Requests Documentation

Wydanie 1.2.3

Kenneth Reitz

13 cze 2017

Spis treści

Wydanie v1.2.3. (*Instalacja*)

Requests jest biblioteką HTTP na *licencji Apache2*, w języku Python, dla istot ludzkich.

Wbudowany w Pythona moduł **urllib2** oferuje większość potrzebnych możliwości HTTP, ale API jest całkowicie **zepsute**. Zostało zbudowane dla innych czasów — i dla innej sieci. Wymaga *olbrzymiej* ilości pracy (nawet nadpisywania metod) żeby wykonać najprostsze zadania.

Tak nie powinno to wyglądać. Nie w Pythonie.

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

Zobacz podobny kod, bez Requests.

Requests bierze na siebie całą trudną pracę z HTTP/1.1 w Pythonie — czyniąc twoją integrację z usługami sieciowymi bezszwową. Nie ma potrzeby ręcznie dodawać ciągów zapytań do URL–i, albo poddawać twoje dane POST metodzie form-encode. Keep-alive i conection pooling są automatycznew 100% za sprawą urllib3, wbudowanego w Requests.

ROZDZIAŁ 1

Testimonials

Rząd Jej Królewskiej Mości, Amazon, Google, Twilio, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Twitter, SoundCloud, Kippt, Readability, i Instytucje Federalne Stanów Zjednoczonych używają Requests wewnętrznie. Requests zostało pobrane ponad 3 000 000 razy z PyPI.

Armin Ronacher Requests to perfekcyjny przykład, jak piękne może być API z prawidłowym poziomem abstrakcji.

Matt DeBoard Wytatuuję sobie jakoś moduł Pythona requests autorstwa @kennethreitz'a na moim ciele. W całości.

Daniel Greenfeld Zastąpiłem kod spaghetti o długości 1200 LOC z 10 liniami kodu dzięki bibliotece Requests autorstwa @kennethreitz'a. Dziesiejszy dzień był NIESAMOWITY.

Kenny Meyers HTTP w Pythonie: W razie wątpliwości, albo w razie ich braku, użyj Requests. Piękne, proste, Pythoniczne.

ROZDZIAŁ 2

Wspierane funkcje

Requests jest gotowy na dziesiejszą sieć.

- Międzynarodowe domeny i URL-e
- Keep-Alive i Connection Pooling
- Sesje z zachowywaniem Cookies (ciasteczek)
- Weryfikacja SSL w stylu przeglądarek
- Basic/Digest Authentication
- Eleganckie Cookies (klucz/wartość)
- Automatyczna dekompresja
- Odpowiedzi Unicode
- Przesyłanie plików multipart
- Timeout połączeń
- `.netrc` support
- Python 2.6—3.3
- Wątkowo-bezpieczny

ROZDZIAŁ 3

Instrukcja użytkownika

Ta część dokumentacji, w większości proza, zaczyna się od podstawowych informacji o Requests, a potem skupia się na instrukcjach krok po kroku uzyskiwania jak najwięcej z Requests.

Wprowadzenie

Filozofia

Requests były zaprojektowane ze zwróceniem uwagi na kilka idiomów z **PEP 20**.

1. Piękne jest lepsze niż brzydkie.
2. Jawne jest lepsze niż domniemane.
3. Proste jest lepsze niż kompleksowe.
4. Kompleksowy jest lepszy niż skompliowany.
5. Czytelność się liczy.

Wszystkie kontrybucje do Requests powinny pamiętać o tych ważnych regułach.

Licencja Apache2

Obecnie wiele projektów open source jest na licencji GPL. GPL ma swój czas i miejsce, ale nie powinna to być licencja dla twojego następnego projektu open source.

Projekt wydany na licencji GPL nie może być używany w żadnym produkcie komercjalnym bez tego produktu oferowanego jako open source.

Licencje MIT, BSD, ISC i Apache2 są świetnymi alternatywami dla GPL pozwalającymi na dowolne używanie oprogramowania open-source we własnościowym oprogramowaniu closed-source

Requests jest wydany na licencji Apache2.

Licencja Requests (język angielski)

Instalacja

Ta część dokumentacji pokrywa instalację Requests. Pierwszym krokiem używania jakiegokolwiek pakietu oprogramowania jest prawidłowa instalacja.

Distribute & Pip

Instalowanie requests jest proste z pip:

```
$ pip install requests
```

albo z easy_install:

```
$ easy_install requests
```

Ale naprawdę nie powinieneś tego robić.

Mirror dla Cheeseshop (PyPI)

Jeśli Cheeseshop (znany też jako PyPI) nie działa, możesz również zainstalować Requests z jednego z mirrorów. Crate.io jest jednym z nich:

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

Zdobądź kod

Requests jest aktywnie rozwijany na GitHubie, gdzie kod jest zawsze dostępny.

Możesz sklonować publiczne repozytorium:

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

Pobrać tarball:

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

Lub pobrać zipball:

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

Kiedy już masz kopię źródeł, możesz wbudować je w swój pakiet lub łatwo zainstalować je do site-packages:

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

Quickstart

Chcesz zacząć? Ta strona to dobre wprowadzenie jak zacząć pracować z Requests. To wprowadzenie zakład, że już zainstalowałeś Requests. Jeśli tego jeszcze nie zrobiłeś, sprawdź sekcję *Instalacja*.

Po pierwsze, upewnij się, że:

- Requests są *zainstalowane*
- Requests są *aktualne*

Zacznijmy od kilku prostych przykładów.

Wykonaj Żądanie

Wykonywanie żądania z Requests jest bardzo proste.

Zacznij od zaimportowania modułu Requests:

```
>>> import requests
```

Teraz spróbujemy pobrać stronę. W tym przykładzie, spróbujmy pobrać publiczną oś czasu na GitHubie:

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

Teraz mamy obiekt klasy `Response` zwany `r`. Możemy uzyskać wszelkie potrzebne informacje z tego obiektu.

Proste API Requests oznacza, że wszystkie formy żądań HTTP są równie oczywiste. Na przykład, tak wykonuje się żądanie POST:

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

Nieźle, prawda? A jak wykonuje się inne żądania HTTP: PUT, DELETE, HEAD i OPTIONS? Równie prosto:

```
>>> 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")
```

To jest dobre, ale to dopiero początek tego, co może zrobić Requests.

Podawanie parametrów w URL-ach

Bardzo często chcesz wysłać jakieś dane w ciągu zapytania URL-a. Jeśli konstruowałbyś URL ręcznie, byłyby to pary klucz/wartość w URL-u po znaku zapytania, np. `httpbin.org/get?key=val`. Requests pozwala podawać te argumenty jako słownik (`dict`), używając keyword argumentu `params`. Na przykład, jeśli chciałbyś przekazać `key1=value1` i `key2=value2` do `httpbin.org/get`, użyłbyś poniższego kodu:

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

Możesz zobaczyć, że URL został poprawnie zakodowany przez wydrukowanie URL-a:

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

Zauważ, że klucz o wartości `None` nie zostanie dodany do URL-a.

Zawartość odpowiedzi

Możemy przeczytać zawartość odpowiedzi serwera. Skorzystamy ponownie z osi czasu z GitHuba:

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

Requests automatycznie zdekoduje treść z serwera. Większość charsetów Unicode jest poprawnie i bezszwowo dekodowana.

Kiedy wykonujesz żądanie, Requests inteligentnie zgaduje kodowanie na podstawie nagłówków HTTP. To kodowanie jest używane przez `r.text`. Możesz dowiedzieć się, jakiego kodowanie Requests używa, i je zmienić, używając własności `r.encoding`:

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

Jeśli zmienisz kodowanie, Requests użyje nowej wartości `r.encoding` przy whenever you call `r.text`.

Requests może też używać twoich własnych kodowań jeśli będziesz ich potrzebował. Jeśli stworzyłeś swoje własne kodowanie i zarejestrowałeś je w module `codecs`, możesz po prostu użyć nazwy kodeka jako wartość `r.encoding` i Requests zajmie się dekodowaniem za ciebie.

Binarna zawartość odpowiedzi

Możesz też uzyskać dostęp do body odpowiedzi jako bajty, dla żądań nietekstowych:

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

Transfer-encodings: `gzip` i `deflate` są automatycznie dekodowane.

Na przykład, jeśli chcesz stworzyć obrazek z danych binarnych, możesz użyć poniższego kodu:

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

Zwartość odpowiedzi JSON

Istnieje też wbudowany dekoder JSON, jeśli zajmujesz się danymi JSON:

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

Jeśli dekodowanie JSON nie powiedzie się, `r.json` podnosi wyjątek. Na przykład, jeśli odpowiedź wyniesie 401 (Unauthorized), próba użycia `r.json` podnosi `ValueError: No JSON object could be decoded`

Surowa zawratość odpowiedzi

Jeśli chcesz otrzymać surową odpowiedź socket od serwera (a zazwyczaj nie chcesz), możesz użyć `r.raw`. Jeśli chcesz to zrobić, upewnij się, że ustawiłeś `stream=True` w twoim oryginalnym żądaniu. Jeśli to uczynisz, możesz zrobić tak:

```
>>> 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'
```

Własne nagłówki

Jeśli chciałbyś dodać własne nagłówki HTTP do żądania, po prostu użyj parametru `headers` i umieść nagłówki w słowniku (`dict`).

Na przykład, nie podaliśmy content-type w poprzednim przykładzie:

```
>>> 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)
```

Bardziej skompliowane żądania POST

Zazwyczaj, chcesz wysłać dane form-encoded — na przykład z formularza w HTML. Aby to zrobić, po prostu przekaż słownik do argumentu `data`. Twój słownik danych będzie automatycznie zakodowany w formacie formularzy kiedy żądanie zostanie wykonane:

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

Ale czasami chcesz wysłać dane które nie są form-encoded. Jeśli przekażesz `string` zamiast `dict`, dane będą wysłane prosto to serwera.

Na przykład, GitHub API v3 akceptuje dane POST/PATCH zakodowane w JSON:

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

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

POST — plik zakodowany Multipart

Requests sprawia, że dodawanie plików zakodowanych Multipart jest proste:

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

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

Możesz jawnie ustawić nazwę pliku:

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

Jeśli chcesz, możesz wysłać ciągi znaków, które będą otrzymane jako pliki:

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

Kody odpowiedzi

Możemy sprawdzić kod statusu odpowiedzi:

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

Requests ma też wbudowany obiekt sprawdzania kodów dla łatwej referencji:

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

Jeśli wykonaliśmy złe żądanie (odpowiedź 4XX błąd klienta lub 5XX błąd serwera), możemy podnieść wyjątek używając `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
```

Ale, ponieważ `status_code` dla `r` wynosił `200`, `raise_for_status()` wykonuje:

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

Wszystko jest dobrze.

Nagłówki odpowiedzi

Możemy przejrzeć nagłówki odpowiedzi serwera przy użyciu słownika Pythona:

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

Ten słownik jest specjalny: jest on stworzony tylko dla nagłówków HTTP. Zgodnie z RFC 2616, wielkość liter nie ma znaczenia w nagłówkach HTTP.

Więc możemy uzyskać dostęp do nagłówków używając dowolnej wielkości liter:

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

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

Ciasteczka (cookies)

Jeśli odpowiedź zawiera jakieś ciasteczka, możesz szybko uzyskać dostęp do nich:

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

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

Aby wysłać własne ciasteczka do serwera, możemy użyć parametru `cookies`:

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

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

Przekierowania i historia

Requests automatycznie przekieruje żądania przy użyciu GET i OPTIONS.

GitHub przekierowuje wszystkie żądania HTTP na HTTPS. Możemy użyć metody `history` obiektu Response do śledzenia przekierowań. Zobaczmy, co robi GitHub:

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

Lista `Response.history` zawiera obiekty `Request` stworzone w celu zakończenia żądania. Lista jest posortowana od najstarszego do najnowszego żądania.

Jeśli używasz GET lub OPTIONS, możesz zablokować obsługę przekierowań przy użyciu parametru `allow_redirects`:

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

Jeśli używasz POST, PUT, PATCH, DELETE lub HEAD, możesz też włączyć automatyczne przekierowania:

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

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

Timeouty (przekroczenia limitu czasu żądania)

Możesz przerwać czekanie na odpowiedź przez Requests po danej liczbie sekund przy użyciu parametru `timeout`:

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

Note:

`timeout` wpływa tylko na połączenie, nie na pobieranie odpowiedzi.

Błędy i wyjątki

W razie problemu z siecią (np. nieudane żądanie do DNS, odmowa połączenia itd.), Requests podniesie wyjątek *`ConnectionError`*.

W razie rzadkiej nieprawidłowej odpowiedzi HTTP, Requests podniesie wyjątek *`HTTPError`*.

Jeśli żądanie osiągnie timeout, wyjątek `Timeout` jest podnoszony.

Jeśli żądanie przekroczy skonfigurowany limit maksymalnych przekierowań, wyjątek *`TooManyRedirects`* jest podnoszony.

Wszystkie wyjątki podnoszone przez Requests dziedziczą z *`requests.exceptions.RequestException`*.

Gotowy na więcej? Sprawdź sekcję *zaawansowaną*.

Użycie zaawansowane

Ten dokument opisuje niektóre z najważniejszych zaawansowanych funkcji Requests.

Obiekty Session

Obiekty Seesion pozwalają na zachowywanie niektórych parametrów pomiędzy żądaniami. Zachowują one również ciasteczka pomiędzy wszystkimi żądaniami wykonanymi z instancji Session.

Obiekt Session ma wszystkie metody głównego API Requests.

Zachowajmy trochę ciasteczek pomiędzy żadaniami:

```
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"}}'
```

Sesje mogą też być używane do dostarczania domyślnych danych do metod żądań. Robi się to przekazując dane do właściwości obiektu Session:

```
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'})
```

Jakiekolwiek słowniki przekazane do metody żądania będą złączone z ustawionymi wartościami sesyjnymi. Parametry metod nadpisują parametry sesji.

Usuń wartość ze słownika parametru

Czasami chcesz pominąć klucze sesyjne w słowniku parametru. Aby to zrobić, wystarczy ustawić wartość klucza na `None` w parametrze metody. Zostanie on automatycznie pominięty.

Wszystkie wartości w sesji są dostępne bezpośrednio. Sprawdź *dokumentację API sesji* aby dowiedzieć się więcej.

Obiekty Request i Response

Kiedy wywołujesz requests.*() robisz dwie ważne rzeczy: po pierwsze, konstruujesz obiekt `Request` który zostanie wysłany do serwera aby zażądać albo wykonać zapytanie dotyczące jakiegoś zasobu. Po drugie, obiekt `Response` jest generowany kiedy `requests` otrzymuje odpowiedź od serwera. Obiekt Response zawiera wszystkie informacje zwrócone przez serwer oraz oryginalny obiekt `Request`. Oto proste żądanie aby otrzymać parę ważnych informacji z serwerów Wikipedii:

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

Jeśli chcemy uzyskać dostęp do nagłówków, robimy to:

```
>>> 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'}
```

Natomiast, jeśli chcemy otrzymać nagłówki które wysłaliśmy do serwera, po prostu uzyskujemy dostęp do żądania, a potem do jego nagłówków:

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

Przygotowane żądania

Kiedy otrzymasz obiekt `Response` z wywołania API lub Session, atrybut `request` jest tak naprawdę `PreparedRequest`, które było użyte. Czasami chciałbyś coś jeszcze zrobić z body lub nagłówkami (lub czymkolwiek innym) przed wysłaniem żądania. Prostym przepisem na to jest poniższy kod:

```
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)
```

Ponieważ nie robisz nic specjalnego z obiektem `Request`, przygotowujesz go natychmiastowo i zmodyfikowałeś obiekt `PreparedRequest`. Potem możesz wysłać go z innymi parametrami, które przekazałbyś do `requests.*` lub `Sesssion.*`.

Weryfikacja certyfikatów SSL

Requests może weryfikować certyfikaty SSL dla żądań HTTPS, tak jak przeglądarka. Aby sprawdzić certyfikat SSL hosta, możesz użyć argumentu `verify`:

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

Nie mam SSL ustawionego na tej domenie, a więc żądanie nie powodzi się. Świetnie. GitHub natomiast ma certyfikat:

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

Możesz też przekazać ścieżkę do pliku CA_BUNDLE dla prywatnych certyfikatów parametrowi `verify`. Możesz też ustawić zmienną środowiskową `REQUESTS_CA_BUNDLE`.

Requests może też ignorować weryfikację certyfikatu SSL jeśli ustawisz `verify` na False.

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

Domyślnie, `verify` jest ustawiony na True. Opcja `verify` dotyczy tylko certyfikatów hostów.

Możesz też podać lokalny certyfikat do użycia jako certyfikat po stronie klienta, jako pojedynczy plik (zawierając klucz prywatny i certyfikat) lub jako krotkę (tuple) zawierającą ścieżki obu plików:

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

Jeśli podasz złą ścieżkę lub niewłaściwy certyfikat:

```
>>> 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 lib
```

Workflow zawartości body

Domyślnie, kiedy wykonujesz żądanie, body odpowiedzi jest pobierane od razu. Możesz nadpisać to działanie i opóźnić pobieranie do czasu, kiedy wywołasz atrybut `Response.content` przy użyciu parametru `stream`:

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

W tej chwili tylko nagłówki zostały pobrane, a połączenie jest wciąż otwarte, co pozwala nam na pobieranie zawartości pod pewnymi warunkami:

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

Możesz dalej kontrolować workflow używając metod `Response.iter_content` i `Response.iter_lines`, albo czytając z klasy urllib3 `urllib3.HTTPResponse` w `Response.raw`.

Keep-Alive

Dobre wieści — dzięki urllib3, keep-alive jest w 100% automatyczne w sesji! Jakiekolwiek żądanie które wykonasz w sesji automatycznie wykorzysta odpowiednie połączenie!

Zauważ, że połączenia są zwracane do pool do ponownego użycia kiedy wszystkie dane body zostaną przeczytane; upewnij się, że albo ustawiłeś `stream` na `False` albo przeczytałeś własność `content` obiektu `Response`.

Strumieniowanie Uploadów

Requests wspiera strumieniowanie uplaodów, co pozwala na wysyłanie dużych strumieni lub plików bez wczytywania ich do pamięci. Aby strumieniować i uploadować, po prostu podaj obiekt plikopodobny jako twoje body:

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

Żądania Chunk-Encoded

Requests wspiera również kodowanie transferu Chunked dla żądań przychodzących i wychodzących. Aby wysłać żądanie Chunk-encoded, po prostu podaj generator (albo jakikolwiek iterator bez określonej długości) jako twoje body:

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

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

Hooki zdarzeń

Requests ma system hooków który możesz użyć do manipulowania częściami procesu żądania lub procesowania sygnałów zdarzeń.

Dostępne hooki:

response: Odpowiedź wygenerowana z Request.

Możesz przypisać funkcję hooka do każdego żądania osobno przez przekazanie słownika `{hook_name: callback_function}` do parametru `hooks` żądania:

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

Ta `callback_function` otrzyma kawałek danych jako swój pierwszy argument.

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

Jeśli nastąpi błąd podczas wykonywania callbacku, nastąpi ostrzeżenie.

Jeśli funkcja callbacku zwraca wartość, przyjmuje się, że ta wartość ma zastąpić dane podane dla funkcji. Jeśli funkcja nic nie zwraca, nic się nie dzieje.

Wydrukujmy niektóre argumenty metody żądania podczas działania (*at runtime*):

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

Własne uwierzytelnienie

Requests pozwala na użycie własnego mechanizmu uwierzytelnienia.

Jakiekolwiek callable przekazane jako argument `auth` dla metody żądania będzie miał możliwość zmodyfikowania żądania zanim zostanie wysłane.

Implementacje uwierzytelnienia są subklasami `requests.auth.AuthBase`, i można je bardzo prosto zdefiniować. Requests oferuje dwa popularne schematy uwierzytelnienia w `requests.auth`: `HTTPBasicAuth` i `HTTPDigestAuth`.

Załóżmy że mamy usługę sieciową która odpowie tylko jeśli nagłówek `X-Pizza` jest ustawiony na wartość hasła. Mało prawdopodobne, ale po prostu zignoruj to.

```
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
```

Później, możemy wykonać żądanie używając naszego Pizza Auth:

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

Żądania Strumieniowe

Z `requests.Response.iter_lines()` możesz łatwo iterować na strumieniowych API takich jak Twitter Streaming API.

Aby użyć Twitter Streaming API do śledzenia słowa kluczowego „requests”:

```
import json
import requests

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

for line in r.iter_lines():

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

Proxies

Jeśli musisz użyć proxy, możesz skonfigurować indywidualne żądania przy użyciu argumentu `proxies` do każdej metody żądania:

```
import requests

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

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

Możesz też skonfigurować proxy przy użyciu zmiennych środowiskowych `HTTP_PROXY` i `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")
```

Aby użyć HTTP Basic Auth z twoim proxy, użyj składni *http://user:password@host/*:

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

Zgodność

Requests w zamiarze ma być zgodny ze wszystkimi specyfikacjami i RFC odpowiednimi dla Requests jeśli ta zgodność nie będzie sprawiała problemów użytkownikom. Ta uwaga na specyfikację może doprowadzić do niektórych zachowań, które osoby nie znające specyfikacji mogą uznać za dziwne.

Kodowania

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`.

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/
↪a050faf084662f3a352dd1a941f2c7c9f886d4ad')
```

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

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

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

```
>>> commit_data = r.json()
>>> print commit_data.keys()
[u'committer', u'author', u'url', u'tree', u'sha', u'parents', u'message']
>>> print commit_data[u'committer']
{u'date': u'2012-05-10T11:10:50-07:00', u'email': u'me@kennethreitz.com', u'name': u
↪'Kenneth Reitz'}
>>> print commit_data[u'message']
makin' history
```

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

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

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

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

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

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

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

Link Headers

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

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

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

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

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

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

Transport Adapters

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

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

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

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

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

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

Blocking Or Non-Blocking?

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

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

Uwierzytelnienie

Ten dokument opisuje różne formy uwierzytelniania z Requests.

Wiele usług sieciowych wymaga uwierzytelnienia, istnieje też wiele różnych typów. Poniżej opisujemy różne formy uwierzytelniania dostępne w Requests, od tych prostszych do tych bardziej kompleksowych.

Basic Authentication

Wiele usług sieciowych wymagających uwierzytelnienia akceptuje HTTP Basic Auth. Jest to najprostszy rodzaj uwierzytelnienia, i Requests wspiera go od razu *po wyjęciu z pudełka* (out of the box).

Wykonywanie żądań z HTTP Basic Auth jest bardzo proste:

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

W rzeczywistości HTTP Basic Auth jest tak pospolity, że Requests oferuje skrót do używania go:

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

Podawanie danych logowania w krotce (`tuple`) w ten sposób jest identyczne z przykładem `HTTPBasicAuth` wyżej.

Uwierzytelnienie netrc

Jeśli metoda uwierzytelnienia nie jest podana z argumentem `auth`, Requests spróbuje zdobyć dane do logowania dla nazwy hosta URL-a z pliku netrc użytkownika.

Jeśli dane do logowania zostaną znalezione, żądanie jest wysyłane z HTTP Basic Auth.

Digest Authentication

Inną popularną formą uwierzytelnienia HTTP jest Digest Authentication, a Requests wspiera ją również *po wyjęciu z pudełka*:

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

Uwierzytelnienie OAuth 1

Popularną formą uwierzytelnienia w niektórych API sieciowych jest OAuth. Biblioteka `requests-oauthlib` pozwala użytkownikom Requests prosto wykonywać uwierzytelnione żądania OAuth:

```
>>> 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]>
```

Aby uzyskać więcej informacji o tym, jak działa OAuth, zobacz oficjalną stronę OAuth. Aby uzyskać przykłady i dokumentację do requests-oauthlib, zobacz repozytorium requests_oauthlib na GitHubie.

Inne uwierzytelnienie

Requests jest stworzony aby umożliwić łatwe i szybkie dodawanie innych form uwierzytelnienia. Członkowie społeczności open-source często tworzą dodatki do obsługi innych, bardziej skomplikowanych lub rzadziej używanych metod uwierzytelnienia. Najlepsze są częścią organizacji Requests, w tym:

- Kerberos
- NTLM

Jeśli chcesz używać którejś z tych form uwierzytelnienia, idź do ich strony na GitHubie i podążaj za instrukcjami.

Nowe formy uwierzytelnienia

Jeśli nie możesz znaleźć dobrej implementacji formy uwierzytelnienia którą chcesz użyć, możesz sam ją zaimplementować. Requests ułatwia dodawanie własnych form uwierzytelnienia.

Aby to zrobić, stwórz subklasę `requests.auth.AuthBase` i zaimplementuj metodę `__call__()`. Kiedy handler uwierzytelnienia jest dołączony do żądania, jest on wywoływany (*call*) podczas przygotowywania żądania. Metoda `__call__` musi więc zrobić wszystko co trzeba, aby uwierzytelnienie działało. Niektóre formy uwierzytelnienia dodają hooki do oferowania dodatkowej funkcjonalności.

Przykłady można znaleźć w organizacji Requests i w pliku `auth.py`.

ROZDZIAŁ 4

Informacje o społeczności

Ta część dokumentacji, w większości proza, opisuje ekosystem i społeczność Requests.

Często zadawane pytania

Ta część dokumentacji odpowiada na częste pytania o Requests.

Kodowane dane?

Requests automatycznie dekoduje odpowiedzi gzip-encoded, i czyni co w jego mocy, aby zdekodować odpowiedź do Unicode.

Możesz bezpośrednio dostać się do surowej odpowiedzi (a nawet socketu) jeśli zaistnieje taka potrzeba.

Własny User-Agent?

Requests pozwala na łatwe zmienianie ciągów User-Agent, wraz z wszystkimi innymi nagłówkami HTTP.

Dlaczego nie Httplib2?

Chris Adams podsumował to świetnie na Hacker News:

> httplib2 jest częścią "dlaczego powinieneś używać requests": jest bardziej respektowalny jako klient ale nie jest dobrze udokumentowany i potrzeba za dużo kodu dla podstawowych operacji. Doceniam to, co httplib2 próbuje zrobić, że jest wiele niskopoziomowych kłopotóæ w budowaniu nowoczesnego klienta HTTP, ale naprawdę, po prostu użyj requests. Kenneth Reitz jest bardzo zmotywowany i rozumie do jakiego stopnia proste rzeczy powinny być proste, tymczasem httplib2 jest bardziej akademickim ćwiczeniem niż czymś co powinno być używane do budowania systemów w produkcji[1].

Uwaga: jestem w pliku AUTHORS dla request ale jestem odpowiedzialny tylko za około 0.0001% wspaniałości.

1. http://code.google.com/p/httplib2/issues/detail?id=96 jest świetnym przykładem: dokuczliwy bug który dotyczy wielu ludzi, przez miesiące istniała poprawka, która działała świetnie na forku przetestowanym paroma TB danych, ale ponad rok zajęło dostanie się tego do trunk i jeszcze dłużej do PyPI gdzie każdy inny projekt wymagający "httplib2" dostałby działającą wersję.

Wsparcie dla Python 3

Tak! Oto lista oficjalnie wspieranych platform Pythona:

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

Integracja

ScraperWiki

ScraperWiki jest świetną usługą, pozwalającą na uruchamiania scraperów w Pythonie, Rubym i PHP. Teraz Requests v0.6.1 jest dostępny dla twoich scraperów!

Aby spróbować, po prostu:

```
import requests
```

Python dla iOS

Requests is built into the wonderful Python for iOS runtime!

Aby spróbować, po prostu:

```
import requests
```

Artykuły i Wykłady

- Python for the Web uczy jak używać Pythona w intarakcjach z siecią przy pomocy Requests.
- Recenzja Requests Daniela Greenfielda
- Mój wykład 'Python for Humans' (audio)
- Wykład Issaca Kelly'ego 'Consuming Web APIs'
- Post na blogu o Requests via Yum

- Post na rosyjskim blogu o Requests
- Post na francuskim blogu o Requests

Wsparcie

Jeśli masz pytania albo problemy z requests, istnieje kilka opcji:

Tweetnij

Jeśli twoje pytanie (po angielsku) jest krótsze niż 140 znaków, tweetnij do @kennethreitz.

Zgłoś problem

Jeśli zauważyłeś nieoczekiwane zachowanie w Requests, albo chcesz wsparcia dla nowej funkcji, zgłoś problem na GitHub (po angielsku).

E-mail

Chętnie odpowien ma personalne albo dogłębne pytania o Requests. Śmiało pisz (po angielsku) na adres requests@kennethreitz.com.

IRC

Istnieje oficjalny kanał na freenode — #python-requests

Jestem też dostępny jako **kennethreitz** na freenode.

Aktualizacje

Jeżeli chcesz mieć aktualne informacje o społeczności i pracach nad Requests, masz kilka opcji:

GitHub

Najlepszym sposobem na śledzenie prac nad Requests jest repozytorium na GitHub.

Twitter

Bardzo często tweetuję o nowych funkcjach i wydaniach Requests.

Obserwuj @kennethreitz, aby uzyskać więcej informacji.

Lista mailingowa

Istnieje lista mailingowa dla Requests. Aby się zapisać, wyślij maila na adres requests@librelist.org.

ROZDZIAŁ 5

Dokumentacja API

Jeśli poszukujesz informacji o specyficznej funkcji, klasie lub metodzie, ta część dokumentacji jest dla ciebie.

Interfejs dewelopera

Ta część dokumentacji obejmuje wszystkie interfejsy Requests. Tam, gdzie Requests zależy od zewnętrznych bibliotek, dokumentujemy najważniejsze tutaj i oferujemy linki do oryginalnej dokumentacji.

Główny interfejs

Cała funkcjonalność Requests jest dostępna w poniższych 7 metodach. Wszystkie zwracają instancję obiektu *Response*.

requests.**request**(*method*, *url*, ***kwargs*)
Constructs and sends a *Request*. Returns *Response* object.

Parametry

- **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.

Parametry

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

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

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

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

Klasy niższego poziomu

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.

Parametry

- **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.

Parametry ****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.

Sesje żądań

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.

Parametry

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

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

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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*.

Parametry

- **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*

Parametry

- **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*.

Parametry

- **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`.

Parametry

- **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*.

Parametry

- **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*.

Parametry

- **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.

Parametry

- **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.

Wyjątki

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.

Sprawdzanie kodów odpowiedzi

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

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

>>> requests.codes.teapot
418

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

Ciasteczka (cookies)

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

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

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

Parametry **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.

Parametry

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

Kodowania

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

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

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

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

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

Parametry **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

Klasy

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.

Parametry ****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.

Parametry

- **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.

Parametry

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

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

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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.

Parametry

- **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*.

Parametry

- **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*

Parametry

- **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*.

Parametry

- **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`.

Parametry

- **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*.

Parametry

- **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*.

Parametry

- **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.

Parametry

- **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.

Migracja do wersji 1.x

Ta sekcja opisuje najważniejsze różnice pomiędzy 0.x a 1.x i ma na celu ułatwienie aktualizacji.

Zmiany w API

- `Response.json` jest teraz wywoływalne (callable), a nie właściwością (property) odpowiedzi.

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()    # *wywołanie* podnosi wyjątek jeśli dekodowanie JSON nie uda się
```

- API dla `Session` zmieniło się. Obiekty sesji nie przyjmują już parametrów. `Session` jest teraz pisane wielką literą, ale wciąż można używać `session` dla kompatybilności wstecznej.

```
s = requests.Session()     # wcześniej sesja przyjmowała parametry
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- Wszystkie request hooki zostały usunięte, za wyjątkiem `response`.

- Pomocnicy uwierzytelniania są teraz w oddzielnych modułach. Patrz requests-oauthlib i requests-kerberos.
- Parametr dla żądań streamingowych został zmieniony z `prefetch` na `stream` i logika została odwróćona. Dodatkowo, `stream` jest teraz wymagany do odczytu surowej odpowiedzi.

```
# w 0.x, prefetch=False miałoby taki sam efekt
r = requests.get('https://github.com/timeline.json', stream=True)
r.raw.read(10)
```

- Parametr `config` metody requests został usunięty. Niektóre opcje są teraz konfigurowalne w `Session`, np. keep-alive i maksymalna ilość przekierowań. Opcja gadatliwości powinna być skonfigurowana przez logging.

```
import requests
import logging

# te dwie linie włączają debugowanie na poziomie httlib (requests->urllib3->
↪httplib)
# zobaczysz REQUEST, wraz z HEADERS i DATA, oraz RESPONSE z HEADERS i bez DATA.
# jedyną brakującą rzeczą będzie response.body, które nie jest logowane.
import httplib
httplib.HTTPConnection.debuglevel = 1

logging.basicConfig() # musisz zainicjować logging, w przeciwnym wypadku nie␣
↪zobaczysz nic z requests
logging.getLogger().setLevel(logging.DEBUG)
requests_log = logging.getLogger("requests.packages.urllib3")
requests_log.setLevel(logging.DEBUG)
requests_log.propagate = True

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

Licencja

Jedną z kluczowych zmian która nie dotyczy API jest zmiana licencji z licencji ISC na licencję Apache 2.0. Licensja Apache 2.0 zapewnia licencjonowanie kontrybucji na tejże licencji.

ROZDZIAŁ 6

Instrukcja współpracownika

Jeśli chcesz wnieść wkład do projektu, ta część dokumentacji jest dla ciebie.

Filozofia rozwoju

Requests jest otwartą ale zadufaną w sobie biblioteką, tworzoną przez otwartego ale zadufanego w sobie developera.

Dobrotliwy Dyktator (BDFL)

Kenneth Reitz jest BDFL-em. Ma ostateczny głos w każdej decyzji dotyczącej Requests.

Walory

- Prostota jest zawsze lepsza niż funkcjonalność.
- Wysłuchaj wszystkich, a potem to zignoruj.
- Tylko API jest ważne. Wszystko inne jest drugorzędne.
- Mieść się w use-case 90%. Zignoruj krytykantów.

Semantic Versioning

Przez wiele lat społeczność open-source była nękana przez dystonię numerów wersji. Numery różnią się tak bardzo między projektami, że są praktycznie bez znaczenia.

Requests używa Semantic Versioning. Specyfikacja próbuje skończyć to szaleństwo przy użyciu małego zestawu praktycznych zasad, które ty i twoi koledzy powinniście zastosować w następnym projekcie.

Biblioteka Standardowa?

Requests nie ma *aktywnych* planów bycia dołączonym w bibliotece standardowej. Było to przedyskutowane z Guido i wieloma developerami.

Istotnie, biblioteka standardowa to miejsce gdzie biblioteka przychodzi umrzeć. Jest dobra dla modułu którego ciągły rozwój nie jest potrzebny.

Requests ostatnio osiągnęło v1.0.0. Ten wielki kamień milowy oznacza duży krok w dobrą stronę.

Pakiety dystrybucji Linuksa

Istnieją dystrybucje dla wielu repozytoriów Linuksowych, w tym: Ubuntu, Debian, RHEL i Arch.

Te dystrybucje są czasami rozbieżnymi forkami, albo są w inny sposób nieaktualne w stosunku do ostatniego kodu i poprawek. PyPI (i jego mirrory) oraz GitHub są oficjalnymi źródłami dystrybucji; alternatywy nie są wspierane przez projekt Requests.

Jak pomóc

Requests są aktywnie rozwijane, a kontrybucje są więcej niż mile widziane!

1. Poszukaj otwartych problemów albo otwórz nowy aby rozpocząć dyskusję nad bugiem. Istnieje tag *Contributor Friendly* dla problemów idealnych dla osób nieobeznanych z kodem.
2. Sforkuj repozytorium na GitHubie i zacznij dokonywać zmian na nowej gałęzi (branch).
3. Napisz test pokazujący, że bug został naprawiony.
4. Wyślij pull request i wkurzaj maintainera dopóki nie zostanie zmerge'owany i opublikowany. Upewnij się, że dodałes się do pliku AUTHORS.

Zamrożenie feature'ów

Od v1.0.0, Requests wszedł w status zamrożenia feature'ów. Prośby o nowe funkcje i Pull Requesty je implementujące nie będą akceptowane.

Zależności do developmentu

Będziesz musiał zainstalować py.test aby uruchomić testy Requests:

```
$ 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
```

Środowiska runtime

Requests obecnie wspiera następujące wersje Pythona:

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

Wsparcie dla Pythona 3.1 i 3.2 może zostać usunięte w każdej chwili.

Google App Engine nigdy nie będzie oficjanie wspierane. Pull requesty dot. kompatybilności będą akceptowane, o ile nie skomplikują kodu.

Jesteś szalony?

- wsparcie dla SPDY byłoby świetne. Bez rozszerzeń w C.

Paczkowanie w Downstreamie

Jeśli paczkujesz Requests, zauważ że musisz też redystrybuować plik `cacerts.pem` aby uzyskać poprawne funkcjonowanie SSL.

Autorzy

Requests jest tworzony i zarządzany przez Kennetha Reitza i różnych współpracowników:

Dowódca rozwoju

- Kenneth Reitz <me@kennethreitz.com>

Urllib3

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

Łatki i sugestie

- 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
- Bob Carroll <bob.carroll@alum.rit.edu> @rcarz
- Hugo Osvaldo Barrera <hugo@osvaldobarrera.com.ar> @hobarrera
- Łukasz Langa <lukasz@langa.pl> @llanga
- Dave Shawley <daveshawley@gmail.com>

- James Clarke (jam)
- Kevin Burke <kev@inburke.com>
- Flavio Curella

Indeks modułów pythona

r

Indeks

P

R

S

T

U

V