Requests Documentation

Release 1.2.3

Kenneth Reitz

September 24, 2013

CONTENTS

Release v1.2.3. (*Installation*)

Requests is an *Apache2 Licensed* HTTP library, written in Python, for human beings.

Python's standard **urllib2** module provides most of the HTTP capabilities you need, but the API is thoroughly **broken**. It was built for a different time — and a different web. It requires an *enormous* amount of work (even method overrides) to perform the simplest of tasks.

Things shouldn't be this way. Not in Python.

```
>>> 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, without Requests.

Requests takes all of the work out of Python HTTP/1.1 — making your integration with web services seamless. 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, powered by urllib3, which is embedded within Requests.

CHAPTER

ONE

TESTIMONIALS

Her Majesty's Government, Amazon, Google, Twilio, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Twitter, SoundCloud, Kippt, Readability, and Federal US Institutions use Requests internally. It has been downloaded over 3,000,000 times from PyPI.

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 @kennethreitz'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 @kennethreitz's request library. Today has been AWESOME.

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

CHAPTER

TWO

FEATURE SUPPORT

Requests is ready for today's web.

- International Domains and URLs
- Keep-Alive & Connection Pooling
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Basic/Digest Authentication
- Elegant Key/Value Cookies
- Automatic Decompression
- Unicode Response Bodies
- Multipart File Uploads
- Connection Timeouts
- `.netrc` support
- Python 2.6—3.3
- Thread-safe.

CHAPTER

THREE

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.

3.1 Introduction

3.1.1 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.

3.1.2 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.

3.1.3 Requests License

Copyright 2013 Kenneth Reitz

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

3.2 Installation

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

3.2.1 Distribute & Pip

Installing requests is simple with pip:

```
$ pip install requests
```

or, with easy_install:

```
$ easy_install requests
```

But, you really shouldn't do that.

3.2.2 Cheeseshop Mirror

If the Cheeseshop is down, you can also install Requests from one of the mirrors. Crate.io is one of them:

```
$ pip install -i http://simple.crate.io/ requests
```

3.2.3 Get the 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/kennethreitz/requests.git
```

Download the tarball:

```
$ curl -OL https://github.com/kennethreitz/requests/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/kennethreitz/requests/zipball/master
```

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

```
$ python setup.py install
```

3.3 Quickstart

Eager to get started? This page gives a good introduction in how to get started with Requests. This assumes you already have Requests installed. If you do not, head over to the *Installation* section.

First, make sure that:

- Requests is *installed*
- Requests is *up-to-date*

Let's get started with some simple examples.

3.3.1 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://github.com/timeline.json')
```

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")
```

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")
>>> 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.

3.3.2 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, 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
u'http://httpbin.org/get?key2=value2&key1=value1'
```

3.3.3 Response Content

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

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> r.text
'[{"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`.

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.

3.3.4 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 StringIO import StringIO
>>> i = Image.open(StringIO(r.content))
```

3.3.5 JSON Response Content

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

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> 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 401 (Unauthorized), attempting `r.json` raises `ValueError: No JSON object could be decoded`

3.3.6 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://github.com/timeline.json', 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'
```

3.3.7 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 content-type in the previous example:

```
>>> import json
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}
>>> headers = {'content-type': 'application/json'}

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

3.3.8 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"
  },
  ...
}
```

There are many times that you 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))
```

3.3.9 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 explicitly:

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

>>> 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"
  },
  ...
}
```

3.3.10 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 (non-200 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.

3.3.11 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; charset=utf-8'
}
```

The dictionary is special, though: it's made just for HTTP headers. According to RFC 2616, HTTP Headers are case-insensitive.

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

```
>>> r.headers['Content-Type']
'application/json; charset=utf-8'

>>> r.headers.get('content-type')
'application/json; charset=utf-8'
```

If a header doesn't exist in the Response, its value defaults to `None`:

```
>>> r.headers['X-Random']
None
```

3.3.12 Cookies

If a response contains some Cookies, you can get quick access to 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"}}'
```

3.3.13 Redirection and History

Requests will automatically perform location redirection while using the GET and OPTIONS verbs.

GitHub redirects all HTTP requests to HTTPS. We can use the `history` method of the Response object to track redirection. Let's see what Github does:

```
>>> r = requests.get('http://github.com')
>>> r.url
'https://github.com/'
>>> r.status_code
200
>>> r.history
[<Response [301]>]
```

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

If you're using GET or OPTIONS, 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 POST, PUT, PATCH, DELETE or HEAD, you can enable redirection as well:

```
>>> r = requests.post('http://github.com', allow_redirects=True)
>>> r.url
'https://github.com/'
>>> r.history
[<Response [301]>]
```

3.3.14 Timeouts

You can tell requests to stop waiting for a response after a given number of seconds with the `timeout` parameter:

```
>>> 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. (time
```

Note:

`timeout` only effects the connection process itself, not the downloading of the response body.

3.3.15 Errors and Exceptions

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

In the event of the rare invalid HTTP response, Requests will raise an `HTTPError` exception.

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.

3.4 Advanced Usage

This document covers some of Requests more advanced features.

3.4.1 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.

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.

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.

3.4.2 Request and Response Objects

Whenever a call is made to requests.*() you are doing two major things. First, you are constructing a `Request` object which will be sent of 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'}
```

3.4.3 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()
prepped = Request('GET',  # or any other method, 'POST', 'PUT', etc.
                  url,
                  data=data
                  headers=headers
                  # ...
                  ).prepare()
# do something with prepped.body
# do something with prepped.headers
resp = s.send(prepped,
              stream=stream,
              verify=verify,
              proxies=proxies,
              cert=cert,
              timeout=timeout,
              # etc.
              )
print(resp.status_code)
```

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

3.4.4 SSL Cert Verification

Requests can verify SSL certificates for HTTPS requests, just like a web browser. To check a host's SSL certificate, you can use the `verify` argument:

```
>>> requests.get('https://kennethreitz.com', verify=True)
requests.exceptions.SSLError: hostname 'kennethreitz.com' doesn't match either of '*.herokuapp.com',
```

I don't have SSL setup on this domain, so it fails. Excellent. Github does though:

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

You can also pass `verify` the path to a CA_BUNDLE file for private certs. You can also set the `REQUESTS_CA_BUNDLE` environment variable.

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

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

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

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 file's path:

```
>>> requests.get('https://kennethreitz.com', cert=('/path/server.crt', '/path/key'))
<Response [200]>
```

If you specify a wrong path or an invalid cert:

```
>>> requests.get('https://kennethreitz.com', cert='/wrong_path/server.pem')
SSLError: [Errno 336265225] _ssl.c:347: error:140B0009:SSL routines:SSL_CTX_use_PrivateKey_file:PEM
```

3.4.5 Body Content Workflow

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

```
tarball_url = 'https://github.com/kennethreitz/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, or reading from the underlying urllib3 `urllib3.HTTPResponse` at `Response.raw`.

3.4.6 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.

3.4.7 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') as f:
    requests.post('http://some.url/streamed', data=f)
```

3.4.8 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())
```

3.4.9 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):
    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]>
```

3.4.10 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 `requests.auth.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]>
```

3.4.11 Streaming Requests

With `requests.Response.iter_lines()` you can easily iterate over streaming APIs such as the Twitter Streaming API.

To use the Twitter Streaming API to track the keyword "requests":

```
import json
import requests

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

for line in r.iter_lines():

    # filter out keep-alive new lines
    if line:
        print json.loads(line)
```

3.4.12 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 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/",
}
```

3.4.13 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 call the `Response.text` method. Requests will first check for an encoding in the HTTP header, and if none is present, will use charade 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`.

3.4.14 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/kennethreitz/requests/git/commits/a050faf084662f3a
```

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/kennethreitz/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/kennethreitz/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/kennethreitz/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.

3.4.15 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.githu
```

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'}
```

3.4.16 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.

Implementing a Transport Adapter is beyond the scope of this documentation, but a good start would be to subclass the `requests.adapters.BaseAdapter` class.

3.5 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.

3.5.1 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.

3.5.2 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]>
```

3.5.3 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 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

3.5.4 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.

3.5.5 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 `requests.auth.AuthBase` and implement the `__call__()` method. 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.

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

CHAPTER

FOUR

COMMUNITY GUIDE

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

4.1 Frequently Asked Questions

This part of the documentation answers common questions about Requests.

4.1.1 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.

4.1.2 Custom User-Agents?

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

4.1.3 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.

4.1.4 Python 3 Support?

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

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

4.2 Integrations

4.2.1 ScraperWiki

ScraperWiki is an excellent service that allows you to run Python, Ruby, and PHP scraper scripts on the web. Now, Requests v0.6.1 is available to use in your scrapers!

To give it a try, simply:

```
import requests
```

4.2.2 Python for iOS

Requests is built into the wonderful Python for iOS runtime!

To give it a try, simply:

```
import requests
```

4.3 Articles & Talks

- Python for the Web teaches how to use Python to interact with the web, using Requests.
- Daniel Greenfield'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
- French blog post introducing Requests

4.4 Support

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

4.4.1 Send a Tweet

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

4.4.2 File an Issue

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

4.4.3 E-mail

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

4.4.4 IRC

The official Freenode channel for Requests is #python-requests

I'm also available as **kennethreitz** on Freenode.

4.5 Updates

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

4.5.1 GitHub

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

4.5.2 Twitter

I often tweet about new features and releases of Requests.

Follow @kennethreitz for updates.

4.5.3 Mailing List

There's a low-volume mailing list for Requests. To subscribe to the mailing list, send an email to requests@librelist.org.

CHAPTER

FIVE

API DOCUMENTATION

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

5.1 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.

5.1.1 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`. 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`.
- **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': ('filename', fileobj)}) for multipart encoding upload.
- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True if POST/PUT/DELETE redirect following is allowed.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **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.

Usage:

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

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

Parameters

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

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

Parameters

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

`requests.`**`post`**(*url*, *data=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`.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`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.

`requests.`**`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.

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

Parameters

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

Lower-Level Classes

class `requests.`**`Request`**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=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 the request. If a dictionary is provided, form-encoding will take place.
- **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 lovely Charade library (Thanks, Ian!).

`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)

`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.

`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.

`iter_lines`(*chunk_size=512*, *decode_unicode=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.

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

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

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

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

`raw` = None
File-like object representation of response (for advanced usage). Requires that ‘‘stream=True‘ on the request.

`status_code` = None
Integer Code of responded HTTP Status.

`text`
Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

`url` = None
Final URL location of Response.

5.1.2 Request Sessions

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

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

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

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

`cert` = None
SSL certificate default.

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

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

`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.

`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.

`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.

`post` (*url*, *data=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`.
- ****kwargs** – Optional arguments that `request` takes.

`proxies` = None
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) 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.

`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*)
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 or bytes 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** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

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

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

`stream` = None
Stream response content default.

`trust_env` = None
Should we trust the environment?

`verify` = None
SSL Verification default.

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.
- **pool_block** – Whether the connection pool should block for connections.

Usage:

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

`add_headers`(*request*, ***kwargs*)
Add any headers needed by the connection. Currently this adds a Proxy-Authorization header.

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.

`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** – Whether we should actually verify the certificate.
- **cert** – The SSL certificate to verify.

close()

Disposes of any internal state.

Currently, this just closes the PoolManager, which closes 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.

init_poolmanager(*connections*, *maxsize*, *block=False*)

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.

request_url(*request*, *proxies*)

Obtain the url to use when making the final request.

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

This shoudl 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 to proxy URLs.

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** – (optional) The timeout on the request.
- **verify** – (optional) Whether to verify SSL certificates.
- **vert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

Exceptions

exception `requests.exceptions.`**`RequestException`**
There was an ambiguous exception that occurred while handling your request.

exception `requests.exceptions.`**`ConnectionError`**
A Connection error occurred.

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

exception `requests.exceptions.`**`URLRequired`**
A valid URL is required to make a request.

exception `requests.exceptions.`**`TooManyRedirects`**
Too many redirects.

Status Code Lookup

`requests.`**`codes`**(*name=None*)
Dictionary lookup object.

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

>>> requests.codes.teapot
418

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

Cookies

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

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

`requests.utils.`**`cookiejar_from_dict`**(*cookie_dict*, *cookiejar=None*)
Returns a CookieJar from a key/value dictionary.

Parameters **cookie_dict** – Dict of key/values to insert into CookieJar.

`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.

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.

`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.every encodings from `<meta ... charset=XXX>`

3.fall back and replace all unicode characters

Classes

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

`apparent_encoding`
The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

`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)

`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.

`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.

`iter_lines`(*chunk_size=512*, *decode_unicode=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.

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

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

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

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

raw = None
File-like object representation of response (for advanced usage). Requires that ‘‘stream=True‘ on the request.

status_code = None
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

url = None
Final URL location of Response.

class requests.**Request**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=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 the request. If a dictionary is provided, form-encoding will take place.
- **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.**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_auth (*auth*, *url=''*)
Prepares the given HTTP auth data.

prepare_body (*data*, *files*)
Prepares the given HTTP body data.

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

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.**Session**
A Requests session.

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

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

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

cert = None
SSL certificate default.

close()
Closes all adapters and as such the session

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

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.

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.

`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.

`post`(*url*, *data=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`.
- ****kwargs** – Optional arguments that `request` takes.

`proxies` = **None**
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) 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.

`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*)
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 or bytes 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** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True by default.

- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

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

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

`stream` = None
Stream response content default.

`trust_env` = None
Should we trust the environment?

`verify` = None
SSL Verification default.

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.
- **pool_block** – Whether the connection pool should block for connections.

Usage:

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

`add_headers`(*request*, ***kwargs*)
Add any headers needed by the connection. Currently this adds a Proxy-Authorization header.

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.

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** – Whether we should actually verify the certificate.
- **cert** – The SSL certificate to verify.

close()

Disposes of any internal state.

Currently, this just closes the PoolManager, which closes 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.

init_poolmanager(*connections*, *maxsize*, *block=False*)

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.

request_url(*request*, *proxies*)

Obtain the url to use when making the final request.

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

This shoudl 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 to proxy URLs.

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** – (optional) The timeout on the request.
- **verify** – (optional) Whether to verify SSL certificates.
- **vert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

5.1.3 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)
r.raw.read(10)
```

- 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

# these two lines enable debugging at httplib level (requests->urllib3->httplib)
# you will see the REQUEST, including HEADERS and DATA, and RESPONSE with HEADERS but without DA
# the only thing missing will be the response.body which is not logged.
import httplib
httplib.HTTPConnection.debuglevel = 1

logging.basicConfig() # you need to initialize logging, otherwise you will not see anything from
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.

CHAPTER

SIX

CONTRIBUTOR GUIDE

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

6.1 Development Philosophy

Requests is an open but opinionated library, created by an open but opinionated developer.

6.1.1 Benevolent Dictator

Kenneth Reitz is the BDFL. He has final say in any decision related to Requests.

6.1.2 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.

6.1.3 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.

6.1.4 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.

Requests just reached v1.0.0. This huge milestone marks a major step in the right direction.

6.1.5 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 bugfixes. PyPI (and its mirrors) and GitHub are the official distribution sources; alternatives are not supported by the requests project.

6.2 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.

6.2.1 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.

6.2.2 Development Dependencies

You'll need to install py.test in order to run the Requests' test suite:

```
$ pip install -r requirements.txt
$ invoke test
py.test
platform darwin -- Python 2.7.3 -- pytest-2.3.4
collected 25 items

test_requests.py .........................
25 passed in 3.50 seconds
```

6.2.3 Runtime Environments

Requests currently supports the following versions of Python:

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

Support for Python 3.1 and 3.2 may be dropped at any time.

Google App Engine will never be officially supported. Pull requests for compatibility will be accepted, as long as they don't complicate the codebase.

6.2.4 Are you crazy?

- SPDY support would be awesome. No C extensions.

6.3 Authors

Requests is written and maintained by Kenneth Reitz and various contributors:

6.3.1 Development Lead

- Kenneth Reitz <me@kennethreitz.com>

6.3.2 Urllib3

- Andrey Petrov <andrey.petrov@shazow.net>

6.3.3 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)
- Cory Benfield (Lukasa)
- 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>
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24
- 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> @dontYetKnow
- Dmitry Medvinsky <me@dmedvinsky.name>
- Bryce Boe <bbzbryce@gmail.com> @bboe
- Colin Dunklau <colin.dunklau@gmail.com> @cdunklau
- Hugo Osvaldo Barrera <hugo@osvaldobarrera.com.ar> @hobarrera
- Łukasz Langa <lukasz@langa.pl> @llanga
- Dave Shawley <daveshawley@gmail.com>

PYTHON MODULE INDEX