21.12. `http.client` — HTTP protocol client

Source code: Lib/http/client.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module `urllib.request` uses it to handle URLs that use HTTP and HTTPS.

Note: HTTPS support is only available if Python was compiled with SSL support (through the `ssl` module).

The module provides the following classes:

class `http.client.`**`HTTPConnection`**(*host, port=None[, strict][, timeout], source_address=None*)

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = http.client.HTTPConnection('www.cwi.nl')
>>> h2 = http.client.HTTPConnection('www.cwi.nl:80')
>>> h3 = http.client.HTTPConnection('www.cwi.nl', 80)
>>> h3 = http.client.HTTPConnection('www.cwi.nl', 80, timeout=10)
```

Changed in version 3.2: source_address was added.

Deprecated since version 3.2, will be removed in version 3.4: The *strict* parameter is deprecated. HTTP 0.9-style “Simple Responses” are not supported anymore.

class `http.client.`**`HTTPSConnection`**(*host, port=None, key_file=None, cert_file=None[, strict][, timeout], source_address=None, *, context=None, check_hostname=None*)

A subclass of `HTTPConnection` that uses SSL for communication with secure servers. Default port is `443`. If *context* is specified, it must be a `ssl.SSLContext` instance describing the various SSL options. If *context* is specified and has a `verify_mode` of either `CERT_OPTIONAL` or `CERT_REQUIRED`, then by default *host* is matched against the host name(s) allowed by the server's certificate. If you want to change that behaviour, you can explicitly set *check_hostname* to False.

key_file and *cert_file* are deprecated, please use `ssl.SSLContext.load_cert_chain()` instead.

If you access arbitrary hosts on the Internet, it is recommended to require certificate checking and feed the *context* with a set of trusted CA certificates:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
context.verify_mode = ssl.CERT_REQUIRED
context.load_verify_locations('/etc/pki/tls/certs/ca-bundle.crt')
h = client.HTTPSConnection('svn.python.org', 443, context=context
```

Changed in version 3.2: source_address, *context* and *check_hostname* were added.

Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if `ssl.HAS_SNI` is true).

Deprecated since version 3.2, will be removed in version 3.4: The *strict* parameter is deprecated. HTTP 0.9-style "Simple Responses" are not supported anymore.

class `http.client.`**`HTTPResponse`**(*sock*, *debuglevel=0*[, *strict*], *method=None*, *url=None*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

Deprecated since version 3.2, will be removed in version 3.4: The *strict* parameter is deprecated. HTTP 0.9-style "Simple Responses" are not supported anymore.

The following exceptions are raised as appropriate:

exception `http.client.`**`HTTPException`**

The base class of the other exceptions in this module. It is a subclass of `Exception`.

exception `http.client.`**`NotConnected`**

A subclass of `HTTPException`.

exception `http.client.`**`InvalidURL`**

A subclass of `HTTPException`, raised if a port is given and is either non-numeric or empty.

exception `http.client.`**`UnknownProtocol`**

A subclass of `HTTPException`.

exception `http.client.`**`UnknownTransferEncoding`**

A subclass of `HTTPException`.

exception `http.client.`**`UnimplementedFileMode`**

A subclass of `HTTPException`.

exception `http.client.`**`IncompleteRead`**

A subclass of `HTTPException`.

exception `http.client.`**`ImproperConnectionState`**

A subclass of `HTTPException`.

exception `http.client.`**`CannotSendRequest`**

A subclass of `ImproperConnectionState`.

exception `http.client.`**`CannotSendHeader`**

A subclass of `ImproperConnectionState`.

exception `http.client.`**`ResponseNotReady`**

A subclass of `ImproperConnectionState`.

exception `http.client.`**`BadStatusLine`**

A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don't understand.

The constants defined in this module are:

`http.client.`**`HTTP_PORT`**

The default port for the HTTP protocol (always `80`).

`http.client.`**`HTTPS_PORT`**

The default port for the HTTPS protocol (always `443`).

and also the following constants for integer status codes:

Constant	Value	Definition
`CONTINUE`	`100`	HTTP/1.1, RFC 2616, Section 10.1.1
`SWITCHING_PROTOCOLS`	`101`	HTTP/1.1, RFC 2616, Section

		10.1.2
PROCESSING	102	WEBDAV, RFC 2518, Section 10.1
OK	200	HTTP/1.1, RFC 2616, Section 10.2.1
CREATED	201	HTTP/1.1, RFC 2616, Section 10.2.2
ACCEPTED	202	HTTP/1.1, RFC 2616, Section 10.2.3
NON_AUTHORITATIVE_INFORMATION	203	HTTP/1.1, RFC 2616, Section 10.2.4
NO_CONTENT	204	HTTP/1.1, RFC 2616, Section 10.2.5
RESET_CONTENT	205	HTTP/1.1, RFC 2616, Section 10.2.6
PARTIAL_CONTENT	206	HTTP/1.1, RFC 2616, Section 10.2.7
MULTI_STATUS	207	WEBDAV RFC 2518, Section 10.2
IM_USED	226	Delta encoding in HTTP, **RFC 3229**, Section 10.4.1
MULTIPLE_CHOICES	300	HTTP/1.1, RFC 2616, Section 10.3.1
MOVED_PERMANENTLY	301	HTTP/1.1, RFC 2616, Section 10.3.2
FOUND	302	HTTP/1.1, RFC 2616, Section 10.3.3
SEE_OTHER	303	HTTP/1.1, RFC 2616, Section 10.3.4
NOT_MODIFIED	304	HTTP/1.1, RFC 2616, Section 10.3.5
USE_PROXY	305	HTTP/1.1, RFC 2616, Section 10.3.6
TEMPORARY_REDIRECT	307	HTTP/1.1, RFC 2616, Section 10.3.8
BAD_REQUEST	400	HTTP/1.1, RFC 2616, Section 10.4.1
UNAUTHORIZED	401	HTTP/1.1, RFC 2616, Section 10.4.2
PAYMENT_REQUIRED	402	HTTP/1.1, RFC 2616, Section 10.4.3
FORBIDDEN	403	HTTP/1.1, RFC 2616, Section

		10.4.4
`NOT_FOUND`	`404`	HTTP/1.1, RFC 2616, Section 10.4.5
`METHOD_NOT_ALLOWED`	`405`	HTTP/1.1, RFC 2616, Section 10.4.6
`NOT_ACCEPTABLE`	`406`	HTTP/1.1, RFC 2616, Section 10.4.7
`PROXY_AUTHENTICATION_REQUIRED`	`407`	HTTP/1.1, RFC 2616, Section 10.4.8
`REQUEST_TIMEOUT`	`408`	HTTP/1.1, RFC 2616, Section 10.4.9
`CONFLICT`	`409`	HTTP/1.1, RFC 2616, Section 10.4.10
`GONE`	`410`	HTTP/1.1, RFC 2616, Section 10.4.11
`LENGTH_REQUIRED`	`411`	HTTP/1.1, RFC 2616, Section 10.4.12
`PRECONDITION_FAILED`	`412`	HTTP/1.1, RFC 2616, Section 10.4.13
`REQUEST_ENTITY_TOO_LARGE`	`413`	HTTP/1.1, RFC 2616, Section 10.4.14
`REQUEST_URI_TOO_LONG`	`414`	HTTP/1.1, RFC 2616, Section 10.4.15
`UNSUPPORTED_MEDIA_TYPE`	`415`	HTTP/1.1, RFC 2616, Section 10.4.16
`REQUESTED_RANGE_NOT_SATISFIABLE`	`416`	HTTP/1.1, RFC 2616, Section 10.4.17
`EXPECTATION_FAILED`	`417`	HTTP/1.1, RFC 2616, Section 10.4.18
`UNPROCESSABLE_ENTITY`	`422`	WEBDAV, RFC 2518, Section 10.3
`LOCKED`	`423`	WEBDAV RFC 2518, Section 10.4
`FAILED_DEPENDENCY`	`424`	WEBDAV, RFC 2518, Section 10.5
`UPGRADE_REQUIRED`	`426`	HTTP Upgrade to TLS, **RFC 2817**, Section 6
`PRECONDITION_REQUIRED`	`428`	Additional HTTP Status Codes, **RFC 6585**, Section 3
`TOO_MANY_REQUESTS`	`429`	Additional HTTP Status Codes, **RFC 6585**, Section 4

`REQUEST_HEADER_FIELDS_TOO_LARGE`	`431`	Additional HTTP Status Codes, **RFC 6585**, Section 5
`INTERNAL_SERVER_ERROR`	`500`	HTTP/1.1, RFC 2616, Section 10.5.1
`NOT_IMPLEMENTED`	`501`	HTTP/1.1, RFC 2616, Section 10.5.2
`BAD_GATEWAY`	`502`	HTTP/1.1 RFC 2616, Section 10.5.3
`SERVICE_UNAVAILABLE`	`503`	HTTP/1.1, RFC 2616, Section 10.5.4
`GATEWAY_TIMEOUT`	`504`	HTTP/1.1 RFC 2616, Section 10.5.5
`HTTP_VERSION_NOT_SUPPORTED`	`505`	HTTP/1.1, RFC 2616, Section 10.5.6
`INSUFFICIENT_STORAGE`	`507`	WEBDAV, RFC 2518, Section 10.6
`NOT_EXTENDED`	`510`	An HTTP Extension Framework, **RFC 2774**, Section 7
`NETWORK_AUTHENTICATION_REQUIRED`	`511`	Additional HTTP Status Codes, **RFC 6585**, Section 6

Changed in version 3.3: Added codes `428`, `429`, `431` and `511` from **RFC 6585**.

`http.client.`**`responses`**

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `http.client.responses[http.client.NOT_FOUND]` is `'Not Found'`.

21.12.1. HTTPConnection Objects

`HTTPConnection` instances have the following methods:

`HTTPConnection.`**`request`**(*method*, *url*, *body=None*, *headers={}*)

This will send a request to the server using the HTTP request method *method* and the selector *url*. If the *body* argument is present, it should be string or bytes object of data to send after the headers are finished. Strings are encoded as ISO-8859-1, the default charset for HTTP. To use other encodings, pass a bytes object. The Content-Length header is set to the length of the string.

The *body* may also be an open *file object*, in which case the contents of the file is sent; this file object should support `fileno()` and `read()` methods. The header Content-Length is automatically set to the length of the file as reported by stat. The *body* argument may also be an iterable and Content-Length header should be explicitly provided when the body is an iterable.

The *headers* argument should be a mapping of extra HTTP headers to send with the request.

New in version 3.2: body can now be an iterable.

`HTTPConnection.`**`getresponse`**()

Should be called after a request is sent to get the response from the server. Returns an `HTTPResponse` instance.

> **Note:** Note that you must have read the whole response before you can send a new request to the server.

`HTTPConnection.`**`set_debuglevel`**(*level*)

Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new `HTTPResponse` objects that are created.

New in version 3.1.

`HTTPConnection.`**`set_tunnel`**(*host*, *port=None*, *headers=None*)

Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to a HTTPS Connection through a proxy server.

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

New in version 3.2.

`HTTPConnection.`**`connect`**()

Connect to the server specified when the object was created.

`HTTPConnection.`**`close`**()

Close the connection to the server.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

`HTTPConnection.`**`putrequest`**(*request*, *selector*, *skip_host=False*, *skip_accept_encoding=False*)

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *request* string, the *selector* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

`HTTPConnection.`**`putheader`**(*header*, *argument*[, ...])

Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

`HTTPConnection.`**`endheaders`**(*message_body=None*)

Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request. The message body will be sent in the same packet as the message headers if it is string, otherwise it is sent in a separate packet.

`HTTPConnection.`**`send`**(*data*)

Send data to the server. This should be used directly only after the `endheaders()` method has been called and before `getresponse()` is called.

21.12.2. HTTPResponse Objects

An `HTTPResponse` instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

`HTTPResponse.`**`read`**([*amt*])

Reads and returns the response body, or up to the next *amt* bytes.

`HTTPResponse.`**`readinto`**(*b*)

Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

New in version 3.3.

`HTTPResponse.`**`getheader`**(*name*, *default=None*)

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If 'default' is any iterable other than a single string, its elements are similarly returned joined by commas.

`HTTPResponse.`**`getheaders`**()

Return a list of (header, value) tuples.

`HTTPResponse.`**`fileno`**()

Return the `fileno` of the underlying socket.

`HTTPResponse.`**`msg`**

A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of `email.message.Message`.

`HTTPResponse.`**`version`**

HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

`HTTPResponse.`**`status`**

Status code returned by server.

`HTTPResponse.`**`reason`**

Reason phrase returned by server.

`HTTPResponse.`**`debuglevel`**

A debugging hook. If `debuglevel` is greater than zero, messages will be printed to stdout as the response is read and parsed.

`HTTPResponse.`**`closed`**

Is `True` if the stream is closed.

21.12.3. Examples

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPConnection("www.python.org")
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200)) # 200 bytes
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"...
...
>>> # Example of an invalid request
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPConnection("www.python.org")
>>> conn.request("HEAD","/index.html")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to `POST` requests:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issu
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://b
>>> conn.close()
```

Client side `HTTP PUT` requests are very similar to `POST` requests. The difference lies only the server side where HTTP server will allow resources to be created via `PUT` request. It should be noted that custom HTTP methods +are also handled in `urllib.request.Request` by sending the appropriate +method attribute.Here is an example session that shows how to do `PUT` request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
```

```
200, OK
```

21.12.4. HTTPMessage Objects

An `http.client.HTTPMessage` instance holds the headers from an HTTP response. It is implemented using the `email.message.Message` class.