XML-RPC, SOAP, and REST*, introduces you to various API protocols such as XML-RPC, SOAP, and REST. You can programatically ask any website or web service for information and interact with them. For example, you can search for products on Amazon or Google.

Chapter 9, *Network Monitoring and Security*, introduces you to various techniques for capturing, storing, analyzing, and manipulating network packets. This encourages you to go further to investigate your network security issues using concise Python scripts.

For More Information:  
In this chapter, we will cover the following topics:

- Downloading data from an HTTP server
- Serving HTTP requests from your machine
- Extracting cookie information after visiting a website
- Submitting web forms
- Sending web requests through a proxy server
- Checking whether a web page exists with the HEAD request
- Spoofing Mozilla Firefox in your client code
- Saving bandwidth in web requests with the HTTP compression
- Writing an HTTP fail-over client with resume and partial downloading
- Writing a simple HTTPS server code with Python and OpenSSL

Introduction

This chapter explains Python HTTP networking library functions with a few third-party libraries. For example, the requests library deals with the HTTP requests in a nicer and cleaner way. The OpenSSL library is used in one of the recipes to create a SSL-enabled web server.

Many common HTTP protocol features have been illustrated in a few recipes, for example, the web form submission with POST, manipulating header information, use of compression, and so on.

For More Information:
Downloading data from an HTTP server

You would like to write a simple HTTP client to fetch some data from any web server using the native HTTP protocol. This can be the very first steps towards creating your own HTTP browser.

How to do it...

Let us access www.python.org with our Pythonic minimal browser that uses Python's httplib.

Listing 4.1 explains the following code for a simple HTTP client:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import argparse
import httplib

REMOTE_SERVER_HOST = 'www.python.org'
REMOTE_SERVER_PATH = '/'

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

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

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

        try:
            errcode, errmsg, headers = http.getreply()

        except Exception, e:
```

For More Information:
print "Client failed error code: %s message:%s headers:%s"
%(errcode, errmsg, headers)
else:
    print "Got homepage from %s" %self.host

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

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='HTTP Client Example')
    parser.add_argument('--host', action="store", dest="host", default=REMOTE_SERVER_HOST)
    parser.add_argument('--path', action="store", dest="path", default=REMOTE_SERVER_PATH)
    given_args = parser.parse_args()
    host, path = given_args.host, given_args.path
    client = HTTPClient(host)
    print client.fetch(path)

This recipe will by default fetch a page from www.python.org. You can run this recipe with or without the host and path arguments. If this script is run, it will show the following output:

$ python 4_1_download_data.py --host=www.python.org
Got homepage from www.python.org
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Python Programming Language &ndash; Official Website</title>
    ....
</head>

If you run this recipe with an invalid path, it will show the following server response:

$ python 4_1_download_data.py --host='www.python.org' --path='/not-exist'
Got homepage from www.python.org
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

For More Information:
How it works...

This recipe defines an HTTPClient class that fetches data from the remote host. It is built using Python's native `httplib` library. In the `fetch()` method, it uses the `HTTP()` function and other auxiliary functions to create a dummy HTTP client, such as `putrequest()` or `putheader()`. It first puts the `GET/path` string that is followed by setting up a user agent, which is the name of the current script (`__file__`).

The main request `getreply()` method is put inside a try-except block. The response is retrieved from the `getfile()` method and the stream's content is read.

Serving HTTP requests from your machine

You would like to create your own web server. Your web server should handle client requests and send a simple `hello` message.

How to do it...

Python ships with a very simple web server that can be launched from the command line as follows:

```
$ python -m SimpleHTTPServer 8080
```

This will launch an HTTP web server on port 8080. You can access this web server from your browser by typing `http://localhost:8080`. This will show the contents of the current directory from where you run the preceding command. If there is any web server index file, for example, `index.html`, inside that directory, your browser will show the contents of `index.html`. However, if you like to have full control over your web server, you need to launch your customized HTTP server.

Listing 4.2 gives the following code for the custom HTTP web server:

```
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import argparse
```

For More Information:

import sys
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

DEFAULT_HOST = '127.0.0.1'
DEFAULT_PORT = 8800

class RequestHandler(BaseHTTPRequestHandler):
    """ Custom request handler""

def do_GET(self):
    """ Handler for the GET requests ""
    self.send_response(200)
    self.send_header('Content-type','text/html')
    self.end_headers()
    # Send the message to browser
    self.wfile.write("Hello from server!")

class CustomHTTPServer(HTTPServer):
    "A custom HTTP server"
    def __init__(self, host, port):
        server_address = (host, port)
        HTTPServer.__init__(self, server_address, RequestHandler)

def run_server(port):
    try:
        server = CustomHTTPServer(DEFAULT_HOST, port)
        print "Custom HTTP server started on port: %s" % port
        server.serve_forever()
    except Exception, err:
        print "Error:%s" % err
except KeyboardInterrupt:
    print "Server interrupted and is shutting down..."
    server.socket.close()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Simple HTTP Server
Example')
    parser.add_argument('--port', action="store", dest="port",
    type=int, default=DEFAULT_PORT)
    given_args = parser.parse_args()
    port = given_args.port
    run_server(port)

For More Information:
The following screenshot shows a simple HTTP server:

If you run this web server and access the URL from a browser, this will send the one line text 
Hello from server! to the browser, as follows:

```
$ python 4_2_simple_http_server.py --port=8800
Custom HTTP server started on port: 8800
localhost - - [18/Apr/2013 13:39:33] "GET / HTTP/1.1" 200 -
localhost - - [18/Apr/2013 13:39:33] "GET /favicon.ico HTTP/1.1" 200 -
```

How it works...

In this recipe, we created the `CustomHTTPServer` class inherited from the `HTTPServer` class. In the constructor method, the `CustomHTTPServer` class sets up the server address and port received as a user input. In the constructor, our web server's `RequestHandler` class has been set up. Every time a client is connected, the server handles the request according to this class.

The `RequestHandler` defines the action to handle the client's GET request. It sends an HTTP header (code 200) with a success message Hello from server! using the `write()` method.

For More Information:
Extracting cookie information after visiting a website

Many websites use cookies to store their various information on to your local disk. You would like to see this cookie information and perhaps log in to that website automatically using cookies.

How to do it...

Let us try to pretend to log in to a popular code-sharing website, www.bitbucket.org. We would like to submit the login information on the login page, https://bitbucket.org/account/signin/?next=/.

The following screenshot shows the login page:

![Login Page Screenshot]

So, we note down the form element IDs and decide which fake values should be submitted. We access this page the first time, and the next time, we access the home page to observe what cookies have been set up.

Listing 4.3 explains extracting cookie information as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import cookielib
import urllib
import urllib2

ID_USERNAME = 'id_username'
```

For More Information:
ID_PASSWORD = 'id_password'
USERNAME = 'you@email.com'
PASSWORD = 'mypassword'
LOGIN_URL = 'https://bitbucket.org/account/signin/?next=/'
NORMAL_URL = 'https://bitbucket.org/

def extract_cookie_info():
    """ Fake login to a site with cookie"
    # setup cookie jar
    cj = cookielib.CookieJar()
    login_data = urllib.urlencode({'ID_USERNAME': USERNAME,
                                   'ID_PASSWORD': PASSWORD})
    # create url opener
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    resp = opener.open(LOGIN_URL, login_data)

    # send login info
    for cookie in cj:
        print "----First time cookie: %s --> %s" % (cookie.name,
                                                  cookie.value)
        print "Headers: %s" % resp.headers

    # now access without any login info
    resp = opener.open(NORMAL_URL)
    for cookie in cj:
        print "++++Second time cookie: %s --> %s" % (cookie.name,
                                                   cookie.value)
        print "Headers: %s" % resp.headers

if __name__ == '__main__':
    extract_cookie_info()

Running this recipe results in the following output:

$ python 4_3_extract_cookie_information.py
----First time cookie: bb_session --> aed58dde1228571bf60466581790566d
Headers: Server: nginx/1.2.4
Date: Sun, 05 May 2013 15:13:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 21167
Connection: close
X-Served-By: bitbucket04
Content-Language: en
X-Static-Version: c67fb01467cf

For More Information:
How it works...

We have used Python’s cookielib and set up a cookie jar, cj. The login data has been encoded using urllib.urlencode. urllib2 has a build_opener() method, which takes the predefined cookie jar with an instance of HTTPCookieProcessor() and returns a URL opener. We call this opener twice: once for the login page and once for the home page of the website. It seems that only one cookie, bb_session, was set with the set-cookie directive present in the page header. More information about cookielib can be found on the official Python documentation site at http://docs.python.org/2/library/cookielib.html.
Submitting web forms

During web browsing, we submit web forms many times in a day. Now, you would like do that using the Python code.

Getting ready

This recipe uses a third-party Python module called requests. You can install the compatible version of this module by following the instructions from http://docs.python-requests.org/en/latest/user/install/. For example, you can use pip to install requests from the command line as follows:

$ pip install requests

How to do it...

Let us submit some fake data to register with www.twitter.com. Each form submission has two methods: GET and POST. The less sensitive data, for example, search queries, are usually submitted by GET and the more sensitive data is sent via the POST method. Let us try submitting data with both of them.

Listing 4.4 explains the submit web forms, as follows:

```
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import requests
import urllib
import urllib2

ID_USERNAME = 'signup-user-name'
ID_EMAIL = 'signup-user-email'
ID_PASSWORD = 'signup-user-password'
USERNAME = 'username'
EMAIL = 'you@email.com'
PASSWORD = 'yourpassword'
SIGNUP_URL = 'https://twitter.com/account/create'

def submit_form():
    """Submit a form""
    payload = {ID_USERNAME : USERNAME,
```
ID_EMAIL    : EMAIL,
ID_PASSWORD : PASSWORD,

# make a get request
resp = requests.get(SIGNUP_URL)
print "Response to GET request: %s" %resp.content

# send POST request
resp = requests.post(SIGNUP_URL, payload)
print "Headers from a POST request response: %s" %resp.headers
# print "HTML Response: %s" %resp.read()

if __name__ == '__main__':
    submit_form()

If you run this script, you will see the following output:

$ python 4_4_submit_web_form.py
Response to GET request: <?xml version="1.0" encoding="UTF-8"?>
<hash>
    <error>This method requires a POST.</error>
    <request>/account/create</request>
</hash>

Headers from a POST request response: { 'status': '200 OK', 'content-length': '21064', 'set-cookie': '_twitter_sess=BAh7CD--
d2865d4d0d1365eeb21755559dc5e6b99f64ea39ff; domain=.twitter.com;
path=/; HttpOnly', 'expires': 'Tue, 31 Mar 1981 05:00:00 GMT',
'vary': 'Accept-Encoding', 'last-modified': 'Sun, 05 May 2013
15:59:27 GMT', 'pragma': 'no-cache', 'date': 'Sun, 05 May 2013
15:59:27 GMT', 'x-xss-protection': '1; mode=block', 'x-transaction':
'a4b425eda23b5312', 'content-encoding': 'gzip', 'strict-transport-
security': 'max-age=631138519', 'server': 'tfe', 'x-mid':
f7de9a3f3d11310427116ad90bf3e8c95e868', 'x-runtime': '0.09969',
etag': '7af6f92a7f7b4d37a6454caa6094071d', 'cache-control': 'no-
cache, no-store, must-revalidate, pre-check=0, post-check=0', 'x-
frame-options': 'SAMEORIGIN', 'content-type': 'text/html;
charset=utf-8'}
This recipe uses a third-party module, requests. It has convenient wrapper methods, `get()` and `post()`, that do the URL encoding of data and submit forms properly.

In this recipe, we created a data payload with a username, password, and e-mail for creating the Twitter account. When we first submit the form with the `GET` method, the Twitter website returns an error saying that the page only supports `POST`. After we submit the data with `POST`, the page processes it. We can confirm this from the header data.

### Sending web requests through a proxy server

You would like to browse web pages through a proxy. If you have configured your browser with a proxy server and that works, you can try this recipe. Otherwise, you can use any of the public proxy servers available on the Internet.

### Getting ready

You need to have access to a proxy server. You can find a free proxy server by searching on Google or on any other search engine. Here, for the sake of demonstration, we have used 165.24.10.8.

### How to do it...

Let us send our HTTP request through a public domain proxy server.

Listing 4.5 explains proxying web requests across a proxy server as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import urllib

URL = 'https://www.github.com'
PROXY_ADDRESS = "165.24.10.8:8080"

if __name__ == '__main__':
    resp = urllib.urlopen(URL, proxies = {"http" : PROXY_ADDRESS})
    print "Proxy server returns response headers: %s "
    %resp.headers
```

For More Information:

If you run this script, it will show the following output:

```bash
$ python 4_5_proxy_web_request.py
Proxy server returns response headers: Server: GitHub.com
Date: Sun, 05 May 2013 16:16:04 GMT
Content-Type: text/html; charset=utf-8
Connection: close
Status: 200 OK
Cache-Control: private, max-age=0, must-revalidate
Strict-Transport-Security: max-age=2592000
X-Frame-Options: deny
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Thu, 05-May-2033 16:16:04 GMT; HttpOnly
Set-Cookie: _gh_sess=BAh7...; path=/; expires=Sun, 01-Jan-2023 00:00:00 GMT; secure; HttpOnly
X-Runtime: 8
ETag: "66fcc37865eb05c19b2d15fbb44cd7a9"
Content-Length: 10643
Vary: Accept-Encoding
```

**How it works...**

This is a short recipe where we access the social code-sharing site, www.github.com, with a public proxy server found on Google search. The proxy address argument has been passed to the `urlopen()` method of `urllib`. We print the HTTP header of response to show that the proxy settings work here.

**Checking whether a web page exists with the HEAD request**

You would like to check the existence of a web page without downloading the HTML content. This means that we need to send a `GET` request with a browser client. According to Wikipedia, the HEAD request asks for the response identical to the one that would correspond to a `GET` request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

**How to do it...**

We would like to send a HEAD request to www.python.org. This will not download the content of the homepage, rather it checks whether the server returns one of the valid responses, for example, OK, FOUND, MOVED PERMANENTLY, and so on.

For More Information:
Listing 4.6 explains checking a web page with the HEAD request as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.
import argparse
import httplib
import urlparse
import re
import urllib

DEFAULT_URL = 'http://www.python.org'
HTTP_GOOD_CODES = [httplib.OK, httplib.FOUND, httplib.MOVED_PERMANENTLY]

def get_server_status_code(url):
    """Download just the header of a URL and return the server's status code.
    """
    host, path = urlparse.urlparse(url)[1:3]
    try:
        conn = httplib.HTTPConnection(host)
        conn.request('HEAD', path)
        return conn.getresponse().status
    except StandardError:
        return None

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Example HEAD Request')
    parser.add_argument('--url', action="store", dest="url",
    default=DEFAULT_URL)
    given_args = parser.parse_args()
    url = given_args.url
    if get_server_status_code(url) in HTTP_GOOD_CODES:
        print "Server: %s status is OK: " %url
    else:
        print "Server: %s status is NOT OK!" %url
```

Running this script shows the success or error if the page is found by the HEAD request as follows:

```
$ python 4_6_checking_webpage_with_HEAD_request.py
Server: http://www.python.org status is OK!
$ python 4_6_checking_webpage_with_HEAD_request.py --url=http://www.zytho.org
Server: http://www.zytho.org status is NOT OK!
```
**How it works...**

We used the `HTTPConnection()` method of `httplib`, which can make a `HEAD` request to a server. We can specify the path if necessary. Here, the `HTTPConnection()` method checks the home page or path of `www.python.org`. However, if the URL is not correct, it can't find the return response inside the accepted list of return codes.

**Spoofing Mozilla Firefox in your client code**

From your Python code, you would like to pretend to the web server that you are browsing from Mozilla Firefox.

**How to do it...**

You can send the custom user-agent values in the HTTP request header.

Listing 4.7 explains spoofing Mozilla Firefox in your client code as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.

import urllib2

BROWSER = 'Mozilla/5.0 (Windows NT 5.1; rv:20.0) Gecko/20100101 Firefox/20.0'
URL = 'http://www.python.org'

def spoof_firefox():
    opener = urllib2.build_opener()
    opener.addheaders = [('User-agent', BROWSER)]
    result = opener.open(URL)
    print "Response headers:"
    for header in result.headers.headers:
        print "\t", header

if __name__ == '__main__':
    spoof_firefox()
```

If you run this script, you will see the following output:

```
$ python 4_7_spoof_mozilla_firefox_in_client_code.py
Response headers:
  Date: Sun, 05 May 2013 16:56:36 GMT
  Server: Apache/2.2.16 (Debian)
```

For More Information:

How it works...

We used the `build_opener()` method of `urllib2` to create our custom browser whose user-agent string has been set up as `Mozilla/5.0 (Windows NT 5.1; rv:20.0) Gecko/20100101 Firefox/20.0`.

Saving bandwidth in web requests with the HTTP compression

You would like to give your web server users better performance in downloading web pages. By compressing HTTP data, you can speed up the serving of web contents.

How to do it...

Let us create a web server that serves contents after compressing it to the `gzip` format.

Listing 4.8 explains the HTTP compression as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.
import argparse
import string
import os
import sys
import gzip
import cStringIO
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

DEFAULT_HOST = '127.0.0.1'
DEFAULT_PORT = 8800
HTML_CONTENT = """<html><body><h1>Compressed Hello World!</h1></body></html>""
```

For More Information:

class RequestHandler(BaseHTTPRequestHandler):
    """ Custom request handler""

def do_GET(self):
    """ Handler for the GET requests ""
    self.send_response(200)
    self.send_header('Content-type','text/html')
    self.send_header('Content-Encoding','gzip')

    zbuf = self.compress_buffer(HTML_CONTENT)
    sys.stdout.write("Content-Encoding: gzip\r\n")
    self.send_header('Content-Length',len(zbuf))
    self.end_headers()

    # Send the message to browser
    zbuf = self.compress_buffer(HTML_CONTENT)
    sys.stdout.write("Content-Encoding: gzip\r\n")
    sys.stdout.write("Content-Length: %d\r\n" % (len(zbuf)))
    sys.stdout.write("\r\n")
    self.wfile.write(zbuf)
    return

def compress_buffer(self, buf):
    zbuf = cStringIO.StringIO()
    zfile = gzip.GzipFile(mode = 'wb',  fileobj = zbuf,
                          compresslevel = 6)
    zfile.write(buf)
    zfile.close()
    return zbuf.getvalue()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Simple HTTP Server Example')
    parser.add_argument('--port', action="store", dest="port",
                        type=int, default=DEFAULT_PORT)
    given_args = parser.parse_args()
    port = given_args.port
    server_address = (DEFAULT_HOST, port)
    server = HTTPServer(server_address, RequestHandler)
    server.serve_forever()
You can run this script and see the Compressed Hello World! text (as a result of the HTTP compression) on your browser screen when accessing http://localhost:8800 as follows:

```
$ python 4_8_http_compression.py
localhost - - [22/Feb/2014 12:01:26] "GET / HTTP/1.1" 200 -
Content-Encoding: gzip
Content-Length: 71

localhost - - [22/Feb/2014 12:01:26] "GET /favicon.ico HTTP/1.1" 200 -
Content-Encoding: gzip
Content-Length: 71
```

The following screenshot illustrates serving compressed content by a web server:

Compressed Hello World!

**How it works...**

We created a web server by instantiating the `HTTPServer` class from the `BaseHTTPServer` module. We attached a custom request handler to this server instance, which compresses every client response using a `compress_buffer()` method. A predefined HTML content has been supplied to the clients.
Writing an HTTP fail-over client with resume and partial downloading

You would like to create a fail-over client that will resume downloading a file if it fails for any reason in the first instance.

How to do it...

Let us download the Python 2.7 code from www.python.org. A resume_download() file will resume any unfinished download of that file.

Listing 4.9 explains resume downloading as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7. It may run on any other
# version with/without modifications.

import urllib, os
TARGET_URL = 'http://python.org/ftp/python/2.7.4/
TARGET_FILE = 'Python-2.7.4.tgz'

class CustomURLOpener(urllib.FancyURLopener):
    """Override FancyURLopener to skip error 206 (when a
    partial file is being sent)
    """
    def http_error_206(self, url, fp, errcode, errmsg, headers, data=None):
        pass

    def resume_download():
        file_exists = False
        CustomURLClass = CustomURLOpener()
        if os.path.exists(TARGET_FILE):
            out_file = open(TARGET_FILE, "ab")
            file_exists = os.path.getsize(TARGET_FILE)
            # If the file exists, then only download the unfinished part
            CustomURLClass.addheader("Download range","bytes=%s- " %
            (file_exists))
        else:
            out_file = open(TARGET_FILE, "wb")

        web_page = CustomURLClass.open(TARGET_URL + TARGET_FILE)

        # If the file exists, but we already have the whole thing, don't
```
download again
    if int(web_page.headers['Content-Length']) == file_exists:
        loop = 0
        print "File already downloaded!"

    byte_count = 0
    while True:
        data = web_page.read(8192)
        if not data:
            break
        out_file.write(data)
        byte_count = byte_count + len(data)

    web_page.close()
    out_file.close()

    for k,v in web_page.headers.items():
        print k, "=".v
    print "File copied", byte_count, "bytes from", web_page.url

if __name__ == '__main__':
    resume_download()

Running this script will result in the following output:

$ python 4_9_http_fail_over_client.py
content-length = 14489063
content-encoding = x-gzip
accept-ranges = bytes
connection = close
server = Apache/2.2.16 (Debian)
last-modified = Sat, 06 Apr 2013 14:16:10 GMT
content-range = bytes 0-14489062/14489063
etag = "1748016-dd15e7-4d9b1d8685e80"
date = Tue, 07 May 2013 12:51:31 GMT
content-type = application/x-tar
File copied 14489063 bytes from http://python.org/ftp/python/2.7.4/
Python-2.7.4.tgz

How it works...

In this recipe, we created a custom URL opener class inheriting from the FancyURLopener method of urllib, but http_error_206() is overridden where partial content is downloaded. So, our method checks the existence of the target file and if it is not present, it tries to download with the custom URL opener class.
Writing a simple HTTPS server code with Python and OpenSSL

You need a secure web server code written in Python. You already have your SSL keys and certificate files ready with you.

Getting ready

You need to install the third-party Python module, pyOpenSSL. This can be grabbed from PyPI (https://pypi.python.org/pypi/pyOpenSSL). Both on Windows and Linux hosts, you may need to install some additional packages, which are documented at http://pythonhosted.org/pyOpenSSL/.

How to do it...

After placing a certificate file on the current working folder, we can create a web server that makes use of this certificate to serve encrypted content to the clients.

Listing 4.10 explains the code for a secure HTTP server as follows:

```python
#!/usr/bin/env python
# Python Network Programming Cookbook -- Chapter - 4
# This program is optimized for Python 2.7.
# It may run on any other version with/without modifications.
# Requires pyOpenSSL and SSL packages installed

import socket, os
from SocketServer import BaseServer
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
from OpenSSL import SSL

class SecureHTTPServer(HTTPServer):
    def __init__(self, server_address, HandlerClass):
        BaseServer.__init__(self, server_address, HandlerClass)
        ctx = SSL.Context(SSL.SSLv23_METHOD)
        fpem = 'server.pem' # location of the server private key and
        the server certificate
        ctx.use_privatekey_file (fpem)
        ctx.use_certificate_file(fpem)
        self.socket = SSL.Connection(ctx,
        socket.socket(self.address_family, self.socket_type))
        self.server_bind()
        self.server_activate()
```

For More Information:

class SecureHTTPRequestHandler(SimpleHTTPRequestHandler):
    def setup(self):
        self.connection = self.request
        self.rfile = socket._fileobject(self.request, "rb",
        self.rbufsize)
        self.wfile = socket._fileobject(self.request, "wb",
        self.wbufsize)

    def run_server(HandlerClass = SecureHTTPRequestHandler,
    ServerClass = SecureHTTPServer):
        server_address = ('', 4443) # port needs to be accessible by
        user
        server = ServerClass(server_address, HandlerClass)
        running_address = server.socket.getsockname()
        print "Serving HTTPS Server on %s:%s ..."
        %(running_address[0], running_address[1])
        server.serve_forever()

        if __name__ == '__main__':
            run_server()

If you run this script, it will result in the following output:

$ python 4_10_https_server.py
Serving HTTPS Server on 0.0.0.0:4443 ...

How it works...

If you notice the previous recipes that create the web server, there is not much difference
in terms of the basic procedure. The main difference is in applying the SSL Context() method with the SSLv23_METHOD argument. We have created the SSL socket with the Python OpenSSL third-party module's Connection() class. This class takes this context object along with the address family and socket type.

The server's certificate file is kept in the current directory, and this has been applied with the context object. Finally, the server has been activated with the server_activate() method.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.