Requests Documentation

Release 1.2.0

Kenneth Reitz

07.04.2016

Inhaltsverzeichnis

Release v1.2.0. (*Installation*)

Requests ist eine *Apache2 lizensierte* HTTP Bibliothek, geschrieben in Python, für die einfache Nutzung durch Menschen.

Das **urllib2** Standard-Modul in Python bietet Ihnen die meisten HTTP-Funktionalitäten, die Sie benötigen, aber die API ist **definitiv kaputt**. Sie wurde für eine andere Zeit geschrieben - und ein anderes Web. Sie erfordert eine *enorme* Menge an Arbeit (inkl. dem Überschreiben von Methoden), um auch nur die einfachsten Dinge zu erledigen.

So sollten die Dinge nicht laufen. Nicht 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, ...}
```

Der gleiche Code, aber ohne Requests.

Requests nimmt Ihnen die ganze harte Arbeit für HTTP/1.1 in Python ab - und macht damit die Integration von Webdiensten schnell und einfach. Keine Notwendigkeit, manuell query strings zu Ihren URLs hinzu zu fügen oder die Daten für ein POST erst in form-encoding umzuwandeln. Keep-Alive und das Pooling von Verbindungen laufen 100% automatisch ab, erledigt von urllib3, die in Requests eingebettet ist.

KAPITEL 1

Stimmen zu Requests

Die Regierung Ihrer Majestät, Amazon, Google, Twilio, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Twitter, SoundCloud, Kippt, Readability, and US-Bundesbehörden benutzen Requests in Ihren internen Anwendungen. Requests wurde mehr als 2 Millionen mal von PyPI herunter geladen.

Armin Ronacher Requests ist das perfekte Beispiel dafür, wie elegant eine API mit dem richtigen Grad an Abstraktion sein kann.

Matt DeBoard Ich werde mir @kennethreitz's Python requests Modul auf meinen Körper tätowieren lassen. Das ganze Ding.

Daniel Greenfeld Ich habe eine 1200 Zeilen Bibliothek aus Spaghetticode überflüssig gemacht und durch 10 Zeilen Code ersetzt, dank @kennethreitz's request Bibliothek. Heute war ein genialer Tag.

Kenny Meyers Python HTTP: Im Zweifel, ach was, auch ohne Zweifel, benutzen Sie Requests. Elegant, einfach, Pythonisch.

KAPITEL 2

Features

- Internationale Domains und URLs
- Keep-Alive & Connection Pooling
- Sessions mit persistierten Cookies
- SSL-Verifizierung wie im Browser
- Basic/Digest Authentifizierung
- Elegante Schlüssel/Wert Cookies
- Automatische Dekomprimierung
- Unicode Antwortdaten
- Hochladen von mehrteiligen Dateien
- Verbindungs-Timeouts
- `.netrc` Unterstützung
- Python 2.6-3.3
- Thread-Sicherheit

KAPITEL 3

Benutzeranleitung

Dieser Teil der Dokumentation, der zum Großteil aus Beschreibungsprosa besteht, beginnt mit einigen Hintergrundinformationen über Requests. Danach liegt der Schwerpunkt auf einer Schritt-für-Schritt Anleitung, um den maximalen Nutzen aus Requests zu ziehen.

3.1 Einführung

3.1.1 Philosophie

Requests wurde in Anlehnung an einige **PEP 20** Idiome entwickelt.

1. Schön ist besser als häßlich.
2. Explizit ist besser als implizit.
3. Einfach ist besser als komplex.
4. Komplex ist besser als kompliziert.
5. Lesbarkeit zählt.

Alle Beiträge zu Requests sollen diese wichtigen Regeln beachten.

3.1.2 Apache2 Lizenz

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 wird unter den Bedingungen der Apache2 License bereit gestellt.

3.1.3 Requests Lizenz

3.2 Installation

Dieser Teil der Dokumentation behandelt die Installation von Requests. Der erste Schritt zur Benutzung jeder Software ist es, diese korrekt zu installieren.

3.2.1 Distribute & Pip

Die Installation von Requests funktioniert sehr einfach über pip:

```
$ pip install requests
```

order mit easy_install:

```
$ easy_install requests
```

Aber das sollten Sie wirklich nicht machen.

3.2.2 Cheeseshop Mirror

Falls der Cheeseshop offline ist, können Sie Requests von einem der Mirrors installieren. Crate.io ist einer von ihnen:

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

3.2.3 Holen Sie sich den Code

Requests wird aktiv auf Guthub entwickelt, wo der Code immer verfügbar ist.

Sie können entweder das öffentliche Repository clonen:

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

Laden Sie den tarball herunter:

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

Oder sie laden den zipball:

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

Sobald Sie eine Kopie des Quellcodes haben, können Sie diesen in Ihr Python-Package einbetten oder einfach zu Ihren site-packages hinzufügen:

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

3.3 Schnellstart

Sie wollen loslegen? Diese Seite bietet eine gute Einführung, wie Sie mit Requests starten. Wir nehmen an, Sie haben Requests bereits installiert. Falls nicht, finden Sie im Abschnitt *Installation* die notwendigen Informationen dazu.

Zuerst stellen Sie bitte sicher, dass:

- Requests *installiert* ist.
- Requests *auf dem akutellen Stand* ist.

Lassen Sie und mit einigen einfachen Beispielen beginnen.

3.3.1 Eine HTTP-Anforderung senden

Es ist sehr einfach, eine Anforderung mit Requests zu senden.

Beginnen Sie mit dem Import des Requests Moduls:

```
>>> import requests
```

Lassen Sie uns nun versuchen, eine Webseite zu laden. Für dieses Beispiel werden wir die öffnentliche Zeitleiste von Github lesen:

```
>>> r = requests.get('https://github.com/timeline.json')
```

Jetzt haben wir ein Objekt `Response` mit dem Namen `r`. Wir können alle benötigten Informationen von diesem Objekt lesen.

Die einfache API von Requests bedeutet, dass alle Formen von HTTP-Anfragen genau so leicht zu erfassen sind. Zum Beispiel führen Sie ein HTTP POST wie folgt durch:

```
>>> r = requests.post("http://httpbin.org/post")
```

Nett, nicht wahr? Wie sieht es mit den anderen HTTP Anforderungen aus: PUT, DELETE, HEAD und OPTIONS? Die sind alle ebenso einfach:

```
>>> 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")
```

Das ist alle schön und gut, stellt aber nur den Anfang von dem dar, was Requests leistet.

3.3.2 Parameter in URLs übergeben

In vielen Fällen werden Sie Daten in irgendeiner Form im Query-String der URL senden wollen. Falls Sie die URL per Hand zusammenstellen, würden diese Daten als Schlüssel/Wert-Paare nach einem Fragezeichen in der URL übergeben werden, z.B. `httpbin.org/get?schluessel=wert`. Requests erlaubt es Ihnen, diese Daten in einem Dictionary zu übergeben, in dem Sie das `params` Schlüsselwortargument benutzen. Als Beispiel nehmen wir an, dass Sie `schluessel1=wert1` und `schluessel2=wert2` an `httpbin.org/get` übergeben wollen. Dazu benutzen Sie den folgenden Code:

```
>>> payload = {'schluessel1': 'wert1', 'schluessel2': 'wert2'}
>>> r = requests.get("http://httpbin.org/get", params=payload)
```

Wenn Sie die erzeugte URL ausgeben lassen, sehen Sie, dass diese korrekt erzeugt wurde:

```
>>> print r.url
u'http://httpbin.org/get?schluessel2=wert2&schluessel1=wert1'
```

3.3.3 Daten aus der Antwort

Wir können den Inhalt der Antwort des Servers lesen. Betrachten wir noch einmal die Zeitleiste von Github:

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

Requests dekodiert automatisch den Inhalt der Antwort vom Server. Die meisten Unicode-Zeichensätze werden problemlos erkannt.

Wenn Sie eine Anfrage absenden, versucht Requests anhand der Header die Kodierung der Daten in der Antwort zu ermitteln. Die Kodierung, die Requests ermittelt hat, wird verwendet, wenn Sie auf ‘‘r.text‘‘zugreifen. Sie können herausfinden, welche Zeichenkodierung Requests verwendet (und diese auch ändern), in dem Sie die Eigenschaft ‘‘r.encoding‘‘verwenden:

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

Falls Sie die Zeichenkodierung ändern, benutzt Requests diese Zuordnung, sobald Sie auf ‘‘r.text‘‘zugreifen.

Requests wird auch benutzerdefinierte Zeichenkodierungen benutzen, wenn Sie diese benötigen. Wenn Sie Ihre eigene Kodierung erstellt und im Modul `codecs` registriert haben, können Sie einfach den Namen der Kodierung als Wert für `r.encoding` benutzen und Requests übernimmt die Dekodierung für Sie.

3.3.4 Binäre Antwortdaten

Sie können auch byteweise auf die Serverantwort zugreifen, um nicht-Text Anfragen zu erledigen:

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

Die `gzip` und `deflate` transfer-encodings werden automatisch für Sie dekodiert.

Um zum Beispiel ein Bild aus den von einer Anfrage gelieferten Binärdaten zu erzeugen, können Sie den folgenden Code benutzen:

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

3.3.5 JSON-basierte Antwortdaten

Es gibt auch einen eingebauten JSON Decoder, falls Sie mit JSON Daten arbeiten:

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

Sollte die Dekodierung der JSON-Daten fehlschlagen, erzeugt `r.json` eine Exception. Erhalten Sie beispielsweise einen HTTP-Statuscode 401 (Unauthorized) zurück, wird ein Zugriff auf `r.json` einen `ValueError: No JSON object could be decoded` auslösen.

3.3.6 Rohdaten der Antwort

Für den seltenen Fall, dass Sie auf die unverarbeiteten Daten des raw sockets der Antwort zugreifen wollen, können Sie `r.raw` bneutzen. Falls Sie das wollen, stellen Sie sicher, dass Sie `stream=True` in Ihrer Anfrage setzen. Nachdem das erledigt ist, können Sie folgenden Code benutzen:

```
>>> 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 Benutzerdefinierte Header

Wenn Sie bestimmte Header zu der HTTP-Anfrage hinzufügen wollen, übergeben Sie einfach ein Dictionary über den `headers` Parameter.

Wir haben zum Beispiel im letzten Code-Beispiel nicht den gewünschten Content-Type angegeben:

```
>>> 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 Komplexere POST Anfragen

Typischerweise möchten Sie einige formular-kodierte Daten senden - so wie in einem HTML-Formular. Um dies zu erledigen, übergeben Sie einfach ein Dictionary an das *data* Argument. Dieses Dictionary mit den Daten wird automatisch formular-kodiert, wenn die Anfrage ausgeführt wird:

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

In vielen Fällen werden Sie auch Daten senden wollen, die nicht formular-kodiert sind. Wenn Sie an Stelle eines Dictionary einen `string` übergeben, werden diese Daten direkt übertragen.

So akzeptiert die GitHub API v3 beispielsweise JSON-kodierte Strings für POST/PATCH-Daten:

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

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

3.3.9 POST mit Multipart-Dateien

Requests macht das Hochladen von Dateien in Multipart-Kodierung einfach:

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

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<ganz...geheime...binaere...daten>"
  },
  ...
}
```

Sie können den Dateinamen explizit angeben:

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

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<ganz...geheime...binaere...daten>"
  },
  ...
}
```

Falls Sie das wollen, können Sie Strings als Dateien hochladen:

```
>>> url = 'http://httpbin.org/post'
.. >>> files = {'file': ('report.csv', 'einige,daten,zum,senden\\nnoch,eine,zweite,zeile\\n')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "einige,daten,zum,senden\\nnoch,eine,zweite,zeile\\n"
  },
  ...
}
```

3.3.10 Status Codes der Antwort

Wir können den Statuscode der Antwort prüfen:

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

Requests hat auch ein eingebautes Objekt zum Nachschlagen von Statuscodes:

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

Falls wir eine ungültige Anfrage (eine Antwort mit einem Status ungleich 200) ausgeführt haben, können wir über `Response.raise_for_status()` eine Exception werfen:

```
>>> 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
```

Aber, da unser `status_code` für `r` ein `200` war, erhalten wir beim Aufruf von `raise_for_status()`

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

Alles ist gut.

3.3.11 Header der Antwort

Wir können die Header der Serverantwort als Python Dictionary lesen:

```
>>> 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'
}
```

Dieses Dictionary ist aber ein spezielles: es ist nur für HTTP-Header gemacht. Nach dem RFC 2616 sind HTTP Header nicht abhängig von Groß- oder Kleinschreibung.

Daher können wir in beliebiger Schreibweise auf die Header zugreifen:

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

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

Falls ein Header nicht in der Antwort enthalten ist, wird als Wert der Default von `None` geliefert:

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

3.3.12 Cookies

Falls eine Antwort ein oder mehrere Cookies enthält, können Sie einfach und schnell darauf zugreifen:

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

>>> r.cookies['beispiel_cookie_name']
'beispiel_cookie_wert'
```

Um eigene Cookies an den Server zu senden, können Sie den `cookies` Parameter benutzen:

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_sind='keksig')

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

3.3.13 Redirections und Verlauf

Requets führt automatisch Weiterleitungen aus, wenn eines der beiden Verben GET oder OPTIONS benutzt wird.

GitHub zum Beispiel leitet alle HTTP-Anfragen auf HTTPS um. Wir können die `history` Methode des Antwortobjekts benutzen, um diese Weiterleitungen zu verfolgen. Sehen wir und an, was GitHub macht:

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

Die `Response.history` Liste enthält eine Liste der `Request` Objekte, die erzeugt wurden, um die Anfrage abzuschließen. Diese Liste ist vom ältesten zum neuesten Objekt sortiert.

Falls Sie GET oder OPTIONS benutzen, können Sie die Handhabung von Weiterleitungen mit Hilfe des `allow_redirects` Parameters kontrollieren:

```
>>> r = requests.get('http://github.com', allow_redirects=False)
>>> r.status_code
301
>>> r.history
[]
```

Falls Sie POST, PUT, PATCH, DELETE oder HEAD benutzen, können Sie bei Bedarf damit ebenso die Behandlung von Weiterleitungen aktivieren:

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

3.3.14 Timeouts

Sie können Anfragen nach einer bestimmten Anzahl von Sekunden anweisen, nicht länger zu warten, in dem Sie den `timeout` Parameter benutzen:

```
>>> 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
```

Hinweis:

`timeout` betrifft nur den Prozess des Verbindungsaufbaus, nicht das Herunterladen der Antwort selbst.

3.3.15 Fehler und Exceptions

Im Falle eines Netzwerkproblems (z.B. DNS-Fehler, abgewiesene Verbindung, etc.) löst Requests einen `ConnectionError` Ausnahmefehler aus.

Im Fall einer (seltenen) ungültigen HTTP Antwort löst Requests einen `HTTPError` Ausnahmefehler aus.

Falls eine Anfrage eine Zeitüberschreitung auslöst, wird ein `Timeout` Ausnahmefehler ausgelöst.

Falls eine Anfrage die konfigurierte maximale Anzahl von Weiterleitungen überschreitet, wird ein `TooManyRedirects` Ausnahmefehler ausgelöst.

Alle Ausnahmefehler, die Requests explizit auslöst, erben von `requests.exceptions.RequestException`.

Bereit für mehr? Dann sehen Sie sich den Abschnitt *Weitergehende Informationen* an.

3.4 Fortgeschrittene Nutzung

Dieses Dokument behandelt einige der fortgeschrittenen Features von Requests.

3.4.1 Sesssion-Objekte

Das Session-Objekt erlaubt das Persistieren einiger Parameter über einzelne Anfragen hinweg. Es persistiert auch Cookies für die Dauer aller Anfragen einer Session-Instanz.

Ein Session-Objekt besitzt alle Methoden der Haupt-API von Requests (get,put, post, delete, head, options).

Lassen Sie und mit Cookies über mehrere Anfragen hinweg arbeiten:

```
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 können auch dazu benutzt werden, um für die Methoden der Anfrage Standardwerte vorzuhalten. Dies geschieht durch die Angaben von Eigenschaften eines Session-Objekts:

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

# sowohl 'x-test' als auch 'x-test2' werden als Header übermittelt
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

Alle Dictionaries, die Sie einer Anfragemethode übergeben, werden mit den Werten der Session zusammengeführt, bevor sie gesendet werden. Falls an beiden Stellen Angaben gemacht werden, überschreiben die Parameter auf Methodenebene die des Session-Objekts.

Einen Wert aus einem Dictionary-Parameter entfernen

Unter Umständen möchten Sie einzelne Werte aus den Session-Daten nicht übermitteln. Dies können Sie dadurch erreichen, dass Sie den Schlüssel im Dictionary beim Aufruf der Methode angeben, aber als Wert ‘‘None‘‘angeben. Dieser Schlüssel wird dann automatisch ausgelassen.

Alle Werte, die innerhalb einer Session verwendet werden, stehen Ihnen direkt zur Verfügug. Sehen Sie dazu unter *Session API* nach, um mehr zu erfahren.

3.4.2 Objekte für Anfrage und Antwort

Wann immer ein Aufruf von requests.*() erfolgt, passieren zwei Dinge. Zuerst erzeugen Sie damit ein `Request` Objekt, dass an einen Server geschickt wird, um eine Anfrage auszuführen oder eine Ressource zu lesen. Zweitens wird, sobald Requests eine Antwort vom Server erhält, ein `Response` Objekt erzeugt. Dieses `Response` Objekt enthält alle vom Server gelieferten Informationen sowie das ursprünglich erzeugte `Request` Objekt. Hier ist eine einfaxche Anfrage, um einige sehr wichtige Daten aus der Wikipedia zu lesen:

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

Wenn Sie wissen wollen, welche Header der Server an uns zurück geschickt hat, tun Sie folgendes:

```
>>> 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'}
```

Wollen Sie dagegen wissen, welche Header wir ursprünglich an den Server gesendet haben, greifen wir eindach auf die Anfrage und darin auf die Header von `request` zu:

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

3.4.3 Prepared Requests

Wann immer SIe ein `Response` Objekt von einem API-Aufruf oder einem Session-Aufruf zurück erhalten, handelt es sich bei der `request` Eigenschaft tatsächlich um die `PreparedRequest` Eigenschaft, die benutzt wurde. In

einigen Fällen möchten Sie vielleicht zusätzliche Bearbeitungen am Body oder den Headern (oder etwas anderem) vornehmen, bevor Sie die Anfrage absenden. Der einfache Weg dazu ist folgender:

```
from requests import Request, Session

s = Session()
prepped = Request('GET',  # oder irgendeine andere Methode, 'POST', 'PUT', etc.
                  url,
                  data=data
                  headers=headers
                  # ...
                  ).prepare()
# führen Sie mit prepped.body eine Aktion durch
# oder
# führen Sie eine Aktion mit prepped.headers durch
resp = s.send(prepped,
              stream=stream,
              verify=verify,
              proxies=proxies,
              cert=cert,
              timeout=timeout,
              # etc.
              )
print(resp.status_code)
```

Nachdem Sie nichts Besonderes mit dem `Request` Objekt gemacht haben, bereiten Sie das sofort durch den Aufruf von prepare() vor und veändern das `PreparedRequest` Objekt. Dieses senden Sie dann mit den anderen Parametern, die Sie auch an `requests.*` oder an `Session.*` gesendet hätten.

3.4.4 Überprüfen von SSL-Zertifikaten

Requests kann SSL-Zertifikate für HTTPS-Anfragen überprüfen, genau wie ein Web Browser. Um das SSL Zertifikat eines Hosts zu überprüfen, können Sie den `verify` Parameter benutzen:

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

Ich habe SSL für diese Domain nicht aktiviert, deshalb schlägt die Überprüfung fehl. Exzellent. GitHub dagegen hat SSL aktiviert:

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

Sie können auch anstelle von `True` den Pfad zu einer CA_BUNDLE Datei für private Zertifikate übergeben. Ebenso können Sie die `REQUESTS_CA_BUNDLE` Umgebungsvariable setzen.

Requests kann auch die Überprüfung des SSL Zertifikates ignorieren, wenn Sie `verify` auf `False` setzen.

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

Als Standard steht `verfiy` auf `True` und kann nur für Hostzertifikate verwendet werden.

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]>
```

Falls Sie einen ungültigen Pfad oder ein ungültiges Zertifikat angeben:

```
>>> 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 Workflow für Antwortdaten

Standardmäßig wird, wenn Sie eine Anfrage ausführen, der Inhalt (body) der Antwort sofort herunter geladen. Sie können dieses Verhalten überschreiben und das Herunterladen der Antwort verzögern, bis Sie auf die `Response.content` Eigenschaft zugreifen. Benutzen Sie dazu den `stream` Parameter:

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

Zu diesem Zeitpunkt wurden nur die Header der Antwort herunter geladen und die Verbindung bleibt offen. Damit wird und erlaubt, das Lesen des Antwortsinhalts optional zu gestalten:

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

Sie können den Ablauf durch die Methoden `Response.iter_content` und `Response.iter_lines` weiter kontrollieren oder indem Sie von der darunter liegenden urllib3-Klasse `urllib3.HTTPResponse` die Eigenschaft `Response.raw` lesen.

3.4.6 Keep-Alive

Gute Nachrichten - dank der urllib3 funktioniert keep-alive 100% automatisch innerhalb einer Session. Jede Anfrage, die Sie innerhalb einer Session ausführen, wird automatisch die jeweilige Verbindung wieder verwenden!

Bitte beachten Sie, dass Verbindungen nur dann an den Pool zurück gegeben werden, nachdem alle Daten der Antwort gelesen wurden. Stellen Sie sicher, dass entweder `stream` auf `False` gesetzt wurde oder Sie die `content` Eigenschaft des `Response` Objekts gelesen haben.

3.4.7 Streaming Uploads

Requests unterstützt Streaming bei Uploads, damit Sie in der Lage sind, große Streams oder Dateien zu senden, ohne diese erst in den Speicher laden zu müssen. Um den Upload zu streamen, geben Sie für die Nutzdaten einfach ein Dateiobjekt an:

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

3.4.8 Paketierte Anforderungen (*chunked encoding*)

Requests unterstützt auch die paketierte Kodierung für die Übertragung (sog. *chunked encoding*) für ausgehende und ankommende Anfragen. Um eine Anfrage in Datenpakete zerteilt zu übertragen, geben Sie einfach einen Generator (oder einen Iterator ohne Länge) für die Daten an:

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

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

3.4.9 Event Hooks

Reqeusts hat ein System für Event Hooks, das Sie benutzen können, um sich in den Anfrageprozess einzuhängen oder Ereignisse zu signalisieren.

Verfügbare Hooks:

response: Die Antwort auf eine Anforderung.

Sie können eine Funktion für einen Hook auf Anfragebasis zuweisen, in dem Sie ein `{hook_name: callback_function}` Dictionary an den `hooks` Parameter übergeben:

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

Diese `callback_function` erhält ein Datenpaket als erstes Argument.

```
def print_url(r):
    print(r.url)
```

Falls während der Ausführung des Callbacks ein Fehler passiert, erhalten Sie eine Warnung.

Wenn die Callback-Funktion einen Wert zurück liefert, wird angenommen, dass dieser die Datenersetzen soll, die übergeben wurden. Wenn die Funktion nichts zurück liefert, wird auch nichts verändert.

Lassen Sie uns einige Argumente der Anfrage-Methode zur Laufzeit ausgeben:

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

3.4.10 Benutzerdefinierte Authentifizierung

Reqeusts erlaubt es Ihnen, Ihren eigenen Authentifizierungsmechanismus anzugeben.

Jedes Callable, das als das `auth` Argument einer Anfrage übergeben wird, hat die Möglichkeit, die Anfrage vor der Weiterleitung zu modifizieren.

Implementierungen für eine Authentifizierung sind Unterklassen von `requests.auth.AuthBase` und einfach zu definieren. Reqeusts bietet zwei Implementierungen üblicher Authentifizierungs-Schemata in `requests.auth`: `HTTPBasicAuth` and `HTTPDigestAuth`.

Nehmen wir an, dass wir einen Webservice haben, der nur dann reagiert, wenn der `X-Pizza` Header auf einen Wert mit einer Kennung gesetzt wurde. Unwahrscheinlich, aber für das Beispiel nehmen wir das einfach mal an.

```
from requests.auth import AuthBase

class PizzaAuth(AuthBase):
    """Bindet HTTP Pizza Authentifizierung an das angegebene Anfrageobjekt."""
    def __init__(self, username):
        # hier erfolgt die vorbereitung für auth-relevante daten
        self.username = username

    def __call__(self, r):
        # wir verändern die anfrage und liefern diese zurück
        r.headers['X-Pizza'] = self.username
        return r
```

Jetzt können wir unsere Anfrage mit der Pizza-Authentifizierung durchführen:

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

3.4.11 Streaming mit Requests

Mit `requests.Response.iter_lines()` können Sie einfach über Streaming APIs wie z.B. die Twitter Streaming API iterieren.

Sagen wir der Twitter Streaming API, dass wir das Schlüsselwort "requests" verfolgen wollen:

```
import requests
import json

r = requests.post('https://stream.twitter.com/1/statuses/filter.json',
    data={'track': 'requests'}, auth=('username', 'password'), stream=True)

for line in r.iter_lines():
    if line: # keep-alive zeilen ausfiltern
        print json.loads(line)
```

3.4.12 Proxies

Falls Sie einen Proxy benutzen müssen, können Sie individuelle Anfragen mit dem `proxies` Argument ausstatten:

```
import requests

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

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

Sie können Proxies auch über die Umgebungsvariablen `HTTP_PROXY` und `HTTPS_PROXY` konfigurieren.

```
$ 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")
```

Um HTTP Basic Auth mit ihrem Proxy zu benutzen, verwenden Sie die übliche *http://user:password@host/* Syntax:

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

3.4.13 Standardkonformität

Es ist die Absicht des Requests-Projektes, alle relevanten Spezifikationen und RFCs zu beachten, wo diese Konformität keine Schwierigkeiten für die Benutzer bedeutet. Dieser Fokus auf die Beachtung der Standards kann zu verhalten führen, dass für nicht mit diesen Standards und RFCs vertrauten Benutzern auf den ersten Blick ungewöhnlich aussieht.

Zeichenkodierung

Wenn Sie eine Antwort vom Server erhalten, versucht Requests das für die Dekodierung der Antwortdaten nötige Encoding zu erschließen, wenn Sie die `Response.text` Methode aufrufen. Requests prüft zuerst, ob in den HTTP Headern eine Zeichenkodierung angegeben ist. Falls kein Encoding im Header vorhanden ist, benutzt Requests charade für einen Versuch, das Encoding zu erschließen.

Der einzige Fall, bei dem Requests nicht versuchen wird, die Zeichenkodierung zu erraten, ist der, dass keine explizite Angabe des Encodings im Header vorhanden ist **und** dass der `Content-Type` Header den Inhalt `text` besitzt. In diesem Fall gibt der RFC 2616 an, dass der Standardzeichensatz `ISO-8859-1` sein muss. Requests folgt in diesem Fall der RFC-Spezifikation. Falls Sie ein anderes Encoding benötigen, können Sie manuell die Eigenschaft `Response.encoding` setzen. Alternativ können Sie auch den unverarbeiteten (raw) `Response.content` benutzen.

3.4.14 HTTP Verben

Requests erlaubt den Zugriff auf fast alle HTTP Verben: GET, OPTIONS, HEAD, POST, PUT, PATCH und DELETE. Der folgende Abschnitt bietet Ihnen detaillierte Beispiele, wie diese Verben in Requests benutzt werden. Als Beispiel dient uns die GitHub API.

Wir beginnen mit dem am meisten verwendeten Verb: GET. HTTP GET ist eine idempotente Methode, die eine Ressource von einer angegebenen URL liest. Daher sollten Sie dieses Verb benutzen, wenn Sie Daten von einer URL im Web lesen wollen. Ein Beispiel wäre der Versuch, Informationen über einen bestimmten Commit in GitHub zu erhalten. Nehmen wir an, wir wollten den Commit *à050faf‘* aus dem Requests-Repository liesen. Dies erreichen Sie mit dem folgenden Code:

```
>>> import requests
>>> r = requests.get('https://api.github.com/repos/kennethreitz/requests/git/commits/a050faf084662f3a
```

Wir sollten sicherstellen, dass GitHub korrekt geantwortet hat. Falls dies der Fall ist, wollen wir herausfinden, wie der Inhalt der Antwort aussieht. Dies geht so:

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

GitHub liefert also JSON. Das ist großartig, denn wir können die `r.json` Methode benutzen, um die Antwort in Python-Objekte zu zerlegen.

```
>>> 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 weit, so einfach. Lassen Sie uns die GitHub API etwas genauer betrachten. Wir könnten dazu die Dokumentation lesen, aber es macht mehr Spaß, wenn wir das stattdessen mit Requests tun. Wir können dazu das OPTIONS Verb benutzen, um zu sehen, welche HTTP Methoden von der gerade benutzten URL unterstützt werden.

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

He! Was? Das ist nicht hilfreich! Wie sich heraus stellt, implementiert GitHub, wie viele andere Dienste, die im WEB eine API anbieten, die OPTIONS-Methode nicht wirklich. Das ist etwas ärgerlich, denm jetzt müssen wir doch

die langweilige Dokumentation lesen. Würde GitHub OPTIONS korrekt implementiert haben, dann könnte man mit diesem Verb alle erlaubten HTTP Methoden für die URL zurück erhalten, wie zum Beispiel so:

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

Nachdem wir die Dokumentation gelesen haben, sehen wir, dass die einzige andere Methode, die für Commits erlaubt wird, ein POST ist. Dies erzeugt einen neuen Commit. Da wir für unsere Beispiele bisher das Requests-Repository benutzt haben, wäre es vielleicht besser, nicht einfach wilde Änderungen per POST abzusenden. Lassen Sie und daher etwas mit dem Ticket-Feature von GitHub spielen.

Diese Dokumentation wurde als Antwort auf Ticket #482 hinzugefügt. Da wir deshalb davon ausgehen können, dass es dieses Ticket gibt, werden wir es in den Beispielen benutzen. Starten wir damit, dass wir das Ticket lesen.

```
>>> 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, wir haben drei Kommentare zu Ticket #482. Sehen wir und den letzten an.

```
>>> 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
```

Der Abschnitt “advanced”? Na dann schreiben wir doch einen Kommentar, dass wir uns gleich auf diese Aufgabe stürzen. Aber wer ist der Kerl überhaupt? ;-)

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

OK, sagen wir diesem Kenneth, dass wir los legen. Nach der Dokumentation der GitHub API können wir dies tun, indem wir mit einem POST einen Kommentar hinzufügen. Dann machen wir das auch.

```
>>> body = json.dumps({u"body": u"Klingt großartig! Ich mach mich gleich daran!"})
>>> url = u"https://api.github.com/repos/kennethreitz/requests/issues/482/comments"
>>> r = requests.post(url=url, data=body)
>>> r.status_code
404
```

Oha, das ist merkwürdig! Wahrscheinlich müssen wir und anmelden. Das wir bestimmt schwierig, nicht? Falsch. Requests macht es uns sehr einfach, verschiedene Formen der Authentifizierung zu benutzen, unter anderem die sehr verbreitete *Basic Authentication*.

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

Perfekt. Oh nein, doch nicht! Bevor wir die Dokumentation schreiben, muss zuerst die Katze gefüttert werden, das dauert eine Weile. Könnten wir doch nur den Kommentar bearbeiten! Glücklicherweise erlaubt uns die GitHub API ein anderes HTTP verb zu benutzen: PATCH. Verwenden wir also PATCH, um den Kommentar zu ändern.

```
>>> print content[u"id"]
5804413
>>> body = json.dumps({u"body": u"Klingt gut! Ich mache mich dran, sobald ich die Katze gefüttert hab
>>> 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
```

Exzellent. Jetzt quälen wir diesen Kenneth etwas und bringen ihn ins Schwitzen, weil wir ihm nicht sagen, dass wir an der Dokumentation arbeiten. Dazu löschen wir diesen Kommentar wieder. GitHub lässt und Kommentare durch das sehr passend benannte Verb DELETE löschen. Gut, werden wir den Kommentar los ...

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

Nochmal exzellent. Alles weg. Alles, was ich jetzt noch möchte, ist eine Auskunft, wie viel von meiner Änderungsrate ich verbraucht habe. Lassen Sie uns das heraus finden. GitHub sendet diese Informationen in einem header, daher werde ich, anstatt die komplette Seite zu laden, nur über das Verb HEAD die Header lesen.

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

Sehr gut. Zeit, noch mehr Python Code zu schreiben, der die GitHub APU auf alle erdenklichen Arten missbraucht; und das noch 4995 mal in der nächsten Stunde.

3.4.15 Link Header

Viele HTTP APIs benutzen sogenannte Links Header. Diese machen APIs leichter verstehbar und auch leichter zu erforschen.

GitHub beispielsweise benutzt solche Link Header für die Pagination, (das seitenweise Blättern) in seiner API:

```
>>> 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 analysiert diese Header automatisch und bietet sie in leicht erreichbarer Form an:

```
>>> 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-Adapter

Mit dem Erreichen der Version 1.0.0 wurde Requests auf ein modulares internes Design umgestellt. Mit ein Grund für diese Umstellung war die Implementierung von Transportadaptern, die ursprünglich hier beschrieben wurden. Transportadapter bieten einen Mechanismus zu Definition von Interaktionsmethoden für einen HTTP-basierten Dienst. Im speziellen erlauben sie die Konfiguration auf Dienstebasis.

Requests wird mit einem einzelnen Transportadapter ausgeliefert, dem *HTTPAdapter*. Dieser Adapter bietet die Standardinteraktion mit HTTP und HTTPs über die mächtige Bibliothek urllib3. Immer dann, wenn eine Requests `Session` initialisiert wird, wird eine davon an das `Session` Objekt für HTTP und eine für HTTPS gebunden.

Requests ermöglicht es Benutzern, ihre eigenen Transportadapter zu erstellen und zu benutzen, die spezielle Funktionen bereit stellen. Einmal erzeugt, kann ein Transportadapter an ein Session-Objekt gebunden werden, zusammen mit der Informationen, bei welchen Webdiensten es angewendet werden soll.

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

Der `mount`-Aufruf registriert eine spezielle Instanz eines Transportadapters für einen Domänen-Prefix. Nachdem die Bindung erfolgt ist, wird jede HTTP-Anforderung einer Session, deren URL mit dem Domänen-Prefix beginnt, den angegebenen Transportadapter benutzen.

Die Implementierung von Transportadaptern ist nicht Gegenstand dieser Dokumentation, aber ein guter Startpunkt wäre es, von der Klasse `requests.adapters.BaseAdapter` zu erben..

3.5 Authentifizierung

Dieses Dokument behandelt verschiedene Arten der Authentifizierung mit Requests.

Viele Webdienste erforderne eine Anmeldung und es gibt eine ganze Reihe verschiedener Typen. Nachfolgend diskutieren wir verschiedene Formen der Authentifizierung, die in Requests verfügbar sind, vom Einfachen zum Komplexen.

3.5.1 Basic Authentication

Viele Webdienste, die eine Anmeldung erfordern, akzeptieren HTTP Basic Auth. Dies ist die einfachste Art und Weise und Requests unterstützt dies direkt.

Anforderungen mit HTTP Basic Auth zu senden, ist sehr einfach:

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

tatsächlich ist HTTP Basic Auth so verbreitet, dass Requests dafür ein nützliches Kürzel zur Verfügung stellt:

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

Die Anmeldedaten in einem Tupel wie diesem bereit zu stellen ist exakt das Gleiche als in dem `HTTPBasicAuth` Beispiel oben.

3.5.2 Digest Authentication

Eine andere sehr verbreitete Art der HTTP Authentifizierung ist Digest Authentication, die ebenfalls von Requests direkt unterstützt wird:

```
>>> 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 Andere Authentifizierungsarten

Requests wurde so entworfen, dass andere Formen der Authentifizierung leicht schnell hinzugefügt werden können. Mitglieder der Open-Source Gemeinschaft schreiben oft Authentifizierungs-Handler für komplexere oder weniger verbreitete Formen der Anmeldung. Einige der besten wurden unter der Requests organization zusammengefasst, darunter:

- OAuth
- Kerberos
- NTLM

Falls Sie eine dieser Authentifizierungen benutzen wollen, gehen Sie einfach zu ihrer Github-Seite und folgen Sie den Anweisungen.

3.5.4 Neue Formen der Authentifizierung

Falls Sie keine gute Implementierung für die Anmeldung, die Sie nutzen wollen, finden können, können Sie diese selbst implementieren. Requests macht es Ihnen leicht, eigene Authentifizierungsmodule hinzuzufügen.

Dazu leiten Sie von der Klasse `requests.auth.AuthBase` ab und implementieren den Aufruf der `__call__()` Methode. Wenn ein Authentifizierungs-Handler an eine Anforderung gebunden ist, wird er während des Setups der Anforderung aufgerufen. Die `__call__` Methode muss daher alles, was für das Gelingen der Anmeldung nötig ist, implementieren. Einige Formen der Authentifizierung stellen zusätzlich hooks für zusätzliche Funktionalitäten bereit.

Beispiele dazu finden Sie unter Requests organization und in der Datei `auth.py`.

KAPITEL 4

Community Guide

Dieser Abschnitt der Dokumentation befasst sich mit dem Requests-Ökosystem und der Nutzergemeinde.

4.1 Häufig gestelle Fragen

Dieser Teil der Dokumentation beantwortet häufig gestellte Fragen zu Requests.

4.1.1 Encoded Data?

Requests dekomprimiert automatisch Antworten, die gzip-kodiert sind und versucht sein Bestes, um Antworten des Servers wenn möglich in Unicode umzuwandeln.

Falls Sie das benötigen, können Sie auch die die Rohdaten der Antwort (und sogar auf den Socket) zugreifen.

4.1.2 Benutzerdefinierter User-Agents?

Requests erlaubt es Ihnen, auch einfache Art und Weise den User-Agent-String zu überschreiben, zusammen mit jedem anderem HTTP-Header.

4.1.3 Warum nicht Httplib2?

Chris Adams hat dazu eine exizellante Zusammenfassung auf Hacker News geschrieben:

> 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 Unterstützung für Python 3?

Ja! Hier ist die Liste der Python-Versionen, die offiziell unterstützt werden:

- 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 Unterstützung

Wenn Sie eine Frage oder Fehlerberichte zu Reqeusts haben, sind hier einige Möglichkeiten:

4.4.1 Schicken Sie einen Tweet

Falls Ihre Frage in weniger als 140 Zeichen passt, senden Sie einfach einen Tweet an @kennethreitz.

4.4.2 Erstellen Sie ein Ticket

Falls Ihnen ein unerwartetes Verhalten in Requests auffällt oder Sie gerne ein neues Features unterstützt haben würden, erstellen Sie ein Ticket auf Github.

4.4.3 E-mail

Ich antworte gerne auf persönliche oder tiefergehende Fragen zu Requests. Schreiben Sie doch einfach eine Mail an requests@kennethreitz.com.

4.4.4 IRC

Der offizielle IRC Kabele für Requests ist #python-requests

Ich bin auch unter **kennethreitz** auf Freenode.

4.5 Updates

Wenn Sie gerne auf dem Laufenden bleiben wollen, was die Community und die Weiterentwicklung von Requests angeht, haben Sie verschiedene Möglichkeiten:

4.5.1 GitHub

Der beste Weg, um die Entwicklung von Requests zu verfolgen, ist das GitHub repo.

4.5.2 Twitter

I tweittere oft über neue Features und Versionen von Requests.

Folgen Sie @kennethreitz für Updates.

4.5.3 Mailingliste

Es gibt eine Mailingliste für Requests, die nicht zu viel an Mails produziert. Um diese zu abonnieren, senden Sie eine Mail an requests@librelist.org.

KAPITEL 5

API Dokumentation

Falls Sie nach Informationen zu einer bestimmten Funktion, Klasse oder Methode suchen, dann ist dies der richtige Abschnitt der Dokumentation.

5.1 Entwickler-Schnittstelle

Dieser Teil der Dokumentation deckt alle Schnittstellen von Requests ab. Für die Teile von Requests, die externe Bibliotheken benötigen, dokumentieren wir die wichtigsten hier an dieser Stelle und geben Links zur eigentlichen Dokumentation an.

5.1.1 Haupt-Schnittstelle

Die gesamte Funktionalität von Requests kann mit den folgenden sieben Methoden erreicht werden. Alle liefern eine Instanz der Klasse **:class:‘Response <Response>‘_** zurück.

`requests.`**`request`**(*method*, *url*, ***kwargs*)

Constructs and sends a *Request*. Returns *Response* object.

Parameter

- **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’, file-obj)}) 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.

Parameter

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

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

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

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

Untergeordnete Klassen

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.

Parameter

- **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.

Parameter ****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.

Parameter

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

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

Parameter

- **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.

Parameter

- **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 to follow.

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

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

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

- **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*)
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.

Parameter

- **pool_connections** – The number of urllib3 connection pools to cache.
- **pool_maxsize** – The maximum number of connections to save in the pool.

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*.

Parameter

- **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*

Parameter

- **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*.

Parameter

- **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`.

Parameter

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

init_poolmanager(*connections*, *maxsize*)
Initializes a urllib3 PoolManager. This method should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameter

- **connections** – The number of urllib3 connection pools to cache.
- **maxsize** – The maximum number of connections to save in the pool.

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*.

Parameter

- **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.

Parameter

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

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

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

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

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

Nachschlagen von Statuscodes

`requests.`**`codes`**()
Dictionary lookup object.

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

>>> requests.codes.teapot
418

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

Cookies

Encodings

Klassen

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.

Parameter ****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.

Parameter

- **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.

Parameter

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

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

Parameter

- **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.

Parameter

- **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 to follow.

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

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

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

- **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.

Parameter

- **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*)
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.

Parameter

- **pool_connections** – The number of urllib3 connection pools to cache.
- **pool_maxsize** – The maximum number of connections to save in the pool.

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*.

Parameter

- **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*

Parameter

- **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*.

Parameter

- **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.

Parameter

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

init_poolmanager(*connections*, *maxsize*)
Initializes a urllib3 PoolManager. This method should not be called from user code, and is only exposed for use when subclassing the *HTTPAdapter*.

Parameter

- **connections** – The number of urllib3 connection pools to cache.
- **maxsize** – The maximum number of connections to save in the pool.

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*.

Parameter

- **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.

Parameter

- **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 Migration auf 1.x

Dieser Abschnitt zeigt die Hauptunterschiede zwischen 0.x und 1.x auf und ist dazu gedacht, den Schmerz eines Upgrades möglichst gering zu halten.

Änderungen der API

- `Response.json` ist jetzt ein callable und nicht länger eine Eigenschaft einer Serverantwort.

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()    # Dieser *Aufruf* wirft eine Exception, sollte die JSON-Dekodierung fehlschlagen
```

- Die `Session` API hat sich geändert. Session-Objekte erhalten keine Parameter mehr. `Session` wird jetzt mit großem Anfangsbuchstaben geschrieben, aber das Objekt kann aus Kompatibilitätsgründen weiterhin klein geschrieben als `session` instantiiert werden.

```
s = requests.Session()     # früher konnten Parameter übergeben werden
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- Alle Hooks für Anfragen wurden entfernt, mit Ausnahme von 'response'.
- Die Hilfsroutinen für die Authentifizierung wurden in separate Module ausgegliedert. Sehen Sie sich dazu requests-oauthlib and requests-kerberos an.
- Der Parameter für Streaming-Anfragen wurde von `prefetch` in `stream` geändert und die Logik invertiert. Zusätzlich muss `stream` jetzt angegeben werden, um die Rohdaten der Serverantwort zu lesen.

```
# in 0.x, wurde das gleiche Ergebnis mit der Angaben von prefetch=False erreicht
r = requests.get('https://github.com/timeline.json', stream=True)
r.raw.read(10)
```

- Der `config` Parameter für Anfragen wurde entfernt. Einige der davon betroffenen Optionen werden jetzt in einer `Session` konfiguriert, wie z.B. keep-alive und die meximale Anzahl von Redirects. Die Option für ausführliche Informationen (verbosity) sollte über die Konfiguration des Loggings gesetzt werden.

```
# Ausfürhliche Informationen sollten jetzt über das Logging konfiguriert werden
my_config = {'verbose': sys.stderr}
requests.get('http://httpbin.org/headers', config=my_config)  # schlecht!
```

Lizensierung

Ein Hauptunterschied, der nichts mit der API zu tun hat, ist eine Änderung der Lizensierung von der ISC Lizenz zur Apache 2.0 Lizenz. Die Apache 2.0 Lizenz stellt sicher, dass Beiträge zu Requests ebenfalls von der Apache 2.0 Lizenz abgedeckt sind.

KAPITEL 6

Hinweise für Mitmacher

Wenn Sie sich am Projekt beteiligen wollen, ist dies der Teil der Dokumentation, den Sie lesen sollten.

6.1 Entwicklungsphilosophie

Requests ist eine offene, aber etwas eigenwillige Bibliothek, entwickelt von einem offenen, aber etwas eigenwilligen Entwickler.

6.1.1 Wohlwollender Diktator

Kenneth Reitz ist der BDFL. Er hat das letzte Wort hinsichtlich jedes Aspekts von Requests.

6.1.2 Werte

- Einfachheit ist immer besser als Funktionalität.
- Hör jedem zu und dann ziehe es nicht in Betracht..
- Die API ist alles, was zählt. Alles andere ist zweitrangig.
- Kümmere Dich um die 90% der Anwendungsfälle. Ignoriere die Neinsager.

6.1.3 Semantische Versionierung

Über viele Jahre wurde die Open-Source Gemeinschaft von der Versionsnummer-Störung heimgesucht. Die Nummern unterscheiden sich von Projekt zu Projekt so grundlegend, dass diese Information mehr oder weniger nutzlos ist.

Requests benutzt Semantische Versionierung. Dieses Spezifikation versucht, diesem Wahnsinn mit einem kleinen Satz an praxisorientierten Regeln ein Ende zu bereiten.

6.1.4 Standardbibliothek?

Es gibt keinen *aktiven* Plan, dass Requests Teil der Standardbilbiothek wird. Diese Entscheidung wurde intensiv mit Guido und vielen anderen Kernentwicklern besprochen.

Zusammengefasst ist die Standardbibliothek der Platz, an den eine Bibliothek zum Sterben geht. Es ist für ein Modul dann angebracht, in die Standardbibliothek aufgenommen zu werden, wenn eine aktive Entwicklung nicht länger nötig ist.

Requests hat jetzt gerade knapp die Version 1.0.0 hinter sich gelassen. Diese große Meilenstein ist ein wichtiger Schritt in die richtige Richtung.

6.1.5 Linux Distributionen

Distributionen wurde für viele Linux Repositories erstellt, einschließlich Ubuntu, Debian, RHEL, and Arch.

These Distributionen sind manchmal abweichende Forks or werden auf andere Weise nicht auf dem aktuellen Stand der Codebasis und der Bugfixes gehalten. PyPI (und seine Mirrors) und GitHub sind die offiziellen Distributionsquellen; Alternativen werden vom Requests-Projekt nicht unterstützt.

6.2 Wie helfen?

Requests wird aktiv entwickelt und Beiträge sind mehr als willkommen!

1. Suchen Sie nach offenen Tickets oder erstellen Sie selbst ein Ticket, um eine Diskussion über ein neues Feature oder einen Bug zu starten. Es gibt ein Tag Contributor Friendly für Tickets, die ideal für Leute sind, die mit der Codebasis noch nicht so vertraut sind.
2. Spalten Sie das Repository auf Github ab und beginnen Sie damit, Ihre Änderungen im **master** branch (oder erstellen Sie davon einen neuen Branch).
3. Schreiben Sie einen Test, der zegit, dass der Bug behoben wurde oder das neue Feature wie erwartet funktioniert.
4. Senden Sie einen pull request und gehen Sie dem Maintainer so lange auf die Nerven, bis die Änderungen zusammengeführt und publiziert wurden. :) Stellen Sie sicher, dass Sie sich unter AUTHORS hinzugefügt haben.

6.2.1 Feature Freeze

Mit der Version 1.0.0 wurde die Erstellung neuer Features für Requests eingefroren. Anforderungen für neue Features und pull requests, die neue Features implementieren, werden nicht akzeptiert.

6.2.2 Entwicklungs-Abhängigkeiten

Um die Requests Testsuite auszuführen, mpssen Sie py.test installieren:

```
$ 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 Laufzeitumgebungen

Requests unterstützt derzeit die folgenden Versionen von Python:

- Python 2.6
- Python 2.7

- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

Unterstützung für Python 3.1 und 3.2 kann zu jeder Zeit wegfallen.

Die Google App Engine wird nie offiziell unterstützt werden. Pull requests aus Gründen der Kompatibilität werden akzeptiert, so lange sie die Codebasis nicht verkomplizieren.

6.2.4 Sind Sie verrückt?

- Unterstüzung für SPDY wäre großartig. Keine C extensions.

6.3 Autoren

Requests wird geschrieben und betreut von Kenneth Reitz und verschiedenen Autoren:

6.3.1 Entwicklungsleitung

- Kenneth Reitz <me@kennethreitz.com>

6.3.2 Urllib3

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

6.3.3 Patches und Vorschläge

- Verschiedene Mitglieder von Pocoo
- 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>
- Armin Hanisch <@derlinkshaender>

r

R

S

T

U

V