Hip Documentation

Release 0.1.dev0

The Trio Collective

May 07, 2020

Contents

CHAPTER 1

User Guide

1.1 Making your first HTTP request

First things first, import the hip module:

```
>>> import hip
```

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 = hip.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.

1.2 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', ...})
```

1.2.1 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'}
```

1.2.2 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.

1.2.3 Using `io` Wrappers with Response content

Sometimes you want to use `io.TextIOWrapper` or similar objects like a CSV reader directly with *HTTPResponse* data. Making these two interfaces play nice together requires using the `auto_close` attribute by setting it to `False`. By default HTTP responses are closed after reading all bytes, this disables that behavior:

```
>>> import io
>>> r = http.request('GET', 'https://example.com', preload_content=False)
>>> r.auto_close = False
>>> for line in io.TextIOWrapper(r):
>>>     print(line)
```

1.3 Request Data

1.3.1 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', ...}
```

1.3.2 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'}
```

1.3.3 Form Data

For `PUT` and `POST` requests, hip 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'}
```

1.3.4 JSON

You can send a JSON 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',
```

(continues on next page)

(continued from previous page)

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

1.3.5 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'...'
```

1.4 Certificate Verification

While you can disable certification verification, it is highly recommend to leave it on.

Unless otherwise specified hip will try to load the default system certificate stores. The most reliable cross-platform method is to use the certifi package which provides Mozilla's root certificate bundle:

```
python -m pip install certifi
```

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

```
>>> import certifi
>>> import hip
>>> http = hip.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')
hip.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.

1.5 Using Timeouts

Timeouts allow you to control how long (in seconds) 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)
<hip.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=hip.Timeout(connect=1.0))
<hip.response.HTTPResponse>
>>> http.request(
...     'GET',
...     'http://httpbin.org/delay/3',
...     timeout=hip.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 = hip.PoolManager(timeout=3.0)
>>> http = hip.PoolManager(
...     timeout=hip.Timeout(connect=1.0, read=2.0))
```

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

1.6 Retrying Requests

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

To change the number of retries just specify an integer:

```
>>> http.request('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=hip.Retry(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=hip.Retry(
...         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 = hip.PoolManager(retries=False)
>>> http = hip.PoolManager(
...     retries=hip.Retry(5, redirect=2))
```

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

1.7 Errors & Exceptions

hip wraps lower-level exceptions, for example:

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

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

1.8 Logging

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

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

CHAPTER 2

Advanced Usage

2.1 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 hip
>>> http = hip.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 hip
>>> http = hip.PoolManager(maxsize=10)
# Alternatively
>>> http = hip.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 = hip.PoolManager(maxsize=10, block=True)
# Alternatively
>>> http = hip.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.

2.2 Streaming and IO

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

```
>>> import hip
>>> http = hip.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 Hip 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()
```

2.3 Proxies

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

```
>>> import hip
>>> proxy = hip.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 hip with the `socks` extra:

```
python -m pip install hip[socks]
```

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

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

2.4 Custom SSL 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 hip
>>> http = hip.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.

2.5 Client Certificates

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

If you have an encrypted client certificate private key you can use the `key_password` parameter to specify a password to decrypt the key.

```
>>> http = hip.PoolManager(
...     cert_file='/path/to/your/client_cert.pem',
...     cert_reqs='CERT_REQUIRED',
```

(continues on next page)

(continued from previous page)

```
...     key_file='/path/to/your/client.key',
...     key_password='keyfile_password')
```

If your key isn't encrypted the `key_password` parameter isn't required.

2.6 Certificate Validation and macOS

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.

2.7 SSL Warnings

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

- ***InsecureRequestWarning*** This happens when a 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.
- ***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.

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 hip
>>> hip.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.

2.8 Brotli Encoding

Brotli is a compression algorithm created by Google with better compression than gzip and deflate and is supported by Hip if the brotlipy package is installed. You may also request the package be installed via the `hip[brotli]` extra:

```
python -m pip install hip[brotli]
```

Here's an example using brotli encoding via the `Accept-Encoding` header:

```
>>> from hip import PoolManager
>>> http = PoolManager()
>>> http.request('GET', 'https://www.google.com/', headers={'Accept-Encoding': 'br'})
```

CHAPTER 3

Reference

3.1 Subpackages

3.1.1 hip.util package

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

At the very core, just like its predecessors, *hip* 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, *hip* provides various helper methods which are used with the higher level components but can also be used independently.

hip.util.connection module

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

hip.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.

hip.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.

hip.util.request module

hip.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-hip/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'}
```

hip.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.

hip.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.

hip.util.retry module

class hip.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 hip.util.retry.**Retry**(*total=10*, *connect=None*, *read=None*, *redirect=None*, *status=None*, *method_whitelist=frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTIONS'})*, *status_forcelist=None*, *backoff_factor=0*, *raise_on_redirect=True*, *raise_on_status=True*, *history=None*, *respect_retry_after_header=True*, *remove_headers_on_redirect=frozenset({'Authorization'})*)

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). Hip 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.
- **remove_headers_on_redirect** (*iterable*) – Sequence of headers to remove from the request when a response indicating a redirect is returned before firing off the redirected request.

DEFAULT_METHOD_WHITELIST = frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTION

RETRY_AFTER_STATUS_CODES = frozenset({503, 413, 429})

DEFAULT_REDIRECT_HEADERS_BLACKLIST = frozenset({'Authorization'})

BACKOFF_MAX = 120
Maximum backoff time.

new(***kw*)

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

parse_retry_after(*retry_after*)

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

sleep_for_retry(*response=None*)

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.

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)

is_exhausted()
Are we out of retries?

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.

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

hip.util.timeout module

class hip.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 (in seconds) 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 (in seconds) 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 Hip 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

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*

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*

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

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

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

Returns Elapsed time in seconds.

Return type float

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

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

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

hip.util.url module

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

Data structure 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.

request_uri
Absolute path including the query string.

netloc
Network location including host and port

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'
```

hip.util.url.**split_first**(*s*, *delims*)
Deprecated since version 1.25.

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.

hip.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. This parser is RFC 3986 compliant.

The parser logic and helper functions are based heavily on work done in the `rfc3986` module.

Parameters **url** (*str*) – URL to parse into a *Url* namedtuple.

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', ...)
```

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

Module contents

class hip.util.**SSLContext**
Bases: _ssl._SSLContext

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

sslsocket_class
alias of SSLSocket

sslobject_class
alias of SSLObject

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

wrap_bio(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

set_npn_protocols(*npn_protocols*)

set_servername_callback(*server_name_callback*)

set_alpn_protocols(*alpn_protocols*)

load_default_certs(*purpose=<Purpose.SERVER_AUTH: _ASN1Object(nid=129, shortname='serverAuth', longname='TLS Web Server Authentication', oid='1.3.6.1.5.5.7.3.1')>*)

options

hostname_checks_common_name

protocol

verify_flags

verify_mode

class hip.util.**Retry**(*total=10*, *connect=None*, *read=None*, *redirect=None*, *status=None*, *method_whitelist=frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTIONS'})*, *status_forcelist=None*, *backoff_factor=0*, *raise_on_redirect=True*, *raise_on_status=True*, *history=None*, *respect_retry_after_header=True*, *remove_headers_on_redirect=frozenset({'Authorization'})*)
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). Hip 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.

- **remove_headers_on_redirect** (*iterable*) – Sequence of headers to remove from the request when a response indicating a redirect is returned before firing off the redirected request.

DEFAULT_METHOD_WHITELIST = frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTION

RETRY_AFTER_STATUS_CODES = frozenset({503, 413, 429})

DEFAULT_REDIRECT_HEADERS_BLACKLIST = frozenset({'Authorization'})

BACKOFF_MAX = 120
Maximum backoff time.

new(***kw*)

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

parse_retry_after(*retry_after*)

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

sleep_for_retry(*response=None*)

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.

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)

is_exhausted()
Are we out of retries?

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.

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

class hip.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 (in seconds) 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 (in seconds) 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 Hip 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

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*

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*

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

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

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

Returns Elapsed time in seconds.

Return type float

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

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

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

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

Data structure 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.

request_uri
Absolute path including the query string.

netloc
Network location including host and port

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'
```

hip.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.

hip.util.**current_time**()
monotonic() -> float

Monotonic clock, cannot go backward.

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

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

hip.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. This parser is RFC 3986 compliant.

The parser logic and helper functions are based heavily on work done in the `rfc3986` module.

Parameters **url** (*str*) – URL to parse into a *Url* namedtuple.

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', ...)
```

hip.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-hip/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'}
```

hip.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 abbreviation. (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.

hip.util.**resolve_ssl_version**(*candidate*)
like resolve_cert_reqs

hip.util.**split_first**(*s*, *delims*)
Deprecated since version 1.25.

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.

hip.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*, *key_password=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_ssl_context().
- **ciphers** – A string of ciphers we wish the client to support.
- **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().
- **key_password** – Optional password if the keyfile is encrypted.

hip.util.**wait_for_read**(*sock*, *timeout=None*)
Waits for reading to be available on a given socket. Returns True if the socket is readable, or False if the timeout expired.

hip.util.**wait_for_write**(*sock*, *timeout=None*)
Waits for writing to be available on a given socket. Returns True if the socket is readable, or False if the timeout expired.

hip.util.**wait_for_socket**(**args*, ***kwargs*)

exception hip.util.**SSLWantReadError**
Bases: ssl.SSLError

Non-blocking SSL socket needs to read more data before the requested operation can be completed.

exception hip.util.**SSLWantWriteError**
Bases: ssl.SSLError

Non-blocking SSL socket needs to write more data before the requested operation can be completed.

3.2 Submodules

3.3 hip.connectionpool module

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

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

scheme = None

QueueCls
alias of hip.util.queue.LifoQueue

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

class hip.connectionpool.**HTTPConnectionPool**(*host*, *port=None*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, ***conn_kw*)
Bases: hip._sync.connectionpool.ConnectionPool, hip._sync.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 *hip.util.Timeout* which gives you more fine-grained control over request timeouts. After the constructor has been parsed, this is always a *hip.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 hip.connectionpool.ProxyManager"

- **_proxy_headers** – A dictionary with proxy headers, should not be used directly, instead, see hip.connectionpool.ProxyManager"
- ****conn_kw** – Additional parameters are used to create fresh hip.connection.HTTPConnection, hip.connection.HTTPSConnection instances.

scheme = 'http'

ConnectionCls
alias of hip._sync.connection.HTTP1Connection

ResponseCls
alias of hip._sync.response.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.

urlopen(*method*, *url*, *body=None*, *headers=None*, *retries=None*, *timeout=<object object>*, *pool_timeout=None*, *body_pos=None*, *preload_content=True*, ***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** (*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.
- **timeout** – If specified, overrides the default timeout for this one request. It may be a float (in seconds) or an instance of *hip.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.
- **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 hip will auto-populate the value when needed.
- **preload_content** – If True, the response's body will be preloaded during construction.

- ****response_kw** – Additional parameters are passed to *hip.response.HTTPResponse.from_base()*

class hip.connectionpool.**HTTPSConnectionPool**(*host*, *port=None*, *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*, *key_password=None*, *ca_certs=None*, *ssl_version=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*, *ssl_context=None*, *server_hostname=None*, ***conn_kw*)

Bases: hip._sync.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, ssl_version, key_password are only used if ssl is available and are fed into *hip.util.ssl_wrap_socket()* to upgrade the connection socket into an SSL socket.

scheme = 'https'

hip.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', '/')
```

3.4 hip.exceptions module

exception hip.exceptions.**HTTPError**

Bases: Exception

Base exception used by this module.

exception hip.exceptions.**HTTPWarning**

Bases: Warning

Base warning used by this module.

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

Base exception for errors caused within a pool.

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

Base exception for PoolErrors that have associated URLs.

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

Raised when SSL certificate fails in an HTTPS connection.

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

Raised when the connection to a proxy fails.

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

Raised when automatic decoding based on Content-Type fails.

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

Raised when something unexpected happens mid-request/response.

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

alias of *hip.exceptions.ProtocolError*

exception hip.exceptions.**MaxRetryError**(*pool*, *url*, *reason=None*)
Bases: *hip.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 hip.exceptions.**TimeoutStateError**
Bases: *hip.exceptions.HTTPError*

Raised when passing an invalid state to a timeout

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

Raised when a socket timeout error occurs.

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

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

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

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

Raised when a socket timeout occurs while connecting to a server

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

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

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

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

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

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

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

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

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

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

exception hip.exceptions.**ResponseError**
Bases: *hip.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 hip.exceptions.**SecurityWarning**
Bases: *hip.exceptions.HTTPWarning*

Warned when performing security reducing actions

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

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

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

Warned when making an unverified HTTPS request.

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

Warned when system time is suspected to be wrong

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

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

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

Warned when making a HTTPS request without SNI available.

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

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

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

The header provided was somehow invalid.

exception hip.exceptions.**BadVersionError**(*version*)
Bases: *hip.exceptions.ProtocolError*

The HTTP version in the response is unsupported.

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

ProxyManager does not support the supplied scheme

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

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

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

Hip encountered an error when trying to rewind a body

exception hip.exceptions.**FailedTunnelError**(*message*, *response*)
Bases: *hip.exceptions.HTTPError*

An attempt was made to set up a CONNECT tunnel, but that attempt failed.

exception hip.exceptions.**InvalidBodyError**
Bases: *hip.exceptions.HTTPError*

An attempt was made to send a request with a body object that Hip does not support.

3.5 hip.fields module

hip.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.

hip.fields.**format_header_param_rfc2231**(*name*, *value*)
Helper function to format and quote a single header parameter using the strategy defined in RFC 2231.

Particularly useful for header parameters which might contain non-ASCII values, like file names. This follows 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 bytes or str`.

Ret An RFC-2231-formatted unicode string.

hip.fields.**format_header_param_html5**(*name*, *value*)
Helper function to format and quote a single header parameter using the HTML5 strategy.

Particularly useful for header parameters which might contain non-ASCII values, like file names. This follows the HTML5 Working Draft Section 4.10.22.7 and matches the behavior of curl and modern browsers.

Parameters

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

Ret A unicode string, stripped of troublesome characters.

hip.fields.**format_header_param**(*name*, *value*)
Helper function to format and quote a single header parameter using the HTML5 strategy.

Particularly useful for header parameters which might contain non-ASCII values, like file names. This follows the HTML5 Working Draft Section 4.10.22.7 and matches the behavior of curl and modern browsers.

Parameters

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

Ret A unicode string, stripped of troublesome characters.

class hip.fields.**RequestField**(*name*, *data*, *filename=None*, *headers=None*, *header_formatter=<function format_header_param_html5>*)
Bases: `object`

A data container for request body parameters.

Parameters

- **name** – The name of this request field. Must be unicode.
- **data** – The data/value body.
- **filename** – An optional filename of the request field. Must be unicode.
- **headers** – An optional dict-like object of headers to initially use for the field.
- **header_formatter** – An optional callable that is used to encode and format the headers. By default, this is *format_header_param_html5()*.

classmethod from_tuples(*fieldname*, *value*, *header_formatter=<function format_header_param_html5>*)
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.

render_headers()
Renders the headers for this request field.

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.

3.6 hip.filepost module

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

for ... in hip.filepost.**iter_field_objects**(*fields*)
Iterate over fields.

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

hip.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.

hip.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 *hip.filepost.choose_boundary()*.

3.7 hip.poolmanager module

class hip.poolmanager.**PoolManager**(*num_pools=10*, *headers=None*, *backend=None*, ***connection_pool_kw*)
Bases: `hip._sync.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 *hip.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
```

proxy = None

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_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 `hip.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_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_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 *hip.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 *hip.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.

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

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

class hip.poolmanager.**ProxyManager**(*proxy_url*, *num_pools=10*, *headers=None*, *proxy_headers=None*, ***connection_pool_kw*)
Bases: `hip._sync.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 containing 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 = hip.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.

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

3.8 hip.request module

class hip.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.

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

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_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.

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 *hip.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 mimic 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.

3.9 hip.response module

class hip.response.**DeflateDecoder**

Bases: `object`

decompress(*data*)

class hip.response.**GzipDecoder**

Bases: `object`

decompress(*data*)

class hip.response.**HTTPResponse**(*body=''*, *headers=None*, *status=0*, *version=0*, *reason=None*, *strict=0*, *decode_content=True*, *original_response=None*, *pool=None*, *connection=None*, *msg=None*, *retries=None*, *enforce_content_length=False*, *request_method=None*, *request_url=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

- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.
- **retries** – The retries contains the last *Retry* that was used during the request.

CONTENT_DECODERS = ['gzip', 'deflate']

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

preload_content()

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.

release_conn()

data

connection

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).

DECODER_ERROR_CLASSES = (<class 'OSError'>, <class 'zlib.error'>)

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.)

for ... in stream(*decode_content=None*)
A generator wrapper for the read() method.

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

classmethod from_base(*r*, ***response_kw*)
Given an hip.base.Response instance r, return a corresponding *hip.response.HTTPResponse* object.

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

getheaders()

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

info()

close()

closed

fileno()

readable()

readinto(*b*)

3.10 Module contents

Hip: A new Python HTTP client for Everyone

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

Returns the handler after adding it.

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

class hip.**HTTPConnectionPool**(*host*, *port=None*, *timeout=<object object>*, *maxsize=1*, *block=False*, *headers=None*, *retries=None*, *_proxy=None*, *_proxy_headers=None*, ***conn_kw*)
Bases: hip._sync.connectionpool.ConnectionPool, hip._sync.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 *hip.util.Timeout* which gives you more fine-grained control over request timeouts. After the constructor has been parsed, this is always a *hip.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 `hip.connectionpool.ProxyManager`"
- **_proxy_headers** – A dictionary with proxy headers, should not be used directly, instead, see `hip.connectionpool.ProxyManager`"
- ****conn_kw** – Additional parameters are used to create fresh `hip.connection.HTTPConnection`, `hip.connection.HTTPSConnection` instances.

scheme = 'http'

ConnectionCls
alias of `hip._sync.connection.HTTP1Connection`

ResponseCls
alias of `hip._sync.response.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.

urlopen(*method*, *url*, *body=None*, *headers=None*, *retries=None*, *timeout=<object object>*, *pool_timeout=None*, *body_pos=None*, *preload_content=True*, ***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** (*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.
- **timeout** – If specified, overrides the default timeout for this one request. It may be a float (in seconds) or an instance of *hip.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.
- **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 hip will auto-populate the value when needed.

- **preload_content** – If True, the response's body will be preloaded during construction.
- ****response_kw** – Additional parameters are passed to *hip.response.HTTPResponse.from_base()*

class hip.**HTTPSConnectionPool**(*host*, *port=None*, *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*, *key_password=None*, *ca_certs=None*, *ssl_version=None*, *assert_hostname=None*, *assert_fingerprint=None*, *ca_cert_dir=None*, *ssl_context=None*, *server_hostname=None*, ***conn_kw*)

Bases: hip._sync.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, ssl_version, key_password are only used if ssl is available and are fed into *hip.util.ssl_wrap_socket()* to upgrade the connection socket into an SSL socket.

scheme = 'https'

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

Bases: hip._sync.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 *hip.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
```

proxy = None

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_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 `hip.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_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_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 *hip.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 *hip.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.

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

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

class hip.**ProxyManager**(*proxy_url*, *num_pools=10*, *headers=None*, *proxy_headers=None*, ***connection_pool_kw*)
Bases: `hip._sync.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 containing 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 = hip.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 hip.**HTTPResponse**(*body=''*, *headers=None*, *status=0*, *version=0*, *reason=None*, *strict=0*, *decode_content=True*, *original_response=None*, *pool=None*, *connection=None*, *msg=None*, *retries=None*, *enforce_content_length=False*, *request_method=None*, *request_url=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

- **decode_content** – If True, will attempt to decode the body based on the 'content-encoding' header.
- **retries** – The retries contains the last *Retry* that was used during the request.

CONTENT_DECODERS = ['gzip', 'deflate']

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

preload_content()

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.

release_conn()

data

connection

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).

DECODER_ERROR_CLASSES = (<class 'OSError'>, <class 'zlib.error'>)

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.)

for ... in stream(*decode_content=None*)
A generator wrapper for the read() method.

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

classmethod from_base(*r*, ***response_kw*)
Given an `hip.base.Response` instance `r`, return a corresponding *hip.response.HTTPResponse* object.

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

getheaders()

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

info()

close()

closed

fileno()

readable()

readinto(*b*)

class hip.**Retry**(*total=10*, *connect=None*, *read=None*, *redirect=None*, *status=None*, *method_whitelist=frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTIONS'})*, *status_forcelist=None*, *backoff_factor=0*, *raise_on_redirect=True*, *raise_on_status=True*, *history=None*, *respect_retry_after_header=True*, *remove_headers_on_redirect=frozenset({'Authorization'})*)
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). Hip 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.
- **remove_headers_on_redirect** (*iterable*) – Sequence of headers to remove from the request when a response indicating a redirect is returned before firing off the redirected request.

DEFAULT_METHOD_WHITELIST = frozenset({'GET', 'HEAD', 'DELETE', 'PUT', 'TRACE', 'OPTION

RETRY_AFTER_STATUS_CODES = frozenset({503, 413, 429})

DEFAULT_REDIRECT_HEADERS_BLACKLIST = frozenset({'Authorization'})

BACKOFF_MAX = 120
Maximum backoff time.

new(***kw*)

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

parse_retry_after(*retry_after*)

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

sleep_for_retry(*response=None*)

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.

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)

is_exhausted()
Are we out of retries?

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.

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

class hip.**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 (in seconds) 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 (in seconds) 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 Hip 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

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*

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*

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

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

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

Returns Elapsed time in seconds.

Return type float

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

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

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

hip.**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', '/')
```

hip.**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 *hip.filepost.choose_boundary()*.

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

hip.**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-hip/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'}
```

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

hip.**AsyncHTTPConnectionPool**
alias of `hip._async.connectionpool.HTTPConnectionPool`

hip.**AsyncHTTPSConnectionPool**
alias of hip._async.connectionpool.HTTPSConnectionPool

hip.**AsyncPoolManager**
alias of hip._async.poolmanager.PoolManager

hip.**AsyncProxyManager**
alias of hip._async.poolmanager.ProxyManager

hip.**AsyncHTTPResponse**
alias of hip._async.response.HTTPResponse

CHAPTER 4

Contributing

Hip 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 Hip 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. Format your changes with black using command `$ nox -s blacken` and lint your changes using command `nox -s lint`.
5. Send a pull request and bug the maintainer until it gets merged and published.

4.1 Setting up your development environment

In order to setup the development environment all that you need is nox installed in your machine:

```
$ python -m pip install --user --upgrade nox
```

4.2 Running the tests

We use some external dependencies, multiple interpreters and code coverage analysis while running test suite. Our `noxfile.py` handles much of this for you:

```
$ nox --sessions test-2.7 test-3.7
[ Nox will create virtualenv, install the specified dependencies, and run the␣
↪commands in order.]
```

(continues on next page)

(continued from previous page)

```
nox > Running session test-2.7
.......
.......
nox > Session test-2.7 was successful.
.......
.......
nox > Running session test-3.7
.......
.......
nox > Session test-3.7 was successful.
```

There is also a nox command for running all of our tests and multiple python versions.

$ nox –sessions test

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 Hip's supported platforms, you can run our `tox` suite:

```
$ nox --sessions test
[ Nox will create virtualenv, install the specified dependencies, and run the␣
↪commands in order.]
.......
.......
nox > Session test-2.7 was successful.
nox > Session test-3.4 was successful.
nox > Session test-3.5 was successful.
nox > Session test-3.6 was successful.
nox > Session test-3.7 was successful.
nox > Session test-3.8 was successful.
nox > Session test-pypy was successful.
```

4.3 Releases

A release candidate can be created by any contributor by creating a branch named `release-x.x` where `x.x` is the version of the proposed release.

- Update `CHANGES.rst` and `hip/__init__.py` with the proper version number and commit the changes to `release-x.x`.
- Open a pull request to merge the `release-x.x` branch into the `master` branch.
- Integration tests are run against the release candidate on Travis. From here on all the steps below will be handled by a maintainer so unless you receive review comments you are done here.
- Once the pull request is squash merged into master the merging maintainer will tag the merge commit with the version number:
  - `git tag -a 1.24.1 [commit sha]`
  - `git push origin master --tags`
- After the commit is tagged Travis will build the tagged commit and upload the sdist and wheel to PyPI and create a draft release on GitHub for the tag. The merging maintainer will ensure that the PyPI sdist and wheel are properly uploaded.
- The merging maintainer will mark the draft release on GitHub as an approved release.

CHAPTER 5

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 6

License

Hip is made available under both the MIT License and Apache-2.0 License. For more details, see LICENSE.

CHAPTER 7

Contributing

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

Python Module Index

h

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W