urllib3 Documentation

Release 1.0.2

Andrey Petrov

February 05, 2012

CONTENTS

CHAPTER

ONE

CONNECTIONPOOLS

A connection pool is a container for a collection of connections to a specific host.

If you need to make requests to the same host repeatedly, then you should use a `HTTPConnectionPool`.

```
>>> from urllib3 import HTTPConnectionPool
>>> pool = HTTPConnectionPool('ajax.googleapis.com', maxsize=1)
>>> r = pool.request('GET', '/ajax/services/search/web',
...                  fields={'q': 'urllib3', 'v': '1.0'})
>>> r.status
200
>>> r.headers['content-type']
'text/javascript; charset=utf-8'
>>> len(r.data) # Content of the response
3318
>>> r = pool.request('GET', '/ajax/services/search/web',
...                  fields={'q': 'python', 'v': '1.0'})
>>> len(r.data) # Content of the response
2960
>>> pool.num_connections
1
>>> pool.num_requests
2
```

By default, the pool will cache just one connection. If you're planning on using such a pool in a multithreaded environment, you should set the `maxsize` of the pool to a higher number, such as the number of threads. You can also control many other variables like timeout, blocking, and default headers.

1.1 Helpers

There are various helper functions provided for instantiating these ConnectionPools more easily:

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.connectionpool.**make_headers**(*keep_alive=None*, *accept_encoding=None*, *user_agent=None*, *basic_auth=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.

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'}
```

1.2 API

`urllib3.connectionpool` comes with two connection pools:

class urllib3.connectionpool.**HTTPConnectionPool**(*host*, *port=None*, *strict=False*, *timeout=None*, *maxsize=1*, *block=False*, *headers=None*)

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`.
- **timeout** – Socket timeout for each individual connection, can be a float. None disables timeout.
- **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.

get_url (*url*, *fields=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

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

post_url (*url*, *fields=None*, *headers=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

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) tuple. For example:

```
fields = {
    'foo': 'bar',
    'fakefile': ('foofile.txt', 'contents of foofile'),
    'realfile': ('barfile.txt', open('realfile').read()),
    '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*, ***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*, *retries=3*, *redirect=True*, *assert_same_host=True*, *timeout=<object object at 0x7fa948670410>*, *pool_timeout=None*, *release_conn=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()`.

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** – Number of retries to allow before raising a MaxRetryError exception.
- **redirect** – Automatically handle redirects (status codes 301, 302, 303, 307), each redirect counts as a retry.
- **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.
- **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. 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)`.
- ****response_kw** – Additional parameters are passed to `urllib3.response.HTTPResponse.from_httplib()`

class `urllib3.connectionpool.`**`HTTPSConnectionPool`**(*host*, *port=None*, *strict=False*, *timeout=None*, *maxsize=1*, *block=False*, *headers=None*, *key_file=None*, *cert_file=None*, *cert_reqs='CERT_NONE'*, *ca_certs=None*)

Same as `HTTPConnectionPool`, but HTTPS.

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

The `key_file`, `cert_file`, `cert_reqs`, and `ca_certs` parameters are only used if `ssl` is available and are fed into `ssl.wrap_socket()` to upgrade the connection socket into an SSL socket.

All of these pools inherit from a common base class:

class `urllib3.connectionpool.`**`ConnectionPool`**

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

CHAPTER

TWO

POOLMANAGER

A pool manager is an abstraction for a collection of *ConnectionPools*.

If you need to make requests to multiple hosts, then you can use a `PoolManager`, which takes care of maintaining your pools so you don't have to.

```
>>> from urllib3 import PoolManager
>>> manager = PoolManager(10)
>>> r = manager.request('GET', 'http://google.com/')
>>> r.headers['server']
'gws'
>>> r = manager.request('GET', 'http://yahoo.com/')
>>> r.headers['server']
'YTS/1.20.0'
>>> r = manager.request('POST', 'http://google.com/mail')
>>> r = manager.request('HEAD', 'http://google.com/calendar')
>>> len(manager.pools)
2
>>> conn = manager.connection_from_host('google.com')
>>> conn.num_requests
3
```

The API of a `PoolManager` object is similar to that of a *ConnectionPool*, so they can be passed around interchangeably.

The PoolManager uses a Least Recently Used (LRU) policy for discarding old pools. That is, if you set the PoolManager `num_pools` to 10, then after making requests to 11 or more different hosts, the least recently used pools will be cleaned up eventually.

Cleanup of stale pools does not happen immediately. You can read more about the implementation and the various adjustable variables within `RecentlyUsedContainer`.

2.1 API

class `urllib3.poolmanager.`**`PoolManager`**(*num_pools=10*, ***connection_pool_kw*)

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.

- ****connection_pool_kw** – Additional parameters are used to create fresh `urllib3.connectionpool.ConnectionPool` instances.

Example:

```
>>> manager = PoolManager()
>>> r = manager.urlopen("http://google.com/")
>>> r = manager.urlopen("http://google.com/mail")
>>> r = manager.urlopen("http://yahoo.com/")
>>> len(r.pools)
2
```

connection_from_host(*host*, *port=80*, *scheme='http'*)
Get a `ConnectionPool` based on the host, port, and scheme.

Note that an appropriate `port` value is required here to normalize connection pools in our container most effectively.

connection_from_url(*url*)
Similar to `urllib3.connectionpool.connection_from_url()` but doesn't pass any additional parameters to the `urllib3.connectionpool.ConnectionPool` constructor.

Additional parameters are taken from the `PoolManager` constructor.

get_url(*url*, *fields=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

post_url(*url*, *fields=None*, *headers=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

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) tuple. For example:

```
fields = {
    'foo': 'bar',
    'fakefile': ('foofile.txt', 'contents of foofile'),
    'realfile': ('barfile.txt', open('realfile').read()),
```

```
    '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*, ***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*, ***kw*)
Same as `urllib3.connectionpool.HTTPConnectionPool.urlopen()`.

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

CHAPTER

THREE

HELPERS

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

3.1 Filepost

`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. The key is treated as the field name, and the value as the body of the form-data. If the value is a tuple of two elements, then the first element is treated as the filename of the form-data section.
- **boundary** – If not specified, then a random boundary will be generated using `mimetools.choose_boundary()`.

3.2 Request

class `urllib3.request.`**`RequestMethods`**
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-orm-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.

`get_url`(*url*, *fields=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

`post_url`(*url*, *fields=None*, *headers=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

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) tuple. For example:

```
fields = {
    'foo': 'bar',
    'fakefile': ('foofile.txt', 'contents of foofile'),
    'realfile': ('barfile.txt', open('realfile').read()),
    '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*, ***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.

3.3 Response

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*)
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.

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.

CONTENT_DECODERS = {'gzip': <function decode_gzip at 0x215b488>, 'deflate': <function decode_deflate at 0x215b500>

data

static **from_httplib**(*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`.

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

getheaders()

read(*amt=None*, *decode_content=True*, *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, decoding and caching is skipped because we can't decode partial content nor does it 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. (Overridden if `amt` is set.)
- **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.)

release_conn()

urllib3.response.**decode_deflate**(*data*)

urllib3.response.**decode_gzip**(*data*)

CHAPTER

FOUR

COLLECTIONS

These datastructures are used to implement the behaviour of various urllib3 components in a decoupled and application-agnostic design.

class `urllib3._collections.`**`RecentlyUsedContainer`**(*maxsize=10*)

Provides a dict-like that maintains up to `maxsize` keys while throwing away the least-recently-used keys beyond `maxsize`.

CHAPTER

FIVE

HIGHLIGHTS

- Re-use the same socket connection for multiple requests, with optional client-side certificate verification. See: `HTTPConnectionPool` and `HTTPSConnectionPool`
- File posting. See: `encode_multipart_formdata()`
- Built-in redirection and retries (optional).
- Supports gzip and deflate decoding. See: `decode_gzip()` and `decode_deflate()`
- Thread-safe and sanity-safe.
- Small and easy to understand codebase perfect for extending and building upon. For a more comprehensive solution, have a look at Requests.

CHAPTER

SIX

GETTING STARTED

6.1 Installing

`pip install urllib3` or fetch the latest source from github.com/shazow/urllib3.

6.2 Usage

```
>>> import urllib3
>>> http = urllib3.PoolManager()
>>> r = http.request('GET', 'http://google.com/')
>>> r.status
200
>>> r.headers['server']
'gws'
>>> r.data
...
```

CHAPTER

SEVEN

COMPONENTS

`urllib3` tries to strike a fine balance between power, extendability, and sanity. To achieve this, the codebase is a collection of small reusable utilities and abstractions composed together in a few helpful layers.

7.1 PoolManager

The highest level is the *PoolManager(...)*.

The `PoolManager` will take care of reusing connections for you whenever you request the same host. this should cover most scenarios without significant loss of efficiency, but you can always drop down to a lower level component for more granular control.

```
>>> http = urllib3.PoolManager(10)
>>> r1 = http.request('GET', 'http://google.com/')
>>> r2 = http.request('GET', 'http://google.com/mail')
>>> r3 = http.request('GET', 'http://yahoo.com/')
>>> len(http.pools)
2
```

A `PoolManager` is a proxy for a collection of `ConnectionPool` objects. They both inherit from `RequestMethods` to make sure that their API is similar, so that instances of either can be passed around interchangeably.

7.2 ConnectionPool

The next layer is the *ConnectionPool(...)*.

The `HTTPConnectionPool` and `HTTPSConnectionPool` classes allow you to define a pool of connections to a single host and make requests against this pool with automatic **connection reusing** and **thread safety**.

When the `ssl` module is available, then `HTTPSConnectionPool` objects can be configured to check SSL certificates against specific provided certificate authorities.

```
>>> conn = urllib3.connection_from_url('http://google.com')
>>> r1 = conn.request('GET', 'http://google.com/')
>>> r2 = conn.request('GET', '/mail')
>>> r3 = conn.request('GET', 'http://yahoo.com/')
Traceback (most recent call last)
  ...
HostChangedError: Connection pool with host 'http://google.com' tried to
open a foreign host: http://yahoo.com/
```

Again, a ConnectionPool is a pool of connections to a specific host. Trying to access a different host through the same pool will raise a `HostChangedError` exception unless you specify `assert_same_host=False`. Do this at your own risk as the outcome is completely dependent on the behaviour of the host server.

If you need to access multiple hosts and don't want to manage your own collection of `ConnectionPool` objects, then you should use a `PoolManager`.

A `ConnectionPool` is composed of a collection of `httplib.HTTPConnection` objects.

7.3 Foundation

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.

7.3.1 Helpers

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

Filepost

`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. The key is treated as the field name, and the value as the body of the form-data. If the value is a tuple of two elements, then the first element is treated as the filename of the form-data section.
- **boundary** – If not specified, then a random boundary will be generated using `mimetools.choose_boundary()`.

Request

class `urllib3.request.`**`RequestMethods`**

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-orm-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.

get_url (*url*, *fields=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

post_url (*url*, *fields=None*, *headers=None*, ***urlopen_kw*)
Deprecated since version 1.0. Use `request()` instead.

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) tuple. For example:

```
fields = {
    'foo': 'bar',
    'fakefile': ('foofile.txt', 'contents of foofile'),
    'realfile': ('barfile.txt', open('realfile').read()),
    '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*, ***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.

Response

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*)
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.

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.

CONTENT_DECODERS = {'gzip': <function decode_gzip at 0x215b488>, 'deflate': <function decode_deflate at 0x215b500>

data

static **from_httplib**(*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`.

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

getheaders()

read(*amt=None*, *decode_content=True*, *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, decoding and caching is skipped because we can't decode partial content nor does it 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. (Overridden if `amt` is set.)
- **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.)

release_conn()

urllib3.response.**decode_deflate**(*data*)

urllib3.response.**decode_gzip**(*data*)

CHAPTER

EIGHT

CONTRIBUTING

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`.

PYTHON MODULE INDEX

u