```
"""HTTP server base class.

Note: the class in this module doesn't implement any HTTP request; see
SimpleHTTPServer for simple implementations of GET, HEAD and POST
(including CGI scripts).  It does, however, optionally implement HTTP/1.1
persistent connections, as of version 0.3.

Contents:

- BaseHTTPRequestHandler: HTTP request handler base class
- test: test function

XXX To do:

- log requests even later (to capture byte count)
- log user-agent header and other interesting goodies
- send error log to separate file
"""



# See also:
#
# HTTP Working Group                                        T. Berners-Lee
# INTERNET-DRAFT                                            R. T. Fielding
# <draft-ietf-http-v10-spec-00.txt>                      H. Frystyk Nielsen
# Expires September 8, 1995                                    March 8, 1995
#
# URL: http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v10-spec-00.txt
#
# and
#
# Network Working Group                                        R. Fielding
# Request for Comments: 2616                                       et al
# Obsoletes: 2068                                            June 1999
# Category: Standards Track
#
# URL: http://www.faqs.org/rfcs/rfc2616.html

# Log files
# ---------
#
# Here's a quote from the NCSA httpd docs about log file format.
#
# | The logfile format is as follows. Each line consists of:
# |
# | host rfc931 authuser [DD/Mon/YYYY:hh:mm:ss] "request" ddd bbbb
# |
# |        host: Either the DNS name or the IP number of the remote client
# |        rfc931: Any information returned by identd for this person,
# |                - otherwise.
# |        authuser: If user sent a userid for authentication, the user name,
# |                  - otherwise.
# |        DD: Day
# |        Mon: Month (calendar name)
# |        YYYY: Year
# |        hh: hour (24-hour format, the machine's timezone)
# |        mm: minutes
```

```
58 # |         ss: seconds
59 # |         request: The first line of the HTTP request as sent by the
       client.
60 # |         ddd: the status code returned by the server, - if not available.
61 # |         bbbb: the total number of bytes sent,
62 # |               *not including the HTTP/1.0 header*, - if not available
63 # |
64 # | You can determine the name of the file accessed through request.
65 #
66 # (Actually, the latter is only true if you know the server configuration
67 # at the time the request was made!)
68
69  __version__ = "0.3"
70
71  __all__ = ["HTTPServer", "BaseHTTPRequestHandler"]
72
73  import sys
74  import time
75  import socket # For gethostbyaddr()
76  from warnings import filterwarnings, catch_warnings
77  with catch_warnings():
78      if sys.py3kwarning:
79          filterwarnings("ignore", ".*mimetools has been removed",
80                         DeprecationWarning)
81      import mimetools
82  import SocketServer
83
84  # Default error message template
85  DEFAULT_ERROR_MESSAGE = """\
86  <head>
87  <title>Error response</title>
88  </head>
89  <body>
90  <h1>Error response</h1>
91  <p>Error code %(code)d.
92  <p>Message: %(message)s.
93  <p>Error code explanation: %(code)s = %(explain)s.
94  </body>
95  """
96
97  DEFAULT_ERROR_CONTENT_TYPE = "text/html"
98
99  def _quote_html(html):
100     return html.replace("&", "&amp;").replace("<", "&lt;").replace(">", "&
          gt;")
101
102 class HTTPServer(SocketServer.TCPServer):
103
104     allow_reuse_address = 1    # Seems to make sense in testing environment
105
106     def server_bind(self):
107         """Override server_bind to store the server name."""
108         SocketServer.TCPServer.server_bind(self)
109         host, port = self.socket.getsockname()[:2]
110         self.server_name = socket.getfqdn(host)
111         self.server_port = port
112
113
```

```
class BaseHTTPRequestHandler(SocketServer.StreamRequestHandler):

    """HTTP request handler base class.

    The following explanation of HTTP serves to guide you through the
    code as well as to expose any misunderstandings I may have about
    HTTP (so you don't need to read the code to figure out I'm wrong
    :-).

    HTTP (HyperText Transfer Protocol) is an extensible protocol on
    top of a reliable stream transport (e.g. TCP/IP).  The protocol
    recognizes three parts to a request:

    1. One line identifying the request type and path
    2. An optional set of RFC-822-style headers
    3. An optional data part

    The headers and data are separated by a blank line.

    The first line of the request has the form

    <command> <path> <version>

    where <command> is a (case-sensitive) keyword such as GET or POST,
    <path> is a string containing path information for the request,
    and <version> should be the string "HTTP/1.0" or "HTTP/1.1".
    <path> is encoded using the URL encoding scheme (using %xx to signify
    the ASCII character with hex code xx).

    The specification specifies that lines are separated by CRLF but
    for compatibility with the widest range of clients recommends
    servers also handle LF.  Similarly, whitespace in the request line
    is treated sensibly (allowing multiple spaces between components
    and allowing trailing whitespace).

    Similarly, for output, lines ought to be separated by CRLF pairs
    but most clients grok LF characters just fine.

    If the first line of the request has the form

    <command> <path>

    (i.e. <version> is left out) then this is assumed to be an HTTP
    0.9 request; this form has no optional headers and data part and
    the reply consists of just the data.

    The reply form of the HTTP 1.x protocol again has three parts:

    1. One line giving the response code
    2. An optional set of RFC-822-style headers
    3. The data

    Again, the headers and data are separated by a blank line.

    The response code line has the form

    <version> <responsecode> <responsestring>
```

```
172     where <version> is the protocol version ("HTTP/1.0" or "HTTP/1.1"),
173     <responsecode> is a 3-digit response code indicating success or
174     failure of the request, and <responsestring> is an optional
175     human-readable string explaining what the response code means.
176
177     This server parses the request and the headers, and then calls a
178     function specific to the request type (<command>). Specifically,
179     a request SPAM will be handled by a method do_SPAM(). If no
180     such method exists the server sends an error response to the
181     client. If it exists, it is called with no arguments:
182
183     do_SPAM()
184
185     Note that the request name is case sensitive (i.e. SPAM and spam
186     are different requests).
187
188     The various request details are stored in instance variables:
189
190     - client_address is the client IP address in the form (host,
191     port);
192
193     - command, path and version are the broken-down request line;
194
195     - headers is an instance of mimetools.Message (or a derived
196     class) containing the header information;
197
198     - rfile is a file object open for reading positioned at the
199     start of the optional input data part;
200
201     - wfile is a file object open for writing.
202
203     IT IS IMPORTANT TO ADHERE TO THE PROTOCOL FOR WRITING!
204
205     The first thing to be written must be the response line. Then
206     follow 0 or more header lines, then a blank line, and then the
207     actual data (if any). The meaning of the header lines depends on
208     the command executed by the server; in most cases, when data is
209     returned, there should be at least one header line of the form
210
211     Content-type: <type>/<subtype>
212
213     where <type> and <subtype> should be registered MIME types,
214     e.g. "text/html" or "text/plain".
215
216     """
217
218     # The Python system version, truncated to its first component.
219     sys_version = "Python/" + sys.version.split()[0]
220
221     # The server software version. You may want to override this.
222     # The format is multiple whitespace-separated strings,
223     # where each string is of the form name[/version].
224     server_version = "BaseHTTP/" + __version__
225
226     # The default request version. This only affects responses up until
227     # the point where the request line is parsed, so it mainly decides what
228     # the client gets back when sending a malformed request line.
229     # Most web servers default to HTTP 0.9, i.e. don't send a status line.
```

```
default_request_version = "HTTP/0.9"
def parse_request(self):
    """Parse a request (internal).
    The request should be stored in self.raw_requestline; the results
    are in self.command, self.path, self.request_version and
    self.headers.
    Return True for success, False for failure; on failure, an
    error is sent back.
    """
    self.command = None  # set in case of error on the first line
    self.request_version = version = self.default_request_version
    self.close_connection = 1
    requestline = self.raw_requestline
    requestline = requestline.rstrip('\r\n')
    self.requestline = requestline
    words = requestline.split()
    if len(words) == 3:
        command, path, version = words
        if version[:5] != 'HTTP/':
            self.send_error(400, "Bad request version (%r)" % version)
            return False
        try:
            base_version_number = version.split('/', 1)[1]
            version_number = base_version_number.split(".")
            # RFC 2145 section 3.1 says there can be only one "." and
            #   - major and minor numbers MUST be treated as
            #      separate integers;
            #   - HTTP/2.4 is a lower version than HTTP/2.13, which in
            #      turn is lower than HTTP/12.3;
            #   - Leading zeros MUST be ignored by recipients.
            if len(version_number) != 2:
                raise ValueError
            version_number = int(version_number[0]), int(version_number
                [1])
        except (ValueError, IndexError):
            self.send_error(400, "Bad request version (%r)" % version)
            return False
        if version_number >= (1, 1) and self.protocol_version >= "HTTP
            /1.1":
            self.close_connection = 0
        if version_number >= (2, 0):
            self.send_error(505,
                      "Invalid HTTP Version (%s)" % base_version_number
                        )
            return False
    elif len(words) == 2:
        command, path = words
        self.close_connection = 1
        if command != 'GET':
            self.send_error(400,
                            "Bad HTTP/0.9 request type (%r)" % command)
            return False
    elif not words:
        return False
```

```
285         else:
286             self.send_error(400, "Bad request syntax (%r)" % requestline)
287             return False
288         self.command, self.path, self.request_version = command, path,
                version
289
290         # Examine the headers and look for a Connection directive
291         self.headers = self.MessageClass(self.rfile, 0)
292
293         conntype = self.headers.get('Connection', "")
294         if conntype.lower() == 'close':
295             self.close_connection = 1
296         elif (conntype.lower() == 'keep-alive' and
297               self.protocol_version >= "HTTP/1.1"):
298             self.close_connection = 0
299         return True
300
301     def handle_one_request(self):
302         """Handle a single HTTP request.
303
304         You normally don't need to override this method; see the class
305         __doc__ string for information on how to handle specific HTTP
306         commands such as GET and POST.
307
308         """
309         try:
310             self.raw_requestline = self.rfile.readline(65537)
311             if len(self.raw_requestline) > 65536:
312                 self.requestline = ''
313                 self.request_version = ''
314                 self.command = ''
315                 self.send_error(414)
316                 return
317             if not self.raw_requestline:
318                 self.close_connection = 1
319                 return
320             if not self.parse_request():
321                 # An error code has been sent, just exit
322                 return
323             mname = 'do_' + self.command
324             if not hasattr(self, mname):
325                 self.send_error(501, "Unsupported method (%r)" % self.
                        command)
326                 return
327             method = getattr(self, mname)
328             method()
329             self.wfile.flush() #actually send the response if not already
                    done.
330         except socket.timeout, e:
331             #a read or a write timed out.  Discard this connection
332             self.log_error("Request timed out: %r", e)
333             self.close_connection = 1
334             return
335
336     def handle(self):
337         """Handle multiple requests if necessary."""
338         self.close_connection = 1
339
```

```
        self.handle_one_request()                                          340
        while not self.close_connection:                                   341
            self.handle_one_request()                                      342
                                                                           343
    def send_error(self, code, message=None):                              344
        """Send and log an error reply.                                    345
                                                                           346
        Arguments are the error code, and a detailed message.              347
        The detailed message defaults to the short entry matching the      348
        response code.                                                     349
                                                                           350
        This sends an error response (so it must be called before any      351
        output has been generated), logs the error, and finally sends      352
        a piece of HTML explaining the error to the user.                  353
                                                                           354
        """                                                                355
                                                                           356
        try:                                                               357
            short, long = self.responses[code]                             358
        except KeyError:                                                   359
            short, long = '???', '???'                                     360
        if message is None:                                                361
            message = short                                                362
        explain = long                                                     363
        self.log_error("code %d, message %s", code, message)               364
        # using _quote_html to prevent Cross Site Scripting attacks (see   365
            bug #1100201)
        content = (self.error_message_format %                             366
                   {'code': code, 'message': _quote_html(message), 'explain 367
                       ': explain})
        self.send_response(code, message)                                  368
        self.send_header("Content-Type", self.error_content_type)          369
        self.send_header('Connection', 'close')                            370
        self.end_headers()                                                 371
        if self.command != 'HEAD' and code >= 200 and code not in (204,    372
            304):
            self.wfile.write(content)                                      373
                                                                           374
    error_message_format = DEFAULT_ERROR_MESSAGE                           375
    error_content_type = DEFAULT_ERROR_CONTENT_TYPE                        376
                                                                           377
    def send_response(self, code, message=None):                           378
        """Send the response header and log the response code.             379
                                                                           380
        Also send two standard headers with the server software            381
        version and the current date.                                      382
                                                                           383
        """                                                                384
        self.log_request(code)                                             385
        if message is None:                                                386
            if code in self.responses:                                     387
                message = self.responses[code][0]                          388
            else:                                                          389
                message = ''                                               390
        if self.request_version != 'HTTP/0.9':                             391
            self.wfile.write("%s %d %s\r\n" %                              392
                             (self.protocol_version, code, message))       393
            # print (self.protocol_version, code, message)                 394
```

```
395         self.send_header('Server', self.version_string())
396         self.send_header('Date', self.date_time_string())
397
398     def send_header(self, keyword, value):
399         """Send a MIME header."""
400         if self.request_version != 'HTTP/0.9':
401             self.wfile.write("%s: %s\r\n" % (keyword, value))
402
403         if keyword.lower() == 'connection':
404             if value.lower() == 'close':
405                 self.close_connection = 1
406             elif value.lower() == 'keep-alive':
407                 self.close_connection = 0
408
409     def end_headers(self):
410         """Send the blank line ending the MIME headers."""
411         if self.request_version != 'HTTP/0.9':
412             self.wfile.write("\r\n")
413
414     def log_request(self, code='-', size='-'):
415         """Log an accepted request.
416
417         This is called by send_response().
418
419         """
420
421         self.log_message('"%s" %s %s',
422                          self.requestline, str(code), str(size))
423
424     def log_error(self, format, *args):
425         """Log an error.
426
427         This is called when a request cannot be fulfilled.  By
428         default it passes the message on to log_message().
429
430         Arguments are the same as for log_message().
431
432         XXX This should go to the separate error log.
433
434         """
435
436         self.log_message(format, *args)
437
438     def log_message(self, format, *args):
439         """Log an arbitrary message.
440
441         This is used by all other logging functions.  Override
442         it if you have specific logging wishes.
443
444         The first argument, FORMAT, is a format string for the
445         message to be logged.  If the format string contains
446         any % escapes requiring parameters, they should be
447         specified as subsequent arguments (it's just like
448         printf!).
449
450         The client ip address and current date/time are prefixed to every
451         message.
452
```

```
    """                                                                    453
                                                                           454
    sys.stderr.write("%s - - [%s] %s\n" %                                  455
                     (self.client_address[0],                              456
                      self.log_date_time_string(),                         457
                      format%args))                                        458
                                                                           459
def version_string(self):                                                  460
    """Return the server software version string."""                       461
    return self.server_version + ' ' + self.sys_version                    462
                                                                           463
def date_time_string(self, timestamp=None):                                464
    """Return the current date and time formatted for a message header     465
    .."""
    if timestamp is None:                                                  466
        timestamp = time.time()                                            467
    year, month, day, hh, mm, ss, wd, y, z = time.gmtime(timestamp)        468
    s = "%s, %02d %3s %4d %02d:%02d:%02d GMT" % (                          469
            self.weekdayname[wd],                                          470
            day, self.monthname[month], year,                              471
            hh, mm, ss)                                                    472
    return s                                                               473
                                                                           474
def log_date_time_string(self):                                            475
    """Return the current time formatted for logging."""                   476
    now = time.time()                                                      477
    year, month, day, hh, mm, ss, x, y, z = time.localtime(now)            478
    s = "%02d/%3s/%04d %02d:%02d:%02d" % (                                 479
            day, self.monthname[month], year, hh, mm, ss)                  480
    return s                                                               481
                                                                           482
weekdayname = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']            483
                                                                           484
monthname = [None,                                                         485
             'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',                     486
             'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']                     487
                                                                           488
def address_string(self):                                                  489
    """Return the client address formatted for logging.                    490
                                                                           491
    This version looks up the full hostname using gethostbyaddr(),         492
    and tries to find a name that contains at least one dot.               493
                                                                           494
    """                                                                    495
                                                                           496
    host, port = self.client_address[:2]                                   497
    return socket.getfqdn(host)                                            498
                                                                           499
# Essentially static class variables                                       500
                                                                           501
# The version of the HTTP protocol we support.                             502
# Set this to HTTP/1.1 to enable automatic keepalive                       503
protocol_version = "HTTP/1.0"                                              504
                                                                           505
# The Message-like class used to parse headers                             506
MessageClass = mimetools.Message                                           507
                                                                           508
# Table mapping response codes to messages; entries have the               509
```

```
510     # form {code: (shortmessage, longmessage)}.
511     # See RFC 2616.
512     responses = {
513         100: ('Continue', 'Request received, please continue'),
514         101: ('Switching Protocols',
515               'Switching to new protocol; obey Upgrade header'),
516
517         200: ('OK', 'Request fulfilled, document follows'),
518         201: ('Created', 'Document created, URL follows'),
519         202: ('Accepted',
520               'Request accepted, processing continues off-line'),
521         203: ('Non-Authoritative Information', 'Request fulfilled from
                cache'),
522         204: ('No Content', 'Request fulfilled, nothing follows'),
523         205: ('Reset Content', 'Clear input form for further input.'),
524         206: ('Partial Content', 'Partial content follows.'),
525
526         300: ('Multiple Choices',
527               'Object has several resources -- see URI list'),
528         301: ('Moved Permanently', 'Object moved permanently -- see URI
                list'),
529         302: ('Found', 'Object moved temporarily -- see URI list'),
530         303: ('See Other', 'Object moved -- see Method and URL list'),
531         304: ('Not Modified',
532               'Document has not changed since given time'),
533         305: ('Use Proxy',
534               'You must use proxy specified in Location to access this '
535               'resource.'),
536         307: ('Temporary Redirect',
537               'Object moved temporarily -- see URI list'),
538
539         400: ('Bad Request',
540               'Bad request syntax or unsupported method'),
541         401: ('Unauthorized',
542               'No permission -- see authorization schemes'),
543         402: ('Payment Required',
544               'No payment -- see charging schemes'),
545         403: ('Forbidden',
546               'Request forbidden -- authorization will not help'),
547         404: ('Not Found', 'Nothing matches the given URI'),
548         405: ('Method Not Allowed',
549               'Specified method is invalid for this resource.'),
550         406: ('Not Acceptable', 'URI not available in preferred format.'),
551         407: ('Proxy Authentication Required', 'You must authenticate with
                ',
552               'this proxy before proceeding.'),
553         408: ('Request Timeout', 'Request timed out; try again later.'),
554         409: ('Conflict', 'Request conflict.'),
555         410: ('Gone',
556               'URI no longer exists and has been permanently removed.'),
557         411: ('Length Required', 'Client must specify Content-Length.'),
558         412: ('Precondition Failed', 'Precondition in headers is false.'),
559         413: ('Request Entity Too Large', 'Entity is too large.'),
560         414: ('Request-URI Too Long', 'URI is too long.'),
561         415: ('Unsupported Media Type', 'Entity body in unsupported format
                .'),
562         416: ('Requested Range Not Satisfiable',
563               'Cannot satisfy request range.'),
```

```
        417: ('Expectation Failed',
              'Expect condition could not be satisfied.'),

        500: ('Internal Server Error', 'Server got itself in trouble'),
        501: ('Not Implemented',
              'Server does not support this operation'),
        502: ('Bad Gateway', 'Invalid responses from another server/proxy
            .'),
        503: ('Service Unavailable',
              'The server cannot process the request due to a high load'),
        504: ('Gateway Timeout',
              'The gateway server did not receive a timely response'),
        505: ('HTTP Version Not Supported', 'Cannot fulfill request.'),
        }


def test(HandlerClass = BaseHTTPRequestHandler,
         ServerClass = HTTPServer, protocol="HTTP/1.0"):
    """Test the HTTP request handler class.

    This runs an HTTP server on port 8000 (or the first command line
    argument).

    """

    if sys.argv[1:]:
        port = int(sys.argv[1])
    else:
        port = 8000
    server_address = ('', port)

    HandlerClass.protocol_version = protocol
    httpd = ServerClass(server_address, HandlerClass)

    sa = httpd.socket.getsockname()
    print "Serving HTTP on", sa[0], "port", sa[1], "..."
    httpd.serve_forever()


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