urllib3 Documentation

Release 1.22

Andrey Petrov

Jul 20, 2017

Contents

CHAPTER 1

User Guide

Making requests

First things first, import the urllib3 module:

```
>>> import urllib3
```

You'll need a *PoolManager* instance to make requests. This object handles all of the details of connection pooling and thread safety so that you don't have to:

```
>>> http = urllib3.PoolManager()
```

To make a request use `request()`:

```
>>> r = http.request('GET', 'http://httpbin.org/robots.txt')
>>> r.data
b'User-agent: *\nDisallow: /deny\n'
```

`request()` returns a *HTTPResponse* object, the *Response content* section explains how to handle various responses.

You can use `request()` to make requests using any HTTP verb:

```
>>> r = http.request(
...     'POST',
...     'http://httpbin.org/post',
...     fields={'hello': 'world'})
```

The *Request data* section covers sending other kinds of requests data, including JSON, files, and binary data.

Response content

The *HTTPResponse* object provides `status`, *data*, and `header` attributes:

```
>>> r = http.request('GET', 'http://httpbin.org/ip')
>>> r.status
200
>>> r.data
b'{\n  "origin": "104.232.115.37"\n}\n'
>>> r.headers
HTTPHeaderDict({'Content-Length': '33', ...})
```

JSON content

JSON content can be loaded by decoding and deserializing the `data` attribute of the request:

```
>>> import json
>>> r = http.request('GET', 'http://httpbin.org/ip')
>>> json.loads(r.data.decode('utf-8'))
{'origin': '127.0.0.1'}
```

Binary content

The `data` attribute of the response is always set to a byte string representing the response content:

```
>>> r = http.request('GET', 'http://httpbin.org/bytes/8')
>>> r.data
b'\xaa\xa5H?\x95\xe9\x9b\x11'
```

Note: For larger responses, it's sometimes better to *stream* the response.

Request data

Headers

You can specify headers as a dictionary in the `headers` argument in `request()`:

```
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/headers',
...     headers={
...         'X-Something': 'value'
...     })
>>> json.loads(r.data.decode('utf-8'))['headers']
{'X-Something': 'value', ...}
```

Query parameters

For `GET`, `HEAD`, and `DELETE` requests, you can simply pass the arguments as a dictionary in the `fields` argument to `request()`:

```
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/get',
...     fields={'arg': 'value'})
>>> json.loads(r.data.decode('utf-8'))['args']
{'arg': 'value'}
```

For `POST` and `PUT` requests, you need to manually encode query parameters in the URL:

```
>>> from urllib.parse import urlencode
>>> encoded_args = urlencode({'arg': 'value'})
>>> url = 'http://httpbin.org/post?' + encoded_args
>>> r = http.request('POST', url)
>>> json.loads(r.data.decode('utf-8'))['args']
{'arg': 'value'}
```

Form data

For `PUT` and `POST` requests, urllib3 will automatically form-encode the dictionary in the `fields` argument provided to `request()`:

```
>>> r = http.request(
...     'POST',
...     'http://httpbin.org/post',
...     fields={'field': 'value'})
>>> json.loads(r.data.decode('utf-8'))['form']
{'field': 'value'}
```

JSON

You can sent JSON a request by specifying the encoded data as the `body` argument and setting the `Content-Type` header when calling `request()`:

```
>>> import json
>>> data = {'attribute': 'value'}
>>> encoded_data = json.dumps(data).encode('utf-8')
>>> r = http.request(
...     'POST',
...     'http://httpbin.org/post',
...     body=encoded_data,
...     headers={'Content-Type': 'application/json'})
>>> json.loads(r.data.decode('utf-8'))['json']
{'attribute': 'value'}
```

Files & binary data

For uploading files using `multipart/form-data` encoding you can use the same approach as *Form data* and specify the file field as a tuple of `(file_name, file_data)`:

```
>>> with open('example.txt') as fp:
...     file_data = fp.read()
>>> r = http.request(
```

```
...     'POST',
...     'http://httpbin.org/post',
...     fields={
...         'filefield': ('example.txt', file_data),
...     })
>>> json.loads(r.data.decode('utf-8'))['files']
{'filefield': '...'}
```

While specifying the filename is not strictly required, it's recommended in order to match browser behavior. You can also pass a third item in the tuple to specify the file's MIME type explicitly:

```
>>> r = http.request(
...     'POST',
...     'http://httpbin.org/post',
...     fields={
...         'filefield': ('example.txt', file_data, 'text/plain'),
...     })
```

For sending raw binary data simply specify the `body` argument. It's also recommended to set the `Content-Type` header:

```
>>> with open('example.jpg', 'rb') as fp:
...     binary_data = fp.read()
>>> r = http.request(
...     'POST',
...     'http://httpbin.org/post',
...     body=binary_data,
...     headers={'Content-Type': 'image/jpeg'})
>>> json.loads(r.data.decode('utf-8'))['data']
b'...'
```

Certificate verification

It is highly recommended to always use SSL certificate verification. **By default, urllib3 does not verify HTTPS requests**.

In order to enable verification you will need a set of root certificates. The easiest and most reliable method is to use the certifi package which provides Mozilla's root certificate bundle:

```
pip install certifi
```

You can also install certifi along with urllib3 by using the `secure` extra:

```
pip install urllib3[secure]
```

> **Warning:** If you're using Python 2 you may need additional packages. See the *section below* for more details.

Once you have certificates, you can create a *`PoolManager`* that verifies certificates when making requests:

```
>>> import certifi
>>> import urllib3
>>> http = urllib3.PoolManager(
```

```
...     cert_reqs='CERT_REQUIRED',
...     ca_certs=certifi.where())
```

The *PoolManager* will automatically handle certificate verification and will raise *SSLError* if verification fails:

```
>>> http.request('GET', 'https://google.com')
(No exception)
>>> http.request('GET', 'https://expired.badssl.com')
urllib3.exceptions.SSLError ...
```

Note: You can use OS-provided certificates if desired. Just specify the full path to the certificate bundle as the `ca_certs` argument instead of `certifi.where()`. For example, most Linux systems store the certificates at `/etc/ssl/certs/ca-certificates.crt`. Other operating systems can be difficult.

Certificate verification in Python 2

Older versions of Python 2 are built with an `ssl` module that lacks *SNI support* and can lag behind security updates. For these reasons it's recommended to use pyOpenSSL.

If you install urllib3 with the `secure` extra, all required packages for certificate verification on Python 2 will be installed:

```
pip install urllib3[secure]
```

If you want to install the packages manually, you will need `pyOpenSSL`, `cryptography`, `idna`, and `certifi`.

Note: If you are not using macOS or Windows, note that cryptography requires additional system packages to compile. See building cryptography on Linux for the list of packages required.

Once installed, you can tell urllib3 to use pyOpenSSL by using *urllib3.contrib.pyopenssl*:

```
>>> import urllib3.contrib.pyopenssl
>>> urllib3.contrib.pyopenssl.inject_into_urllib3()
```

Finally, you can create a *PoolManager* that verifies certificates when performing requests:

```
>>> import certifi
>>> import urllib3
>>> http = urllib3.PoolManager(
...     cert_reqs='CERT_REQUIRED',
...     ca_certs=certifi.where())
```

If you do not wish to use pyOpenSSL, you can simply omit the call to *urllib3.contrib.pyopenssl.inject_into_urllib3()*. urllib3 will fall back to the standard-library `ssl` module. You may experience *several warnings* when doing this.

Warning: If you do not use pyOpenSSL, Python must be compiled with ssl support for certificate verification to work. It is uncommon, but it is possible to compile Python without SSL support. See this Stackoverflow thread for more details.

If you are on Google App Engine, you must explicitly enable SSL support in your `app.yaml`:

```
libraries:
- name: ssl
  version: latest
```

Using timeouts

Timeouts allow you to control how long requests are allowed to run before being aborted. In simple cases, you can specify a timeout as a `float` to `request()`:

```
>>> http.request(
...     'GET', 'http://httpbin.org/delay/3', timeout=4.0)
<urllib3.response.HTTPResponse>
>>> http.request(
...     'GET', 'http://httpbin.org/delay/3', timeout=2.5)
MaxRetryError caused by ReadTimeoutError
```

For more granular control you can use a *Timeout* instance which lets you specify separate connect and read timeouts:

```
>>> http.request(
...     'GET',
...     'http://httpbin.org/delay/3',
...     timeout=urllib3.Timeout(connect=1.0))
<urllib3.response.HTTPResponse>
>>> http.request(
...     'GET',
...     'http://httpbin.org/delay/3',
...     timeout=urllib3.Timeout(connect=1.0, read=2.0))
MaxRetryError caused by ReadTimeoutError
```

If you want all requests to be subject to the same timeout, you can specify the timeout at the *PoolManager* level:

```
>>> http = urllib3.PoolManager(timeout=3.0)
>>> http = urllib3.PoolManager(
...     timeout=urllib3.Timeout(connect=1.0, read=2.0))
```

You still override this pool-level timeout by specifying `timeout` to `request()`.

Retrying requests

urllib3 can automatically retry idempotent requests. This same mechanism also handles redirects. You can control the retries using the `retries` parameter to `request()`. By default, urllib3 will retry requests 3 times and follow up to 3 redirects.

To change the number of retries just specify an integer:

```
>>> http.requests('GET', 'http://httpbin.org/ip', retries=10)
```

To disable all retry and redirect logic specify `retries=False`:

```
>>> http.request(
...     'GET', 'http://nxdomain.example.com', retries=False)
NewConnectionError
```

```
>>> r = http.request(
...     'GET', 'http://httpbin.org/redirect/1', retries=False)
>>> r.status
302
```

To disable redirects but keep the retrying logic, specify `redirect=False`:

```
>>> r = http.request(
...     'GET', 'http://httpbin.org/redirect/1', redirect=False)
>>> r.status
302
```

For more granular control you can use a *Retry* instance. This class allows you far greater control of how requests are retried.

For example, to do a total of 3 retries, but limit to only 2 redirects:

```
>>> http.request(
...     'GET',
...     'http://httpbin.org/redirect/3',
...     retries=urllib3.Retries(3, redirect=2))
MaxRetryError
```

You can also disable exceptions for too many redirects and just return the `302` response:

```
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/redirect/3',
...     retries=urllib3.Retries(
...         redirect=2, raise_on_redirect=False))
>>> r.status
302
```

If you want all requests to be subject to the same retry policy, you can specify the retry at the *PoolManager* level:

```
>>> http = urllib3.PoolManager(retries=False)
>>> http = urllib3.PoolManager(
...     retries=urllib3.Retry(5, redirect=2))
```

You still override this pool-level retry policy by specifying `retries` to `request()`.

Errors & Exceptions

urllib3 wraps lower-level exceptions, for example:

```
>>> try:
...     http.request('GET', 'nx.example.com', retries=False)
>>> except urllib3.exceptions.NewConnectionError:
...     print('Connection failed.')
```

See *exceptions* for the full list of all exceptions.

Logging

If you are using the standard library `logging` module urllib3 will emit several logs. In some cases this can be undesirable. You can use the standard logger interface to change the log level for urllib3's logger:

```
>>> logging.getLogger("urllib3").setLevel(logging.WARNING)
```

CHAPTER 2

Advanced Usage

Customizing pool behavior

The *PoolManager* class automatically handles creating *ConnectionPool* instances for each host as needed. By default, it will keep a maximum of 10 *ConnectionPool* instances. If you're making requests to many different hosts it might improve performance to increase this number:

```
>>> import urllib3
>>> http = urllib3.PoolManager(num_pools=50)
```

However, keep in mind that this does increase memory and socket consumption.

Similarly, the *ConnectionPool* class keeps a pool of individual *HTTPConnection* instances. These connections are used during an individual request and returned to the pool when the request is complete. By default only one connection will be saved for re-use. If you are making many requests to the same host simultaneously it might improve performance to increase this number:

```
>>> import urllib3
>>> http = urllib3.PoolManager(maxsize=10)
# Alternatively
>>> http = urllib3.HTTPConnectionPool('google.com', maxsize=10)
```

The behavior of the pooling for *ConnectionPool* is different from *PoolManager*. By default, if a new request is made and there is no free connection in the pool then a new connection will be created. However, this connection will not be saved if more than `maxsize` connections exist. This means that `maxsize` does not determine the maximum number of connections that can be open to a particular host, just the maximum number of connections to keep in the pool. However, if you specify `block=True` then there can be at most `maxsize` connections open to a particular host:

```
>>> http = urllib3.PoolManager(maxsize=10, block=True)
# Alternatively
>>> http = urllib3.HTTPConnectionPool('google.com', maxsize=10, block=True)
```

Any new requests will block until a connection is available from the pool. This is a great way to prevent flooding a host with too many connections in multi-threaded applications.

Streaming and IO

When dealing with large responses it's often better to stream the response content:

```
>>> import urllib3
>>> http = urllib3.PoolManager()
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/bytes/1024',
...     preload_content=False)
>>> for chunk in r.stream(32):
...     print(chunk)
b'...'
b'...'
...
>>> r.release_conn()
```

Setting `preload_content` to `False` means that urllib3 will stream the response content. *`stream()`* lets you iterate over chunks of the response content.

Note: When using `preload_content=False`, you should call *`release_conn()`* to release the http connection back to the connection pool so that it can be re-used.

However, you can also treat the *`HTTPResponse`* instance as a file-like object. This allows you to do buffering:

```
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/bytes/1024',
...     preload_content=False)
>>> r.read(4)
b'\x88\x1f\x8b\xe5'
```

Calls to *`read()`* will block until more response data is available.

```
>>> import io
>>> reader = io.BufferedReader(r, 8)
>>> reader.read(4)
>>> r.release_conn()
```

You can use this file-like object to do things like decode the content using `codecs`:

```
>>> import codecs
>>> reader = codecs.getreader('utf-8')
>>> r = http.request(
...     'GET',
...     'http://httpbin.org/ip',
...     preload_content=False)
>>> json.load(reader(r))
{'origin': '127.0.0.1'}
>>> r.release_conn()
```

Proxies

You can use *ProxyManager* to tunnel requests through an HTTP proxy:

```
>>> import urllib3
>>> proxy = urllib3.ProxyManager('http://localhost:3128/')
>>> proxy.request('GET', 'http://google.com/')
```

The usage of *ProxyManager* is the same as *PoolManager*.

You can use *SOCKSProxyManager* to connect to SOCKS4 or SOCKS5 proxies. In order to use SOCKS proxies you will need to install PySocks or install urllib3 with the `socks` extra:

```
pip install urllib3[socks]
```

Once PySocks is installed, you can use *SOCKSProxyManager*:

```
>>> from urllib3.contrib.socks import SOCKSProxyManager
>>> proxy = SOCKSProxyManager('socks5://localhost:8889/')
>>> proxy.request('GET', 'http://google.com/')
```

Custom SSL certificates and client certificates

Instead of using certifi you can provide your own certificate authority bundle. This is useful for cases where you've generated your own certificates or when you're using a private certificate authority. Just provide the full path to the certificate bundle when creating a *PoolManager*:

```
>>> import urllib3
>>> http = urllib3.PoolManager(
...     cert_reqs='CERT_REQUIRED',
...     ca_certs='/path/to/your/certificate_bundle')
```

When you specify your own certificate bundle only requests that can be verified with that bundle will succeed. It's recommended to use a separate *PoolManager* to make requests to URLs that do not need the custom certificate.

You can also specify a client certificate. This is useful when both the server and the client need to verify each other's identity. Typically these certificates are issued from the same authority. To use a client certificate, provide the full path when creating a *PoolManager*:

```
>>> http = urllib3.PoolManager(
...     cert_file='/path/to/your/client_cert.pem',
...     cert_reqs='CERT_REQUIRED',
...     ca_certs='/path/to/your/certificate_bundle')
```

Certificate validation and Mac OS X

Apple-provided Python and OpenSSL libraries contain a patches that make them automatically check the system keychain's certificates. This can be surprising if you specify custom certificates and see requests unexpectedly succeed. For example, if you are specifying your own certificate for validation and the server presents a different certificate you would expect the connection to fail. However, if that server presents a certificate that is in the system keychain then the connection will succeed.

This article has more in-depth analysis and explanation.

SSL Warnings

urllib3 will issue several different warnings based on the level of certificate verification support. These warning indicate particular situations and can resolved in different ways.

- ***InsecureRequestWarning*** This happens when an request is made to an HTTPS URL without certificate verification enabled. Follow the *certificate verification* guide to resolve this warning.
- ***InsecurePlatformWarning*** This happens on Python 2 platforms that have an outdated `ssl` module. These older `ssl` modules can cause some insecure requests to succeed where they should fail and secure requests to fail where they should succeed. Follow the *pyOpenSSL* guide to resolve this warning.
- ***SNIMissingWarning*** This happens on Python 2 versions older than 2.7.9. These older versions lack SNI support. This can cause servers to present a certificate that the client thinks is invalid. Follow the *pyOpenSSL* guide to resolve this warning.

Making unverified HTTPS requests is **strongly** discouraged, however, if you understand the risks and wish to disable these warnings, you can use *disable_warnings()*:

```
>>> import urllib3
>>> urllib3.disable_warnings()
```

Alternatively you can capture the warnings with the standard `logging` module:

```
>>> logging.captureWarnings(True)
```

Finally, you can suppress the warnings at the interpreter level by setting the `PYTHONWARNINGS` environment variable or by using the -W flag.

Google App Engine

urllib3 supports Google App Engine with some caveats.

If you're using the Flexible environment, you do not have to do any configuration- urllib3 will just work. However, if you're using the Standard environment then you either have to use *urllib3.contrib.appengine*'s *AppEngineManager* or use the Sockets API

To use *AppEngineManager*:

```
>>> from urllib3.contrib.appengine import AppEngineManager
>>> http = AppEngineManager()
>>> http.request('GET', 'https://google.com/')
```

To use the Sockets API, add the following to your app.yaml and use *PoolManager* as usual:

```
env_variables:
    GAE_USE_SOCKETS_HTTPLIB : 'true'
```

For more details on the limitations and gotchas, see *urllib3.contrib.appengine*.

CHAPTER 3

Reference

Subpackages

urllib3.contrib package

These modules implement various extra features, that may not be ready for prime time or that require optional third-party dependencies.

urllib3.contrib.appengine module

This module provides a pool manager that uses Google App Engine's URLFetch Service.

Example usage:

```
from urllib3 import PoolManager
from urllib3.contrib.appengine import AppEngineManager, is_appengine_sandbox

if is_appengine_sandbox():
    # AppEngineManager uses AppEngine's URLFetch API behind the scenes
    http = AppEngineManager()
else:
    # PoolManager uses a socket-level API behind the scenes
    http = PoolManager()

r = http.request('GET', 'https://google.com/')
```

There are limitations to the URLFetch service and it may not be the best choice for your application. There are three options for using urllib3 on Google App Engine:

1. You can use *AppEngineManager* with URLFetch. URLFetch is cost-effective in many circumstances as long as your usage is within the limitations.
2. You can use a normal *PoolManager* by enabling sockets. Sockets also have limitations and restrictions and have a lower free quota than URLFetch. To use sockets, be sure to specify the following in your `app.yaml`:

```
env_variables:
    GAE_USE_SOCKETS_HTTPLIB : 'true'
```

3. If you are using App Engine Flexible, you can use the standard `PoolManager` without any configuration or special environment variables.

class `urllib3.contrib.appengine.`**`AppEngineManager`**(*headers=None*, *retries=None*, *validate_certificate=True*, *urlfetch_retries=True*)

Bases: *urllib3.request.RequestMethods*

Connection manager for Google App Engine sandbox applications.

This manager uses the URLFetch service directly instead of using the emulated httplib, and is subject to URLFetch limitations as described in the App Engine documentation here.

Notably it will raise an *AppEnginePlatformError* if:

- URLFetch is not available.
- If you attempt to use this on App Engine Flexible, as full socket support is available.
- If a request size is more than 10 megabytes.
- If a response size is more than 32 megabtyes.
- If you use an unsupported request method such as OPTIONS.

Beyond those cases, it will raise normal urllib3 errors.

`urlopen`(*method*, *url*, *body=None*, *headers=None*, *retries=None*, *redirect=True*, *timeout=<object object>*, ***response_kw*)

exception `urllib3.contrib.appengine.`**`AppEnginePlatformError`**

Bases: *urllib3.exceptions.HTTPError*

exception `urllib3.contrib.appengine.`**`AppEnginePlatformWarning`**

Bases: *urllib3.exceptions.HTTPWarning*

`urllib3.contrib.appengine.`**`is_appengine`**()

urllib3.contrib.appengine.**is_appengine_sandbox**()

urllib3.contrib.appengine.**is_local_appengine**()

urllib3.contrib.appengine.**is_prod_appengine**()

urllib3.contrib.appengine.**is_prod_appengine_mvms**()

urllib3.contrib.ntlmpool module

NTLM authenticating pool, contributed by erikcederstran

Issue #10, see: http://code.google.com/p/urllib3/issues/detail?id=10

class urllib3.contrib.ntlmpool.**NTLMConnectionPool**(*user*, *pw*, *authurl*, **args*, ***kwargs*)

Bases: *urllib3.connectionpool.HTTPSConnectionPool*

Implements an NTLM authentication version of an urllib3 connection pool

scheme = 'https'

urlopen(*method*, *url*, *body=None*, *headers=None*, *retries=3*, *redirect=True*, *assert_same_host=True*)

urllib3.contrib.pyopenssl module

SSL with SNI-support for Python 2. Follow these instructions if you would like to verify SSL certificates in Python 2. Note, the default libraries do *not* do certificate checking; you need to do additional work to validate certificates yourself.

This needs the following packages installed:

- pyOpenSSL (tested with 16.0.0)
- cryptography (minimum 1.3.4, from pyopenssl)
- idna (minimum 2.0, from cryptography)

However, pyopenssl depends on cryptography, which depends on idna, so while we use all three directly here we end up having relatively few packages required.

You can install them with the following command:

```
pip install pyopenssl cryptography idna
```

To activate certificate checking, call *inject_into_urllib3()* from your Python code before you begin making HTTP requests. This can be done in a `sitecustomize` module, or at any other time before your application begins using `urllib3`, like this:

```
try:
    import urllib3.contrib.pyopenssl
    urllib3.contrib.pyopenssl.inject_into_urllib3()
except ImportError:
    pass
```

Now you can use *urllib3* as you normally would, and it will support SNI when the required modules are installed.

Activating this module also has the positive side effect of disabling SSL/TLS compression in Python 2 (see CRIME attack).

If you want to configure the default list of supported cipher suites, you can set the `urllib3.contrib.pyopenssl.DEFAULT_SSL_CIPHER_LIST` variable.

urllib3.contrib.pyopenssl.**inject_into_urllib3**()
Monkey-patch urllib3 with PyOpenSSL-backed SSL-support.

urllib3.contrib.pyopenssl.**extract_from_urllib3**()
Undo monkey-patching by *inject_into_urllib3()*.

urllib3.contrib.socks module

This module contains provisional support for SOCKS proxies from within urllib3. This module supports SOCKS4 (specifically the SOCKS4A variant) and SOCKS5. To enable its functionality, either install PySocks or install this module with the `socks` extra.

The SOCKS implementation supports the full range of urllib3 features. It also supports the following SOCKS features:

- SOCKS4
- SOCKS4a
- SOCKS5
- Usernames and passwords for the SOCKS proxy

Known Limitations:

- Currently PySocks does not support contacting remote websites via literal IPv6 addresses. Any such connection attempt will fail. You must use a domain name.
- Currently PySocks does not support IPv6 connections to the SOCKS proxy. Any such connection attempt will fail.

class urllib3.contrib.socks.**SOCKSConnection**(**args*, ***kwargs*)
Bases: *urllib3.connection.HTTPConnection*

A plain-text HTTP connection that connects via a SOCKS proxy.

class urllib3.contrib.socks.**SOCKSHTTPConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, ***conn_kw*)
Bases: *urllib3.connectionpool.HTTPConnectionPool*

ConnectionCls
alias of *SOCKSConnection*

class urllib3.contrib.socks.**SOCKSHTTPSConnection**(**args*, ***kwargs*)
Bases: *urllib3.contrib.socks.SOCKSConnection*, *urllib3.connection.VerifiedHTTPSConnection*

class urllib3.contrib.socks.**SOCKSHTTPSConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, *key_file=None*, *cert_file=None*, *cert_reqs=None*, *ca_certs=None*, *ssl_version=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*, ***conn_kw*)

Bases: *urllib3.connectionpool.HTTPSConnectionPool*

ConnectionCls
alias of *SOCKSHTTPSConnection*

class urllib3.contrib.socks.**SOCKSProxyManager**(*proxy_url*, *username=None*, *password=None*, *num_pools=10*, *headers=None*, ***connection_pool_kw*)

Bases: *urllib3.poolmanager.PoolManager*

A version of the urllib3 ProxyManager that routes connections via the defined SOCKS proxy.

pool_classes_by_scheme = {'http': <class 'urllib3.contrib.socks.SOCKSHTTPConnectionPool'>, 'https': <class 'ur

urllib3.util package

Useful methods for working with `httplib`, completely decoupled from code specific to **urllib3**.

At the very core, just like its predecessors, *urllib3* is built on top of `httplib` – the lowest level HTTP library included in the Python standard library.

To aid the limited functionality of the `httplib` module, *urllib3* provides various helper methods which are used with the higher level components but can also be used independently.

urllib3.util.connection module

urllib3.util.connection.**allowed_gai_family**()
This function is designed to work in the context of getaddrinfo, where family=socket.AF_UNSPEC is the default and will perform a DNS search for both IPv6 and IPv4 records.

urllib3.util.connection.**create_connection**(*address*, *timeout=<object object>*, *source_address=None*, *socket_options=None*)
Connect to *address* and return the socket object.

Convenience function. Connect to *address* (a 2-tuple `(host, port)`) and return the socket object. Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used. If *source_address* is set it must be a tuple of (host, port) for the socket to bind as a source address before making the connection. An host of '' or port 0 tells the OS to use the default.

urllib3.util.connection.**is_connection_dropped**(*conn*)
Returns True if the connection is dropped and should be closed.

Parameters **conn** – `httplib.HTTPConnection` object.

Note: For platforms like AppEngine, this will always return `False` to let the platform handle connection recycling transparently for us.

urllib3.util.request module

urllib3.util.request.**make_headers**(*keep_alive=None*, *accept_encoding=None*, *user_agent=None*, *basic_auth=None*, *proxy_basic_auth=None*, *disable_cache=None*)

Shortcuts for generating request headers.

Parameters

- **keep_alive** – If `True`, adds 'connection: keep-alive' header.
- **accept_encoding** – Can be a boolean, list, or string. `True` translates to 'gzip,deflate'. List will get joined by comma. String will be used as provided.
- **user_agent** – String representing the user-agent you want, such as "python-urllib3/0.6"
- **basic_auth** – Colon-separated username:password string for 'authorization: basic ...' auth header.
- **proxy_basic_auth** – Colon-separated username:password string for 'proxy-authorization: basic ...' auth header.
- **disable_cache** – If `True`, adds 'cache-control: no-cache' header.

Example:

```
>>> make_headers(keep_alive=True, user_agent="Batman/1.0")
{'connection': 'keep-alive', 'user-agent': 'Batman/1.0'}
>>> make_headers(accept_encoding=True)
{'accept-encoding': 'gzip,deflate'}
```

urllib3.util.request.**rewind_body**(*body*, *body_pos*)

Attempt to rewind body to a certain position. Primarily used for request redirects and retries.

Parameters

- **body** – File-like object that supports seek.
- **pos** (*int*) – Position to seek to in file.

urllib3.util.request.**set_file_position**(*body*, *pos*)

If a position is provided, move file to that point. Otherwise, we'll attempt to record a position for future use.

urllib3.util.response module

urllib3.util.response.**assert_header_parsing**(*headers*)

Asserts whether all headers have been successfully parsed. Extracts encountered errors from the result of parsing headers.

Only works on Python 3.

Parameters **headers** (*httplib.HTTPMessage.*) – Headers to verify.

Raises ***urllib3.exceptions.HeaderParsingError*** – If parsing errors are found.

urllib3.util.response.**is_fp_closed**(*obj*)

Checks whether a given file-like object is closed.

Parameters **obj** – The file-like object to check.

urllib3.util.response.**is_response_to_head**(*response*)

Checks whether the request of a response has been a HEAD-request. Handles the quirks of AppEngine.

Parameters **conn** (`httplib.HTTPResponse`) –

urllib3.util.retry module

class `urllib3.util.retry.`**RequestHistory**(*method*, *url*, *error*, *status*, *redirect_location*)
Bases: `tuple`

error
Alias for field number 2

method
Alias for field number 0

redirect_location
Alias for field number 4

status
Alias for field number 3

url
Alias for field number 1

class `urllib3.util.retry.`**Retry**(*total=10*, *connect=None*, *read=None*, *redirect=None*, *status=None*, *method_whitelist=frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE'])*, *status_forcelist=None*, *backoff_factor=0*, *raise_on_redirect=True*, *raise_on_status=True*, *history=None*, *respect_retry_after_header=True*)
Bases: `object`

Retry configuration.

Each retry attempt will create a new Retry object with updated values, so they can be safely reused.

Retries can be defined as a default for a pool:

```
retries = Retry(connect=5, read=2, redirect=5)
http = PoolManager(retries=retries)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', retries=Retry(10))
```

Retries can be disabled by passing `False`:

```
response = http.request('GET', 'http://example.com/', retries=False)
```

Errors will be wrapped in *`MaxRetryError`* unless retries are disabled, in which case the causing exception will be raised.

Parameters

- **total** (*int*) – Total number of retries to allow. Takes precedence over other counts.

  Set to `None` to remove this constraint and fall back on other counts. It's a good idea to set this to some sensibly-high value to account for unexpected edge cases and avoid infinite retry loops.

  Set to `0` to fail on the first retry.

  Set to `False` to disable and imply `raise_on_redirect=False`.

- **connect** (*int*) – How many connection-related errors to retry on.

  These are errors raised before the request is sent to the remote server, which we assume has not triggered the server to process the request.

  Set to `0` to fail on the first retry of this type.

- **read** (*int*) – How many times to retry on read errors.

  These errors are raised after the request was sent to the server, so the request may have side-effects.

  Set to `0` to fail on the first retry of this type.

- **redirect** (*int*) – How many redirects to perform. Limit this to avoid infinite redirect loops.

  A redirect is a HTTP response with a status code 301, 302, 303, 307 or 308.

  Set to `0` to fail on the first retry of this type.

  Set to `False` to disable and imply `raise_on_redirect=False`.

- **status** (*int*) – How many times to retry on bad status codes.

  These are retries made on responses, where status code matches `status_forcelist`.

  Set to `0` to fail on the first retry of this type.

- **method_whitelist** (*iterable*) – Set of uppercased HTTP method verbs that we should retry on.

  By default, we only retry on methods which are considered to be idempotent (multiple requests with the same parameters end with the same state). See *Retry.DEFAULT_METHOD_WHITELIST*.

  Set to a `False` value to retry on any verb.

- **status_forcelist** (*iterable*) – A set of integer HTTP status codes that we should force a retry on. A retry is initiated if the request method is in `method_whitelist` and the response status code is in `status_forcelist`.

  By default, this is disabled with `None`.

- **backoff_factor** (*float*) – A backoff factor to apply between attempts after the second try (most errors are resolved immediately by a second try without a delay). urllib3 will sleep for:

```
{backoff factor} * (2 ^ ({number of total retries} - 1))
```

  seconds. If the backoff_factor is 0.1, then `sleep()` will sleep for [0.0s, 0.2s, 0.4s, ...] between retries. It will never be longer than *Retry.BACKOFF_MAX*.

  By default, backoff is disabled (set to 0).

- **raise_on_redirect** (*bool*) – Whether, if the number of redirects is exhausted, to raise a MaxRetryError, or to return a response with a response code in the 3xx range.

- **raise_on_status** (*bool*) – Similar meaning to `raise_on_redirect`: whether we should raise an exception, or return a response, if status falls in `status_forcelist` range and retries have been exhausted.

- **history** (*tuple*) – The history of the request encountered during each call to *increment()*. The list is in the order the requests occurred. Each list item is of class *RequestHistory*.

- **respect_retry_after_header** (*bool*) – Whether to respect Retry-After header on status codes defined as *Retry.RETRY_AFTER_STATUS_CODES* or not.

BACKOFF_MAX = 120
Maximum backoff time.

DEFAULT = Retry(total=3, connect=None, read=None, redirect=None, status=None)

DEFAULT_METHOD_WHITELIST = frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE'])

RETRY_AFTER_STATUS_CODES = frozenset([503, 413, 429])

classmethod from_int(*retries*, *redirect=True*, *default=None*)
Backwards-compatibility for the old retries format.

get_backoff_time()
Formula for computing the current backoff

Return type float

get_retry_after(*response*)
Get the value of Retry-After in seconds.

increment(*method=None*, *url=None*, *response=None*, *error=None*, *_pool=None*, *_stacktrace=None*)
Return a new Retry object with incremented retry counters.

Parameters

- **response** (*HTTPResponse*) – A response object, or None, if the server did not return a response.
- **error** (*Exception*) – An error encountered during the request, or None if the response was received successfully.

Returns A new `Retry` object.

is_exhausted()
Are we out of retries?

is_retry(*method*, *status_code*, *has_retry_after=False*)
Is this method/status code retryable? (Based on whitelists and control variables such as the number of total retries to allow, whether to respect the Retry-After header, whether this header is present, and whether the returned status code is on the list of status codes to be retried upon on the presence of the aforementioned header)

new(***kw*)

parse_retry_after(*retry_after*)

sleep(*response=None*)
Sleep between retry attempts.

This method will respect a server's `Retry-After` response header and sleep the duration of the time requested. If that is not present, it will use an exponential backoff. By default, the backoff factor is 0 and this method will return immediately.

sleep_for_retry(*response=None*)

urllib3.util.timeout module

class `urllib3.util.timeout.`**Timeout**(*total=None*, *connect=<object object>*, *read=<object object>*)

Bases: `object`

Timeout configuration.

Timeouts can be defined as a default for a pool:

```
timeout = Timeout(connect=2.0, read=7.0)
http = PoolManager(timeout=timeout)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', timeout=Timeout(10))
```

Timeouts can be disabled by setting all the parameters to `None`:

```
no_timeout = Timeout(connect=None, read=None)
response = http.request('GET', 'http://example.com/, timeout=no_timeout)
```

Parameters

- **total** (*integer, float, or None*) – This combines the connect and read timeouts into one; the read timeout will be set to the time leftover from the connect attempt. In the event that both a connect timeout and a total are specified, or a read timeout and a total are specified, the shorter timeout will be applied.

  Defaults to None.

- **connect** (*integer, float, or None*) – The maximum amount of time to wait for a connection attempt to a server to succeed. Omitting the parameter will default the connect timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout for connection attempts.

- **read** (*integer, float, or None*) – The maximum amount of time to wait between consecutive read operations for a response from the server. Omitting the parameter will default the read timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout.

Note: Many factors can affect the total amount of time for urllib3 to return an HTTP response.

For example, Python's DNS resolver does not obey the timeout specified on the socket. Other factors that can affect total request time include high CPU load, high swap, the program running at a low priority level, or other behaviors.

In addition, the read and total timeouts only measure the time between read operations on the socket connecting the client and the server, not the total amount of time for the request to return a complete response. For most requests, the timeout is raised because the server has not sent the first byte in the specified time. This is not always the case; if a server streams one byte every fifteen seconds, a timeout of 20 seconds will not trigger, even though the request will take several minutes to complete.

If your goal is to cut off any request after a set amount of wall clock time, consider having a second "watcher" thread to cut off a slow request.

DEFAULT_TIMEOUT = <object object>
A sentinel object representing the default timeout value

clone()
Create a copy of the timeout object

Timeout properties are stored per-pool but each request needs a fresh Timeout object to ensure each one has its own start/stop configured.

Returns a copy of the timeout object

Return type *Timeout*

connect_timeout
Get the value to use when setting a connection timeout.

This will be a positive float or integer, the value None (never timeout), or the default system timeout.

Returns Connect timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

classmethod from_float(*timeout*)
Create a new Timeout from a legacy timeout value.

The timeout value used by httplib.py sets the same timeout on the connect(), and recv() socket requests. This creates a *Timeout* object that sets the individual timeouts to the `timeout` value passed to this function.

Parameters **timeout** (*integer, float, sentinel default object, or None*) – The legacy timeout value.

Returns Timeout object

Return type *Timeout*

get_connect_duration()
Gets the time elapsed since the call to *start_connect()*.

Returns Elapsed time.

Return type float

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to get duration for a timer that hasn't been started.

read_timeout
Get the value for the read timeout.

This assumes some time has elapsed in the connection timeout and computes the read timeout appropriately.

If self.total is set, the read timeout is dependent on the amount of time taken by the connect timeout. If the connection time has not been established, a *TimeoutStateError* will be raised.

Returns Value to use for the read timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

Raises ***urllib3.exceptions.TimeoutStateError*** – If *start_connect()* has not yet been called on this object.

start_connect()
Start the timeout clock, used during a connect() attempt

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to start a timer that has been started already.

urllib3.util.url module

class urllib3.util.url.**Url**
Bases: *urllib3.util.url.Url*

Datastructure for representing an HTTP URL. Used as a return value for *parse_url()*. Both the scheme and host are normalized as they are both case-insensitive according to RFC 3986.

hostname
For backwards-compatibility with urlparse. We're nice like that.

netloc
Network location including host and port

request_uri
Absolute path including the query string.

url
Convert self into a url

This function should more or less round-trip with *parse_url()*. The returned url may not be exactly the same as the url inputted to *parse_url()*, but it should be equivalent by the RFC (e.g., urls with a blank port will have : removed).

Example:

```
>>> U = parse_url('http://google.com/mail/')
>>> U.url
'http://google.com/mail/'
>>> Url('http', 'username:password', 'host.com', 80,
... '/path', 'query', 'fragment').url
'http://username:password@host.com:80/path?query#fragment'
```

urllib3.util.url.**get_host**(*url*)
Deprecated. Use *parse_url()* instead.

urllib3.util.url.**parse_url**(*url*)
Given a url, return a parsed *Url* namedtuple. Best-effort is performed to parse incomplete urls. Fields not provided will be None.

Partly backwards-compatible with urlparse.

Example:

```
>>> parse_url('http://google.com/mail/')
Url(scheme='http', host='google.com', port=None, path='/mail/', ...)
>>> parse_url('google.com:80')
Url(scheme=None, host='google.com', port=80, path=None, ...)
>>> parse_url('/foo?bar')
Url(scheme=None, host=None, port=None, path='/foo', query='bar', ...)
```

urllib3.util.url.**split_first**(*s*, *delims*)
Given a string and an iterable of delimiters, split on the first found delimiter. Return two split parts and the matched delimiter.

If not found, then the first part is the full input string.

Example:

```
>>> split_first('foo/bar?baz', '?/=')
('foo', 'bar?baz', '/')
```

```
>>> split_first('foo/bar?baz', '123')
('foo/bar?baz', '', None)
```

Scales linearly with number of delims. Not ideal for large number of delims.

Module contents

class urllib3.util.**SSLContext**(*protocol*)

Bases: _ssl._SSLContext

An SSLContext holds various SSL-related configuration options and data, such as certificates and possibly a private key.

load_default_certs(*purpose=_ASN1Object(nid=129, shortname='serverAuth', longname='TLS Web Server Authentication', oid='1.3.6.1.5.5.7.3.1')*)

protocol

set_alpn_protocols(*alpn_protocols*)

set_npn_protocols(*npn_protocols*)

wrap_socket(*sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None*)

class urllib3.util.**Retry**(*total=10, connect=None, read=None, redirect=None, status=None, method_whitelist=frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE']), status_forcelist=None, backoff_factor=0, raise_on_redirect=True, raise_on_status=True, history=None, respect_retry_after_header=True*)

Bases: object

Retry configuration.

Each retry attempt will create a new Retry object with updated values, so they can be safely reused.

Retries can be defined as a default for a pool:

```
retries = Retry(connect=5, read=2, redirect=5)
http = PoolManager(retries=retries)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', retries=Retry(10))
```

Retries can be disabled by passing False:

```
response = http.request('GET', 'http://example.com/', retries=False)
```

Errors will be wrapped in *MaxRetryError* unless retries are disabled, in which case the causing exception will be raised.

Parameters

- **total** (*int*) – Total number of retries to allow. Takes precedence over other counts.

  Set to None to remove this constraint and fall back on other counts. It's a good idea to set this to some sensibly-high value to account for unexpected edge cases and avoid infinite retry loops.

Set to `0` to fail on the first retry.

Set to `False` to disable and imply `raise_on_redirect=False`.

- **connect** (*int*) – How many connection-related errors to retry on.

  These are errors raised before the request is sent to the remote server, which we assume has not triggered the server to process the request.

  Set to `0` to fail on the first retry of this type.

- **read** (*int*) – How many times to retry on read errors.

  These errors are raised after the request was sent to the server, so the request may have side-effects.

  Set to `0` to fail on the first retry of this type.

- **redirect** (*int*) – How many redirects to perform. Limit this to avoid infinite redirect loops.

  A redirect is a HTTP response with a status code 301, 302, 303, 307 or 308.

  Set to `0` to fail on the first retry of this type.

  Set to `False` to disable and imply `raise_on_redirect=False`.

- **status** (*int*) – How many times to retry on bad status codes.

  These are retries made on responses, where status code matches `status_forcelist`.

  Set to `0` to fail on the first retry of this type.

- **method_whitelist** (*iterable*) – Set of uppercased HTTP method verbs that we should retry on.

  By default, we only retry on methods which are considered to be idempotent (multiple requests with the same parameters end with the same state). See *Retry.DEFAULT_METHOD_WHITELIST*.

  Set to a `False` value to retry on any verb.

- **status_forcelist** (*iterable*) – A set of integer HTTP status codes that we should force a retry on. A retry is initiated if the request method is in `method_whitelist` and the response status code is in `status_forcelist`.

  By default, this is disabled with `None`.

- **backoff_factor** (*float*) – A backoff factor to apply between attempts after the second try (most errors are resolved immediately by a second try without a delay). urllib3 will sleep for:

```
{backoff factor} * (2 ^ ({number of total retries} - 1))
```

  seconds. If the backoff_factor is 0.1, then `sleep()` will sleep for [0.0s, 0.2s, 0.4s, ...] between retries. It will never be longer than *Retry.BACKOFF_MAX*.

  By default, backoff is disabled (set to 0).

- **raise_on_redirect** (*bool*) – Whether, if the number of redirects is exhausted, to raise a MaxRetryError, or to return a response with a response code in the 3xx range.

- **raise_on_status** (*bool*) – Similar meaning to `raise_on_redirect`: whether we should raise an exception, or return a response, if status falls in `status_forcelist` range and retries have been exhausted.

- **history** (*tuple*) – The history of the request encountered during each call to *increment()*. The list is in the order the requests occurred. Each list item is of class RequestHistory.
- **respect_retry_after_header** (*bool*) – Whether to respect Retry-After header on status codes defined as *Retry.RETRY_AFTER_STATUS_CODES* or not.

BACKOFF_MAX = 120

DEFAULT = Retry(total=3, connect=None, read=None, redirect=None, status=None)

DEFAULT_METHOD_WHITELIST = frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE'])

RETRY_AFTER_STATUS_CODES = frozenset([503, 413, 429])

classmethod from_int(*retries*, *redirect=True*, *default=None*)
Backwards-compatibility for the old retries format.

get_backoff_time()
Formula for computing the current backoff

Return type float

get_retry_after(*response*)
Get the value of Retry-After in seconds.

increment(*method=None*, *url=None*, *response=None*, *error=None*, *_pool=None*, *_stacktrace=None*)
Return a new Retry object with incremented retry counters.

Parameters

- **response** (*HTTPResponse*) – A response object, or None, if the server did not return a response.
- **error** (*Exception*) – An error encountered during the request, or None if the response was received successfully.

Returns A new Retry object.

is_exhausted()
Are we out of retries?

is_retry(*method*, *status_code*, *has_retry_after=False*)
Is this method/status code retryable? (Based on whitelists and control variables such as the number of total retries to allow, whether to respect the Retry-After header, whether this header is present, and whether the returned status code is on the list of status codes to be retried upon on the presence of the aforementioned header)

new(***kw*)

parse_retry_after(*retry_after*)

sleep(*response=None*)
Sleep between retry attempts.

This method will respect a server's Retry-After response header and sleep the duration of the time requested. If that is not present, it will use an exponential backoff. By default, the backoff factor is 0 and this method will return immediately.

sleep_for_retry(*response=None*)

class urllib3.util.**Timeout**(*total=None*, *connect=<object object>*, *read=<object object>*)
Bases: object

Timeout configuration.

Timeouts can be defined as a default for a pool:

```
timeout = Timeout(connect=2.0, read=7.0)
http = PoolManager(timeout=timeout)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', timeout=Timeout(10))
```

Timeouts can be disabled by setting all the parameters to `None`:

```
no_timeout = Timeout(connect=None, read=None)
response = http.request('GET', 'http://example.com/, timeout=no_timeout)
```

Parameters

- **total** (*integer, float, or None*) – This combines the connect and read timeouts into one; the read timeout will be set to the time leftover from the connect attempt. In the event that both a connect timeout and a total are specified, or a read timeout and a total are specified, the shorter timeout will be applied.

  Defaults to None.

- **connect** (*integer, float, or None*) – The maximum amount of time to wait for a connection attempt to a server to succeed. Omitting the parameter will default the connect timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout for connection attempts.

- **read** (*integer, float, or None*) – The maximum amount of time to wait between consecutive read operations for a response from the server. Omitting the parameter will default the read timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout.

Note: Many factors can affect the total amount of time for urllib3 to return an HTTP response.

For example, Python's DNS resolver does not obey the timeout specified on the socket. Other factors that can affect total request time include high CPU load, high swap, the program running at a low priority level, or other behaviors.

In addition, the read and total timeouts only measure the time between read operations on the socket connecting the client and the server, not the total amount of time for the request to return a complete response. For most requests, the timeout is raised because the server has not sent the first byte in the specified time. This is not always the case; if a server streams one byte every fifteen seconds, a timeout of 20 seconds will not trigger, even though the request will take several minutes to complete.

If your goal is to cut off any request after a set amount of wall clock time, consider having a second "watcher" thread to cut off a slow request.

DEFAULT_TIMEOUT = <object object>

clone()

Create a copy of the timeout object

Timeout properties are stored per-pool but each request needs a fresh Timeout object to ensure each one has its own start/stop configured.

Returns a copy of the timeout object

Return type *Timeout*

connect_timeout
Get the value to use when setting a connection timeout.

This will be a positive float or integer, the value None (never timeout), or the default system timeout.

Returns Connect timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

classmethod from_float(*timeout*)
Create a new Timeout from a legacy timeout value.

The timeout value used by httplib.py sets the same timeout on the connect(), and recv() socket requests. This creates a *Timeout* object that sets the individual timeouts to the `timeout` value passed to this function.

Parameters **timeout** (*integer, float, sentinel default object, or None*) – The legacy timeout value.

Returns Timeout object

Return type *Timeout*

get_connect_duration()
Gets the time elapsed since the call to *start_connect()*.

Returns Elapsed time.

Return type float

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to get duration for a timer that hasn't been started.

read_timeout
Get the value for the read timeout.

This assumes some time has elapsed in the connection timeout and computes the read timeout appropriately.

If self.total is set, the read timeout is dependent on the amount of time taken by the connect timeout. If the connection time has not been established, a *TimeoutStateError* will be raised.

Returns Value to use for the read timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

Raises ***urllib3.exceptions.TimeoutStateError*** – If *start_connect()* has not yet been called on this object.

start_connect()
Start the timeout clock, used during a connect() attempt

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to start a timer that has been started already.

class urllib3.util.**Url**
Bases: *urllib3.util.url.Url*

Datastructure for representing an HTTP URL. Used as a return value for *parse_url()*. Both the scheme and host are normalized as they are both case-insensitive according to RFC 3986.

hostname
For backwards-compatibility with urlparse. We're nice like that.

netloc
Network location including host and port

request_uri
Absolute path including the query string.

url
Convert self into a url

This function should more or less round-trip with *parse_url()*. The returned url may not be exactly the same as the url inputted to *parse_url()*, but it should be equivalent by the RFC (e.g., urls with a blank port will have : removed).

Example:

```
>>> U = parse_url('http://google.com/mail/')
>>> U.url
'http://google.com/mail/'
>>> Url('http', 'username:password', 'host.com', 80,
... '/path', 'query', 'fragment').url
'http://username:password@host.com:80/path?query#fragment'
```

urllib3.util.**assert_fingerprint**(*cert*, *fingerprint*)
Checks if given fingerprint matches the supplied certificate.

Parameters

- **cert** – Certificate as bytes object.
- **fingerprint** – Fingerprint as string of hexdigits, can be interspersed by colons.

urllib3.util.**current_time**()
time() -> floating point number

Return the current time in seconds since the Epoch. Fractions of a second may be present if the system clock provides them.

urllib3.util.**is_connection_dropped**(*conn*)
Returns True if the connection is dropped and should be closed.

Parameters **conn** – `httplib.HTTPConnection` object.

Note: For platforms like AppEngine, this will always return `False` to let the platform handle connection recycling transparently for us.

urllib3.util.**is_fp_closed**(*obj*)
Checks whether a given file-like object is closed.

Parameters **obj** – The file-like object to check.

urllib3.util.**get_host**(*url*)
Deprecated. Use *parse_url()* instead.

urllib3.util.**parse_url**(*url*)
Given a url, return a parsed *Url* namedtuple. Best-effort is performed to parse incomplete urls. Fields not provided will be None.

Partly backwards-compatible with `urlparse`.

Example:

```
>>> parse_url('http://google.com/mail/')
Url(scheme='http', host='google.com', port=None, path='/mail/', ...)
>>> parse_url('google.com:80')
```

```
Url(scheme=None, host='google.com', port=80, path=None, ...)
>>> parse_url('/foo?bar')
Url(scheme=None, host=None, port=None, path='/foo', query='bar', ...)
```

urllib3.util.**make_headers**(*keep_alive=None*, *accept_encoding=None*, *user_agent=None*, *basic_auth=None*, *proxy_basic_auth=None*, *disable_cache=None*)

Shortcuts for generating request headers.

Parameters

- **keep_alive** – If `True`, adds 'connection: keep-alive' header.
- **accept_encoding** – Can be a boolean, list, or string. `True` translates to 'gzip,deflate'. List will get joined by comma. String will be used as provided.
- **user_agent** – String representing the user-agent you want, such as "python-urllib3/0.6"
- **basic_auth** – Colon-separated username:password string for 'authorization: basic ...' auth header.
- **proxy_basic_auth** – Colon-separated username:password string for 'proxy-authorization: basic ...' auth header.
- **disable_cache** – If `True`, adds 'cache-control: no-cache' header.

Example:

```
>>> make_headers(keep_alive=True, user_agent="Batman/1.0")
{'connection': 'keep-alive', 'user-agent': 'Batman/1.0'}
>>> make_headers(accept_encoding=True)
{'accept-encoding': 'gzip,deflate'}
```

urllib3.util.**resolve_cert_reqs**(*candidate*)

Resolves the argument to a numeric constant, which can be passed to the wrap_socket function/method from the ssl module. Defaults to `ssl.CERT_NONE`. If given a string it is assumed to be the name of the constant in the `ssl` module or its abbrevation. (So you can specify *REQUIRED* instead of *CERT_REQUIRED*. If it's neither *None* nor a string we assume it is already the numeric constant which can directly be passed to wrap_socket.

urllib3.util.**resolve_ssl_version**(*candidate*)

like resolve_cert_reqs

urllib3.util.**split_first**(*s*, *delims*)

Given a string and an iterable of delimiters, split on the first found delimiter. Return two split parts and the matched delimiter.

If not found, then the first part is the full input string.

Example:

```
>>> split_first('foo/bar?baz', '?/=')
('foo', 'bar?baz', '/')
>>> split_first('foo/bar?baz', '123')
('foo/bar?baz', '', None)
```

Scales linearly with number of delims. Not ideal for large number of delims.

urllib3.util.**ssl_wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *cert_reqs=None*, *ca_certs=None*, *server_hostname=None*, *ssl_version=None*, *ciphers=None*, *ssl_context=None*, *ca_cert_dir=None*)

All arguments except for server_hostname, ssl_context, and ca_cert_dir have the same meaning as they do when using `ssl.wrap_socket()`.

Parameters

- **server_hostname** – When SNI is supported, the expected hostname of the certificate
- **ssl_context** – A pre-made *SSLContext* object. If none is provided, one will be created using `create_urllib3_context()`.
- **ciphers** – A string of ciphers we wish the client to support. This is not supported on Python 2.6 as the ssl module does not support it.
- **ca_cert_dir** – A directory containing CA certificates in multiple separate files, as supported by OpenSSL's -CApath flag or the capath argument to SSLContext.load_verify_locations().

`urllib3.util.`**wait_for_read**(*socks*, *timeout=None*)
Waits for reading to be available from a list of sockets or optionally a single socket if passed in. Returns a list of sockets that can be read from immediately.

`urllib3.util.`**wait_for_write**(*socks*, *timeout=None*)
Waits for writing to be available from a list of sockets or optionally a single socket if passed in. Returns a list of sockets that can be written to immediately.

Submodules

urllib3.connection module

exception `urllib3.connection.`**ConnectionError**
Bases: `exceptions.Exception`

class `urllib3.connection.`**DummyConnection**
Bases: `object`

Used to detect a failed ConnectionCls import.

class `urllib3.connection.`**HTTPConnection**(**args*, ***kw*)
Bases: `httplib.HTTPConnection`, `object`

Based on httplib.HTTPConnection but provides an extra constructor backwards-compatibility layer between older and newer Pythons.

Additional keyword parameters are used to configure attributes of the connection. Accepted parameters include:

- `strict`: See the documentation on *urllib3.connectionpool.HTTPConnectionPool*
- `source_address`: Set the source address for the current connection.

> **Note:** This is ignored for Python 2.6. It is only applied for 2.7 and 3.x

- `socket_options`: Set specific options on the underlying socket. If not specified, then defaults are loaded from `HTTPConnection.default_socket_options` which includes disabling Nagle's algorithm (sets TCP_NODELAY to 1) unless the connection is behind a proxy.

For example, if you wish to enable TCP Keep Alive in addition to the defaults, you might pass:

```
HTTPConnection.default_socket_options + [
    (socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1),
]
```

Or you may want to disable the defaults by passing an empty list (e.g., `[]`).

connect()

default_port = 80

default_socket_options = [(6, 1, 1)]
Disable Nagle's algorithm by default. `[(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)]`

is_verified = False
Whether this connection verifies the host's certificate.

request_chunked(*method*, *url*, *body=None*, *headers=None*)
Alternative to the common request method, which sends the body with chunked encoding and not as one block

socket_options = None
The socket options provided by the user. If no options are provided, we use the default options.

urllib3.connection.**HTTPSConnection**
alias of *VerifiedHTTPSConnection*

urllib3.connection.**UnverifiedHTTPSConnection**
alias of *HTTPSConnection*

class urllib3.connection.**VerifiedHTTPSConnection**(*host*, *port=None*, *key_file=None*, *cert_file=None*, *strict=None*, *timeout=<object object>*, *ssl_context=None*, ***kw*)
Bases: *urllib3.connection.HTTPSConnection*

Based on httplib.HTTPSConnection but wraps the socket with SSL certification.

assert_fingerprint = None

ca_cert_dir = None

ca_certs = None

cert_reqs = None

connect()

set_cert(*key_file=None*, *cert_file=None*, *cert_reqs=None*, *ca_certs=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*)
This method should only be called once, before the connection is used.

ssl_version = None

urllib3.connectionpool module

class urllib3.connectionpool.**ConnectionPool**(*host*, *port=None*)
Bases: `object`

Base class for all connection pools, such as *HTTPConnectionPool* and *HTTPSConnectionPool*.

QueueCls
alias of `LifoQueue`

close()
Close all pooled connections and disable the pool.

scheme = None

class urllib3.connectionpool.**HTTPConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, ***conn_kw*)

Bases: *urllib3.connectionpool.ConnectionPool*, *urllib3.request.RequestMethods*

Thread-safe connection pool for one host.

Parameters

- **host** – Host used for this HTTP Connection (e.g. "localhost"), passed into `httplib.HTTPConnection`.
- **port** – Port used for this HTTP Connection (None is equivalent to 80), passed into `httplib.HTTPConnection`.
- **strict** – Causes BadStatusLine to be raised if the status line can't be parsed as a valid HTTP/1.0 or 1.1 status line, passed into `httplib.HTTPConnection`.

  Note: Only works in Python 2. This parameter is ignored in Python 3.

- **timeout** – Socket timeout in seconds for each individual connection. This can be a float or integer, which sets the timeout for the HTTP request, or an instance of *urllib3.util.Timeout* which gives you more fine-grained control over request timeouts. After the constructor has been parsed, this is always a *urllib3.util.Timeout* object.
- **maxsize** – Number of connections to save that can be reused. More than 1 is useful in multithreaded situations. If `block` is set to False, more connections will be created but they will not be saved once they've been used.
- **block** – If set to True, no more than `maxsize` connections will be used at a time. When no free connections are available, the call will block until a connection has been released. This is a useful side effect for particular multithreaded situations where one does not want to use more than maxsize connections per host to prevent flooding.
- **headers** – Headers to include with all requests, unless other headers are given explicitly.
- **retries** – Retry configuration to use by default with requests in this pool.
- **_proxy** – Parsed proxy URL, should not be used directly, instead, see `urllib3.connectionpool.ProxyManager`"
- **_proxy_headers** – A dictionary with proxy headers, should not be used directly, instead, see `urllib3.connectionpool.ProxyManager`"
- ****conn_kw** – Additional parameters are used to create fresh *urllib3.connection.HTTPConnection*, *urllib3.connection.HTTPSConnection* instances.

ConnectionCls
alias of `HTTPConnection`

ResponseCls
alias of `HTTPResponse`

close()
Close all pooled connections and disable the pool.

is_same_host(*url*)
Check if the given `url` is a member of the same host as this connection pool.

scheme = 'http'

urlopen(*method*, *url*, *body=None*, *headers=None*, *retries=None*, *redirect=True*, *assert_same_host=True*, *timeout=<object object>*, *pool_timeout=None*, *release_conn=None*, *chunked=False*, *body_pos=None*, ***response_kw*)

Get a connection from the pool and perform an HTTP request. This is the lowest level call for making a request, so you'll need to specify all the raw details.

Note: More commonly, it's appropriate to use a convenience method provided by `RequestMethods`, such as `request()`.

Note: *release_conn* will only behave as expected if *preload_content=False* because we want to make *preload_content=False* the default behaviour someday soon without breaking backwards compatibility.

Parameters

- **method** – HTTP request method (such as GET, POST, PUT, etc.)
- **body** – Data to send in the request body (useful for creating POST requests, see HTTPConnectionPool.post_url for more convenience).
- **headers** – Dictionary of custom headers to send, such as User-Agent, If-None-Match, etc. If None, pool headers are used. If provided, these headers completely replace any pool-specific headers.
- **retries** (`Retry`, False, or an int.) – Configure the number of retries to allow before raising a `MaxRetryError` exception.

  Pass `None` to retry until you receive a response. Pass a `Retry` object for fine-grained control over different types of retries. Pass an integer number to retry connection errors that many times, but no other types of errors. Pass zero to never retry.

  If `False`, then retries are disabled and any exception is raised immediately. Also, instead of raising a MaxRetryError on redirects, the redirect response will be returned.
- **redirect** – If True, automatically handle redirects (status codes 301, 302, 303, 307, 308). Each redirect counts as a retry. Disabling retries will disable redirect, too.
- **assert_same_host** – If `True`, will make sure that the host of the pool requests is consistent else will raise HostChangedError. When False, you can use the pool on an HTTP proxy and request foreign hosts.
- **timeout** – If specified, overrides the default timeout for this one request. It may be a float (in seconds) or an instance of `urllib3.util.Timeout`.
- **pool_timeout** – If set and the pool is set to block=True, then this method will block for `pool_timeout` seconds and raise EmptyPoolError if no connection is available within the time period.
- **release_conn** – If False, then the urlopen call will not release the connection back into the pool once a response is received (but will release if you read the entire contents of the response such as when *preload_content=True*). This is useful if you're not preloading the response's content immediately. You will need to call `r.release_conn()` on the response `r` to return the connection back into the pool. If None, it takes the value of `response_kw.get('preload_content', True)`.
- **chunked** – If True, urllib3 will send the body using chunked transfer encoding. Otherwise, urllib3 will send the body using the standard content-length form. Defaults to False.

- **body_pos** (*int*) – Position to seek to in file-like body in the event of a retry or redirect. Typically this won't need to be set because urllib3 will auto-populate the value when needed.
- ****response_kw** – Additional parameters are passed to *urllib3.response.HTTPResponse.from_httplib()*

class urllib3.connectionpool.**HTTPSConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, *key_file=None*, *cert_file=None*, *cert_reqs=None*, *ca_certs=None*, *ssl_version=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*, ***conn_kw*)

Bases: *urllib3.connectionpool.HTTPConnectionPool*

Same as *HTTPConnectionPool*, but HTTPS.

When Python is compiled with the `ssl` module, then *VerifiedHTTPSConnection* is used, which *can* verify certificates, instead of `HTTPSConnection`.

VerifiedHTTPSConnection uses one of `assert_fingerprint`, `assert_hostname` and `host` in this order to verify connections. If `assert_hostname` is False, no verification is done.

The `key_file`, `cert_file`, `cert_reqs`, `ca_certs`, `ca_cert_dir`, and `ssl_version` are only used if `ssl` is available and are fed into *urllib3.util.ssl_wrap_socket()* to upgrade the connection socket into an SSL socket.

ConnectionCls
alias of `VerifiedHTTPSConnection`

scheme = 'https'

urllib3.connectionpool.**connection_from_url**(*url*, ***kw*)

Given a url, return an *ConnectionPool* instance of its host.

This is a shortcut for not having to parse out the scheme, host, and port of the url before creating an *ConnectionPool* instance.

Parameters

- **url** – Absolute URL string that must include the scheme. Port is optional.
- ****kw** – Passes additional parameters to the constructor of the appropriate *ConnectionPool*. Useful for specifying things like timeout, maxsize, headers, etc.

Example:

```
>>> conn = connection_from_url('http://google.com/')
>>> r = conn.request('GET', '/')
```

urllib3.exceptions module

exception urllib3.exceptions.**BodyNotHttplibCompatible**

Bases: *urllib3.exceptions.HTTPError*

Body should be httplib.HTTPResponse like (have an fp attribute which returns raw chunks) for read_chunked().

exception urllib3.exceptions.**ClosedPoolError**(*pool*, *message*)
Bases: *urllib3.exceptions.PoolError*

Raised when a request enters a pool after the pool has been closed.

exception urllib3.exceptions.**ConnectTimeoutError**
Bases: *urllib3.exceptions.TimeoutError*

Raised when a socket timeout occurs while connecting to a server

urllib3.exceptions.**ConnectionError**
Renamed to ProtocolError but aliased for backwards compatibility.

alias of *ProtocolError*

exception urllib3.exceptions.**DecodeError**
Bases: *urllib3.exceptions.HTTPError*

Raised when automatic decoding based on Content-Type fails.

exception urllib3.exceptions.**DependencyWarning**
Bases: *urllib3.exceptions.HTTPWarning*

Warned when an attempt is made to import a module with missing optional dependencies.

exception urllib3.exceptions.**EmptyPoolError**(*pool*, *message*)
Bases: *urllib3.exceptions.PoolError*

Raised when a pool runs out of connections and no more are allowed.

exception urllib3.exceptions.**HTTPError**
Bases: exceptions.Exception

Base exception used by this module.

exception urllib3.exceptions.**HTTPWarning**
Bases: exceptions.Warning

Base warning used by this module.

exception urllib3.exceptions.**HeaderParsingError**(*defects*, *unparsed_data*)
Bases: *urllib3.exceptions.HTTPError*

Raised by assert_header_parsing, but we convert it to a log.warning statement.

exception urllib3.exceptions.**HostChangedError**(*pool*, *url*, *retries=3*)
Bases: *urllib3.exceptions.RequestError*

Raised when an existing pool gets a request for a foreign host.

exception urllib3.exceptions.**IncompleteRead**(*partial*, *expected*)
Bases: *urllib3.exceptions.HTTPError*, httplib.IncompleteRead

Response length doesn't match expected Content-Length

Subclass of http_client.IncompleteRead to allow int value for *partial* to avoid creating large objects on streamed reads.

exception urllib3.exceptions.**InsecurePlatformWarning**
Bases: *urllib3.exceptions.SecurityWarning*

Warned when certain SSL configuration is not available on a platform.

exception urllib3.exceptions.**InsecureRequestWarning**
Bases: *urllib3.exceptions.SecurityWarning*

Warned when making an unverified HTTPS request.

exception urllib3.exceptions.**InvalidHeader**
Bases: *urllib3.exceptions.HTTPError*

The header provided was somehow invalid.

exception urllib3.exceptions.**LocationParseError**(*location*)
Bases: *urllib3.exceptions.LocationValueError*

Raised when get_host or similar fails to parse the URL input.

exception urllib3.exceptions.**LocationValueError**
Bases: exceptions.ValueError, *urllib3.exceptions.HTTPError*

Raised when there is something wrong with a given URL input.

exception urllib3.exceptions.**MaxRetryError**(*pool*, *url*, *reason=None*)
Bases: *urllib3.exceptions.RequestError*

Raised when the maximum number of retries is exceeded.

Parameters

- **pool** (*HTTPConnectionPool*) – The connection pool
- **url** (*string*) – The requested Url
- **reason** (*exceptions.Exception*) – The underlying error

exception urllib3.exceptions.**NewConnectionError**(*pool*, *message*)
Bases: *urllib3.exceptions.ConnectTimeoutError*, *urllib3.exceptions.PoolError*

Raised when we fail to establish a new connection. Usually ECONNREFUSED.

exception urllib3.exceptions.**PoolError**(*pool*, *message*)
Bases: *urllib3.exceptions.HTTPError*

Base exception for errors caused within a pool.

exception urllib3.exceptions.**ProtocolError**
Bases: *urllib3.exceptions.HTTPError*

Raised when something unexpected happens mid-request/response.

exception urllib3.exceptions.**ProxyError**
Bases: *urllib3.exceptions.HTTPError*

Raised when the connection to a proxy fails.

exception urllib3.exceptions.**ProxySchemeUnknown**(*scheme*)
Bases: exceptions.AssertionError, exceptions.ValueError

ProxyManager does not support the supplied scheme

exception urllib3.exceptions.**ReadTimeoutError**(*pool*, *url*, *message*)
Bases: *urllib3.exceptions.TimeoutError*, *urllib3.exceptions.RequestError*

Raised when a socket timeout occurs while receiving data from a server

exception urllib3.exceptions.**RequestError**(*pool*, *url*, *message*)
Bases: *urllib3.exceptions.PoolError*

Base exception for PoolErrors that have associated URLs.

exception urllib3.exceptions.**ResponseError**
Bases: *urllib3.exceptions.HTTPError*

Used as a container for an error reason supplied in a MaxRetryError.

GENERIC_ERROR = 'too many error responses'

SPECIFIC_ERROR = 'too many {status_code} error responses'

exception urllib3.exceptions.**ResponseNotChunked**
Bases: *urllib3.exceptions.ProtocolError*, exceptions.ValueError

Response needs to be chunked in order to read it as chunks.

exception urllib3.exceptions.**SNIMissingWarning**
Bases: *urllib3.exceptions.HTTPWarning*

Warned when making a HTTPS request without SNI available.

exception urllib3.exceptions.**SSLError**
Bases: *urllib3.exceptions.HTTPError*

Raised when SSL certificate fails in an HTTPS connection.

exception urllib3.exceptions.**SecurityWarning**
Bases: *urllib3.exceptions.HTTPWarning*

Warned when perfoming security reducing actions

exception urllib3.exceptions.**SubjectAltNameWarning**
Bases: *urllib3.exceptions.SecurityWarning*

Warned when connecting to a host with a certificate missing a SAN.

exception urllib3.exceptions.**SystemTimeWarning**
Bases: *urllib3.exceptions.SecurityWarning*

Warned when system time is suspected to be wrong

exception urllib3.exceptions.**TimeoutError**
Bases: *urllib3.exceptions.HTTPError*

Raised when a socket timeout error occurs.

Catching this error will catch both *ReadTimeoutErrors* and *ConnectTimeoutErrors*.

exception urllib3.exceptions.**TimeoutStateError**
Bases: *urllib3.exceptions.HTTPError*

Raised when passing an invalid state to a timeout

exception urllib3.exceptions.**UnrewindableBodyError**
Bases: *urllib3.exceptions.HTTPError*

urllib3 encountered an error when trying to rewind a body

urllib3.fields module

class urllib3.fields.**RequestField**(*name*, *data*, *filename=None*, *headers=None*)
Bases: object

A data container for request body parameters.

Parameters

- **name** – The name of this request field.
- **data** – The data/value body.
- **filename** – An optional filename of the request field.
- **headers** – An optional dict-like object of headers to initially use for the field.

classmethod from_tuples(*fieldname*, *value*)
A *RequestField* factory from old-style tuple parameters.

Supports constructing *RequestField* from parameter of key/value strings AND key/filetuple. A filetuple is a (filename, data, MIME type) tuple where the MIME type is optional. For example:

```
'foo': 'bar',
'fakefile': ('foofile.txt', 'contents of foofile'),
'realfile': ('barfile.txt', open('realfile').read()),
'typedfile': ('bazfile.bin', open('bazfile').read(), 'image/jpeg'),
'nonamefile': 'contents of nonamefile field',
```

Field names and filenames must be unicode.

make_multipart(*content_disposition=None*, *content_type=None*, *content_location=None*)
Makes this request field into a multipart request field.

This method overrides "Content-Disposition", "Content-Type" and "Content-Location" headers to the request parameter.

Parameters

- **content_type** – The 'Content-Type' of the request body.
- **content_location** – The 'Content-Location' of the request body.

render_headers()
Renders the headers for this request field.

urllib3.fields.**format_header_param**(*name*, *value*)
Helper function to format and quote a single header parameter.

Particularly useful for header parameters which might contain non-ASCII values, like file names. This follows RFC 2231, as suggested by RFC 2388 Section 4.4.

Parameters

- **name** – The name of the parameter, a string expected to be ASCII only.
- **value** – The value of the parameter, provided as a unicode string.

urllib3.fields.**guess_content_type**(*filename*, *default='application/octet-stream'*)
Guess the "Content-Type" of a file.

Parameters

- **filename** – The filename to guess the "Content-Type" of using mimetypes.
- **default** – If no "Content-Type" can be guessed, default to *default*.

urllib3.filepost module

urllib3.filepost.**choose_boundary**()
Our embarrassingly-simple replacement for mimetools.choose_boundary.

urllib3.filepost.**encode_multipart_formdata**(*fields*, *boundary=None*)
Encode a dictionary of `fields` using the multipart/form-data MIME format.

Parameters

- **fields** – Dictionary of fields or list of (key, *RequestField*).
- **boundary** – If not specified, then a random boundary will be generated using `mimetools.choose_boundary()`.

urllib3.filepost.**iter_field_objects**(*fields*)
Iterate over fields.

Supports list of (k, v) tuples and dicts, and lists of *RequestField*.

urllib3.filepost.**iter_fields**(*fields*)
Deprecated since version 1.6.

Iterate over fields.

The addition of *RequestField* makes this function obsolete. Instead, use *iter_field_objects()*, which returns *RequestField* objects.

Supports list of (k, v) tuples and dicts.

urllib3.poolmanager module

class urllib3.poolmanager.**PoolManager**(*num_pools=10*, *headers=None*, ***connection_pool_kw*)
Bases: *urllib3.request.RequestMethods*

Allows for arbitrary requests while transparently keeping track of necessary connection pools for you.

Parameters

- **num_pools** – Number of connection pools to cache before discarding the least recently used pool.
- **headers** – Headers to include with all requests, unless other headers are given explicitly.
- ****connection_pool_kw** – Additional parameters are used to create fresh *urllib3.connectionpool.ConnectionPool* instances.

Example:

```
>>> manager = PoolManager(num_pools=2)
>>> r = manager.request('GET', 'http://google.com/')
>>> r = manager.request('GET', 'http://google.com/mail')
>>> r = manager.request('GET', 'http://yahoo.com/')
>>> len(manager.pools)
2
```

clear()
Empty our store of pools and direct them all to close.

This will not affect in-flight connections, but they will not be re-used after completion.

connection_from_context(*request_context*)
Get a `ConnectionPool` based on the request context.

`request_context` must at least contain the `scheme` key and its value must be a key in `key_fn_by_scheme` instance variable.

connection_from_host(*host*, *port=None*, *scheme='http'*, *pool_kwargs=None*)
Get a `ConnectionPool` based on the host, port, and scheme.

If `port` isn't given, it will be derived from the `scheme` using `urllib3.connectionpool.port_by_scheme`. If `pool_kwargs` is provided, it is merged with the instance's `connection_pool_kw` variable and used to create the new connection pool, if one is needed.

connection_from_pool_key(*pool_key*, *request_context=None*)
Get a `ConnectionPool` based on the provided pool key.

`pool_key` should be a namedtuple that only contains immutable objects. At a minimum it must have the `scheme`, `host`, and `port` fields.

connection_from_url(*url*, *pool_kwargs=None*)
Similar to *urllib3.connectionpool.connection_from_url()*.

If `pool_kwargs` is not provided and a new pool needs to be constructed, `self.connection_pool_kw` is used to initialize the *urllib3.connectionpool.ConnectionPool*. If `pool_kwargs` is provided, it is used instead. Note that if a new pool does not need to be created for the request, the provided `pool_kwargs` are not used.

proxy = None

urlopen(*method*, *url*, *redirect=True*, ***kw*)
Same as *urllib3.connectionpool.HTTPConnectionPool.urlopen()* with custom cross-host redirect logic and only sends the request-uri portion of the `url`.

The given `url` parameter must be absolute, such that an appropriate *urllib3.connectionpool.ConnectionPool* can be chosen for it.

class urllib3.poolmanager.**ProxyManager**(*proxy_url*, *num_pools=10*, *headers=None*, *proxy_headers=None*, ***connection_pool_kw*)
Bases: *urllib3.poolmanager.PoolManager*

Behaves just like *PoolManager*, but sends all requests through the defined proxy, using the CONNECT method for HTTPS URLs.

Parameters

- **proxy_url** – The URL of the proxy to be used.
- **proxy_headers** – A dictionary contaning headers that will be sent to the proxy. In case of HTTP they are being sent with each request, while in the HTTPS/CONNECT case they are sent only once. Could be used for proxy authentication.

Example:

```
>>> proxy = urllib3.ProxyManager('http://localhost:3128/')
>>> r1 = proxy.request('GET', 'http://google.com/')
>>> r2 = proxy.request('GET', 'http://httpbin.org/')
>>> len(proxy.pools)
1
>>> r3 = proxy.request('GET', 'https://httpbin.org/')
>>> r4 = proxy.request('GET', 'https://twitter.com/')
>>> len(proxy.pools)
3
```

connection_from_host(*host*, *port=None*, *scheme='http'*, *pool_kwargs=None*)

urlopen(*method*, *url*, *redirect=True*, ***kw*)
Same as HTTP(S)ConnectionPool.urlopen, `url` must be absolute.

urllib3.poolmanager.**proxy_from_url**(*url*, ***kw*)

urllib3.request module

class urllib3.request.**RequestMethods**(*headers=None*)

Bases: object

Convenience mixin for classes who implement a *urlopen()* method, such as *HTTPConnectionPool* and *PoolManager*.

Provides behavior for making common types of HTTP request methods and decides which type of request field encoding to use.

Specifically,

request_encode_url() is for sending requests whose fields are encoded in the URL (such as GET, HEAD, DELETE).

request_encode_body() is for sending requests whose fields are encoded in the *body* of the request using multipart or www-form-urlencoded (such as for POST, PUT, PATCH).

request() is for making any kind of request, it will look up the appropriate encoding format and use one of the above two methods to make the request.

Initializer parameters:

Parameters **headers** – Headers to include with all requests, unless other headers are given explicitly.

request(*method*, *url*, *fields=None*, *headers=None*, ***urlopen_kw*)

Make a request using *urlopen()* with the appropriate encoding of `fields` based on the `method` used.

This is a convenience method that requires the least amount of manual effort. It can be used in most situations, while still having the option to drop down to more specific methods when necessary, such as *request_encode_url()*, *request_encode_body()*, or even the lowest level *urlopen()*.

request_encode_body(*method*, *url*, *fields=None*, *headers=None*, *encode_multipart=True*, *multipart_boundary=None*, ***urlopen_kw*)

Make a request using *urlopen()* with the `fields` encoded in the body. This is useful for request methods like POST, PUT, PATCH, etc.

When `encode_multipart=True` (default), then *urllib3.filepost.encode_multipart_formdata()* is used to encode the payload with the appropriate content type. Otherwise `urllib.urlencode()` is used with the 'application/x-www-form-urlencoded' content type.

Multipart encoding must be used when posting files, and it's reasonably safe to use it in other times too. However, it may break request signing, such as with OAuth.

Supports an optional `fields` parameter of key/value strings AND key/filetuple. A filetuple is a (filename, data, MIME type) tuple where the MIME type is optional. For example:

```
fields = {
    'foo': 'bar',
    'fakefile': ('foofile.txt', 'contents of foofile'),
    'realfile': ('barfile.txt', open('realfile').read()),
    'typedfile': ('bazfile.bin', open('bazfile').read(),
                  'image/jpeg'),
    'nonamefile': 'contents of nonamefile field',
}
```

When uploading a file, providing a filename (the first parameter of the tuple) is optional but recommended to best mimick behavior of browsers.

Note that if `headers` are supplied, the 'Content-Type' header will be overwritten because it depends on the dynamic random boundary string which is used to compose the body of the request. The random boundary string can be explicitly set with the `multipart_boundary` parameter.

request_encode_url(*method*, *url*, *fields=None*, *headers=None*, ***urlopen_kw*)
Make a request using *urlopen()* with the `fields` encoded in the url. This is useful for request methods like GET, HEAD, DELETE, etc.

urlopen(*method*, *url*, *body=None*, *headers=None*, *encode_multipart=True*, *multipart_boundary=None*, ***kw*)

urllib3.response module

class urllib3.response.**DeflateDecoder**
Bases: object

decompress(*data*)

class urllib3.response.**GzipDecoder**
Bases: object

decompress(*data*)

class urllib3.response.**HTTPResponse**(*body=''*, *headers=None*, *status=0*, *version=0*, *reason=None*, *strict=0*, *preload_content=True*, *decode_content=True*, *original_response=None*, *pool=None*, *connection=None*, *retries=None*, *enforce_content_length=False*, *request_method=None*)
Bases: io.IOBase

HTTP Response container.

Backwards-compatible to httplib's HTTPResponse but the response `body` is loaded and decoded on-demand when the `data` property is accessed. This class is also compatible with the Python standard library's `io` module, and can hence be treated as a readable object in the context of that framework.

Extra parameters for behaviour not present in httplib.HTTPResponse:

Parameters

- **preload_content** – If True, the response's body will be preloaded during construction.
- **decode_content** – If True, attempts to decode specific content-encoding's based on headers (like 'gzip' and 'deflate') will be skipped and raw data will be used instead.
- **original_response** – When this HTTPResponse wrapper is generated from an httplib.HTTPResponse object, it's convenient to include the original for debug purposes. It's otherwise unused.
- **retries** – The retries contains the last *Retry* that was used during the request.
- **enforce_content_length** – Enforce content length checking. Body returned by server must match value of Content-Length header, if present. Otherwise, raise error.

CONTENT_DECODERS = ['gzip', 'deflate']

REDIRECT_STATUSES = [301, 302, 303, 307, 308]

close()

closed

connection

data

fileno()

flush()

classmethod from_httplib(*ResponseCls*, *r*, ***response_kw*)
Given an `httplib.HTTPResponse` instance `r`, return a corresponding *urllib3.response.HTTPResponse* object.

Remaining parameters are passed to the HTTPResponse constructor, along with `original_response=r`.

get_redirect_location()
Should we redirect and where to?

Returns Truthy redirect location string if we got a redirect status code and valid location. `None` if redirect status and no location. `False` if not a redirect status code.

getheader(*name*, *default=None*)

getheaders()

info()

read(*amt=None*, *decode_content=None*, *cache_content=False*)
Similar to `httplib.HTTPResponse.read()`, but with two additional parameters: `decode_content` and `cache_content`.

Parameters

- **amt** – How much of the content to read. If specified, caching is skipped because it doesn't make sense to cache partial content as the full response.
- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.
- **cache_content** – If True, will save the returned data such that the same result is returned despite of the state of the underlying file object. This is useful if you want the `.data` property to continue working after having `.read()` the file object. (Overridden if `amt` is set.)

read_chunked(*amt=None*, *decode_content=None*)
Similar to *HTTPResponse.read()*, but with an additional parameter: `decode_content`.

Parameters **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.

readable()

readinto(*b*)

release_conn()

stream(*amt=65536*, *decode_content=None*)
A generator wrapper for the read() method. A call will block until `amt` bytes have been read from the connection or until the connection is closed.

Parameters

- **amt** – How much of the content to read. The generator will return up to much data per iteration, but may return less. This is particularly likely when using compressed data. However, the empty string will never be returned.
- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.

supports_chunked_reads()
Checks if the underlying file-like object looks like a httplib.HTTPResponse object. We do this by testing for the fp attribute. If it is present we assume it returns raw chunks as processed by read_chunked().

tell()
Obtain the number of bytes pulled over the wire so far. May differ from the amount of content returned by :meth:`HTTPResponse.read` if bytes are encoded on the wire (e.g, compressed).

Module contents

urllib3 - Thread-safe connection pooling and re-using.

class urllib3.**HTTPConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, ***conn_kw*)
Bases: *urllib3.connectionpool.ConnectionPool*, *urllib3.request.RequestMethods*

Thread-safe connection pool for one host.

Parameters

- **host** – Host used for this HTTP Connection (e.g. "localhost"), passed into `httplib.HTTPConnection`.
- **port** – Port used for this HTTP Connection (None is equivalent to 80), passed into `httplib.HTTPConnection`.
- **strict** – Causes BadStatusLine to be raised if the status line can't be parsed as a valid HTTP/1.0 or 1.1 status line, passed into `httplib.HTTPConnection`.

  Note: Only works in Python 2. This parameter is ignored in Python 3.

- **timeout** – Socket timeout in seconds for each individual connection. This can be a float or integer, which sets the timeout for the HTTP request, or an instance of *urllib3.util.Timeout* which gives you more fine-grained control over request timeouts. After the constructor has been parsed, this is always a *urllib3.util.Timeout* object.
- **maxsize** – Number of connections to save that can be reused. More than 1 is useful in multithreaded situations. If `block` is set to False, more connections will be created but they will not be saved once they've been used.
- **block** – If set to True, no more than `maxsize` connections will be used at a time. When no free connections are available, the call will block until a connection has been released. This is a useful side effect for particular multithreaded situations where one does not want to use more than maxsize connections per host to prevent flooding.
- **headers** – Headers to include with all requests, unless other headers are given explicitly.
- **retries** – Retry configuration to use by default with requests in this pool.

- **_proxy** – Parsed proxy URL, should not be used directly, instead, see `urllib3.connectionpool.ProxyManager`“
- **_proxy_headers** – A dictionary with proxy headers, should not be used directly, instead, see `urllib3.connectionpool.ProxyManager`“
- ****conn_kw** – Additional parameters are used to create fresh *urllib3.connection.HTTPConnection*, *urllib3.connection.HTTPSConnection* instances.

ConnectionCls
alias of `HTTPConnection`

ResponseCls
alias of *HTTPResponse*

close()
Close all pooled connections and disable the pool.

is_same_host(*url*)
Check if the given `url` is a member of the same host as this connection pool.

scheme = 'http'

urlopen(*method*, *url*, *body=None*, *headers=None*, *retries=None*, *redirect=True*, *assert_same_host=True*, *timeout=<object object>*, *pool_timeout=None*, *release_conn=None*, *chunked=False*, *body_pos=None*, ***response_kw*)
Get a connection from the pool and perform an HTTP request. This is the lowest level call for making a request, so you'll need to specify all the raw details.

Note: More commonly, it's appropriate to use a convenience method provided by *RequestMethods*, such as *request()*.

Note: *release_conn* will only behave as expected if *preload_content=False* because we want to make *preload_content=False* the default behaviour someday soon without breaking backwards compatibility.

Parameters

- **method** – HTTP request method (such as GET, POST, PUT, etc.)
- **body** – Data to send in the request body (useful for creating POST requests, see HTTPConnectionPool.post_url for more convenience).
- **headers** – Dictionary of custom headers to send, such as User-Agent, If-None-Match, etc. If None, pool headers are used. If provided, these headers completely replace any pool-specific headers.
- **retries** (*Retry*, False, or an int.) – Configure the number of retries to allow before raising a *MaxRetryError* exception.

  Pass `None` to retry until you receive a response. Pass a *Retry* object for fine-grained control over different types of retries. Pass an integer number to retry connection errors that many times, but no other types of errors. Pass zero to never retry.

  If `False`, then retries are disabled and any exception is raised immediately. Also, instead of raising a MaxRetryError on redirects, the redirect response will be returned.
- **redirect** – If True, automatically handle redirects (status codes 301, 302, 303, 307, 308). Each redirect counts as a retry. Disabling retries will disable redirect, too.

- **assert_same_host** – If `True`, will make sure that the host of the pool requests is consistent else will raise HostChangedError. When False, you can use the pool on an HTTP proxy and request foreign hosts.
- **timeout** – If specified, overrides the default timeout for this one request. It may be a float (in seconds) or an instance of *urllib3.util.Timeout*.
- **pool_timeout** – If set and the pool is set to block=True, then this method will block for `pool_timeout` seconds and raise EmptyPoolError if no connection is available within the time period.
- **release_conn** – If False, then the urlopen call will not release the connection back into the pool once a response is received (but will release if you read the entire contents of the response such as when *preload_content=True*). This is useful if you're not preloading the response's content immediately. You will need to call `r.release_conn()` on the response `r` to return the connection back into the pool. If None, it takes the value of `response_kw.get('preload_content', True)`.
- **chunked** – If True, urllib3 will send the body using chunked transfer encoding. Otherwise, urllib3 will send the body using the standard content-length form. Defaults to False.
- **body_pos** (*int*) – Position to seek to in file-like body in the event of a retry or redirect. Typically this won't need to be set because urllib3 will auto-populate the value when needed.
- ****response_kw** – Additional parameters are passed to *urllib3.response.HTTPResponse.from_httplib()*

class urllib3.**HTTPSConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, *key_file=None*, *cert_file=None*, *cert_reqs=None*, *ca_certs=None*, *ssl_version=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*, ***conn_kw*)

Bases: *urllib3.connectionpool.HTTPConnectionPool*

Same as *HTTPConnectionPool*, but HTTPS.

When Python is compiled with the `ssl` module, then *VerifiedHTTPSConnection* is used, which *can* verify certificates, instead of `HTTPSConnection`.

VerifiedHTTPSConnection uses one of `assert_fingerprint`, `assert_hostname` and `host` in this order to verify connections. If `assert_hostname` is False, no verification is done.

The `key_file`, `cert_file`, `cert_reqs`, `ca_certs`, `ca_cert_dir`, and `ssl_version` are only used if `ssl` is available and are fed into *urllib3.util.ssl_wrap_socket()* to upgrade the connection socket into an SSL socket.

ConnectionCls
alias of `VerifiedHTTPSConnection`

scheme = 'https'

class urllib3.**PoolManager**(*num_pools=10*, *headers=None*, ***connection_pool_kw*)

Bases: *urllib3.request.RequestMethods*

Allows for arbitrary requests while transparently keeping track of necessary connection pools for you.

Parameters

- **num_pools** – Number of connection pools to cache before discarding the least recently used pool.
- **headers** – Headers to include with all requests, unless other headers are given explicitly.
- ****connection_pool_kw** – Additional parameters are used to create fresh *urllib3.connectionpool.ConnectionPool* instances.

Example:

```
>>> manager = PoolManager(num_pools=2)
>>> r = manager.request('GET', 'http://google.com/')
>>> r = manager.request('GET', 'http://google.com/mail')
>>> r = manager.request('GET', 'http://yahoo.com/')
>>> len(manager.pools)
2
```

clear()
Empty our store of pools and direct them all to close.

This will not affect in-flight connections, but they will not be re-used after completion.

connection_from_context(*request_context*)
Get a `ConnectionPool` based on the request context.

`request_context` must at least contain the `scheme` key and its value must be a key in `key_fn_by_scheme` instance variable.

connection_from_host(*host*, *port=None*, *scheme='http'*, *pool_kwargs=None*)
Get a `ConnectionPool` based on the host, port, and scheme.

If `port` isn't given, it will be derived from the `scheme` using `urllib3.connectionpool.port_by_scheme`. If `pool_kwargs` is provided, it is merged with the instance's `connection_pool_kw` variable and used to create the new connection pool, if one is needed.

connection_from_pool_key(*pool_key*, *request_context=None*)
Get a `ConnectionPool` based on the provided pool key.

`pool_key` should be a namedtuple that only contains immutable objects. At a minimum it must have the `scheme`, `host`, and `port` fields.

connection_from_url(*url*, *pool_kwargs=None*)
Similar to *urllib3.connectionpool.connection_from_url()*.

If `pool_kwargs` is not provided and a new pool needs to be constructed, `self.connection_pool_kw` is used to initialize the *urllib3.connectionpool.ConnectionPool*. If `pool_kwargs` is provided, it is used instead. Note that if a new pool does not need to be created for the request, the provided `pool_kwargs` are not used.

proxy = None

urlopen(*method*, *url*, *redirect=True*, ***kw*)
Same as *urllib3.connectionpool.HTTPConnectionPool.urlopen()* with custom cross-host redirect logic and only sends the request-uri portion of the `url`.

The given `url` parameter must be absolute, such that an appropriate *urllib3.connectionpool.ConnectionPool* can be chosen for it.

class urllib3.**ProxyManager**(*proxy_url*, *num_pools=10*, *headers=None*, *proxy_headers=None*, ***connection_pool_kw*)
Bases: *urllib3.poolmanager.PoolManager*

Behaves just like *PoolManager*, but sends all requests through the defined proxy, using the CONNECT method for HTTPS URLs.

Parameters

- **proxy_url** – The URL of the proxy to be used.
- **proxy_headers** – A dictionary contaning headers that will be sent to the proxy. In case of HTTP they are being sent with each request, while in the HTTPS/CONNECT case they are sent only once. Could be used for proxy authentication.

Example:

```
>>> proxy = urllib3.ProxyManager('http://localhost:3128/')
>>> r1 = proxy.request('GET', 'http://google.com/')
>>> r2 = proxy.request('GET', 'http://httpbin.org/')
>>> len(proxy.pools)
1
>>> r3 = proxy.request('GET', 'https://httpbin.org/')
>>> r4 = proxy.request('GET', 'https://twitter.com/')
>>> len(proxy.pools)
3
```

connection_from_host(*host*, *port=None*, *scheme='http'*, *pool_kwargs=None*)

urlopen(*method*, *url*, *redirect=True*, ***kw*)
Same as HTTP(S)ConnectionPool.urlopen, `url` must be absolute.

class urllib3.**HTTPResponse**(*body=''*, *headers=None*, *status=0*, *version=0*, *reason=None*, *strict=0*, *preload_content=True*, *decode_content=True*, *original_response=None*, *pool=None*, *connection=None*, *retries=None*, *enforce_content_length=False*, *request_method=None*)

Bases: `io.IOBase`

HTTP Response container.

Backwards-compatible to httplib's HTTPResponse but the response `body` is loaded and decoded on-demand when the `data` property is accessed. This class is also compatible with the Python standard library's `io` module, and can hence be treated as a readable object in the context of that framework.

Extra parameters for behaviour not present in httplib.HTTPResponse:

Parameters

- **preload_content** – If True, the response's body will be preloaded during construction.
- **decode_content** – If True, attempts to decode specific content-encoding's based on headers (like 'gzip' and 'deflate') will be skipped and raw data will be used instead.
- **original_response** – When this HTTPResponse wrapper is generated from an httplib.HTTPResponse object, it's convenient to include the original for debug purposes. It's otherwise unused.
- **retries** – The retries contains the last *Retry* that was used during the request.
- **enforce_content_length** – Enforce content length checking. Body returned by server must match value of Content-Length header, if present. Otherwise, raise error.

CONTENT_DECODERS = ['gzip', 'deflate']

REDIRECT_STATUSES = [301, 302, 303, 307, 308]

close()

closed

connection

data

fileno()

flush()

classmethod from_httplib(*ResponseCls*, *r*, ***response_kw*)
Given an `httplib.HTTPResponse` instance `r`, return a corresponding *urllib3.response.HTTPResponse* object.

Remaining parameters are passed to the HTTPResponse constructor, along with `original_response=r`.

get_redirect_location()
Should we redirect and where to?

Returns Truthy redirect location string if we got a redirect status code and valid location. `None` if redirect status and no location. `False` if not a redirect status code.

getheader(*name*, *default=None*)

getheaders()

info()

read(*amt=None*, *decode_content=None*, *cache_content=False*)
Similar to `httplib.HTTPResponse.read()`, but with two additional parameters: `decode_content` and `cache_content`.

Parameters

- **amt** – How much of the content to read. If specified, caching is skipped because it doesn't make sense to cache partial content as the full response.
- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.
- **cache_content** – If True, will save the returned data such that the same result is returned despite of the state of the underlying file object. This is useful if you want the `.data` property to continue working after having `.read()` the file object. (Overridden if `amt` is set.)

read_chunked(*amt=None*, *decode_content=None*)
Similar to *HTTPResponse.read()*, but with an additional parameter: `decode_content`.

Parameters **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.

readable()

readinto(*b*)

release_conn()

stream(*amt=65536*, *decode_content=None*)
A generator wrapper for the read() method. A call will block until `amt` bytes have been read from the connection or until the connection is closed.

Parameters

- **amt** – How much of the content to read. The generator will return up to much data per iteration, but may return less. This is particularly likely when using compressed data. However, the empty string will never be returned.
- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.

supports_chunked_reads()
Checks if the underlying file-like object looks like a httplib.HTTPResponse object. We do this by testing for the fp attribute. If it is present we assume it returns raw chunks as processed by read_chunked().

tell()
Obtain the number of bytes pulled over the wire so far. May differ from the amount of content returned by :meth:`HTTPResponse.read` if bytes are encoded on the wire (e.g, compressed).

class urllib3.**Retry**(*total=10*, *connect=None*, *read=None*, *redirect=None*, *status=None*, *method_whitelist=frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE'])*, *status_forcelist=None*, *backoff_factor=0*, *raise_on_redirect=True*, *raise_on_status=True*, *history=None*, *respect_retry_after_header=True*)
Bases: object

Retry configuration.

Each retry attempt will create a new Retry object with updated values, so they can be safely reused.

Retries can be defined as a default for a pool:

```
retries = Retry(connect=5, read=2, redirect=5)
http = PoolManager(retries=retries)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', retries=Retry(10))
```

Retries can be disabled by passing `False`:

```
response = http.request('GET', 'http://example.com/', retries=False)
```

Errors will be wrapped in *MaxRetryError* unless retries are disabled, in which case the causing exception will be raised.

Parameters

- **total** (*int*) – Total number of retries to allow. Takes precedence over other counts.

  Set to `None` to remove this constraint and fall back on other counts. It's a good idea to set this to some sensibly-high value to account for unexpected edge cases and avoid infinite retry loops.

  Set to `0` to fail on the first retry.

  Set to `False` to disable and imply `raise_on_redirect=False`.

- **connect** (*int*) – How many connection-related errors to retry on.

  These are errors raised before the request is sent to the remote server, which we assume has not triggered the server to process the request.

  Set to `0` to fail on the first retry of this type.

- **read** (*int*) – How many times to retry on read errors.

These errors are raised after the request was sent to the server, so the request may have side-effects.

Set to `0` to fail on the first retry of this type.

- **redirect** (*int*) – How many redirects to perform. Limit this to avoid infinite redirect loops.

  A redirect is a HTTP response with a status code 301, 302, 303, 307 or 308.

  Set to `0` to fail on the first retry of this type.

  Set to `False` to disable and imply `raise_on_redirect=False`.

- **status** (*int*) – How many times to retry on bad status codes.

  These are retries made on responses, where status code matches `status_forcelist`.

  Set to `0` to fail on the first retry of this type.

- **method_whitelist** (*iterable*) – Set of uppercased HTTP method verbs that we should retry on.

  By default, we only retry on methods which are considered to be idempotent (multiple requests with the same parameters end with the same state). See *Retry.DEFAULT_METHOD_WHITELIST*.

  Set to a `False` value to retry on any verb.

- **status_forcelist** (*iterable*) – A set of integer HTTP status codes that we should force a retry on. A retry is initiated if the request method is in `method_whitelist` and the response status code is in `status_forcelist`.

  By default, this is disabled with `None`.

- **backoff_factor** (*float*) – A backoff factor to apply between attempts after the second try (most errors are resolved immediately by a second try without a delay). urllib3 will sleep for:

```
{backoff factor} * (2 ^ ({number of total retries} - 1))
```

  seconds. If the backoff_factor is 0.1, then `sleep()` will sleep for [0.0s, 0.2s, 0.4s, ...] between retries. It will never be longer than *Retry.BACKOFF_MAX*.

  By default, backoff is disabled (set to 0).

- **raise_on_redirect** (*bool*) – Whether, if the number of redirects is exhausted, to raise a MaxRetryError, or to return a response with a response code in the 3xx range.

- **raise_on_status** (*bool*) – Similar meaning to `raise_on_redirect`: whether we should raise an exception, or return a response, if status falls in `status_forcelist` range and retries have been exhausted.

- **history** (*tuple*) – The history of the request encountered during each call to *increment()*. The list is in the order the requests occurred. Each list item is of class `RequestHistory`.

- **respect_retry_after_header** (*bool*) – Whether to respect Retry-After header on status codes defined as *Retry.RETRY_AFTER_STATUS_CODES* or not.

BACKOFF_MAX = 120

DEFAULT = Retry(total=3, connect=None, read=None, redirect=None, status=None)

DEFAULT_METHOD_WHITELIST = frozenset(['HEAD', 'TRACE', 'GET', 'PUT', 'OPTIONS', 'DELETE'])

RETRY_AFTER_STATUS_CODES = frozenset([503, 413, 429])

classmethod from_int(*retries*, *redirect=True*, *default=None*)
Backwards-compatibility for the old retries format.

get_backoff_time()
Formula for computing the current backoff

Return type float

get_retry_after(*response*)
Get the value of Retry-After in seconds.

increment(*method=None*, *url=None*, *response=None*, *error=None*, *_pool=None*, *_stacktrace=None*)
Return a new Retry object with incremented retry counters.

Parameters

- **response** (*HTTPResponse*) – A response object, or None, if the server did not return a response.
- **error** (*Exception*) – An error encountered during the request, or None if the response was received successfully.

Returns A new `Retry` object.

is_exhausted()
Are we out of retries?

is_retry(*method*, *status_code*, *has_retry_after=False*)
Is this method/status code retryable? (Based on whitelists and control variables such as the number of total retries to allow, whether to respect the Retry-After header, whether this header is present, and whether the returned status code is on the list of status codes to be retried upon on the presence of the aforementioned header)

new(***kw*)

parse_retry_after(*retry_after*)

sleep(*response=None*)
Sleep between retry attempts.

This method will respect a server's `Retry-After` response header and sleep the duration of the time requested. If that is not present, it will use an exponential backoff. By default, the backoff factor is 0 and this method will return immediately.

sleep_for_retry(*response=None*)

class `urllib3.`**Timeout**(*total=None*, *connect=<object object>*, *read=<object object>*)
Bases: `object`

Timeout configuration.

Timeouts can be defined as a default for a pool:

```
timeout = Timeout(connect=2.0, read=7.0)
http = PoolManager(timeout=timeout)
response = http.request('GET', 'http://example.com/')
```

Or per-request (which overrides the default for the pool):

```
response = http.request('GET', 'http://example.com/', timeout=Timeout(10))
```

Timeouts can be disabled by setting all the parameters to `None`:

```
no_timeout = Timeout(connect=None, read=None)
response = http.request('GET', 'http://example.com/, timeout=no_timeout)
```

Parameters

- **total** (*integer, float, or None*) – This combines the connect and read timeouts into one; the read timeout will be set to the time leftover from the connect attempt. In the event that both a connect timeout and a total are specified, or a read timeout and a total are specified, the shorter timeout will be applied.

  Defaults to None.

- **connect** (*integer, float, or None*) – The maximum amount of time to wait for a connection attempt to a server to succeed. Omitting the parameter will default the connect timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout for connection attempts.

- **read** (*integer, float, or None*) – The maximum amount of time to wait between consecutive read operations for a response from the server. Omitting the parameter will default the read timeout to the system default, probably the global default timeout in socket.py. None will set an infinite timeout.

Note: Many factors can affect the total amount of time for urllib3 to return an HTTP response.

For example, Python's DNS resolver does not obey the timeout specified on the socket. Other factors that can affect total request time include high CPU load, high swap, the program running at a low priority level, or other behaviors.

In addition, the read and total timeouts only measure the time between read operations on the socket connecting the client and the server, not the total amount of time for the request to return a complete response. For most requests, the timeout is raised because the server has not sent the first byte in the specified time. This is not always the case; if a server streams one byte every fifteen seconds, a timeout of 20 seconds will not trigger, even though the request will take several minutes to complete.

If your goal is to cut off any request after a set amount of wall clock time, consider having a second "watcher" thread to cut off a slow request.

DEFAULT_TIMEOUT = <object object>

clone()

Create a copy of the timeout object

Timeout properties are stored per-pool but each request needs a fresh Timeout object to ensure each one has its own start/stop configured.

Returns a copy of the timeout object

Return type *Timeout*

connect_timeout

Get the value to use when setting a connection timeout.

This will be a positive float or integer, the value None (never timeout), or the default system timeout.

Returns Connect timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

classmethod from_float(*timeout*)
Create a new Timeout from a legacy timeout value.

The timeout value used by httplib.py sets the same timeout on the connect(), and recv() socket requests. This creates a *Timeout* object that sets the individual timeouts to the `timeout` value passed to this function.

Parameters **timeout** (*integer, float, sentinel default object, or None*) – The legacy timeout value.

Returns Timeout object

Return type *Timeout*

get_connect_duration()
Gets the time elapsed since the call to *start_connect()*.

Returns Elapsed time.

Return type float

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to get duration for a timer that hasn't been started.

read_timeout
Get the value for the read timeout.

This assumes some time has elapsed in the connection timeout and computes the read timeout appropriately.

If self.total is set, the read timeout is dependent on the amount of time taken by the connect timeout. If the connection time has not been established, a *TimeoutStateError* will be raised.

Returns Value to use for the read timeout.

Return type int, float, *Timeout.DEFAULT_TIMEOUT* or None

Raises ***urllib3.exceptions.TimeoutStateError*** – If *start_connect()* has not yet been called on this object.

start_connect()
Start the timeout clock, used during a connect() attempt

Raises ***urllib3.exceptions.TimeoutStateError*** – if you attempt to start a timer that has been started already.

urllib3.**add_stderr_logger**(*level=10*)
Helper for quickly adding a StreamHandler to the logger. Useful for debugging.

Returns the handler after adding it.

urllib3.**connection_from_url**(*url*, ***kw*)
Given a url, return an *ConnectionPool* instance of its host.

This is a shortcut for not having to parse out the scheme, host, and port of the url before creating an *ConnectionPool* instance.

Parameters

- **url** – Absolute URL string that must include the scheme. Port is optional.
- ****kw** – Passes additional parameters to the constructor of the appropriate *ConnectionPool*. Useful for specifying things like timeout, maxsize, headers, etc.

Example:

```
>>> conn = connection_from_url('http://google.com/')
>>> r = conn.request('GET', '/')
```

urllib3.**disable_warnings**(*category=<class 'urllib3.exceptions.HTTPWarning'>*)
Helper for quickly disabling all urllib3 warnings.

urllib3.**encode_multipart_formdata**(*fields*, *boundary=None*)
Encode a dictionary of `fields` using the multipart/form-data MIME format.

Parameters

- **fields** – Dictionary of fields or list of (key, *RequestField*).
- **boundary** – If not specified, then a random boundary will be generated using `mimetools.choose_boundary()`.

urllib3.**get_host**(*url*)
Deprecated. Use `parse_url()` instead.

urllib3.**make_headers**(*keep_alive=None*, *accept_encoding=None*, *user_agent=None*, *basic_auth=None*, *proxy_basic_auth=None*, *disable_cache=None*)
Shortcuts for generating request headers.

Parameters

- **keep_alive** – If `True`, adds 'connection: keep-alive' header.
- **accept_encoding** – Can be a boolean, list, or string. `True` translates to 'gzip,deflate'. List will get joined by comma. String will be used as provided.
- **user_agent** – String representing the user-agent you want, such as "python-urllib3/0.6"
- **basic_auth** – Colon-separated username:password string for 'authorization: basic ...' auth header.
- **proxy_basic_auth** – Colon-separated username:password string for 'proxy-authorization: basic ...' auth header.
- **disable_cache** – If `True`, adds 'cache-control: no-cache' header.

Example:

```
>>> make_headers(keep_alive=True, user_agent="Batman/1.0")
{'connection': 'keep-alive', 'user-agent': 'Batman/1.0'}
>>> make_headers(accept_encoding=True)
{'accept-encoding': 'gzip,deflate'}
```

urllib3.**proxy_from_url**(*url*, ***kw*)

CHAPTER 4

Contributing

urllib3 is a community-maintained project and we happily accept contributions.

If you wish to add a new feature or fix a bug:

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug. There is a *Contributor Friendly* tag for issues that should be ideal for people who are not very familiar with the codebase yet.
2. Fork the urllib3 repository on Github to start making your changes.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to `CONTRIBUTORS.txt`.

Running the tests

We use some external dependencies, multiple interpreters and code coverage analysis while running test suite. Our `Makefile` handles much of this for you as long as you're running it inside of a virtualenv:

```
$ make test
[... magically installs dependencies and runs tests on your virtualenv]
Ran 182 tests in 1.633s

OK (SKIP=6)
```

Note that code coverage less than 100% is regarded as a failing run. Some platform-specific tests are skipped unless run in that platform. To make sure the code works in all of urllib3's supported platforms, you can run our `tox` suite:

```
$ make test-all
[... tox creates a virtualenv for every platform and runs tests inside of each]
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
```

```
py33: commands succeeded
py34: commands succeeded
```

Our test suite runs continuously on Travis CI with every pull request.

Sponsorship

Please consider sponsoring urllib3 development, especially if your company benefits from this library.

We welcome your patronage on Bountysource:

- Contribute a recurring amount to the team
- Place a bounty on a specific feature

Your contribution will go towards adding new features to urllib3 and making sure all functionality continues to meet our high quality standards.

Project Grant

A grant for contiguous full-time development has the biggest impact for progress. Periods of 3 to 10 days allow a contributor to tackle substantial complex issues which are otherwise left to linger until somebody can't afford to not fix them.

Contact @shazow to arrange a grant for a core contributor.

Huge thanks to all the companies and individuals who financially contributed to the development of urllib3. Please send a PR if you've donated and would like to be listed.

- Stripe (June 23, 2014)

urllib3 is a powerful, *sanity-friendly* HTTP client for Python. Much of the Python ecosystem *already uses* urllib3 and you should too. urllib3 brings many critical features that are missing from the Python standard libraries:

- Thread safety.
- Connection pooling.
- Client-side SSL/TLS verification.
- File uploads with multipart encoding.
- Helpers for retrying requests and dealing with HTTP redirects.
- Support for gzip and deflate encoding.
- Proxy support for HTTP and SOCKS.
- 100% test coverage.

urllib3 is powerful and easy to use:

```
>>> import urllib3
>>> http = urllib3.PoolManager()
>>> r = http.request('GET', 'http://httpbin.org/robots.txt')
>>> r.status
200
>>> r.data
'User-agent: *\nDisallow: /deny\n'
```

CHAPTER 5

Installing

urllib3 can be installed with pip:

```
$ pip install urllib3
```

Alternatively, you can grab the latest source code from GitHub:

```
$ git clone git://github.com/shazow/urllib3.git
$ python setup.py install
```

CHAPTER 6

Usage

The *User Guide* is the place to go to learn how to use the library and accomplish common tasks. The more in-depth *Advanced Usage* guide is the place to go for lower-level tweaking.

The *Reference* documentation provides API-level documentation.

CHAPTER 7

Who uses urllib3?

- Requests
- Pip
- & more!

CHAPTER 8

License

urllib3 is made available under the MIT License. For more details, see LICENSE.txt.

CHAPTER 9

Contributing

We happily welcome contributions, please see *Contributing* for details.

Python Module Index

u

Index

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W