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.

See also: The Requests package is recommended for a higher-level HTTP client interface.

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*, [*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.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=
```

Changed in version 3.2: source_address was added.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style “Simple Responses” are not longer supported.

class `http.client.`**`HTTPSConnection`**(*host*, *port=None*, *key_file=None*, *cert_file=None*, [*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.

Please read Security considerations for more information on best practices.

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).

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

Changed in version 3.4.3: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

Deprecated since version 3.6: *key_file* and *cert_file* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

The *check_hostname* parameter is also deprecated; the `ssl.SSLContext.check_hostname` attribute of *context* should be used instead.

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

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

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9 style "Simple Responses" are no longer supported.

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.

exception `http.client.`**LineTooLong**

A subclass of `HTTPException`. Raised if an excessively long line is received in the HTTP protocol from the server.

exception `http.client.`**RemoteDisconnected**

A subclass of `ConnectionResetError` and `BadStatusLine`. Raised by `HTTPConnection.getresponse()` when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

New in version 3.5: Previously, `BadStatusLine('')` was raised.

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`).

`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'`.

See HTTP status codes for a list of HTTP status codes that are available in this module as constants.

21.12.1. HTTPConnection Objects

`HTTPConnection` instances have the following methods:

`HTTPConnection.`**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
: This will send a request to the server using the HTTP request method *method* and the selector *url*.

  If *body* is specified, the specified data is sent after the headers are finished. It may be a `str`, a bytes-like object, an open file object, or an iterable of `bytes`. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a file object, the contents of the file is sent; this file object should support at least the `read()` method. If the file object is an instance of `io.TextIOBase`, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

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

  If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a file, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

> **Note:** Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a `str` or bytes-like object that is not also a file as the body representation.

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

Changed in version 3.6: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

`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.

Changed in version 3.5: If a `ConnectionError` or subclass is raised, the `HTTPConnection` object will be ready to reconnect when a new request is sent.

`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. This allows running the connection through a proxy server.

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

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

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the `HTTPSConnection` constructor, and the address of the host that we eventually want to reach to the `set_tunnel()` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

New in version 3.2.

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

Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

`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`**(*method*, *url*, *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 *method* string, the *url* 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*, *, *encode_chunked=False*)

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.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a `collections.Iterable`, each iteration of *message_body* will result in a chunk. If *message_body* is a file object, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

> **Note:** Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

New in version 3.6: Chunked encoding support. The *encode_chunked* parameter was added.

`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.

Changed in version 3.5: The `io.BufferedIOBase` interface is now implemented and all of its reader operations are supported.

`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.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> 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", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200))  # 200 bytes
b'<!doctype html>\n<!--[if"...
...
>>> # 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.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> 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': 'issue
>>> 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://bu
>>> 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/file
...
>>> 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.