Requests Documentation

Release 2.18.1

Kenneth Reitz

Jun 16, 2017

Contents

Release v2.18.1. (*Installation*) **Requests** is the only *Non-GMO* HTTP library for Python, safe for human consumption.

Warning: Recreational use of the Python standard library for HTTP may result in dangerous side-effects, including: security vulnerabilities, verbose code, reinventing the wheel, constantly reading documentation, depression, headaches, or even death.

Behold, the power of Requests:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

See similar code, sans Requests.

Requests allows you to send *organic, grass-fed* HTTP/1.1 requests, without the need for manual labor. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3.

CHAPTER 1

User Testimonials

Twitter, Spotify, Microsoft, Amazon, Lyft, BuzzFeed, Reddit, The NSA, Her Majesty's Government, Google, Twilio, Runscope, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, SoundCloud, Kippt, Sony, and Federal U.S. Institutions that prefer to be unnamed claim to use Requests internally.

Armin Ronacher— *Requests is the perfect example how beautiful an API can be with the right level of abstraction.*

Matt DeBoard— *I'm going to get Kenneth Reitz's Python requests module tattooed on my body, somehow. The whole thing.*

Daniel Greenfeld— *Nuked a 1200 LOC spaghetti code library with 10 lines of code thanks to Kenneth Reitz's request library. Today has been AWESOME.*

Kenny Meyers— *Python HTTP: When in doubt, or when not in doubt, use Requests. Beautiful, simple, Pythonic.*

Requests is one of the most downloaded Python packages of all time, pulling in over 11,000,000 downloads every month. All the cool kids are doing it!

CHAPTER 2

Beloved Features

Requests is ready for today's web.

- Keep-Alive & Connection Pooling
- International Domains and URLs
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Automatic Content Decoding
- Basic/Digest Authentication
- Elegant Key/Value Cookies
- Automatic Decompression
- Unicode Response Bodies
- HTTP(S) Proxy Support
- Multipart File Uploads
- Streaming Downloads
- Connection Timeouts
- Chunked Requests
- `.netrc` Support

Requests officially supports Python 2.6–2.7 & 3.3–3.7, and runs great on PyPy.

CHAPTER 3

The User Guide

This part of the documentation, which is mostly prose, begins with some background information about Requests, then focuses on step-by-step instructions for getting the most out of Requests.

Introduction

Philosophy

Requests was developed with a few **PEP 20** idioms in mind.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

All contributions to Requests should keep these important rules in mind.

Apache2 License

A large number of open source projects you find today are GPL Licensed. While the GPL has its time and place, it should most certainly not be your go-to license for your next open source project.

A project that is released as GPL cannot be used in any commercial product without the product itself also being offered as open source.

The MIT, BSD, ISC, and Apache2 licenses are great alternatives to the GPL that allow your open-source software to be used freely in proprietary, closed-source software.

Requests is released under terms of Apache2 License.

Requests License

Installation of Requests

This part of the documentation covers the installation of Requests. The first step to using any software package is getting it properly installed.

$ pip install requests

To install Requests, simply run this simple command in your terminal of choice:

```
$ pip install requests
```

If you don't have pip installed (tisk tisk!), this Python installation guide can guide you through the process.

Get the Source Code

Requests is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git://github.com/requests/requests.git
```

Or, download the tarball:

```
$ curl -OL https://github.com/requests/requests/tarball/master
# optionally, zipball is also available (for Windows users).
```

Once you have a copy of the source, you can embed it in your own Python package, or install it into your site-packages easily:

```
$ cd requests
$ pip install .
```

Quickstart

Eager to get started? This page gives a good introduction in how to get started with Requests.

First, make sure that:

- Requests is *installed*

- Requests is *up-to-date*

Let's get started with some simple examples.

Make a Request

Making a request with Requests is very simple.

Begin by importing the Requests module:

```
>>> import requests
```

Now, let's try to get a webpage. For this example, let's get GitHub's public timeline:

```
>>> r = requests.get('https://api.github.com/events')
```

Now, we have a *Response* object called `r`. We can get all the information we need from this object.

Requests' simple API means that all forms of HTTP request are as obvious. For example, this is how you make an HTTP POST request:

```
>>> r = requests.post('http://httpbin.org/post', data = {'key':'value'})
```

Nice, right? What about the other HTTP request types: PUT, DELETE, HEAD and OPTIONS? These are all just as simple:

```
>>> r = requests.put('http://httpbin.org/put', data = {'key':'value'})
>>> r = requests.delete('http://httpbin.org/delete')
>>> r = requests.head('http://httpbin.org/get')
>>> r = requests.options('http://httpbin.org/get')
```

That's all well and good, but it's also only the start of what Requests can do.

Passing Parameters In URLs

You often want to send some sort of data in the URL's query string. If you were constructing the URL by hand, this data would be given as key/value pairs in the URL after a question mark, e.g. `httpbin.org/get?key=val`. Requests allows you to provide these arguments as a dictionary of strings, using the `params` keyword argument. As an example, if you wanted to pass `key1=value1` and `key2=value2` to `httpbin.org/get`, you would use the following code:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get('http://httpbin.org/get', params=payload)
```

You can see that the URL has been correctly encoded by printing the URL:

```
>>> print(r.url)
http://httpbin.org/get?key2=value2&key1=value1
```

Note that any dictionary key whose value is `None` will not be added to the URL's query string.

You can also pass a list of items as a value:

```
>>> payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

>>> r = requests.get('http://httpbin.org/get', params=payload)
```

```
>>> print(r.url)
http://httpbin.org/get?key1=value1&key2=value2&key2=value3
```

Response Content

We can read the content of the server's response. Consider the GitHub timeline again:

```
>>> import requests

>>> r = requests.get('https://api.github.com/events')
>>> r.text
u'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests will automatically decode content from the server. Most unicode charsets are seamlessly decoded.

When you make a request, Requests makes educated guesses about the encoding of the response based on the HTTP headers. The text encoding guessed by Requests is used when you access `r.text`. You can find out what encoding Requests is using, and change it, using the `r.encoding` property:

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

If you change the encoding, Requests will use the new value of `r.encoding` whenever you call `r.text`. You might want to do this in any situation where you can apply special logic to work out what the encoding of the content will be. For example, HTTP and XML have the ability to specify their encoding in their body. In situations like this, you should use `r.content` to find the encoding, and then set `r.encoding`. This will let you use `r.text` with the correct encoding.

Requests will also use custom encodings in the event that you need them. If you have created your own encoding and registered it with the `codecs` module, you can simply use the codec name as the value of `r.encoding` and Requests will handle the decoding for you.

Binary Response Content

You can also access the response body as bytes, for non-text requests:

```
>>> r.content
b'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

The `gzip` and `deflate` transfer-encodings are automatically decoded for you.

For example, to create an image from binary data returned by a request, you can use the following code:

```
>>> from PIL import Image
>>> from io import BytesIO

>>> i = Image.open(BytesIO(r.content))
```

JSON Response Content

There's also a builtin JSON decoder, in case you're dealing with JSON data:

```
>>> import requests

>>> r = requests.get('https://api.github.com/events')
>>> r.json()
[{u'repository': {u'open_issues': 0, u'url': 'https://github.com/...
```

In case the JSON decoding fails, `r.json()` raises an exception. For example, if the response gets a 204 (No Content), or if the response contains invalid JSON, attempting `r.json()` raises `ValueError: No JSON object could be decoded`.

It should be noted that the success of the call to `r.json()` does **not** indicate the success of the response. Some servers may return a JSON object in a failed response (e.g. error details with HTTP 500). Such JSON will be decoded and returned. To check that a request is successful, use `r.raise_for_status()` or check `r.status_code` is what you expect.

Raw Response Content

In the rare case that you'd like to get the raw socket response from the server, you can access `r.raw`. If you want to do this, make sure you set `stream=True` in your initial request. Once you do, you can do this:

```
>>> r = requests.get('https://api.github.com/events', stream=True)

>>> r.raw
<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>

>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

In general, however, you should use a pattern like this to save what is being streamed to a file:

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size=128):
        fd.write(chunk)
```

Using `Response.iter_content` will handle a lot of what you would otherwise have to handle when using `Response.raw` directly. When streaming a download, the above is the preferred and recommended way to retrieve the content. Note that `chunk_size` can be freely adjusted to a number that may better fit your use cases.

Custom Headers

If you'd like to add HTTP headers to a request, simply pass in a `dict` to the `headers` parameter.

For example, we didn't specify our user-agent in the previous example:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> headers = {'user-agent': 'my-app/0.0.1'}

>>> r = requests.get(url, headers=headers)
```

Note: Custom headers are given less precedence than more specific sources of information. For instance:

- Authorization headers set with *headers=* will be overridden if credentials are specified in `.netrc`, which in turn will be overridden by the `auth=` parameter.
- Authorization headers will be removed if you get redirected off-host.

- Proxy-Authorization headers will be overridden by proxy credentials provided in the URL.
- Content-Length headers will be overridden when we can determine the length of the content.

Furthermore, Requests does not change its behavior at all based on which custom headers are specified. The headers are simply passed on into the final request.

Note: All header values must be a `string`, bytestring, or unicode. While permitted, it's advised to avoid passing unicode header values.

More complicated POST requests

Typically, you want to send some form-encoded data — much like an HTML form. To do this, simply pass a dictionary to the `data` argument. Your dictionary of data will automatically be form-encoded when the request is made:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}

>>> r = requests.post("http://httpbin.org/post", data=payload)
>>> print(r.text)
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

You can also pass a list of tuples to the `data` argument. This is particularly useful when the form has multiple elements that use the same key:

```
>>> payload = (('key1', 'value1'), ('key1', 'value2'))
>>> r = requests.post('http://httpbin.org/post', data=payload)
>>> print(r.text)
{
  ...
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
```

There are times that you may want to send data that is not form-encoded. If you pass in a `string` instead of a `dict`, that data will be posted directly.

For example, the GitHub API v3 accepts JSON-Encoded POST/PATCH data:

```
>>> import json

>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, data=json.dumps(payload))
```

Instead of encoding the `dict` yourself, you can also pass it directly using the `json` parameter (added in version 2.4.2) and it will be encoded automatically:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, json=payload)
```

POST a Multipart-Encoded File

Requests makes it simple to upload Multipart-encoded files:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

You can set the filename, content_type and headers explicitly:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-
↪excel', {'Expires': '0'})}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

If you want, you can send strings to be received as files:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "some,data,to,send\\nanother,row,to,send\\n"
  },
  ...
}
```

In the event you are posting a very large file as a `multipart/form-data` request, you may want to stream the request. By default, `requests` does not support this, but there is a separate package which does - `requests-toolbelt`. You should read the toolbelt's documentation for more details about how to use it.

For sending multiple files in one request refer to the *advanced* section.

Warning: It is strongly recommended that you open files in binary mode. This is because Requests may attempt to provide the `Content-Length` header for you, and if it does this value will be set to the number of *bytes* in the file. Errors may occur if you open the file in *text mode*.

Response Status Codes

We can check the response status code:

```
>>> r = requests.get('http://httpbin.org/get')
>>> r.status_code
200
```

Requests also comes with a built-in status code lookup object for easy reference:

```
>>> r.status_code == requests.codes.ok
True
```

If we made a bad request (a 4XX client error or 5XX server error response), we can raise it with *`Response.raise_for_status()`*:

```
>>> bad_r = requests.get('http://httpbin.org/status/404')
>>> bad_r.status_code
404

>>> bad_r.raise_for_status()
Traceback (most recent call last):
  File "requests/models.py", line 832, in raise_for_status
    raise http_error
requests.exceptions.HTTPError: 404 Client Error
```

But, since our `status_code` for `r` was `200`, when we call `raise_for_status()` we get:

```
>>> r.raise_for_status()
None
```

All is well.

Response Headers

We can view the server's response headers using a Python dictionary:

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
```

```
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

The dictionary is special, though: it's made just for HTTP headers. According to RFC 7230, HTTP Header names are case-insensitive.

So, we can access the headers using any capitalization we want:

```
>>> r.headers['Content-Type']
'application/json'

>>> r.headers.get('content-type')
'application/json'
```

It is also special in that the server could have sent the same header multiple times with different values, but requests combines them so they can be represented in the dictionary within a single mapping, as per RFC 7230:

> A recipient MAY combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to the combined field value in order, separated by a comma.

Cookies

If a response contains some Cookies, you can quickly access them:

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)

>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

To send your own cookies to the server, you can use the `cookies` parameter:

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')

>>> r = requests.get(url, cookies=cookies)
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

Cookies are returned in a *`RequestsCookieJar`*, which acts like a `dict` but also offers a more complete interface, suitable for use over multiple domains or paths. Cookie jars can also be passed in to requests:

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('tasty_cookie', 'yum', domain='httpbin.org', path='/cookies')
>>> jar.set('gross_cookie', 'blech', domain='httpbin.org', path='/elsewhere')
>>> url = 'http://httpbin.org/cookies'
>>> r = requests.get(url, cookies=jar)
>>> r.text
'{"cookies": {"tasty_cookie": "yum"}}'
```

Redirection and History

By default Requests will perform location redirection for all verbs except HEAD.

We can use the `history` property of the Response object to track redirection.

The *`Response.history`* list contains the *`Response`* objects that were created in order to complete the request. The list is sorted from the oldest to the most recent response.

For example, GitHub redirects all HTTP requests to HTTPS:

```
>>> r = requests.get('http://github.com')

>>> r.url
'https://github.com/'

>>> r.status_code
200

>>> r.history
[<Response [301]>]
```

If you're using GET, OPTIONS, POST, PUT, PATCH or DELETE, you can disable redirection handling with the `allow_redirects` parameter:

```
>>> r = requests.get('http://github.com', allow_redirects=False)

>>> r.status_code
301

>>> r.history
[]
```

If you're using HEAD, you can enable redirection as well:

```
>>> r = requests.head('http://github.com', allow_redirects=True)

>>> r.url
'https://github.com/'

>>> r.history
[<Response [301]>]
```

Timeouts

You can tell Requests to stop waiting for a response after a given number of seconds with the `timeout` parameter. Nearly all production code should use this parameter in nearly all requests. Failure to do so can cause your program to hang indefinitely:

```
>>> requests.get('http://github.com', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
requests.exceptions.Timeout: HTTPConnectionPool(host='github.com', port=80): Request␣
↪timed out. (timeout=0.001)
```

Note

`timeout` is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for `timeout` seconds (more precisely, if no bytes have been received on the underlying socket for `timeout` seconds). If no timeout is specified explicitly, requests do not time out.

Errors and Exceptions

In the event of a network problem (e.g. DNS failure, refused connection, etc), Requests will raise a `ConnectionError` exception.

`Response.raise_for_status()` will raise an `HTTPError` if the HTTP request returned an unsuccessful status code.

If a request times out, a `Timeout` exception is raised.

If a request exceeds the configured number of maximum redirections, a `TooManyRedirects` exception is raised.

All exceptions that Requests explicitly raises inherit from `requests.exceptions.RequestException`.

Ready for more? Check out the *advanced* section.

Advanced Usage

This document covers some of Requests more advanced features.

Session Objects

The Session object allows you to persist certain parameters across requests. It also persists cookies across all requests made from the Session instance, and will use `urllib3`'s connection pooling. So if you're making several requests to the same host, the underlying TCP connection will be reused, which can result in a significant performance increase (see HTTP persistent connection).

A Session object has all the methods of the main Requests API.

Let's persist some cookies across requests:

```
s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get('http://httpbin.org/cookies')

print(r.text)
# '{"cookies": {"sessioncookie": "123456789"}}'
```

Sessions can also be used to provide default data to the request methods. This is done by providing data to the properties on a Session object:

```
s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'})

# both 'x-test' and 'x-test2' are sent
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

Any dictionaries that you pass to a request method will be merged with the session-level values that are set. The method-level parameters override session parameters.

Note, however, that method-level parameters will *not* be persisted across requests, even if using a session. This example will only send the cookies with the first request, but not the second:

```
s = requests.Session()

r = s.get('http://httpbin.org/cookies', cookies={'from-my': 'browser'})
print(r.text)
# '{"cookies": {"from-my": "browser"}}'

r = s.get('http://httpbin.org/cookies')
print(r.text)
# '{"cookies": {}}'
```

If you want to manually add cookies to your session, use the *Cookie utility functions* to manipulate *Session.cookies*.

Sessions can also be used as context managers:

```
with requests.Session() as s:
    s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
```

This will make sure the session is closed as soon as the `with` block is exited, even if unhandled exceptions occurred.

Remove a Value From a Dict Parameter

Sometimes you'll want to omit session-level keys from a dict parameter. To do this, you simply set that key's value to `None` in the method-level parameter. It will automatically be omitted.

All values that are contained within a session are directly available to you. See the *Session API Docs* to learn more.

Request and Response Objects

Whenever a call is made to `requests.get()` and friends, you are doing two major things. First, you are constructing a `Request` object which will be sent off to a server to request or query some resource. Second, a `Response` object is generated once Requests gets a response back from the server. The `Response` object contains all of the information returned by the server and also contains the `Request` object you created originally. Here is a simple request to get some very important information from Wikipedia's servers:

```
>>> r = requests.get('http://en.wikipedia.org/wiki/Monty_Python')
```

If we want to access the headers the server sent back to us, we do this:

```
>>> r.headers
{'content-length': '56170', 'x-content-type-options': 'nosniff', 'x-cache':
'HIT from cp1006.eqiad.wmnet, MISS from cp1010.eqiad.wmnet', 'content-encoding':
'gzip', 'age': '3080', 'content-language': 'en', 'vary': 'Accept-Encoding,Cookie',
'server': 'Apache', 'last-modified': 'Wed, 13 Jun 2012 01:33:50 GMT',
'connection': 'close', 'cache-control': 'private, s-maxage=0, max-age=0,
must-revalidate', 'date': 'Thu, 14 Jun 2012 12:59:39 GMT', 'content-type':
'text/html; charset=UTF-8', 'x-cache-lookup': 'HIT from cp1006.eqiad.wmnet:3128,
MISS from cp1010.eqiad.wmnet:80'}
```

However, if we want to get the headers we sent the server, we simply access the request, and then the request's headers:

```
>>> r.request.headers
{'Accept-Encoding': 'identity, deflate, compress, gzip',
'Accept': '*/*', 'User-Agent': 'python-requests/1.2.0'}
```

Prepared Requests

Whenever you receive a *Response* object from an API call or a Session call, the `request` attribute is actually the `PreparedRequest` that was used. In some cases you may wish to do some extra work to the body or headers (or anything else really) before sending a request. The simple recipe for this is the following:

```
from requests import Request, Session

s = Session()

req = Request('POST', url, data=data, headers=headers)
prepped = req.prepare()

# do something with prepped.body
prepped.body = 'No, I want exactly this as the body.'

# do something with prepped.headers
del prepped.headers['Content-Type']

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)

print(resp.status_code)
```

Since you are not doing anything special with the `Request` object, you prepare it immediately and modify the `PreparedRequest` object. You then send that with the other parameters you would have sent to `requests.*` or `Session.*`.

However, the above code will lose some of the advantages of having a Requests *Session* object. In particular, *Session*-level state such as cookies will not get applied to your request. To get a *PreparedRequest* with that state applied, replace the call to *Request.prepare()* with a call to *Session.prepare_request()*, like this:

```
from requests import Request, Session

s = Session()
req = Request('GET',  url, data=data, headers=headers)

prepped = s.prepare_request(req)

# do something with prepped.body
prepped.body = 'Seriously, send exactly these bytes.'

# do something with prepped.headers
prepped.headers['Keep-Dead'] = 'parrot'

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)
```

```
print(resp.status_code)
```

SSL Cert Verification

Requests verifies SSL certificates for HTTPS requests, just like a web browser. By default, SSL verification is enabled, and Requests will throw a SSLError if it's unable to verify the certificate:

```
>>> requests.get('https://requestb.in')
requests.exceptions.SSLError: hostname 'requestb.in' doesn't match either of '*.
↪herokuapp.com', 'herokuapp.com'
```

I don't have SSL setup on this domain, so it throws an exception. Excellent. GitHub does though:

```
>>> requests.get('https://github.com')
<Response [200]>
```

You can pass `verify` the path to a CA_BUNDLE file or directory with certificates of trusted CAs:

```
>>> requests.get('https://github.com', verify='/path/to/certfile')
```

or persistent:

```
s = requests.Session()
s.verify = '/path/to/certfile'
```

Note: If `verify` is set to a path to a directory, the directory must have been processed using the c_rehash utility supplied with OpenSSL.

This list of trusted CAs can also be specified through the `REQUESTS_CA_BUNDLE` environment variable.

Requests can also ignore verifying the SSL certificate if you set `verify` to False:

```
>>> requests.get('https://kennethreitz.org', verify=False)
<Response [200]>
```

By default, `verify` is set to True. Option `verify` only applies to host certs.

Client Side Certificates

You can also specify a local cert to use as client side certificate, as a single file (containing the private key and the certificate) or as a tuple of both files' paths:

```
>>> requests.get('https://kennethreitz.org', cert=('/path/client.cert', '/path/client.
↪key'))
<Response [200]>
```

or persistent:

```
s = requests.Session()
s.cert = '/path/client.cert'
```

If you specify a wrong path or an invalid cert, you'll get a SSLError:

```
>>> requests.get('https://kennethreitz.org', cert='/wrong_path/client.pem')
SSLError: [Errno 336265225] _ssl.c:347: error:140B0009:SSL routines:SSL_CTX_use_
↪PrivateKey_file:PEM lib
```

Warning: The private key to your local certificate *must* be unencrypted. Currently, Requests does not support using encrypted keys.

CA Certificates

By default, Requests bundles a set of root CAs that it trusts, sourced from the Mozilla trust store. However, these are only updated once for each Requests version. This means that if you pin a Requests version your certificates can become extremely out of date.

From Requests version 2.4.0 onwards, Requests will attempt to use certificates from certifi if it is present on the system. This allows for users to update their trusted certificates without having to change the code that runs on their system.

For the sake of security we recommend upgrading certifi frequently!

Body Content Workflow

By default, when you make a request, the body of the response is downloaded immediately. You can override this behaviour and defer downloading the response body until you access the *Response.content* attribute with the `stream` parameter:

```
tarball_url = 'https://github.com/requests/requests/tarball/master'
r = requests.get(tarball_url, stream=True)
```

At this point only the response headers have been downloaded and the connection remains open, hence allowing us to make content retrieval conditional:

```
if int(r.headers['content-length']) < TOO_LONG:
  content = r.content
  ...
```

You can further control the workflow by use of the *Response.iter_content()* and *Response.iter_lines()* methods. Alternatively, you can read the undecoded body from the underlying urllib3 `urllib3.HTTPResponse` at *Response.raw*.

If you set `stream` to `True` when making a request, Requests cannot release the connection back to the pool unless you consume all the data or call *Response.close*. This can lead to inefficiency with connections. If you find yourself partially reading request bodies (or not reading them at all) while using `stream=True`, you should make the request within a `with` statement to ensure it's always closed:

```
with requests.get('http://httpbin.org/get', stream=True) as r:
    # Do things with the response here.
```

Keep-Alive

Excellent news — thanks to urllib3, keep-alive is 100% automatic within a session! Any requests that you make within a session will automatically reuse the appropriate connection!

Note that connections are only released back to the pool for reuse once all body data has been read; be sure to either set `stream` to `False` or read the `content` property of the `Response` object.

Streaming Uploads

Requests supports streaming uploads, which allow you to send large streams or files without reading them into memory. To stream and upload, simply provide a file-like object for your body:

```
with open('massive-body', 'rb') as f:
    requests.post('http://some.url/streamed', data=f)
```

Warning: It is strongly recommended that you open files in binary mode. This is because Requests may attempt to provide the `Content-Length` header for you, and if it does this value will be set to the number of *bytes* in the file. Errors may occur if you open the file in *text mode*.

Chunk-Encoded Requests

Requests also supports Chunked transfer encoding for outgoing and incoming requests. To send a chunk-encoded request, simply provide a generator (or any iterator without a length) for your body:

```
def gen():
    yield 'hi'
    yield 'there'

requests.post('http://some.url/chunked', data=gen())
```

For chunked encoded responses, it's best to iterate over the data using *`Response.iter_content()`*. In an ideal situation you'll have set `stream=True` on the request, in which case you can iterate chunk-by-chunk by calling `iter_content` with a `chunk_size` parameter of `None`. If you want to set a maximum size of the chunk, you can set a `chunk_size` parameter to any integer.

POST Multiple Multipart-Encoded Files

You can send multiple files in one request. For example, suppose you want to upload image files to an HTML form with a multiple file field 'images':

```
<input type="file" name="images" multiple="true" required="true"/>
```

To do that, just set files to a list of tuples of `(form_field_name, file_info)`:

```
>>> url = 'http://httpbin.org/post'
>>> multiple_files = [
        ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
        ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]
>>> r = requests.post(url, files=multiple_files)
>>> r.text
{
  ...
  'files': {'images': ' ....'}
  'Content-Type': 'multipart/form-data; boundary=3131623adb2043caaeb5538cc7aa0b3a',
  ...
}
```

Warning: It is strongly recommended that you open files in binary mode. This is because Requests may attempt to provide the `Content-Length` header for you, and if it does this value will be set to the number of *bytes* in the file. Errors may occur if you open the file in *text mode*.

Event Hooks

Requests has a hook system that you can use to manipulate portions of the request process, or signal event handling.

Available hooks:

`response`: The response generated from a Request.

You can assign a hook function on a per-request basis by passing a `{hook_name: callback_function}` dictionary to the `hooks` request parameter:

```
hooks=dict(response=print_url)
```

That `callback_function` will receive a chunk of data as its first argument.

```
def print_url(r, *args, **kwargs):
    print(r.url)
```

If an error occurs while executing your callback, a warning is given.

If the callback function returns a value, it is assumed that it is to replace the data that was passed in. If the function doesn't return anything, nothing else is effected.

Let's print some request method arguments at runtime:

```
>>> requests.get('http://httpbin.org', hooks=dict(response=print_url))
http://httpbin.org
<Response [200]>
```

Custom Authentication

Requests allows you to use specify your own authentication mechanism.

Any callable which is passed as the `auth` argument to a request method will have the opportunity to modify the request before it is dispatched.

Authentication implementations are subclasses of *`AuthBase`*, and are easy to define. Requests provides two common authentication scheme implementations in `requests.auth`: *`HTTPBasicAuth`* and *`HTTPDigestAuth`*.

Let's pretend that we have a web service that will only respond if the `X-Pizza` header is set to a password value. Unlikely, but just go with it.

```
from requests.auth import AuthBase

class PizzaAuth(AuthBase):
    """Attaches HTTP Pizza Authentication to the given Request object."""
    def __init__(self, username):
        # setup any auth-related data here
        self.username = username

    def __call__(self, r):
        # modify and return the request
```

```
        r.headers['X-Pizza'] = self.username
        return r
```

Then, we can make a request using our Pizza Auth:

```
>>> requests.get('http://pizzabin.org/admin', auth=PizzaAuth('kenneth'))
<Response [200]>
```

Streaming Requests

With *Response.iter_lines()* you can easily iterate over streaming APIs such as the Twitter Streaming API. Simply set `stream` to `True` and iterate over the response with *iter_lines*:

```
import json
import requests

r = requests.get('http://httpbin.org/stream/20', stream=True)

for line in r.iter_lines():

    # filter out keep-alive new lines
    if line:
        decoded_line = line.decode('utf-8')
        print(json.loads(decoded_line))
```

When using *decode_unicode=True* with *Response.iter_lines()* or *Response.iter_content()*, you'll want to provide a fallback encoding in the event the server doesn't provide one:

```
r = requests.get('http://httpbin.org/stream/20', stream=True)

if r.encoding is None:
    r.encoding = 'utf-8'

for line in r.iter_lines(decode_unicode=True):
    if line:
        print(json.loads(line))
```

Warning: *iter_lines* is not reentrant safe. Calling this method multiple times causes some of the received data being lost. In case you need to call it from multiple places, use the resulting iterator object instead:

```
lines = r.iter_lines()
# Save the first line for later or just skip it

first_line = next(lines)

for line in lines:
    print(line)
```

Proxies

If you need to use a proxy, you can configure individual requests with the `proxies` argument to any request method:

```
import requests

proxies = {
  'http': 'http://10.10.1.10:3128',
  'https': 'http://10.10.1.10:1080',
}

requests.get('http://example.org', proxies=proxies)
```

You can also configure proxies by setting the environment variables `HTTP_PROXY` and `HTTPS_PROXY`.

```
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

$ python
>>> import requests
>>> requests.get('http://example.org')
```

To use HTTP Basic Auth with your proxy, use the *http://user:password@host/* syntax:

```
proxies = {'http': 'http://user:pass@10.10.1.10:3128/'}
```

To give a proxy for a specific scheme and host, use the *scheme://hostname* form for the key. This will match for any request to the given scheme and exact hostname.

```
proxies = {'http://10.20.1.128': 'http://10.10.1.10:5323'}
```

Note that proxy URLs must include the scheme.

SOCKS

New in version 2.10.0.

In addition to basic HTTP proxies, Requests also supports proxies using the SOCKS protocol. This is an optional feature that requires that additional third-party libraries be installed before use.

You can get the dependencies for this feature from `pip`:

```
$ pip install requests[socks]
```

Once you've installed those dependencies, using a SOCKS proxy is just as easy as using a HTTP one:

```
proxies = {
    'http': 'socks5://user:pass@host:port',
    'https': 'socks5://user:pass@host:port'
}
```

Compliance

Requests is intended to be compliant with all relevant specifications and RFCs where that compliance will not cause difficulties for users. This attention to the specification can lead to some behaviour that may seem unusual to those not familiar with the relevant specification.

Encodings

When you receive a response, Requests makes a guess at the encoding to use for decoding the response when you access the *Response.text* attribute. Requests will first check for an encoding in the HTTP header, and if none is present, will use chardet to attempt to guess the encoding.

The only time Requests will not do this is if no explicit charset is present in the HTTP headers **and** the `Content-Type` header contains `text`. In this situation, RFC 2616 specifies that the default charset must be `ISO-8859-1`. Requests follows the specification in this case. If you require a different encoding, you can manually set the *Response.encoding* property, or use the raw *Response.content*.

HTTP Verbs

Requests provides access to almost the full range of HTTP verbs: GET, OPTIONS, HEAD, POST, PUT, PATCH and DELETE. The following provides detailed examples of using these various verbs in Requests, using the GitHub API.

We will begin with the verb most commonly used: GET. HTTP GET is an idempotent method that returns a resource from a given URL. As a result, it is the verb you ought to use when attempting to retrieve data from a web location. An example usage would be attempting to get information about a specific commit from GitHub. Suppose we wanted commit `a050faf` on Requests. We would get it like so:

```
>>> import requests
>>> r = requests.get('https://api.github.com/repos/requests/requests/git/commits/
↪a050faf084662f3a352dd1a941f2c7c9f886d4ad')
```

We should confirm that GitHub responded correctly. If it has, we want to work out what type of content it is. Do this like so:

```
>>> if r.status_code == requests.codes.ok:
...     print(r.headers['content-type'])
...
application/json; charset=utf-8
```

So, GitHub returns JSON. That's great, we can use the *r.json* method to parse it into Python objects.

```
>>> commit_data = r.json()

>>> print(commit_data.keys())
[u'committer', u'author', u'url', u'tree', u'sha', u'parents', u'message']

>>> print(commit_data[u'committer'])
{u'date': u'2012-05-10T11:10:50-07:00', u'email': u'me@kennethreitz.com', u'name': u
↪'Kenneth Reitz'}

>>> print(commit_data[u'message'])
makin' history
```

So far, so simple. Well, let's investigate the GitHub API a little bit. Now, we could look at the documentation, but we might have a little more fun if we use Requests instead. We can take advantage of the Requests OPTIONS verb to see what kinds of HTTP methods are supported on the url we just used.

```
>>> verbs = requests.options(r.url)
>>> verbs.status_code
500
```

Uh, what? That's unhelpful! Turns out GitHub, like many API providers, don't actually implement the OPTIONS method. This is an annoying oversight, but it's OK, we can just use the boring documentation. If GitHub had correctly implemented OPTIONS, however, they should return the allowed methods in the headers, e.g.

```
>>> verbs = requests.options('http://a-good-website.com/api/cats')
>>> print(verbs.headers['allow'])
GET,HEAD,POST,OPTIONS
```

Turning to the documentation, we see that the only other method allowed for commits is POST, which creates a new commit. As we're using the Requests repo, we should probably avoid making ham-handed POSTS to it. Instead, let's play with the Issues feature of GitHub.

This documentation was added in response to Issue #482. Given that this issue already exists, we will use it as an example. Let's start by getting it.

```
>>> r = requests.get('https://api.github.com/repos/requests/requests/issues/482')
>>> r.status_code
200

>>> issue = json.loads(r.text)

>>> print(issue[u'title'])
Feature any http verb in docs

>>> print(issue[u'comments'])
3
```

Cool, we have three comments. Let's take a look at the last of them.

```
>>> r = requests.get(r.url + u'/comments')
>>> r.status_code
200

>>> comments = r.json()

>>> print(comments[0].keys())
[u'body', u'url', u'created_at', u'updated_at', u'user', u'id']

>>> print(comments[2][u'body'])
Probably in the "advanced" section
```

Well, that seems like a silly place. Let's post a comment telling the poster that he's silly. Who is the poster, anyway?

```
>>> print(comments[2][u'user'][u'login'])
kennethreitz
```

OK, so let's tell this Kenneth guy that we think this example should go in the quickstart guide instead. According to the GitHub API doc, the way to do this is to POST to the thread. Let's do it.

```
>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it!"})
>>> url = u"https://api.github.com/repos/requests/requests/issues/482/comments"

>>> r = requests.post(url=url, data=body)
>>> r.status_code
404
```

Huh, that's weird. We probably need to authenticate. That'll be a pain, right? Wrong. Requests makes it easy to use many forms of authentication, including the very common Basic Auth.

```
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth('fake@example.com', 'not_a_real_password')

>>> r = requests.post(url=url, data=body, auth=auth)
>>> r.status_code
201

>>> content = r.json()
>>> print(content[u'body'])
Sounds great! I'll get right on it.
```

Brilliant. Oh, wait, no! I meant to add that it would take me a while, because I had to go feed my cat. If only I could edit this comment! Happily, GitHub allows us to use another HTTP verb, PATCH, to edit this comment. Let's do that.

```
>>> print(content[u"id"])
5804413

>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it once I feed my␣
↪cat."})
>>> url = u"https://api.github.com/repos/requests/requests/issues/comments/5804413"

>>> r = requests.patch(url=url, data=body, auth=auth)
>>> r.status_code
200
```

Excellent. Now, just to torture this Kenneth guy, I've decided to let him sweat and not tell him that I'm working on this. That means I want to delete this comment. GitHub lets us delete comments using the incredibly aptly named DELETE method. Let's get rid of it.

```
>>> r = requests.delete(url=url, auth=auth)
>>> r.status_code
204
>>> r.headers['status']
'204 No Content'
```

Excellent. All gone. The last thing I want to know is how much of my ratelimit I've used. Let's find out. GitHub sends that information in the headers, so rather than download the whole page I'll send a HEAD request to get the headers.

```
>>> r = requests.head(url=url, auth=auth)
>>> print(r.headers)
...
'x-ratelimit-remaining': '4995'
'x-ratelimit-limit': '5000'
...
```

Excellent. Time to write a Python program that abuses the GitHub API in all kinds of exciting ways, 4995 more times.

Custom Verbs

From time to time you may be working with a server that, for whatever reason, allows use or even requires use of HTTP verbs not covered above. One example of this would be the MKCOL method some WEBDAV servers use. Do not fret, these can still be used with Requests. These make use of the built-in `.request` method. For example:

```
>>> r = requests.request('MKCOL', url, data=data)
>>> r.status_code
200 # Assuming your call was correct
```

Utilising this, you can make use of any method verb that your server allows.

Link Headers

Many HTTP APIs feature Link headers. They make APIs more self describing and discoverable.

GitHub uses these for pagination in their API, for example:

```
>>> url = 'https://api.github.com/users/kennethreitz/repos?page=1&per_page=10'
>>> r = requests.head(url=url)
>>> r.headers['link']
'<https://api.github.com/users/kennethreitz/repos?page=2&per_page=10>; rel="next",
↪<https://api.github.com/users/kennethreitz/repos?page=6&per_page=10>; rel="last"'
```

Requests will automatically parse these link headers and make them easily consumable:

```
>>> r.links["next"]
{'url': 'https://api.github.com/users/kennethreitz/repos?page=2&per_page=10', 'rel':
↪'next'}

>>> r.links["last"]
{'url': 'https://api.github.com/users/kennethreitz/repos?page=7&per_page=10', 'rel':
↪'last'}
```

Transport Adapters

As of v1.0.0, Requests has moved to a modular internal design. Part of the reason this was done was to implement Transport Adapters, originally described here. Transport Adapters provide a mechanism to define interaction methods for an HTTP service. In particular, they allow you to apply per-service configuration.

Requests ships with a single Transport Adapter, the *HTTPAdapter*. This adapter provides the default Requests interaction with HTTP and HTTPS using the powerful urllib3 library. Whenever a Requests *Session* is initialized, one of these is attached to the *Session* object for HTTP, and one for HTTPS.

Requests enables users to create and use their own Transport Adapters that provide specific functionality. Once created, a Transport Adapter can be mounted to a Session object, along with an indication of which web services it should apply to.

```
>>> s = requests.Session()
>>> s.mount('http://www.github.com', MyAdapter())
```

The mount call registers a specific instance of a Transport Adapter to a prefix. Once mounted, any HTTP request made using that session whose URL starts with the given prefix will use the given Transport Adapter.

Many of the details of implementing a Transport Adapter are beyond the scope of this documentation, but take a look at the next example for a simple SSL use- case. For more than that, you might look at subclassing the *BaseAdapter*.

Example: Specific SSL Version

The Requests team has made a specific choice to use whatever SSL version is default in the underlying library (urllib3). Normally this is fine, but from time to time, you might find yourself needing to connect to a service-endpoint that uses a version that isn't compatible with the default.

You can use Transport Adapters for this by taking most of the existing implementation of HTTPAdapter, and adding a parameter *ssl_version* that gets passed-through to *urllib3*. We'll make a Transport Adapter that instructs the library to use SSLv3:

```
import ssl

from requests.adapters import HTTPAdapter
from requests.packages.urllib3.poolmanager import PoolManager


class Ssl3HttpAdapter(HTTPAdapter):
    """Transport adapter" that allows us to use SSLv3."""

    def init_poolmanager(self, connections, maxsize, block=False):
        self.poolmanager = PoolManager(
            num_pools=connections, maxsize=maxsize,
            block=block, ssl_version=ssl.PROTOCOL_SSLv3)
```

Blocking Or Non-Blocking?

With the default Transport Adapter in place, Requests does not provide any kind of non-blocking IO. The *Response.content* property will block until the entire response has been downloaded. If you require more granularity, the streaming features of the library (see *Streaming Requests*) allow you to retrieve smaller quantities of the response at a time. However, these calls will still block.

If you are concerned about the use of blocking IO, there are lots of projects out there that combine Requests with one of Python's asynchronicity frameworks. Two excellent examples are grequests and requests-futures.

Header Ordering

In unusual circumstances you may want to provide headers in an ordered manner. If you pass an `OrderedDict` to the `headers` keyword argument, that will provide the headers with an ordering. *However*, the ordering of the default headers used by Requests will be preferred, which means that if you override default headers in the `headers` keyword argument, they may appear out of order compared to other headers in that keyword argument.

If this is problematic, users should consider setting the default headers on a *Session* object, by setting *Session* to a custom `OrderedDict`. That ordering will always be preferred.

Timeouts

Most requests to external servers should have a timeout attached, in case the server is not responding in a timely manner. By default, requests do not time out unless a timeout value is set explicitly. Without a timeout, your code may hang for minutes or more.

The **connect** timeout is the number of seconds Requests will wait for your client to establish a connection to a remote machine (corresponding to the connect()) call on the socket. It's a good practice to set connect timeouts to slightly larger than a multiple of 3, which is the default TCP packet retransmission window.

Once your client has connected to the server and sent the HTTP request, the **read** timeout is the number of seconds the client will wait for the server to send a response. (Specifically, it's the number of seconds that the client will wait *between* bytes sent from the server. In 99.9% of cases, this is the time before the server sends the first byte).

If you specify a single value for the timeout, like this:

```
r = requests.get('https://github.com', timeout=5)
```

The timeout value will be applied to both the `connect` and the `read` timeouts. Specify a tuple if you would like to set the values separately:

```
r = requests.get('https://github.com', timeout=(3.05, 27))
```

If the remote server is very slow, you can tell Requests to wait forever for a response, by passing None as a timeout value and then retrieving a cup of coffee.

```
r = requests.get('https://github.com', timeout=None)
```

Authentication

This document discusses using various kinds of authentication with Requests.

Many web services require authentication, and there are many different types. Below, we outline various forms of authentication available in Requests, from the simple to the complex.

Basic Authentication

Many web services that require authentication accept HTTP Basic Auth. This is the simplest kind, and Requests supports it straight out of the box.

Making requests with HTTP Basic Auth is very simple:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user', 'pass'))
<Response [200]>
```

In fact, HTTP Basic Auth is so common that Requests provides a handy shorthand for using it:

```
>>> requests.get('https://api.github.com/user', auth=('user', 'pass'))
<Response [200]>
```

Providing the credentials in a tuple like this is exactly the same as the `HTTPBasicAuth` example above.

netrc Authentication

If no authentication method is given with the `auth` argument, Requests will attempt to get the authentication credentials for the URL's hostname from the user's netrc file. The netrc file overrides raw HTTP authentication headers set with *headers*=.

If credentials for the hostname are found, the request is sent with HTTP Basic Auth.

Digest Authentication

Another very popular form of HTTP Authentication is Digest Authentication, and Requests supports this out of the box as well:

```
>>> from requests.auth import HTTPDigestAuth
>>> url = 'http://httpbin.org/digest-auth/auth/user/pass'
>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
<Response [200]>
```

OAuth 1 Authentication

A common form of authentication for several web APIs is OAuth. The `requests-oauthlib` library allows Requests users to easily make OAuth 1 authenticated requests:

```
>>> import requests
>>> from requests_oauthlib import OAuth1

>>> url = 'https://api.twitter.com/1.1/account/verify_credentials.json'
>>> auth = OAuth1('YOUR_APP_KEY', 'YOUR_APP_SECRET',
...               'USER_OAUTH_TOKEN', 'USER_OAUTH_TOKEN_SECRET')

>>> requests.get(url, auth=auth)
<Response [200]>
```

For more information on how to OAuth flow works, please see the official OAuth website. For examples and documentation on requests-oauthlib, please see the requests_oauthlib repository on GitHub

OAuth 2 and OpenID Connect Authentication

The `requests-oauthlib` library also handles OAuth 2, the authentication mechanism underpinning OpenID Connect. See the requests-oauthlib OAuth2 documentation for details of the various OAuth 2 credential management flows:

- Web Application Flow
- Mobile Application Flow
- Legacy Application Flow
- Backend Application Flow

Other Authentication

Requests is designed to allow other forms of authentication to be easily and quickly plugged in. Members of the open-source community frequently write authentication handlers for more complicated or less commonly-used forms of authentication. Some of the best have been brought together under the Requests organization, including:

- Kerberos
- NTLM

If you want to use any of these forms of authentication, go straight to their GitHub page and follow the instructions.

New Forms of Authentication

If you can't find a good implementation of the form of authentication you want, you can implement it yourself. Requests makes it easy to add your own forms of authentication.

To do so, subclass *AuthBase* and implement the `__call__()` method:

```
>>> import requests
>>> class MyAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Implement my authentication
...         return r
...
```

```
>>> url = 'http://httpbin.org/get'
>>> requests.get(url, auth=MyAuth())
<Response [200]>
```

When an authentication handler is attached to a request, it is called during request setup. The `__call__` method must therefore do whatever is required to make the authentication work. Some forms of authentication will additionally add hooks to provide further functionality.

Further examples can be found under the Requests organization and in the `auth.py` file.

CHAPTER 4

The Community Guide

This part of the documentation, which is mostly prose, details the Requests ecosystem and community.

Frequently Asked Questions

This part of the documentation answers common questions about Requests.

Encoded Data?

Requests automatically decompresses gzip-encoded responses, and does its best to decode response content to unicode when possible.

You can get direct access to the raw response (and even the socket), if needed as well.

Custom User-Agents?

Requests allows you to easily override User-Agent strings, along with any other HTTP Header.

Why not Httplib2?

Chris Adams gave an excellent summary on Hacker News:

> httplib2 is part of why you should use requests: it's far more respectable as a client but not as well documented and it still takes way too much code for basic operations. I appreciate what httplib2 is trying to do, that there's a ton of hard low-level annoyances in building a modern HTTP client, but really, just use requests instead. Kenneth Reitz is very motivated and he gets the degree to which simple things should be simple whereas httplib2 feels more like an academic exercise than something people should use to build production systems[1].

Disclosure: I'm listed in the requests AUTHORS file but can claim credit for, oh, about 0.0001% of the awesomeness.

1. http://code.google.com/p/httplib2/issues/detail?id=96 is a good example: an annoying bug which affect many people, there was a fix available for months, which worked great when I applied it in a fork and pounded a couple TB of data through it, but it took over a year to make it into trunk and even longer to make it onto PyPI where any other project which required " httplib2" would get the working version.

Python 3 Support?

Yes! Here's a list of Python platforms that are officially supported:

- Python 2.6
- Python 2.7
- Python 3.3
- Python 3.4
- Python 3.5
- Python 3.6
- PyPy

What are "hostname doesn't match" errors?

These errors occur when *SSL certificate verification* fails to match the certificate the server responds with to the hostname Requests thinks it's contacting. If you're certain the server's SSL setup is correct (for example, because you can visit the site with your browser) and you're using Python 2.6 or 2.7, a possible explanation is that you need Server-Name-Indication.

Server-Name-Indication, or SNI, is an official extension to SSL where the client tells the server what hostname it is contacting. This is important when servers are using Virtual Hosting. When such servers are hosting more than one SSL site they need to be able to return the appropriate certificate based on the hostname the client is connecting to.

Python3 and Python 2.7.9+ include native support for SNI in their SSL modules. For information on using SNI with Requests on Python < 2.7.9 refer to this Stack Overflow answer.

Recommended Packages and Extensions

Requests has a great variety of powerful and useful third-party extensions. This page provides an overview of some of the best of them.

Certifi CA Bundle

Certifi is a carefully curated collection of Root Certificates for validating the trustworthiness of SSL certificates while verifying the identity of TLS hosts. It has been extracted from the Requests project.

CacheControl

CacheControl is an extension that adds a full HTTP cache to Requests. This makes your web requests substantially more efficient, and should be used whenever you're making a lot of web requests.

Requests-Toolbelt

Requests-Toolbelt is a collection of utilities that some users of Requests may desire, but do not belong in Requests proper. This library is actively maintained by members of the Requests core team, and reflects the functionality most requested by users within the community.

Requests-OAuthlib

requests-oauthlib makes it possible to do the OAuth dance from Requests automatically. This is useful for the large number of websites that use OAuth to provide authentication. It also provides a lot of tweaks that handle ways that specific OAuth providers differ from the standard specifications.

Betamax

Betamax records your HTTP interactions so the NSA does not have to. A VCR imitation designed only for Python-Requests.

Integrations

Python for iOS

Requests is built into the wonderful Python for iOS runtime!

To give it a try, simply:

```
import requests
```

Articles & Talks

- Python for the Web teaches how to use Python to interact with the web, using Requests.
- Daniel Greenfeld's Review of Requests
- My 'Python for Humans' talk (audio)
- Issac Kelly's 'Consuming Web APIs' talk
- Blog post about Requests via Yum
- Russian blog post introducing Requests
- Sending JSON in Requests

Support

If you have questions or issues about Requests, there are several options:

StackOverflow

If your question does not contain sensitive (possibly proprietary) information or can be properly anonymized, please ask a question on StackOverflow and use the tag `python-requests`.

Send a Tweet

If your question is less than 140 characters, feel free to send a tweet to @kennethreitz, @sigmavirus24, or @lukasaoz.

File an Issue

If you notice some unexpected behaviour in Requests, or want to see support for a new feature, file an issue on GitHub.

E-mail

I'm more than happy to answer any personal or in-depth questions about Requests. Feel free to email requests@kennethreitz.com.

IRC

The official Freenode channel for Requests is #python-requests

The core developers of requests are on IRC throughout the day. You can find them in `#python-requests` as:

- kennethreitz
- lukasa
- sigmavirus24

Vulnerability Disclosure

If you think you have found a potential security vulnerability in requests, please email sigmavirus24 and Lukasa directly. **Do not file a public issue.**

Our PGP Key fingerprints are:

- 0161 BB7E B208 B5E0 4FDC 9F81 D9DA 0A04 9113 F853 (@sigmavirus24)
- 90DC AE40 FEA7 4B14 9B70 662D F25F 2144 EEC1 373D (@lukasa)

If English is not your first language, please try to describe the problem and its impact to the best of your ability. For greater detail, please use your native language and we will try our best to translate it using online services.

Please also include the code you used to find the problem and the shortest amount of code necessary to reproduce it.

Please do not disclose this to anyone else. We will retrieve a CVE identifier if necessary and give you full credit under whatever name or alias you provide. We will only request an identifier when we have a fix and can publish it in a release.

We will respect your privacy and will only publicize your involvement if you grant us permission.

Process

This following information discusses the process the requests project follows in response to vulnerability disclosures. If you are disclosing a vulnerability, this section of the documentation lets you know how we will respond to your disclosure.

Timeline

When you report an issue, one of the project members will respond to you within two days *at the outside*. In most cases responses will be faster, usually within 12 hours. This initial response will at the very least confirm receipt of the report.

If we were able to rapidly reproduce the issue, the initial response will also contain confirmation of the issue. If we are not, we will often ask for more information about the reproduction scenario.

Our goal is to have a fix for any vulnerability released within two weeks of the initial disclosure. This may potentially involve shipping an interim release that simply disables function while a more mature fix can be prepared, but will in the vast majority of cases mean shipping a complete release as soon as possible.

Throughout the fix process we will keep you up to speed with how the fix is progressing. Once the fix is prepared, we will notify you that we believe we have a fix. Often we will ask you to confirm the fix resolves the problem in your environment, especially if we are not confident of our reproduction scenario.

At this point, we will prepare for the release. We will obtain a CVE number if one is required, providing you with full credit for the discovery. We will also decide on a planned release date, and let you know when it is. This release date will *always* be on a weekday.

At this point we will reach out to our major downstream packagers to notify them of an impending security-related patch so they can make arrangements. In addition, these packagers will be provided with the intended patch ahead of time, to ensure that they are able to promptly release their downstream packages. Currently the list of people we actively contact *ahead of a public release* is:

- Jeremy Cline, Red Hat (@jeremycline)
- Daniele Tricoli, Debian (@eriol)

We will notify these individuals at least a week ahead of our planned release date to ensure that they have sufficient time to prepare. If you believe you should be on this list, please let one of the maintainers know at one of the email addresses at the top of this article.

On release day, we will push the patch to our public repository, along with an updated changelog that describes the issue and credits you. We will then issue a PyPI release containing the patch.

At this point, we will publicise the release. This will involve mails to mailing lists, Tweets, and all other communication mechanisms available to the core team.

We will also explicitly mention which commits contain the fix to make it easier for other distributors and users to easily patch their own versions of requests if upgrading is not an option.

Previous CVEs

- Fixed in 2.6.0
  - CVE 2015-2296, reported by Matthew Daley of BugFuzz.
- Fixed in 2.3.0
  - CVE 2014-1829
  - CVE 2014-1830

Community Updates

If you'd like to stay up to date on the community and development of Requests, there are several options:

GitHub

The best way to track the development of Requests is through the GitHub repo.

Twitter

The author, Kenneth Reitz, often tweets about new features and releases of Requests.

Follow @kennethreitz for updates.

Release and Version History

Release History

dev

Improvements

Bugfixes

2.18.1 (2017-06-14)

Bugfixes

- Fix an error in the packaging whereby the *.whl contained incorrect data that regressed the fix in v2.17.3.

2.18.0 (2017-06-14)

Improvements

- `Response` is now a context manager, so can be used directly in a `with` statement without first having to be wrapped by `contextlib.closing()`.

Bugfixes

- Resolve installation failure if multiprocessing is not available
- Resolve tests crash if multiprocessing is not able to determine the number of CPU cores
- Resolve error swallowing in utils set_environ generator

2.17.3 (2017-05-29)

Improvements

- Improved `packages` namespace identity support, for monkeypatching libraries.

2.17.2 (2017-05-29)

Improvements

- Improved `packages` namespace identity support, for monkeypatching libraries.

2.17.1 (2017-05-29)

Improvements

- Improved `packages` namespace identity support, for monkeypatching libraries.

2.17.0 (2017-05-29)

Improvements

- Removal of the 301 redirect cache. This improves thread-safety.

2.16.5 (2017-05-28)

- Improvements to `$ python -m requests.help`.

2.16.4 (2017-05-27)

- Introduction of the `$ python -m requests.help` command, for debugging with maintainers!

2.16.3 (2017-05-27)

- Further restored the `requests.packages` namespace for compatibility reasons.

2.16.2 (2017-05-27)

- Further restored the `requests.packages` namespace for compatibility reasons.

No code modification (noted below) should be neccessary any longer.

2.16.1 (2017-05-27)

- Restored the `requests.packages` namespace for compatibility reasons.
- Bugfix for `urllib3` version parsing.

Note: code that was written to import against the `requests.packages` namespace previously will have to import code that rests at this module-level now.

For example:

```
from requests.packages.urllib3.poolmanager import PoolManager
```

Will need to be re-written to be:

```
from requests.packages import urllib3
urllib3.poolmanager.PoolManager
```

Or, even better:

```
from urllib3.poolmanager import PoolManager
```

2.16.0 (2017-05-26)

- Unvendor ALL the things!

2.15.1 (2017-05-26)

- Everyone makes mistakes.

2.15.0 (2017-05-26)

Improvements

- Introduction of the `Response.next` property, for getting the next `PreparedResponse` from a redirect chain (when `allow_redirects=False`).
- Internal refactoring of `__version__` module.

Bugfixes

- Restored once-optional parameter for `requests.utils.get_environ_proxies()`.

2.14.2 (2017-05-10)

Bugfixes

- Changed a less-than to an equal-to and an or in the dependency markers to widen compatibility with older setuptools releases.

2.14.1 (2017-05-09)

Bugfixes

- Changed the dependency markers to widen compatibility with older pip releases.

2.14.0 (2017-05-09)

Improvements

- It is now possible to pass `no_proxy` as a key to the `proxies` dictionary to provide handling similar to the `NO_PROXY` environment variable.
- When users provide invalid paths to certificate bundle files or directories Requests now raises `IOError`, rather than failing at the time of the HTTPS request with a fairly inscrutable certificate validation error.
- The behavior of `SessionRedirectMixin` was slightly altered. `resolve_redirects` will now detect a redirect by calling `get_redirect_target(response)` instead of directly querying `Response.is_redirect` and `Response.headers['location']`. Advanced users will be able to process malformed redirects more easily.
- Changed the internal calculation of elapsed request time to have higher resolution on Windows.

- Added `win_inet_pton` as conditional dependency for the `[socks]` extra on Windows with Python 2.7.
- Changed the proxy bypass implementation on Windows: the proxy bypass check doesn't use forward and reverse DNS requests anymore
- URLs with schemes that begin with `http` but are not `http` or `https` no longer have their host parts forced to lowercase.

Bugfixes

- Much improved handling of non-ASCII `Location` header values in redirects. Fewer `UnicodeDecodeErrors` are encountered on Python 2, and Python 3 now correctly understands that Latin-1 is unlikely to be the correct encoding.
- If an attempt to `seek` file to find out its length fails, we now appropriately handle that by aborting our content-length calculations.
- Restricted `HTTPDigestAuth` to only respond to auth challenges made on 4XX responses, rather than to all auth challenges.
- Fixed some code that was firing `DeprecationWarning` on Python 3.6.
- The dismayed person emoticon (`/o\\`) no longer has a big head. I'm sure this is what you were all worrying about most.

Miscellaneous

- Updated bundled urllib3 to v1.21.1.
- Updated bundled chardet to v3.0.2.
- Updated bundled idna to v2.5.
- Updated bundled certifi to 2017.4.17.

2.13.0 (2017-01-24)

Features

- Only load the `idna` library when we've determined we need it. This will save some memory for users.

Miscellaneous

- Updated bundled urllib3 to 1.20.
- Updated bundled idna to 2.2.

2.12.5 (2017-01-18)

Bugfixes

- Fixed an issue with JSON encoding detection, specifically detecting big-endian UTF-32 with BOM.

2.12.4 (2016-12-14)

Bugfixes

- Fixed regression from 2.12.2 where non-string types were rejected in the basic auth parameters. While support for this behaviour has been readded, the behaviour is deprecated and will be removed in the future.

2.12.3 (2016-12-01)

Bugfixes

- Fixed regression from v2.12.1 for URLs with schemes that begin with "http". These URLs have historically been processed as though they were HTTP-schemed URLs, and so have had parameters added. This was removed in v2.12.2 in an overzealous attempt to resolve problems with IDNA-encoding those URLs. This change was reverted: the other fixes for IDNA-encoding have been judged to be sufficient to return to the behaviour Requests had before v2.12.0.

2.12.2 (2016-11-30)

Bugfixes

- Fixed several issues with IDNA-encoding URLs that are technically invalid but which are widely accepted. Requests will now attempt to IDNA-encode a URL if it can but, if it fails, and the host contains only ASCII characters, it will be passed through optimistically. This will allow users to opt-in to using IDNA2003 themselves if they want to, and will also allow technically invalid but still common hostnames.
- Fixed an issue where URLs with leading whitespace would raise `InvalidSchema` errors.
- Fixed an issue where some URLs without the HTTP or HTTPS schemes would still have HTTP URL preparation applied to them.
- Fixed an issue where Unicode strings could not be used in basic auth.
- Fixed an issue encountered by some Requests plugins where constructing a Response object would cause `Response.content` to raise an `AttributeError`.

2.12.1 (2016-11-16)

Bugfixes

- Updated setuptools 'security' extra for the new PyOpenSSL backend in urllib3.

Miscellaneous

- Updated bundled urllib3 to 1.19.1.

2.12.0 (2016-11-15)

Improvements

- Updated support for internationalized domain names from IDNA2003 to IDNA2008. This updated support is required for several forms of IDNs and is mandatory for .de domains.
- Much improved heuristics for guessing content lengths: Requests will no longer read an entire `StringIO` into memory.
- Much improved logic for recalculating `Content-Length` headers for `PreparedRequest` objects.
- Improved tolerance for file-like objects that have no `tell` method but do have a `seek` method.
- Anything that is a subclass of `Mapping` is now treated like a dictionary by the `data=` keyword argument.
- Requests now tolerates empty passwords in proxy credentials, rather than stripping the credentials.
- If a request is made with a file-like object as the body and that request is redirected with a 307 or 308 status code, Requests will now attempt to rewind the body object so it can be replayed.

Bugfixes

- When calling `response.close`, the call to `close` will be propagated through to non-urllib3 backends.
- Fixed issue where the `ALL_PROXY` environment variable would be preferred over scheme-specific variables like `HTTP_PROXY`.
- Fixed issue where non-UTF8 reason phrases got severely mangled by falling back to decoding using ISO 8859-1 instead.
- Fixed a bug where Requests would not correctly correlate cookies set when using custom Host headers if those Host headers did not use the native string type for the platform.

Miscellaneous

- Updated bundled urllib3 to 1.19.
- Updated bundled certifi certs to 2016.09.26.

2.11.1 (2016-08-17)

Bugfixes

- Fixed a bug when using `iter_content` with `decode_unicode=True` for streamed bodies would raise `AttributeError`. This bug was introduced in 2.11.
- Strip Content-Type and Transfer-Encoding headers from the header block when following a redirect that transforms the verb from POST/PUT to GET.

2.11.0 (2016-08-08)

Improvements

- Added support for the `ALL_PROXY` environment variable.
- Reject header values that contain leading whitespace or newline characters to reduce risk of header smuggling.

Bugfixes

- Fixed occasional `TypeError` when attempting to decode a JSON response that occurred in an error case. Now correctly returns a `ValueError`.
- Requests would incorrectly ignore a non-CIDR IP address in the `NO_PROXY` environment variables: Requests now treats it as a specific IP.
- Fixed a bug when sending JSON data that could cause us to encounter obscure OpenSSL errors in certain network conditions (yes, really).
- Added type checks to ensure that `iter_content` only accepts integers and `None` for chunk sizes.
- Fixed issue where responses whose body had not been fully consumed would have the underlying connection closed but not returned to the connection pool, which could cause Requests to hang in situations where the `HTTPAdapter` had been configured to use a blocking connection pool.

Miscellaneous

- Updated bundled urllib3 to 1.16.
- Some previous releases accidentally accepted non-strings as acceptable header values. This release does not.

2.10.0 (2016-04-29)

New Features

- SOCKS Proxy Support! (requires PySocks; `$ pip install requests[socks]`)

Miscellaneous

- Updated bundled urllib3 to 1.15.1.

2.9.2 (2016-04-29)

Improvements

- Change built-in CaseInsensitiveDict (used for headers) to use OrderedDict as its underlying datastore.

Bugfixes

- Don't use redirect_cache if allow_redirects=False
- When passed objects that throw exceptions from `tell()`, send them via chunked transfer encoding instead of failing.
- Raise a ProxyError for proxy related connection issues.

2.9.1 (2015-12-21)

Bugfixes

- Resolve regression introduced in 2.9.0 that made it impossible to send binary strings as bodies in Python 3.
- Fixed errors when calculating cookie expiration dates in certain locales.

Miscellaneous

- Updated bundled urllib3 to 1.13.1.

2.9.0 (2015-12-15)

Minor Improvements (Backwards compatible)

- The `verify` keyword argument now supports being passed a path to a directory of CA certificates, not just a single-file bundle.
- Warnings are now emitted when sending files opened in text mode.
- Added the 511 Network Authentication Required status code to the status code registry.

Bugfixes

- For file-like objects that are not seeked to the very beginning, we now send the content length for the number of bytes we will actually read, rather than the total size of the file, allowing partial file uploads.
- When uploading file-like objects, if they are empty or have no obvious content length we set `Transfer-Encoding: chunked` rather than `Content-Length: 0`.
- We correctly receive the response in buffered mode when uploading chunked bodies.
- We now handle being passed a query string as a bytestring on Python 3, by decoding it as UTF-8.
- Sessions are now closed in all cases (exceptional and not) when using the functional API rather than leaking and waiting for the garbage collector to clean them up.

- Correctly handle digest auth headers with a malformed `qop` directive that contains no token, by treating it the same as if no `qop` directive was provided at all.
- Minor performance improvements when removing specific cookies by name.

Miscellaneous

- Updated urllib3 to 1.13.

2.8.1 (2015-10-13)

Bugfixes

- Update certificate bundle to match `certifi` 2015.9.6.2's weak certificate bundle.
- Fix a bug in 2.8.0 where requests would raise `ConnectTimeout` instead of `ConnectionError`
- When using the PreparedRequest flow, requests will now correctly respect the `json` parameter. Broken in 2.8.0.
- When using the PreparedRequest flow, requests will now correctly handle a Unicode-string method name on Python 2. Broken in 2.8.0.

2.8.0 (2015-10-05)

Minor Improvements (Backwards Compatible)

- Requests now supports per-host proxies. This allows the `proxies` dictionary to have entries of the form `{'<scheme>://<hostname>': '<proxy>'}`. Host-specific proxies will be used in preference to the previously-supported scheme-specific ones, but the previous syntax will continue to work.
- `Response.raise_for_status` now prints the URL that failed as part of the exception message.
- `requests.utils.get_netrc_auth` now takes an `raise_errors` kwarg, defaulting to `False`. When `True`, errors parsing `.netrc` files cause exceptions to be thrown.
- Change to bundled projects import logic to make it easier to unbundle requests downstream.
- Changed the default User-Agent string to avoid leaking data on Linux: now contains only the requests version.

Bugfixes

- The `json` parameter to `post()` and friends will now only be used if neither `data` nor `files` are present, consistent with the documentation.
- We now ignore empty fields in the `NO_PROXY` environment variable.
- Fixed problem where `httplib.BadStatusLine` would get raised if combining `stream=True` with `contextlib.closing`.
- Prevented bugs where we would attempt to return the same connection back to the connection pool twice when sending a Chunked body.
- Miscellaneous minor internal changes.
- Digest Auth support is now thread safe.

Updates

- Updated urllib3 to 1.12.

2.7.0 (2015-05-03)

This is the first release that follows our new release process. For more, see our documentation.

Bugfixes

- Updated urllib3 to 1.10.4, resolving several bugs involving chunked transfer encoding and response framing.

2.6.2 (2015-04-23)

Bugfixes

- Fix regression where compressed data that was sent as chunked data was not properly decompressed. (#2561)

2.6.1 (2015-04-22)

Bugfixes

- Remove VendorAlias import machinery introduced in v2.5.2.
- Simplify the PreparedRequest.prepare API: We no longer require the user to pass an empty list to the hooks keyword argument. (c.f. #2552)
- Resolve redirects now receives and forwards all of the original arguments to the adapter. (#2503)
- Handle UnicodeDecodeErrors when trying to deal with a unicode URL that cannot be encoded in ASCII. (#2540)
- Populate the parsed path of the URI field when performing Digest Authentication. (#2426)
- Copy a PreparedRequest's CookieJar more reliably when it is not an instance of RequestsCookieJar. (#2527)

2.6.0 (2015-03-14)

Bugfixes

- CVE-2015-2296: Fix handling of cookies on redirect. Previously a cookie without a host value set would use the hostname for the redirected URL exposing requests users to session fixation attacks and potentially cookie stealing. This was disclosed privately by Matthew Daley of BugFuzz. This affects all versions of requests from v2.1.0 to v2.5.3 (inclusive on both ends).
- Fix error when requests is an `install_requires` dependency and `python setup.py test` is run. (#2462)
- Fix error when urllib3 is unbundled and requests continues to use the vendored import location.
- Include fixes to `urllib3`'s header handling.
- Requests' handling of unvendored dependencies is now more restrictive.

Features and Improvements

- Support bytearrays when passed as parameters in the `files` argument. (#2468)
- Avoid data duplication when creating a request with `str`, `bytes`, or `bytearray` input to the `files` argument.

2.5.3 (2015-02-24)

Bugfixes

- Revert changes to our vendored certificate bundle. For more context see (#2455, #2456, and http://bugs.python.org/issue23476)

2.5.2 (2015-02-23)

Features and Improvements

- Add sha256 fingerprint support. (shazow/urllib3#540)
- Improve the performance of headers. (shazow/urllib3#544)

Bugfixes

- Copy pip's import machinery. When downstream redistributors remove requests.packages.urllib3 the import machinery will continue to let those same symbols work. Example usage in requests' documentation and 3rd-party libraries relying on the vendored copies of urllib3 will work without having to fallback to the system urllib3.
- Attempt to quote parts of the URL on redirect if unquoting and then quoting fails. (#2356)
- Fix filename type check for multipart form-data uploads. (#2411)
- Properly handle the case where a server issuing digest authentication challenges provides both auth and auth-int qop-values. (#2408)
- Fix a socket leak. (shazow/urllib3#549)
- Fix multiple `Set-Cookie` headers properly. (shazow/urllib3#534)
- Disable the built-in hostname verification. (shazow/urllib3#526)
- Fix the behaviour of decoding an exhausted stream. (shazow/urllib3#535)

Security

- Pulled in an updated `cacert.pem`.
- Drop RC4 from the default cipher list. (shazow/urllib3#551)

2.5.1 (2014-12-23)

Behavioural Changes

- Only catch HTTPErrors in raise_for_status (#2382)

Bugfixes

- Handle LocationParseError from urllib3 (#2344)
- Handle file-like object filenames that are not strings (#2379)
- Unbreak HTTPDigestAuth handler. Allow new nonces to be negotiated (#2389)

2.5.0 (2014-12-01)

Improvements

- Allow usage of urllib3's Retry object with HTTPAdapters (#2216)
- The `iter_lines` method on a response now accepts a delimiter with which to split the content (#2295)

Behavioural Changes

- Add deprecation warnings to functions in requests.utils that will be removed in 3.0 (#2309)
- Sessions used by the functional API are always closed (#2326)
- Restrict requests to HTTP/1.1 and HTTP/1.0 (stop accepting HTTP/0.9) (#2323)

Bugfixes

- Only parse the URL once (#2353)
- Allow Content-Length header to always be overridden (#2332)
- Properly handle files in HTTPDigestAuth (#2333)
- Cap redirect_cache size to prevent memory abuse (#2299)
- Fix HTTPDigestAuth handling of redirects after authenticating successfully (#2253)
- Fix crash with custom method parameter to Session.request (#2317)
- Fix how Link headers are parsed using the regular expression library (#2271)

Documentation

- Add more references for interlinking (#2348)
- Update CSS for theme (#2290)
- Update width of buttons and sidebar (#2289)
- Replace references of Gittip with Gratipay (#2282)
- Add link to changelog in sidebar (#2273)

2.4.3 (2014-10-06)

Bugfixes

- Unicode URL improvements for Python 2.
- Re-order JSON param for backwards compat.
- Automatically defrag authentication schemes from host/pass URIs. (#2249)

2.4.2 (2014-10-05)

Improvements

- FINALLY! Add json parameter for uploads! (#2258)
- Support for bytestring URLs on Python 3.x (#2238)

Bugfixes

- Avoid getting stuck in a loop (#2244)

- Multiple calls to iter* fail with unhelpful error. (#2240, #2241)

Documentation

- Correct redirection introduction (#2245)
- Added example of how to send multiple files in one request. (#2227)
- Clarify how to pass a custom set of CAs (#2248)

2.4.1 (2014-09-09)

- Now has a "security" package extras set, `$ pip install requests[security]`
- Requests will now use Certifi if it is available.
- Capture and re-raise urllib3 ProtocolError
- Bugfix for responses that attempt to redirect to themselves forever (wtf?).

2.4.0 (2014-08-29)

Behavioral Changes

- `Connection: keep-alive` header is now sent automatically.

Improvements

- Support for connect timeouts! Timeout now accepts a tuple (connect, read) which is used to set individual connect and read timeouts.
- Allow copying of PreparedRequests without headers/cookies.
- Updated bundled urllib3 version.
- Refactored settings loading from environment – new *Session.merge_environment_settings*.
- Handle socket errors in iter_content.

2.3.0 (2014-05-16)

API Changes

- New `Response` property `is_redirect`, which is true when the library could have processed this response as a redirection (whether or not it actually did).
- The `timeout` parameter now affects requests with both `stream=True` and `stream=False` equally.
- The change in v2.0.0 to mandate explicit proxy schemes has been reverted. Proxy schemes now default to `http://`.
- The `CaseInsensitiveDict` used for HTTP headers now behaves like a normal dictionary when references as string or viewed in the interpreter.

Bugfixes

- No longer expose Authorization or Proxy-Authorization headers on redirect. Fix CVE-2014-1829 and CVE-2014-1830 respectively.
- Authorization is re-evaluated each redirect.
- On redirect, pass url as native strings.

- Fall-back to autodetected encoding for JSON when Unicode detection fails.
- Headers set to `None` on the `Session` are now correctly not sent.
- Correctly honor `decode_unicode` even if it wasn't used earlier in the same response.
- Stop advertising `compress` as a supported Content-Encoding.
- The `Response.history` parameter is now always a list.
- Many, many `urllib3` bugfixes.

2.2.1 (2014-01-23)

Bugfixes

- Fixes incorrect parsing of proxy credentials that contain a literal or encoded '#' character.
- Assorted urllib3 fixes.

2.2.0 (2014-01-09)

API Changes

- New exception: `ContentDecodingError`. Raised instead of `urllib3` `DecodeError` exceptions.

Bugfixes

- Avoid many many exceptions from the buggy implementation of `proxy_bypass` on OS X in Python 2.6.
- Avoid crashing when attempting to get authentication credentials from ~/.netrc when running as a user without a home directory.
- Use the correct pool size for pools of connections to proxies.
- Fix iteration of `CookieJar` objects.
- Ensure that cookies are persisted over redirect.
- Switch back to using chardet, since it has merged with charade.

2.1.0 (2013-12-05)

- Updated CA Bundle, of course.
- Cookies set on individual Requests through a `Session` (e.g. via `Session.get()`) are no longer persisted to the `Session`.
- Clean up connections when we hit problems during chunked upload, rather than leaking them.
- Return connections to the pool when a chunked upload is successful, rather than leaking it.
- Match the HTTPbis recommendation for HTTP 301 redirects.
- Prevent hanging when using streaming uploads and Digest Auth when a 401 is received.
- Values of headers set by Requests are now always the native string type.
- Fix previously broken SNI support.
- Fix accessing HTTP proxies using proxy authentication.
- Unencode HTTP Basic usernames and passwords extracted from URLs.

- Support for IP address ranges for no_proxy environment variable
- Parse headers correctly when users override the default `Host:` header.
- Avoid munging the URL in case of case-sensitive servers.
- Looser URL handling for non-HTTP/HTTPS urls.
- Accept unicode methods in Python 2.6 and 2.7.
- More resilient cookie handling.
- Make `Response` objects pickleable.
- Actually added MD5-sess to Digest Auth instead of pretending to like last time.
- Updated internal urllib3.
- Fixed @Lukasa's lack of taste.

2.0.1 (2013-10-24)

- Updated included CA Bundle with new mistrusts and automated process for the future
- Added MD5-sess to Digest Auth
- Accept per-file headers in multipart file POST messages.
- Fixed: Don't send the full URL on CONNECT messages.
- Fixed: Correctly lowercase a redirect scheme.
- Fixed: Cookies not persisted when set via functional API.
- Fixed: Translate urllib3 ProxyError into a requests ProxyError derived from ConnectionError.
- Updated internal urllib3 and chardet.

2.0.0 (2013-09-24)

API Changes:

- Keys in the Headers dictionary are now native strings on all Python versions, i.e. bytestrings on Python 2, unicode on Python 3.
- Proxy URLs now *must* have an explicit scheme. A `MissingSchema` exception will be raised if they don't.
- Timeouts now apply to read time if `Stream=False`.
- `RequestException` is now a subclass of `IOError`, not `RuntimeError`.
- Added new method to `PreparedRequest` objects: `PreparedRequest.copy()`.
- Added new method to `Session` objects: `Session.update_request()`. This method updates a `Request` object with the data (e.g. cookies) stored on the `Session`.
- Added new method to `Session` objects: `Session.prepare_request()`. This method updates and prepares a `Request` object, and returns the corresponding `PreparedRequest` object.
- Added new method to `HTTPAdapter` objects: `HTTPAdapter.proxy_headers()`. This should not be called directly, but improves the subclass interface.
- `httplib.IncompleteRead` exceptions caused by incorrect chunked encoding will now raise a Requests `ChunkedEncodingError` instead.
- Invalid percent-escape sequences now cause a Requests `InvalidURL` exception to be raised.

- HTTP 208 no longer uses reason phrase "im_used". Correctly uses "already_reported".
- HTTP 226 reason added ("im_used").

Bugfixes:

- Vastly improved proxy support, including the CONNECT verb. Special thanks to the many contributors who worked towards this improvement.
- Cookies are now properly managed when 401 authentication responses are received.
- Chunked encoding fixes.
- Support for mixed case schemes.
- Better handling of streaming downloads.
- Retrieve environment proxies from more locations.
- Minor cookies fixes.
- Improved redirect behaviour.
- Improved streaming behaviour, particularly for compressed data.
- Miscellaneous small Python 3 text encoding bugs.
- .netrc no longer overrides explicit auth.
- Cookies set by hooks are now correctly persisted on Sessions.
- Fix problem with cookies that specify port numbers in their host field.
- BytesIO can be used to perform streaming uploads.
- More generous parsing of the no_proxy environment variable.
- Non-string objects can be passed in data values alongside files.

1.2.3 (2013-05-25)

- Simple packaging fix

1.2.2 (2013-05-23)

- Simple packaging fix

1.2.1 (2013-05-20)

- 301 and 302 redirects now change the verb to GET for all verbs, not just POST, improving browser compatibility.
- Python 3.3.2 compatibility
- Always percent-encode location headers
- Fix connection adapter matching to be most-specific first
- new argument to the default connection adapter for passing a block argument
- prevent a KeyError when there's no link headers

1.2.0 (2013-03-31)

- Fixed cookies on sessions and on requests
- Significantly change how hooks are dispatched - hooks now receive all the arguments specified by the user when making a request so hooks can make a secondary request with the same parameters. This is especially necessary for authentication handler authors
- certifi support was removed
- Fixed bug where using OAuth 1 with body `signature_type` sent no data
- Major proxy work thanks to @Lukasa including parsing of proxy authentication from the proxy url
- Fix DigestAuth handling too many 401s
- Update vendored urllib3 to include SSL bug fixes
- Allow keyword arguments to be passed to `json.loads()` via the `Response.json()` method
- Don't send `Content-Length` header by default on `GET` or `HEAD` requests
- Add `elapsed` attribute to `Response` objects to time how long a request took.
- Fix `RequestsCookieJar`
- Sessions and Adapters are now picklable, i.e., can be used with the multiprocessing library
- Update charade to version 1.0.3

The change in how hooks are dispatched will likely cause a great deal of issues.

1.1.0 (2013-01-10)

- CHUNKED REQUESTS
- Support for iterable response bodies
- Assume servers persist redirect params
- Allow explicit content types to be specified for file data
- Make merge_kwargs case-insensitive when looking up keys

1.0.3 (2012-12-18)

- Fix file upload encoding bug
- Fix cookie behavior

1.0.2 (2012-12-17)

- Proxy fix for HTTPAdapter.

1.0.1 (2012-12-17)

- Cert verification exception bug.
- Proxy fix for HTTPAdapter.

1.0.0 (2012-12-17)

- Massive Refactor and Simplification
- Switch to Apache 2.0 license
- Swappable Connection Adapters
- Mountable Connection Adapters
- Mutable ProcessedRequest chain
- /s/prefetch/stream
- Removal of all configuration
- Standard library logging
- Make Response.json() callable, not property.
- Usage of new charade project, which provides python 2 and 3 simultaneous chardet.
- Removal of all hooks except 'response'
- Removal of all authentication helpers (OAuth, Kerberos)

This is not a backwards compatible change.

0.14.2 (2012-10-27)

- Improved mime-compatible JSON handling
- Proxy fixes
- Path hack fixes
- Case-Insensitive Content-Encoding headers
- Support for CJK parameters in form posts

0.14.1 (2012-10-01)

- Python 3.3 Compatibility
- Simply default accept-encoding
- Bugfixes

0.14.0 (2012-09-02)

- No more iter_content errors if already downloaded.

0.13.9 (2012-08-25)

- Fix for OAuth + POSTs
- Remove exception eating from dispatch_hook
- General bugfixes

0.13.8 (2012-08-21)

- Incredible Link header support :)

0.13.7 (2012-08-19)

- Support for (key, value) lists everywhere.
- Digest Authentication improvements.
- Ensure proxy exclusions work properly.
- Clearer UnicodeError exceptions.
- Automatic casting of URLs to strings (fURL and such)
- Bugfixes.

0.13.6 (2012-08-06)

- Long awaited fix for hanging connections!

0.13.5 (2012-07-27)

- Packaging fix

0.13.4 (2012-07-27)

- GSSAPI/Kerberos authentication!
- App Engine 2.7 Fixes!
- Fix leaking connections (from urllib3 update)
- OAuthlib path hack fix
- OAuthlib URL parameters fix.

0.13.3 (2012-07-12)

- Use simplejson if available.
- Do not hide SSLErrors behind Timeouts.
- Fixed param handling with urls containing fragments.
- Significantly improved information in User Agent.
- client certificates are ignored when verify=False

0.13.2 (2012-06-28)

- Zero dependencies (once again)!
- New: Response.reason
- Sign querystring parameters in OAuth 1.0
- Client certificates no longer ignored when verify=False
- Add openSUSE certificate support

0.13.1 (2012-06-07)

- Allow passing a file or file-like object as data.
- Allow hooks to return responses that indicate errors.
- Fix Response.text and Response.json for body-less responses.

0.13.0 (2012-05-29)

- Removal of Requests.async in favor of grequests
- Allow disabling of cookie persistence.
- New implementation of safe_mode
- cookies.get now supports default argument
- Session cookies not saved when Session.request is called with return_response=False
- Env: no_proxy support.
- RequestsCookieJar improvements.
- Various bug fixes.

0.12.1 (2012-05-08)

- New `Response.json` property.
- Ability to add string file uploads.
- Fix out-of-range issue with iter_lines.
- Fix iter_content default size.
- Fix POST redirects containing files.

0.12.0 (2012-05-02)

- EXPERIMENTAL OAUTH SUPPORT!
- Proper CookieJar-backed cookies interface with awesome dict-like interface.
- Speed fix for non-iterated content chunks.
- Move `pre_request` to a more usable place.
- New `pre_send` hook.

- Lazily encode data, params, files.
- Load system Certificate Bundle if `certify` isn't available.
- Cleanups, fixes.

0.11.2 (2012-04-22)

- Attempt to use the OS's certificate bundle if `certifi` isn't available.
- Infinite digest auth redirect fix.
- Multi-part file upload improvements.
- Fix decoding of invalid %encodings in URLs.
- If there is no content in a response don't throw an error the second time that content is attempted to be read.
- Upload data on redirects.

0.11.1 (2012-03-30)

- POST redirects now break RFC to do what browsers do: Follow up with a GET.
- New `strict_mode` configuration to disable new redirect behavior.

0.11.0 (2012-03-14)

- Private SSL Certificate support
- Remove select.poll from Gevent monkeypatching
- Remove redundant generator for chunked transfer encoding
- Fix: Response.ok raises Timeout Exception in safe_mode

0.10.8 (2012-03-09)

- Generate chunked ValueError fix
- Proxy configuration by environment variables
- Simplification of iter_lines.
- New *trust_env* configuration for disabling system/environment hints.
- Suppress cookie errors.

0.10.7 (2012-03-07)

- *encode_uri* = False

0.10.6 (2012-02-25)

- Allow '=' in cookies.

0.10.5 (2012-02-25)

- Response body with 0 content-length fix.
- New async.imap.
- Don't fail on netrc.

0.10.4 (2012-02-20)

- Honor netrc.

0.10.3 (2012-02-20)

- HEAD requests don't follow redirects anymore.
- raise_for_status() doesn't raise for 3xx anymore.
- Make Session objects picklable.
- ValueError for invalid schema URLs.

0.10.2 (2012-01-15)

- Vastly improved URL quoting.
- Additional allowed cookie key values.
- Attempted fix for "Too many open files" Error
- Replace unicode errors on first pass, no need for second pass.
- Append '/' to bare-domain urls before query insertion.
- Exceptions now inherit from RuntimeError.
- Binary uploads + auth fix.
- Bugfixes.

0.10.1 (2012-01-23)

- PYTHON 3 SUPPORT!
- Dropped 2.5 Support. (*Backwards Incompatible*)

0.10.0 (2012-01-21)

- `Response.content` is now bytes-only. (*Backwards Incompatible*)
- New `Response.text` is unicode-only.
- If no `Response.encoding` is specified and `chardet` is available, `Response.text` will guess an encoding.
- Default to ISO-8859-1 (Western) encoding for "text" subtypes.
- Removal of *decode_unicode*. (*Backwards Incompatible*)

- New multiple-hooks system.
- New `Response.register_hook` for registering hooks within the pipeline.
- `Response.url` is now Unicode.

0.9.3 (2012-01-18)

- SSL verify=False bugfix (apparent on windows machines).

0.9.2 (2012-01-18)

- Asynchronous async.send method.
- Support for proper chunk streams with boundaries.
- session argument for Session classes.
- Print entire hook tracebacks, not just exception instance.
- Fix response.iter_lines from pending next line.
- Fix but in HTTP-digest auth w/ URI having query strings.
- Fix in Event Hooks section.
- Urllib3 update.

0.9.1 (2012-01-06)

- danger_mode for automatic Response.raise_for_status()
- Response.iter_lines refactor

0.9.0 (2011-12-28)

- verify ssl is default.

0.8.9 (2011-12-28)

- Packaging fix.

0.8.8 (2011-12-28)

- SSL CERT VERIFICATION!
- Release of Cerifi: Mozilla's cert list.
- New 'verify' argument for SSL requests.
- Urllib3 update.

0.8.7 (2011-12-24)

- iter_lines last-line truncation fix
- Force safe_mode for async requests
- Handle safe_mode exceptions more consistently
- Fix iteration on null responses in safe_mode

0.8.6 (2011-12-18)

- Socket timeout fixes.
- Proxy Authorization support.

0.8.5 (2011-12-14)

- Response.iter_lines!

0.8.4 (2011-12-11)

- Prefetch bugfix.
- Added license to installed version.

0.8.3 (2011-11-27)

- Converted auth system to use simpler callable objects.
- New session parameter to API methods.
- Display full URL while logging.

0.8.2 (2011-11-19)

- New Unicode decoding system, based on over-ridable *Response.encoding*.
- Proper URL slash-quote handling.
- Cookies with `[`, `]`, and _ allowed.

0.8.1 (2011-11-15)

- URL Request path fix
- Proxy fix.
- Timeouts fix.

0.8.0 (2011-11-13)

- Keep-alive support!
- Complete removal of Urllib2
- Complete removal of Poster
- Complete removal of CookieJars
- New ConnectionError raising
- Safe_mode for error catching
- prefetch parameter for request methods
- OPTION method
- Async pool size throttling
- File uploads send real names
- Vendored in urllib3

0.7.6 (2011-11-07)

- Digest authentication bugfix (attach query data to path)

0.7.5 (2011-11-04)

- Response.content = None if there was an invalid response.
- Redirection auth handling.

0.7.4 (2011-10-26)

- Session Hooks fix.

0.7.3 (2011-10-23)

- Digest Auth fix.

0.7.2 (2011-10-23)

- PATCH Fix.

0.7.1 (2011-10-23)

- Move away from urllib2 authentication handling.
- Fully Remove AuthManager, AuthObject, &c.
- New tuple-based auth system with handler callbacks.

0.7.0 (2011-10-22)

- Sessions are now the primary interface.
- Deprecated InvalidMethodException.
- PATCH fix.
- New config system (no more global settings).

0.6.6 (2011-10-19)

- Session parameter bugfix (params merging).

0.6.5 (2011-10-18)

- Offline (fast) test suite.
- Session dictionary argument merging.

0.6.4 (2011-10-13)

- Automatic decoding of unicode, based on HTTP Headers.
- New `decode_unicode` setting.
- Removal of `r.read/close` methods.
- New `r.faw` interface for advanced response usage.*
- Automatic expansion of parameterized headers.

0.6.3 (2011-10-13)

- Beautiful `requests.async` module, for making async requests w/ gevent.

0.6.2 (2011-10-09)

- GET/HEAD obeys allow_redirects=False.

0.6.1 (2011-08-20)

- Enhanced status codes experience `\o/`
- Set a maximum number of redirects (`settings.max_redirects`)
- Full Unicode URL support
- Support for protocol-less redirects.
- Allow for arbitrary request types.
- Bugfixes

0.6.0 (2011-08-17)

- New callback hook system
- New persistent sessions object and context manager
- Transparent Dict-cookie handling
- Status code reference object
- Removed Response.cached
- Added Response.request
- All args are kwargs
- Relative redirect support
- HTTPError handling improvements
- Improved https testing
- Bugfixes

0.5.1 (2011-07-23)

- International Domain Name Support!
- Access headers without fetching entire body (`read()`)
- Use lists as dicts for parameters
- Add Forced Basic Authentication
- Forced Basic is default authentication type
- `python-requests.org` default User-Agent header
- CaseInsensitiveDict lower-case caching
- Response.history bugfix

0.5.0 (2011-06-21)

- PATCH Support
- Support for Proxies
- HTTPBin Test Suite
- Redirect Fixes
- settings.verbose stream writing
- Querystrings for all methods
- URLErrors (Connection Refused, Timeout, Invalid URLs) are treated as explicitly raised `r.requests.get('hwe://blah'); r.raise_for_status()`

0.4.1 (2011-05-22)

- Improved Redirection Handling
- New 'allow_redirects' param for following non-GET/HEAD Redirects
- Settings module refactoring

0.4.0 (2011-05-15)

- Response.history: list of redirected responses
- Case-Insensitive Header Dictionaries!
- Unicode URLs

0.3.4 (2011-05-14)

- Urllib2 HTTPAuthentication Recursion fix (Basic/Digest)
- Internal Refactor
- Bytes data upload Bugfix

0.3.3 (2011-05-12)

- Request timeouts
- Unicode url-encoded data
- Settings context manager and module

0.3.2 (2011-04-15)

- Automatic Decompression of GZip Encoded Content
- AutoAuth Support for Tupled HTTP Auth

0.3.1 (2011-04-01)

- Cookie Changes
- Response.read()
- Poster fix

0.3.0 (2011-02-25)

- Automatic Authentication API Change
- Smarter Query URL Parameterization
- Allow file uploads and POST data together
- **New Authentication Manager System**
  - Simpler Basic HTTP System

  - Supports all build-in urllib2 Auths
  - Allows for custom Auth Handlers

0.2.4 (2011-02-19)

- Python 2.5 Support
- PyPy-c v1.4 Support
- Auto-Authentication tests
- Improved Request object constructor

0.2.3 (2011-02-15)

- **New HTTPHandling Methods**
  - Response.__nonzero__ (false if bad HTTP Status)
  - Response.ok (True if expected HTTP Status)
  - Response.error (Logged HTTPError if bad HTTP Status)
  - Response.raise_for_status() (Raises stored HTTPError)

0.2.2 (2011-02-14)

- Still handles request in the event of an HTTPError. (Issue #2)
- Eventlet and Gevent Monkeypatch support.
- Cookie Support (Issue #1)

0.2.1 (2011-02-14)

- Added file attribute to POST and PUT requests for multipart-encode file uploads.
- Added Request.url attribute for context and redirects

0.2.0 (2011-02-14)

- Birth!

0.0.1 (2011-02-13)

- Frustration
- Conception

Release Process and Rules

New in version v2.6.2.

Starting with the version to be released after `v2.6.2`, the following rules will govern and describe how the Requests core team produces a new release.

Major Releases

A major release will include breaking changes. When it is versioned, it will be versioned as `vX.0.0`. For example, if the previous release was `v10.2.7` the next version will be `v11.0.0`.

Breaking changes are changes that break backwards compatibility with prior versions. If the project were to change the `text` attribute on a `Response` object to a method, that would only happen in a Major release.

Major releases may also include miscellaneous bug fixes and upgrades to vendored packages. The core developers of Requests are committed to providing a good user experience. This means we're also committed to preserving backwards compatibility as much as possible. Major releases will be infrequent and will need strong justifications before they are considered.

Minor Releases

A minor release will not include breaking changes but may include miscellaneous bug fixes and upgrades to vendored packages. If the previous version of Requests released was `v10.2.7` a minor release would be versioned as `v10.3.0`.

Minor releases will be backwards compatible with releases that have the same major version number. In other words, all versions that would start with `v10.` should be compatible with each other.

Hotfix Releases

A hotfix release will only include bug fixes that were missed when the project released the previous version. If the previous version of Requests released `v10.2.7` the hotfix release would be versioned as `v10.2.8`.

Hotfixes will **not** include upgrades to vendored dependencies after `v2.6.2`

Reasoning

In the 2.5 and 2.6 release series, the Requests core team upgraded vendored dependencies and caused a great deal of headaches for both users and the core team. To reduce this pain, we're forming a concrete set of procedures so expectations will be properly set.

CHAPTER 5

The API Documentation / Guide

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

Developer Interface

This part of the documentation covers all the interfaces of Requests. For parts where Requests depends on external libraries, we document the most important right here and provide links to the canonical documentation.

Main Interface

All of Requests' functionality can be accessed by these 7 methods. They all return an instance of the *Response* object.

`requests.`**`request`**(*method*, *url*, ***kwargs*)
Constructs and sends a *Request*.

Parameters

- **`method`** – method for the new *Request* object.
- **`url`** – URL for the new *Request* object.
- **`params`** – (optional) Dictionary or bytes to be sent in the query string for the *Request*.
- **`data`** – (optional) Dictionary or list of tuples `[(key, value)]` (will be form-encoded), bytes, or file-like object to send in the body of the *Request*.
- **`json`** – (optional) json data to send in the body of the *Request*.
- **`headers`** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **`cookies`** – (optional) Dict or CookieJar object to send with the *Request*.
- **`files`** – (optional) Dictionary of `'name': file-like-objects` (or `{'name': file-tuple}`) for multipart encoding upload. `file-tuple` can be a 2-tuple `('filename', fileobj)`, 3-tuple `('filename', fileobj,`

'content_type') or a 4-tuple ('filename', fileobj, 'content_type', custom_headers), where 'content-type' is a string defining the content type of the given file and custom_headers a dict-like object containing additional headers to add for the file.

- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How many seconds to wait for the server to send data before giving up, as a float, or a *(connect timeout, read timeout)* tuple.
- **allow_redirects** (*bool*) – (optional) Boolean. Enable/disable GET/OPTIONS/POST/PUT/PATCH/DELETE/HEAD redirection. Defaults to True.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. Defaults to True.
- **stream** – (optional) if False, the response content will be immediately downloaded.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Returns *Response* object

Return type *requests.Response*

Usage:

```
>>> import requests
>>> req = requests.request('GET', 'http://httpbin.org/get')
<Response [200]>
```

requests.**head**(*url*, ***kwargs*)
Sends a HEAD request.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that request takes.

Returns *Response* object

Return type *requests.Response*

requests.**get**(*url*, *params=None*, ***kwargs*)
Sends a GET request.

Parameters

- **url** – URL for the new *Request* object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the *Request*.
- ****kwargs** – Optional arguments that request takes.

Returns *Response* object

Return type *requests.Response*

requests.**post**(*url*, *data=None*, *json=None*, ***kwargs*)
Sends a POST request.

Parameters

- **url** – URL for the new *Request* object.

- **data** – (optional) Dictionary (will be form-encoded), bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json data to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Returns *Response* object

Return type *requests.Response*

`requests.`**`put`**(*url*, *data=None*, ***kwargs*)
Sends a PUT request.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary (will be form-encoded), bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json data to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Returns *Response* object

Return type *requests.Response*

`requests.`**`patch`**(*url*, *data=None*, ***kwargs*)
Sends a PATCH request.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary (will be form-encoded), bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json data to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Returns *Response* object

Return type *requests.Response*

`requests.`**`delete`**(*url*, ***kwargs*)
Sends a DELETE request.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Returns *Response* object

Return type *requests.Response*

Exceptions

exception `requests.`**`RequestException`**(**args*, ***kwargs*)
There was an ambiguous exception that occurred while handling your request.

exception `requests.`**`ConnectionError`**(**args*, ***kwargs*)
A Connection error occurred.

exception `requests.`**`HTTPError`**(**args*, ***kwargs*)
An HTTP error occurred.

exception `requests.`**`URLRequired`**(**args*, ***kwargs*)
A valid URL is required to make a request.

exception `requests.`**`TooManyRedirects`**(**args*, ***kwargs*)
Too many redirects.

exception `requests.`**`ConnectTimeout`**(**args*, ***kwargs*)
The request timed out while trying to connect to the remote server.

Requests that produced this error are safe to retry.

exception `requests.`**`ReadTimeout`**(**args*, ***kwargs*)
The server did not send any data in the allotted amount of time.

exception `requests.`**`Timeout`**(**args*, ***kwargs*)
The request timed out.

Catching this error will catch both `ConnectTimeout` and `ReadTimeout` errors.

Request Sessions

class `requests.`**`Session`**
A Requests session.

Provides cookie persistence, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
<Response [200]>
```

Or as a context manager:

```
>>> with requests.Session() as s:
>>>     s.get('http://httpbin.org/get')
<Response [200]>
```

`auth` = None
Default Authentication tuple or object to attach to *`Request`*.

`cert` = None
SSL client certificate default, if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

`close`()
Closes all adapters and as such the session

`cookies` = None
A CookieJar containing all currently outstanding cookies set on this session. By default it is a *`RequestsCookieJar`*, but may be any other `cookielib.CookieJar` compatible object.

`delete`(*url*, ***kwargs*)
Sends a DELETE request. Returns *`Response`* object.

Parameters

- **`url`** – URL for the new *`Request`* object.

- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

get(*url*, ***kwargs*)
Sends a GET request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

get_adapter(*url*)
Returns the appropriate connection adapter for the given URL.

Return type *requests.adapters.BaseAdapter*

get_redirect_target(*resp*)
Receives a Response. Returns a redirect URI or `None`

head(*url*, ***kwargs*)
Sends a HEAD request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

headers = None
A case-insensitive dictionary of headers to be sent on each `Request` sent from this `Session`.

hooks = None
Event-handling hooks.

max_redirects = None
Maximum number of redirects allowed. If the request exceeds this limit, a `TooManyRedirects` exception is raised. This defaults to requests.models.DEFAULT_REDIRECT_LIMIT, which is 30.

merge_environment_settings(*url*, *proxies*, *stream*, *verify*, *cert*)
Check the environment and merge it with some settings.

Return type dict

mount(*prefix*, *adapter*)
Registers a connection adapter to a prefix.

Adapters are sorted in descending order by key length.

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

params = None
Dictionary of querystring data to attach to each *Request*. The dictionary values may be lists for representing multivalued query parameters.

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns *Response* object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

post(*url*, *data=None*, *json=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

prepare_request(*request*)
Constructs a *PreparedRequest* for transmission and returns it. The *PreparedRequest* has settings merged from the *Request* instance and those of the *Session*.

Parameters **request** – *Request* instance to prepare with this session's settings.

Return type *requests.PreparedRequest*

proxies = None
Dictionary mapping protocol or protocol and host to the URL of the proxy (e.g. {'http': 'foo.bar:3128', 'http://host.name': 'foo.bar:4012'}) to be used on each *Request*.

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns *Response* object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Return type *requests.Response*

rebuild_auth(*prepared_request*, *response*)
When being redirected we may want to strip authentication from the request to avoid leaking credentials. This method intelligently removes and reapplies authentication where possible to avoid credential loss.

rebuild_method(*prepared_request*, *response*)
When being redirected we may want to change the method of the request based on certain specs or browser behavior.

rebuild_proxies(*prepared_request*, *proxies*)
This method re-evaluates the proxy configuration by considering the environment variables. If we are redirected to a URL covered by NO_PROXY, we strip the proxy configuration. Otherwise, we set missing proxy keys for this URL (in case they were stripped by a previous redirect).

This method also replaces the Proxy-Authorization header where necessary.

Return type dict

request(*method*, *url*, *params=None*, *data=None*, *headers=None*, *cookies=None*, *files=None*, *auth=None*, *timeout=None*, *allow_redirects=True*, *proxies=None*, *hooks=None*, *stream=None*, *verify=None*, *cert=None*, *json=None*)
Constructs a `Request`, prepares it and sends it. Returns `Response` object.

Parameters

- **method** – method for the new `Request` object.
- **url** – URL for the new `Request` object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*`float or tuple`*) – (optional) How long to wait for the server to send data before giving up, as a float, or a *(connect timeout, read timeout)* tuple.
- **allow_redirects** (*`bool`*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol or protocol and hostname to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. Defaults to `True`.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Return type *requests.Response*

resolve_redirects(*resp*, *req*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*, *yield_requests=False*, ***adapter_kwargs*)
Receives a Response. Returns a generator of Responses or Requests.

send(*request*, ***kwargs*)
Send a given PreparedRequest.

Return type *requests.Response*

stream = None
Stream response content default.

trust_env = None
Trust environment settings for proxy configuration, default authentication and similar.

verify = None
SSL Verification default.

Lower-Level Classes

class requests.**Request**(*method=None*, *url=None*, *headers=None*, *files=None*, *data=None*, *params=None*, *auth=None*, *cookies=None*, *hooks=None*, *json=None*)
A user-created *Request* object.

Used to prepare a *PreparedRequest*, which is sent to the server.

Parameters

- **method** – HTTP method to use.
- **url** – URL to send.
- **headers** – dictionary of headers to send.
- **files** – dictionary of {filename: fileobject} files to multipart upload.
- **data** – the body to attach to the request. If a dictionary is provided, form-encoding will take place.
- **json** – json for the body to attach to the request (if files or data is not specified).
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> req.prepare()
<PreparedRequest [GET]>
```

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

prepare()
Constructs a *PreparedRequest* for transmission and returns it.

register_hook(*event*, *hook*)
Properly register a hook.

class requests.**Response**
The *Response* object, which contains a server's response to an HTTP request.

apparent_encoding
The apparent encoding, provided by the chardet library.

close()
Releases the connection back to the pool. Once this method has been called the underlying `raw` object must not be accessed again.

Note: Should not normally need to be called explicitly.

content
Content of the response, in bytes.

cookies = None
A CookieJar of Cookies the server sent back.

elapsed = None
The amount of time elapsed between sending the request and the arrival of the response (as a timedelta). This property specifically measures the time taken between sending the first byte of the request and finishing parsing the headers. It is therefore unaffected by consuming the response content or the value of the `stream` keyword argument.

encoding = None
Encoding to decode with when accessing r.text.

headers = None
Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

history = None
A list of *Response* objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

is_permanent_redirect
True if this Response one of the permanent versions of redirect.

is_redirect
True if this Response is a well-formed HTTP redirect that could have been processed automatically (by *Session.resolve_redirects*).

iter_content(*chunk_size=1*, *decode_unicode=False*)
Iterates over the response data. When stream=True is set on the request, this avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

chunk_size must be of type int or None. A value of None will function differently depending on the value of *stream*. stream=True will read data as it arrives in whatever size the chunks are received. If stream=False, data is returned as a single chunk.

If decode_unicode is True, content will be decoded using the best available encoding based on the response.

iter_lines(*chunk_size=512*, *decode_unicode=None*, *delimiter=None*)
Iterates over the response data, one line at a time. When stream=True is set on the request, this avoids reading the content at once into memory for large responses.

Note: This method is not reentrant safe.

---

json(***kwargs*)
Returns the json-encoded content of a response, if any.

Parameters ****kwargs** – Optional arguments that `json.loads` takes.

Raises **ValueError** – If the response body does not contain valid json.

links
Returns the parsed header links of the response, if any.

next
Returns a PreparedRequest for the next request in a redirect chain, if there is one.

ok
Returns True if *status_code* is less than 400.

This attribute checks if the status code of the response is between 400 and 600 to see if there was a client error or a server error. If the status code, is between 200 and 400, this will return True. This is **not** a check to see if the response code is `200 OK`.

raise_for_status()
Raises stored *HTTPError*, if one occurred.

raw = None
File-like object representation of response (for advanced usage). Use of `raw` requires that `stream=True` be set on the request.

reason = None
Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".

request = None
The *PreparedRequest* object to which this is a response.

status_code = None
Integer Code of responded HTTP Status, e.g. 404 or 200.

text
Content of the response, in unicode.

If Response.encoding is None, encoding will be guessed using `chardet`.

The encoding of the response content is determined based solely on HTTP headers, following RFC 2616 to the letter. If you can take advantage of non-HTTP knowledge to make a better guess at the encoding, you should set `r.encoding` appropriately before accessing this property.

url = None
Final URL location of Response.

Lower-Lower-Level Classes

class `requests.`**PreparedRequest**
The fully mutable *PreparedRequest* object, containing the exact bytes that will be sent to the server.

Generated from either a *Request* object or manually.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> r = req.prepare()
<PreparedRequest [GET]>

>>> s = requests.Session()
>>> s.send(r)
<Response [200]>
```

body = None
request body to send to the server.

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

headers = None
dictionary of HTTP headers.

hooks = None
dictionary of callback hooks, for internal usage.

method = None
HTTP verb to send to the server.

path_url
Build the path URL to use.

prepare(*method=None*, *url=None*, *headers=None*, *files=None*, *data=None*, *params=None*, *auth=None*, *cookies=None*, *hooks=None*, *json=None*)
Prepares the entire request with the given parameters.

prepare_auth(*auth*, *url=''*)
Prepares the given HTTP auth data.

prepare_body(*data*, *files*, *json=None*)
Prepares the given HTTP body data.

prepare_content_length(*body*)
Prepare Content-Length header based on request method and body

prepare_cookies(*cookies*)
Prepares the given HTTP cookie data.

This function eventually generates a `Cookie` header from the given cookies using cookielib. Due to cookielib's design, the header will not be regenerated if it already exists, meaning this function can only be called once for the life of the *PreparedRequest* object. Any subsequent calls to `prepare_cookies` will have no actual effect, unless the "Cookie" header is removed beforehand.

prepare_headers(*headers*)
Prepares the given HTTP headers.

prepare_hooks(*hooks*)
Prepares the given hooks.

prepare_method(*method*)
Prepares the given HTTP method.

prepare_url(*url*, *params*)
Prepares the given HTTP URL.

register_hook(*event*, *hook*)
Properly register a hook.

url = None
HTTP URL to send the request to.

class `requests.adapters.`**BaseAdapter**
The Base Transport Adapter

close()
Cleans up adapter specific items.

send(*request*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Sends PreparedRequest object. Returns Response object.

Parameters

- **request** – The *PreparedRequest* being sent.
- **stream** – (optional) Whether to stream the request content.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a *(connect timeout, read timeout)* tuple.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use
- **cert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

class requests.adapters.**HTTPAdapter**(*pool_connections=10*, *pool_maxsize=10*, *max_retries=0*, *pool_block=False*)

The built-in HTTP Adapter for urllib3.

Provides a general-case interface for Requests sessions to contact HTTP and HTTPS urls by implementing the Transport Adapter interface. This class will usually be created by the *Session* class under the covers.

Parameters

- **pool_connections** – The number of urllib3 connection pools to cache.
- **pool_maxsize** – The maximum number of connections to save in the pool.
- **max_retries** – The maximum number of retries each connection should attempt. Note, this applies only to failed DNS lookups, socket connections and connection timeouts, never to requests where data has made it to the server. By default, Requests does not retry failed connections. If you need granular control over the conditions under which we retry a request, import urllib3's `Retry` class and pass that instead.
- **pool_block** – Whether the connection pool should block for connections.

Usage:

```
>>> import requests
>>> s = requests.Session()
>>> a = requests.adapters.HTTPAdapter(max_retries=3)
>>> s.mount('http://', a)
```

add_headers(*request*, ***kwargs*)

Add any headers needed by the connection. As of v2.0 this does nothing by default, but is left for overriding by users that subclass the *HTTPAdapter*.

This should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **request** – The *PreparedRequest* to add headers to.
- **kwargs** – The keyword arguments from the call to send().

build_response(*req*, *resp*)

Builds a *Response* object from a urllib3 response. This should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*

Parameters

- **req** – The *PreparedRequest* used to generate the response.
- **resp** – The urllib3 response object.

Return type *requests.Response*

cert_verify(*conn*, *url*, *verify*, *cert*)
Verify a SSL certificate. This method should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **conn** – The urllib3 connection object associated with the cert.
- **url** – The requested URL.
- **verify** – Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use
- **cert** – The SSL certificate to verify.

close()
Disposes of any internal state.

Currently, this closes the PoolManager and any active ProxyManager, which closes any pooled connections.

get_connection(*url*, *proxies=None*)
Returns a urllib3 connection for the given URL. This should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **url** – The URL to connect to.
- **proxies** – (optional) A Requests-style dictionary of proxies used on this request.

Return type urllib3.ConnectionPool

init_poolmanager(*connections*, *maxsize*, *block=False*, ***pool_kwargs*)
Initializes a urllib3 PoolManager.

This method should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **connections** – The number of urllib3 connection pools to cache.
- **maxsize** – The maximum number of connections to save in the pool.
- **block** – Block when no free connections are available.
- **pool_kwargs** – Extra keyword arguments used to initialize the Pool Manager.

proxy_headers(*proxy*)
Returns a dictionary of the headers to add to any request sent through a proxy. This works with urllib3 magic to ensure that they are correctly sent to the proxy, rather than in a tunnelled request if CONNECT is being used.

This should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters **proxies** – The url of the proxy being used for this request.

Return type dict

proxy_manager_for(*proxy*, ***proxy_kwargs*)
Return urllib3 ProxyManager for the given proxy.

This method should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **proxy** – The proxy to return a urllib3 ProxyManager for.
- **proxy_kwargs** – Extra keyword arguments used to configure the Proxy Manager.

Returns ProxyManager

Return type urllib3.ProxyManager

request_url(*request*, *proxies*)
Obtain the url to use when making the final request.

If the message is being sent through a HTTP proxy, the full URL has to be used. Otherwise, we should only use the path portion of the URL.

This should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameters

- **request** – The *PreparedRequest* being sent.
- **proxies** – A dictionary of schemes or schemes and hosts to proxy URLs.

Return type str

send(*request*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Sends PreparedRequest object. Returns Response object.

Parameters

- **request** – The *PreparedRequest* being sent.
- **stream** – (optional) Whether to stream the request content.
- **timeout** (*float or tuple or urllib3 Timeout object*) – (optional) How long to wait for the server to send data before giving up, as a float, or a *(connect timeout, read timeout)* tuple.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use
- **cert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

Return type *requests.Response*

Authentication

class requests.auth.**AuthBase**
Base class that all auth implementations derive from

class requests.auth.**HTTPBasicAuth**(*username*, *password*)
Attaches HTTP Basic Authentication to the given Request object.

class requests.auth.**HTTPProxyAuth**(*username*, *password*)
Attaches HTTP Proxy Authentication to a given Request object.

class requests.auth.**HTTPDigestAuth**(*username*, *password*)
Attaches HTTP Digest Authentication to the given Request object.

Encodings

requests.utils.**get_encodings_from_content**(*content*)
Returns encodings from given content string.

Parameters **content** – bytestring to extract encodings from.

requests.utils.**get_encoding_from_headers**(*headers*)
Returns encodings from given HTTP Header Dict.

Parameters **headers** – dictionary to extract encoding from.

Return type str

requests.utils.**get_unicode_from_response**(*r*)
Returns the requested content back in unicode.

Parameters **r** – Response object to get unicode content from.

Tried:

1.charset from content-type

2.fall back and replace all unicode characters

Return type str

Cookies

requests.utils.**dict_from_cookiejar**(*cj*)
Returns a key/value dictionary from a CookieJar.

Parameters **cj** – CookieJar object to extract cookies from.

Return type dict

requests.utils.**add_dict_to_cookiejar**(*cj*, *cookie_dict*)
Returns a CookieJar from a key/value dictionary.

Parameters

- **cj** – CookieJar to insert cookies into.
- **cookie_dict** – Dict of key/values to insert into CookieJar.

Return type CookieJar

requests.cookies.**cookiejar_from_dict**(*cookie_dict*, *cookiejar=None*, *overwrite=True*)
Returns a CookieJar from a key/value dictionary.

Parameters

- **cookie_dict** – Dict of key/values to insert into CookieJar.
- **cookiejar** – (optional) A cookiejar to add the cookies to.
- **overwrite** – (optional) If False, will not replace cookies already in the jar with new ones.

class requests.cookies.**RequestsCookieJar**(*policy=None*)
Compatibility class; is a cookielib.CookieJar, but exposes a dict interface.

This is the CookieJar we create by default for requests and sessions that don't specify one, since some clients may expect response.cookies and session.cookies to support dict operations.

Requests does not use the dict interface internally; it's just for compatibility with external client code. All requests code should work out of the box with externally provided instances of `CookieJar`, e.g. `LWPCookieJar` and `FileCookieJar`.

Unlike a regular CookieJar, this class is pickleable.

> **Warning:** dictionary operations that are normally O(1) may be O(n).

add_cookie_header(*request*)
Add correct Cookie: header to request (urllib2.Request object).

The Cookie2 header is also added unless policy.hide_cookie2 is true.

clear(*domain=None*, *path=None*, *name=None*)
Clear some cookies.

Invoking this method without arguments will clear all cookies. If given a single argument, only cookies belonging to that domain will be removed. If given two arguments, cookies belonging to the specified path within that domain are removed. If given three arguments, then the cookie with the specified name, path and domain is removed.

Raises KeyError if no matching cookie exists.

clear_expired_cookies()
Discard all expired cookies.

You probably don't need to call this method: expired cookies are never sent back to the server (provided you're using DefaultCookiePolicy), this method is called by CookieJar itself every so often, and the .save() method won't save expired cookies anyway (unless you ask otherwise by passing a true ignore_expires argument).

clear_session_cookies()
Discard all session cookies.

Note that the .save() method won't save session cookies anyway, unless you ask otherwise by passing a true ignore_discard argument.

copy()
Return a copy of this RequestsCookieJar.

extract_cookies(*response*, *request*)
Extract cookies from response, where allowable given the request.

get(*name*, *default=None*, *domain=None*, *path=None*)
Dict-like get() that also supports optional domain and path args in order to resolve naming collisions from using one cookie jar over multiple domains.

> **Warning:** operation is O(n), not O(1).

get_dict(*domain=None*, *path=None*)
Takes as an argument an optional domain and path and returns a plain old Python dict of name-value pairs of cookies that meet the requirements.

Return type dict

items()
Dict-like items() that returns a list of name-value tuples from the jar. Allows client-code to call `dict(RequestsCookieJar)` and get a vanilla python dict of key value pairs.

See also:

keys() and values().

iteritems()
Dict-like iteritems() that returns an iterator of name-value tuples from the jar.

See also:

iterkeys() and itervalues().

iterkeys()
Dict-like iterkeys() that returns an iterator of names of cookies from the jar.

See also:

itervalues() and iteritems().

itervalues()
Dict-like itervalues() that returns an iterator of values of cookies from the jar.

See also:

iterkeys() and iteritems().

keys()
Dict-like keys() that returns a list of names of cookies from the jar.

See also:

values() and items().

list_domains()
Utility method to list all the domains in the jar.

list_paths()
Utility method to list all the paths in the jar.

make_cookies(*response*, *request*)
Return sequence of Cookie objects extracted from response object.

multiple_domains()
Returns True if there are multiple domains in the jar. Returns False otherwise.

Return type bool

pop($k[, d]$) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

set(*name*, *value*, ***kwargs*)
Dict-like set() that also supports optional domain and path args in order to resolve naming collisions from using one cookie jar over multiple domains.

set_cookie_if_ok(*cookie*, *request*)
Set a cookie if policy says it's OK to do so.

setdefault($k[, d]$) → D.get(k,d), also set D[k]=d if k not in D

update(*other*)
Updates this jar with cookies from another CookieJar or dict-like

values()
Dict-like values() that returns a list of values of cookies from the jar.

See also:

keys() and items().

class requests.cookies.**CookieConflictError**
There are two cookies that meet the criteria specified in the cookie jar. Use .get and .set and include domain and path args in order to be more specific.

Status Code Lookup

requests.**codes**

```
>>> requests.codes['temporary_redirect']
307

>>> requests.codes.teapot
418

>>> requests.codes['\o/']
200
```

Migrating to 1.x

This section details the main differences between 0.x and 1.x and is meant to ease the pain of upgrading.

API Changes

- `Response.json` is now a callable and not a property of a response.

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()   # This *call* raises an exception if JSON decoding fails
```

- The `Session` API has changed. Sessions objects no longer take parameters. `Session` is also now capitalized, but it can still be instantiated with a lowercase `session` for backwards compatibility.

```
s = requests.Session()    # formerly, session took parameters
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- All request hooks have been removed except 'response'.
- Authentication helpers have been broken out into separate modules. See requests-oauthlib and requests-kerberos.
- The parameter for streaming requests was changed from `prefetch` to `stream` and the logic was inverted. In addition, `stream` is now required for raw response reading.

```
# in 0.x, passing prefetch=False would accomplish the same thing
r = requests.get('https://github.com/timeline.json', stream=True)
for chunk in r.iter_content(8192):
    ...
```

- The `config` parameter to the requests method has been removed. Some of these options are now configured on a `Session` such as keep-alive and maximum number of redirects. The verbosity option should be handled by configuring logging.

```
import requests
import logging

# Enabling debugging at http.client level (requests->urllib3->http.client)
# you will see the REQUEST, including HEADERS and DATA, and RESPONSE with HEADERS␣
↪but without DATA.
# the only thing missing will be the response.body which is not logged.
try: # for Python 3
    from http.client import HTTPConnection
except ImportError:
    from httplib import HTTPConnection
HTTPConnection.debuglevel = 1

logging.basicConfig() # you need to initialize logging, otherwise you will not␣
↪see anything from requests
logging.getLogger().setLevel(logging.DEBUG)
requests_log = logging.getLogger("requests.packages.urllib3")
requests_log.setLevel(logging.DEBUG)
requests_log.propagate = True

requests.get('http://httpbin.org/headers')
```

Licensing

One key difference that has nothing to do with the API is a change in the license from the ISC license to the Apache 2.0 license. The Apache 2.0 license ensures that contributions to Requests are also covered by the Apache 2.0 license.

Migrating to 2.x

Compared with the 1.0 release, there were relatively few backwards incompatible changes, but there are still a few issues to be aware of with this major release.

For more details on the changes in this release including new APIs, links to the relevant GitHub issues and some of the bug fixes, read Cory's blog on the subject.

API Changes

- There were a couple changes to how Requests handles exceptions. `RequestException` is now a subclass of `IOError` rather than `RuntimeError` as that more accurately categorizes the type of error. In addition, an invalid URL escape sequence now raises a subclass of `RequestException` rather than a `ValueError`.

```
requests.get('http://%zz/')    # raises requests.exceptions.InvalidURL
```

  Lastly, `httplib.IncompleteRead` exceptions caused by incorrect chunked encoding will now raise a Requests `ChunkedEncodingError` instead.

- The proxy API has changed slightly. The scheme for a proxy URL is now required.

```
proxies = {
  "http": "10.10.1.10:3128",    # use http://10.10.1.10:3128 instead
}
```

```
# In requests 1.x, this was legal, in requests 2.x,
#  this raises requests.exceptions.MissingSchema
requests.get("http://example.org", proxies=proxies)
```

Behavioural Changes

- Keys in the `headers` dictionary are now native strings on all Python versions, i.e. bytestrings on Python 2 and unicode on Python 3. If the keys are not native strings (unicode on Python 2 or bytestrings on Python 3) they will be converted to the native string type assuming UTF-8 encoding.
- Values in the `headers` dictionary should always be strings. This has been the project's position since before 1.0 but a recent change (since version 2.11.0) enforces this more strictly. It's advised to avoid passing header values as unicode when possible.

CHAPTER 6

The Contributor Guide

If you want to contribute to the project, this part of the documentation is for you.

Contributor's Guide

If you're reading this, you're probably interested in contributing to Requests. Thank you very much! Open source projects live-and-die based on the support they receive from others, and the fact that you're even considering contributing to the Requests project is *very* generous of you.

This document lays out guidelines and advice for contributing to this project. If you're thinking of contributing, please start by reading this document and getting a feel for how contributing to this project works. If you have any questions, feel free to reach out to either Ian Cordasco or Cory Benfield, the primary maintainers.

If you have non-technical feedback, philosophical ponderings, crazy ideas, or other general thoughts about Requests or its position within the Python ecosystem, the BDFL, Kenneth Reitz, would love to hear from you.

The guide is split into sections based on the type of contribution you're thinking of making, with a section that covers general guidelines for all contributors.

Be Cordial

> **Be cordial or be on your way**. —*Kenneth Reitz*

Requests has one very important rule governing all forms of contribution, including reporting bugs or requesting features. This golden rule is "be cordial or be on your way".

All contributions are welcome, as long as everyone involved is treated with respect.

Get Early Feedback

If you are contributing, do not feel the need to sit on your contribution until it is perfectly polished and complete. It helps everyone involved for you to seek feedback as early as you possibly can. Submitting an early, unfinished version

of your contribution for feedback in no way prejudices your chances of getting that contribution accepted, and can save you from putting a lot of work into a contribution that is not suitable for the project.

Contribution Suitability

Our project maintainers have the last word on whether or not a contribution is suitable for Requests. All contributions will be considered carefully, but from time to time, contributions will be rejected because they do not suit the current goals or needs of the project.

If your contribution is rejected, don't despair! As long as you followed these guidelines, you will have a much better chance of getting your next contribution accepted.

Code Contributions

Steps for Submitting Code

When contributing code, you'll want to follow this checklist:

1. Fork the repository on GitHub.
2. Run the tests to confirm they all pass on your system. If they don't, you'll need to investigate why they fail. If you're unable to diagnose this yourself, raise it as a bug report by following the guidelines in this document: *Bug Reports*.
3. Write tests that demonstrate your bug or feature. Ensure that they fail.
4. Make your change.
5. Run the entire test suite again, confirming that all tests pass *including the ones you just added*.
6. Send a GitHub Pull Request to the main repository's `master` branch. GitHub Pull Requests are the expected method of code collaboration on this project.

The following sub-sections go into more detail on some of the points above.

Code Review

Contributions will not be merged until they've been code reviewed. You should implement any code review feedback unless you strongly object to it. In the event that you object to the code review feedback, you should make your case clearly and calmly. If, after doing so, the feedback is judged to still apply, you must either apply the feedback or withdraw your contribution.

New Contributors

If you are new or relatively new to Open Source, welcome! Requests aims to be a gentle introduction to the world of Open Source. If you're concerned about how best to contribute, please consider mailing a maintainer (listed above) and asking for help.

Please also check the *Get Early Feedback* section.

Kenneth Reitz's Code Style™

The Requests codebase uses the PEP 8 code style.

In addition to the standards outlined in PEP 8, we have a few guidelines:

- Line-length can exceed 79 characters, to 100, when convenient.
- Line-length can exceed 100 characters, when doing otherwise would be *terribly* inconvenient.
- Always use single-quoted strings (e.g. `'#flatearth'`), unless a single-quote occurs within the string.

Additionally, one of the styles that PEP8 recommends for line continuations completely lacks all sense of taste, and is not to be permitted within the Requests codebase:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

No. Just don't. Please.

Docstrings are to follow the following syntaxes:

```
def the_earth_is_flat():
    """NASA divided up the seas into thirty-three degrees."""
    pass
```

```
def fibonacci_spiral_tool():
    """With my feet upon the ground I lose myself / between the sounds
    and open wide to suck it in. / I feel it move across my skin. / I'm
    reaching up and reaching out. / I'm reaching for the random or
    whatever will bewilder me. / Whatever will bewilder me. / And
    following our will and wind we may just go where no one's been. /
    We'll ride the spiral to the end and may just go where no one's
    been.

    Spiral out. Keep going...
    """
    pass
```

All functions, methods, and classes are to contain docstrings. Object data model methods (e.g. `__repr__`) are typically the exception to this rule.

Thanks for helping to make the world a better place!

Documentation Contributions

Documentation improvements are always welcome! The documentation files live in the `docs/` directory of the codebase. They're written in reStructuredText, and use Sphinx to generate the full suite of documentation.

When contributing documentation, please do your best to follow the style of the documentation files. This means a soft-limit of 79 characters wide in your text files and a semi-formal, yet friendly and approachable, prose style.

When presenting Python code, use single-quoted strings (`'hello'` instead of `"hello"`).

Bug Reports

Bug reports are hugely important! Before you raise one, though, please check through the GitHub issues, **both open and closed**, to confirm that the bug hasn't been reported before. Duplicate bug reports are a huge drain on the time of other contributors, and should be avoided as much as possible.

Feature Requests

Requests is in a perpetual feature freeze, only the BDFL can add or approve of new features. The maintainers believe that Requests is a feature-complete piece of software at this time.

One of the most important skills to have while maintaining a largely-used open source project is learning the ability to say "no" to suggested changes, while keeping an open ear and mind.

If you believe there is a feature missing, feel free to raise a feature request, but please do be aware that the overwhelming likelihood is that your feature request will not be accepted.

Development Philosophy

Requests is an open but opinionated library, created by an open but opinionated developer.

Management Style

Kenneth Reitz is the BDFL. He has final say in any decision related to the Requests project. Kenneth is responsible for the direction and form of the library, as well as its presentation. In addition to making decisions based on technical merit, he is responsible for making decisions based on the development philosophy of Requests.

Ian Cordasco and Cory Benfield are the core contributors. They are responsible for triaging bug reports, reviewing pull requests and ensuring that Kenneth is kept up to speed with developments around the library. The day-to-day managing of the project is done by the core contributors. They are responsible for making judgements about whether or not a feature request is likely to be accepted by Kenneth. Their word is, in some ways, more final than Kenneth's.

Values

- Simplicity is always better than functionality.
- Listen to everyone, then disregard it.
- The API is all that matters. Everything else is secondary.
- Fit the 90% use-case. Ignore the nay-sayers.

Semantic Versioning

For many years, the open source community has been plagued with version number dystonia. Numbers vary so greatly from project to project, they are practically meaningless.

Requests uses Semantic Versioning. This specification seeks to put an end to this madness with a small set of practical guidelines for you and your colleagues to use in your next project.

Standard Library?

Requests has no *active* plans to be included in the standard library. This decision has been discussed at length with Guido as well as numerous core developers.

Essentially, the standard library is where a library goes to die. It is appropriate for a module to be included when active development is no longer necessary.

Linux Distro Packages

Distributions have been made for many Linux repositories, including: Ubuntu, Debian, RHEL, and Arch.

These distributions are sometimes divergent forks, or are otherwise not kept up-to-date with the latest code and bug-fixes. PyPI (and its mirrors) and GitHub are the official distribution sources; alternatives are not supported by the Requests project.

How to Help

Requests is under active development, and contributions are more than welcome!

1. Check for open issues or open a fresh issue to start a discussion around 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 repository on GitHub and start making your changes to a new branch.
3. Write a test which shows that the bug was fixed.
4. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to AUTHORS.

Feature Freeze

As of v1.0.0, Requests has now entered a feature freeze. Requests for new features and Pull Requests implementing those features will not be accepted.

Development Dependencies

You'll need to install py.test in order to run the Requests' test suite:

```
$ venv .venv
$ source .venv/bin/activate

$ make
$ python setup.py test
============================= test session starts ==============================
platform darwin -- Python 3.4.4, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
...
collected 445 items

tests/test_hooks.py ...
tests/test_lowlevel.py ............
tests/test_requests.py ..............................................................
tests/test_structures.py ....................
tests/test_testserver.py ...........
tests/test_utils.py ..s.....................................................

============== 442 passed, 1 skipped, 2 xpassed in 46.48 seconds ===============
```

You can also run `$ make tests` to run against all supported Python versions, using tox/detox.

Runtime Environments

Requests currently supports the following versions of Python:

- Python 2.6
- Python 2.7
- Python 3.3
- Python 3.4
- Python 3.5
- Python 3.6
- PyPy

Google AppEngine is not officially supported although support is available with the Requests-Toolbelt.

Authors

Requests is written and maintained by Kenneth Reitz and various contributors:

Keepers of the Four Crystals

- Kenneth Reitz <me@kennethreitz.org> @kennethreitz, Keeper of the Master Crystal.
- Cory Benfield <cory@lukasa.co.uk> @lukasa
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24
- Nate Prewitt <nate.prewitt@gmail.com> @nateprewitt

Urllib3

- Andrey Petrov <andrey.petrov@shazow.net>

Patches and Suggestions

- Various Pocoo Members
- Chris Adams
- Flavio Percoco Premoli
- Dj Gilcrease
- Justin Murphy
- Rob Madole
- Aram Dulyan
- Johannes Gorset
- (Megane Murayama)
- James Rowe

- Daniel Schauenberg
- Zbigniew Siciarz
- Daniele Tricoli 'Eriol'
- Richard Boulton
- Miguel Olivares <miguel@moliware.com>
- Alberto Paro
- Jérémy Bethmont
- (Xu Pan)
- Tamás Gulácsi
- Rubén Abad
- Peter Manser
- Jeremy Selier
- Jens Diemer
- Alex (@alopatin)
- Tom Hogans <tomhsx@gmail.com>
- Armin Ronacher
- Shrikant Sharat Kandula
- Mikko Ohtamaa
- Den Shabalin
- Daniel Miller <danielm@vs-networks.com>
- Alejandro Giacometti
- Rick Mak
- Johan Bergström
- Josselin Jacquard
- Travis N. Vaught
- Fredrik Möllerstrand
- Daniel Hengeveld
- Dan Head
- Bruno Renié
- David Fischer
- Joseph McCullough
- Juergen Brendel
- Juan Riaza
- Ryan Kelly
- Rolando Espinoza La fuente
- Robert Gieseke

- Idan Gazit
- Ed Summers
- Chris Van Horne
- Christopher Davis
- Ori Livneh
- Jason Emerick
- Bryan Helmig
- Jonas Obrist
- Lucian Ursu
- Tom Moertel
- Frank Kumro Jr
- Chase Sterling
- Marty Alchin
- takluyver
- Ben Toews (@mastahyeti)
- David Kemp
- Brendon Crawford
- Denis (@Telofy)
- Matt Giuca
- Adam Tauber
- Honza Javorek
- Brendan Maguire <maguire.brendan@gmail.com>
- Chris Dary
- Danver Braganza <danverbraganza@gmail.com>
- Max Countryman
- Nick Chadwick
- Jonathan Drosdeck
- Jiri Machalek
- Steve Pulec
- Michael Kelly
- Michael Newman <newmaniese@gmail.com>
- Jonty Wareing <jonty@jonty.co.uk>
- Shivaram Lingamneni
- Miguel Turner
- Rohan Jain (@crodjer)
- Justin Barber <barber.justin@gmail.com>

- Roman Haritonov (@reclosedev)
- Josh Imhoff <joshimhoff13@gmail.com>
- Arup Malakar <amalakar@gmail.com>
- Danilo Bargen (@dbrgn)
- Torsten Landschoff
- Michael Holler (@apotheos)
- Timnit Gebru
- Sarah Gonzalez
- Victoria Mo
- Leila Muhtasib
- Matthias Rahlf <matthias@webding.de>
- Jakub Roztocil <jakub@roztocil.name>
- Rhys Elsmore
- André Graf (@dergraf)
- Stephen Zhuang (@everbird)
- Martijn Pieters
- Jonatan Heyman
- David Bonner <dbonner@gmail.com> (@rascalking)
- Vinod Chandru
- Johnny Goodnow <j.goodnow29@gmail.com>
- Denis Ryzhkov <denisr@denisr.com>
- Wilfred Hughes <me@wilfred.me.uk>
- Dmitry Medvinsky <me@dmedvinsky.name>
- Bryce Boe <bbzbryce@gmail.com> (@bboe)
- Colin Dunklau <colin.dunklau@gmail.com> (@cdunklau)
- Bob Carroll <bob.carroll@alum.rit.edu> (@rcarz)
- Hugo Osvaldo Barrera <hugo@osvaldobarrera.com.ar> (@hobarrera)
- Łukasz Langa <lukasz@langa.pl>
- Dave Shawley <daveshawley@gmail.com>
- James Clarke (@jam)
- Kevin Burke <kev@inburke.com>
- Flavio Curella
- David Pursehouse <david.pursehouse@gmail.com> (@dpursehouse)
- Jon Parise (@jparise)
- Alexander Karpinsky (@homm86)
- Marc Schlaich (@schlamar)

- Park Ilsu <daftonshady@gmail.com> (@daftshady)
- Matt Spitz (@mattspitz)
- Vikram Oberoi (@voberoi)
- Can Ibanoglu <can.ibanoglu@gmail.com> (@canibanoglu)
- Thomas Weißschuh <thomas@t-8ch.de> (@t-8ch)
- Jayson Vantuyl <jayson@aggressive.ly>
- Pengfei.X <pengphy@gmail.com>
- Kamil Madac <kamil.madac@gmail.com>
- Michael Becker <mike@beckerfuffle.com> (@beckerfuffle)
- Erik Wickstrom <erik@erikwickstrom.com> (@erikwickstrom)
- (@podshumok)
- Ben Bass (@codedstructure)
- Jonathan Wong <evolutionace@gmail.com> (@ContinuousFunction)
- Martin Jul (@mjul)
- Joe Alcorn (@buttscicles)
- Syed Suhail Ahmed <ssuhail.ahmed93@gmail.com> (@syedsuhail)
- Scott Sadler (@ssadler)
- Arthur Darcet (@arthurdarcet)
- Ulrich Petri (@ulope)
- Muhammad Yasoob Ullah Khalid <yasoob.khld@gmail.com> (@yasoob)
- Paul van der Linden (@pvanderlinden)
- Colin Dickson (@colindickson)
- Smiley Barry (@smiley)
- Shagun Sodhani (@shagunsodhani)
- Robin Linderborg (@vienno)
- Brian Samek (@bsamek)
- Dmitry Dygalo (@Stranger6667)
- piotrjurkiewicz
- Jesse Shapiro <jesse@jesseshapiro.net> (@haikuginger)
- Nate Prewitt <nate.prewitt@gmail.com> (@nateprewitt)
- Maik Himstedt
- Michael Hunsinger
- Brian Bamsch <bbamsch32@gmail.com> (@bbamsch)
- Om Prakash Kumar <omprakash070@gmail.com> (@iamprakashom)
- Philipp Konrad <gardiac2002@gmail.com> (@gardiac2002)
- Hussain Tamboli <hussaintamboli18@gmail.com> (@hussaintamboli)

- Casey Davidson (@davidsoncasey)
- Andrii Soldatenko (@a_soldatenko)
- Moinuddin Quadri <moin18@gmail.com> (@moin18)
- Matt Kohl (@mattkohl)
- Jonathan Vanasco (@jvanasco)
- David Fontenot (@davidfontenot)
- Shmuel Amar (@shmuelamar)
- Gary Wu (@garywu)
- Ryan Pineo (@ryanpineo)
- Ed Morley (@edmorley)
- Matt Liu <liumatt@gmail.com> (@mlcrazy)

There are no more guides. You are now guideless. Good luck.

Python Module Index

r

Index

S

T

U

V