Requests Documentation

Release 2.7.0

Kenneth Reitz

23 set 2017

Indice

Release v2.7.0. (*Installazione*)

Requests è una libreria HTTP con licenza *Apache2*, scritta in Python per gli Esseri Umani.

Il modulo urllib2 della libreria standard Python mette a disposizione quasi tutte le principali funzionalità HTTP ma la sua interfaccia è molto frastagliata. Quel modulo è stato creato per tempi diversi - e un web diverso. Serve molto lavoro (addirittura anche l'overriding di metodi) per realizzare il più semplice dei task.

Le cose non dovrebbero funzionare così. Non 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, ...}
```

Guarda un codice che fa le stesse cose, ma senza Requests.

Requests si fa carico di tutto il lavoro per implementare HTTP/1.1 su Python - rendendo immediata l'integrazione delle tue applicazioni con i web services. Non c'è bisogno di aggiungere manualmente query string agli URL, o di fare form-encoding dei dati di POST. Il Keep-alive e il pooling delle connessioni HTTP sono 100% automatici, tutto ciò grazie a urllib3, che è contenuta dentro Requests.

CAPITOLO 1

Testimonial

Il Governo di Sua Maestà, Amazon, Google, Twilio, Runscope, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, il Washington Post, Twitter, SoundCloud, Kippt, Readability, Sony e Istituzioni Federali degli Stati Uniti che preferiscono rimanere anonime usano Requests al loro interno. E' stato scaricato da PyPI più di 40.000.000 di volte.

Armin Ronacher Requests è l'esempio perfetto di quanto un'API possa essere bella con il giusto livello di astrazione.

Matt DeBoard In un modo o nell'altro, mi farò tatuare addosso il modulo Python requests di @kennethreitz. Il modulo intero.

Daniel Greenfeld Ho rimpiazzato una libreria di 1200 righe di codice spaghetti con sole 10 righe grazie alla libreria requests di @kennethreitz's. Oggi è stata una giornata INCREDIBILE.

Kenny Meyers HTTP con Python: se avete dubbi, o se non ne avete, usate Requests. Bella, semplice, Pythonica.

CAPITOLO 2

Features

Requests è pronto per il web moderno.

- Domini e URL internazionali
- Keep-Alive e Pooling delle connessioni
- Sessioni persistenti attraverso i cookie
- Verifica SSL come la fanno i browser
- Autenticazione Basic/Digest
- Cookie chiave/valore
- Decompressione automatica dei dati
- Corpo delle risposte in Unicode
- Upload di file multipart
- Timeout sulle connessioni
- Supporto per `.netrc`
- Supporto per Python 2.6—3.4
- Thread-safety.

CAPITOLO 3

Documentazione per l'utente

Questa parte della documentazione, che è per lo più in forma di prosa, inizia contestualizzando Requests e prosegue dando istruzioni passo-passo per utilizzare Requests al meglio.

Introduzione

Filosofia

Requests è stata sviluppata tenendo a mente alcuni idiomi della **PEP 20**.

1. Bello è meglio che brutto.
2. Esplicito è meglio che implicito.
3. Semplice è meglio che complesso.
4. Complesso è meglio che complicato.
5. La leggibilità è importante.

Tutti i contributi a Requests dovrebbero seguire queste importanti regole.

Licenza Apache2

Un gran numero di progetti open source moderni hanno licenza GPL. Sebbene la GPL abbia una sua importanza storica, non dovrebbe di certo essere la licenza di riferimento per il vostro prossimo progetto open source.

Un progetto rilasciato in GPL non può essere incluso in prodotti commerciali senza mettere a disposizione in open source anche il prodotto stesso.

Le licenze MIT, BSD, ISC, e Apache2 sono ottime alternative alla GPL che permettono di utilizzare il vostro software open source in codici proprietari e closed source.

Requests è rilasciato nei termini della licenza Apache2.

Licenza di Requests

Installazione

Questa parte della documentazione illustra l'installazione di Requests. Il primo passo per usare qualsiasi pacchetto software è installarlo nel modo corretto.

Distribute e Pip

Installare Requests è semplice con pip, basta eseguire in un terminale:

```
$ pip install requests
```

oppure, con easy_install:

```
$ easy_install requests
```

Ma non dovreste usare il primo dei due.

Ottenere il codice

Requests è sotto sviluppo attivo su GitHub, dove il codice è sempre disponibile.

Potete in alternativa clonare il repository pubblico:

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

oppure scaricare la tarball:

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

oppure, scaricare la zipball:

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

Una volta ottenuta una copia dei sorgenti potete includerli nel vostro package Python, o installarli facilmente nei vostri site-packages:

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

Il primo impatto con Requests

Impazienti di cominciare? Questa pagina vi introdurrà all'uso di Requests.

Prima di tutto, assicuratevi che:

- Requests sia *installato*
- Requests sia *aggiornato*

Iniziamo con alcuni semplici esempi.

Effettuare una Richiesta

Lanciare una richiesta HTTP con Requests è molto semplice.

Per prima cosa importate il modulo Requests:

```
>>> import requests
```

Ora proviamo a leggere una pagina web. Ad esempio, leggiamo la timeline pubblica di GitHub:

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

Abbiamo ottenuto un oggetto di classe *Response* nella variabile `r`. Possiamo ora recuperare tutte le informazioni che ci servono da questo oggetto.

L'API semplicissima di Requests rende le richieste HTTP quasi ovvie. Per esempio, questa è una HTTP POST:

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

Bello, vero? Cosa accade per le altre tipologie di richieste HTTP: PUT, DELETE, HEAD e OPTIONS? Sono anch'esse semplicissime:

```
>>> 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")
```

Finora tutto benissimo, ma è solo un assaggio di ciò che Requests può fare.

Passare parametri negli URL

Spesso dovrete inviare dati nella parte di query string degli URL. Se costruite gli URL a mano, questi dati sarebbero piazzati come coppie chiave/valore dopo il punto interrogativo, es: `httpbin.org/get?key=val`. Requests vi consente di passare i dati sotto forma di dizionario, usando la keyword argument `params`. A titolo di esempio, se volete passare `key1=value1` e `key2=value2` a `httpbin.org/get`, usate:

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

Potete stampare l'URL per verificare che sia stato correttamente codificato:

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

Le chiavi del dizionario di valore `None` non saranno aggiunte alla query string dell'URL.

Per passare come valore una lista di elementi dovete segnalare il fatto che la chiave si riferisce ad una lista appendendo `[]` alla chiave:

```
>>> payload = {'key1': 'value1', 'key2[]': ['value2', 'value3']}
>>> r = requests.get("http://httpbin.org/get", params=payload)
>>> print(r.url)
http://httpbin.org/get?key1=value1&key2%5B%5D=value2&key2%5B%5D=value3
```

Contenuto delle Risposte

Possiamo leggere il contenuto della risposta del server. Leggiamo di nuovo la timeline di GitHub:

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

Requests decodificherà automaticamente il contenuto del server. La maggior parte dei caratteri Unicode sono decodificati senza problemi.

Quando lanciate una richiesta, Requests fa delle ipotesi sull'encoding della risposta sulla base degli header HTTP. L'encoding del testo utilizzato da Requests si applica al contenuto di `r.text`. Potete scoprire quale encoding viene usato da Requests e cambiarlo usando la property `r.encoding`:

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

Se modificate l'encoding, Requests userà il nuovo valore di `r.encoding` ogni volta che chiamate `r.text`. Questo potrebbe esservi utile quando dovete utilizzare una logica custom per determinare quale encoding avrà il contenuto. Ad esempio, HTTP e XML hanno la possibilità di specificare il proprio encoding nel body del documento. In situazioni come questa, dovreste usare `r.content` per ottenere l'encoding del documento e in seguito settare `r.encoding`. Questo permetterà di usare `r.text` con l'encoding corretto.

Requests supporta anche l'impiego di encoding custom nel caso ne abbiate bisogno. Se avete creato il vostro encoding e l'avete registrato nel modulo `codecs`, potete molto semplicemente usare il valore di `r.encoding` e Requests gestirà direttamente il decoding della risposta per voi.

Contenuto binario delle Risposte

Potete anche accedere ai byte che costituiscono il corpo delle risposte non testuali:

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

I transfer-encodings `gzip` e `deflate` sono decodificati automaticamente.

Ad esempio, per creare un'immagine a partire dai dati binari ritornati da una richiesta, potete usare il seguente codice:

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

Contenuto JSON delle Risposte

Se dovete gestire dati in formato JSON, è presente anche un decoder JSON builtin:

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

Nel caso in cui la decodifica JSON fallisca, `r.json` solleva un'eccezione. Ad esempio, se la risposta è un 401 (Unauthorized), l'accesso a `r.json` solleva un `ValueError: No JSON object could be decoded`

Contenuto raw delle Risposte

Nel remoto caso in cui vi servisse il socket raw della risposta del server, potete accedere a `r.raw`. Se lo fate, ricordatevi di impostare `stream=True` nell'effettuare la richiesta.. Quindi potete fare:

```
>>> r = requests.get('https://api.github.com/events', stream=True)
>>> r.raw
<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>
>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

Tuttavia, in generale, dovreste usare questo pattern per salvare su un file i dati in arrivo in streaming:

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

Usare `Response.iter_content` vi risparmierà di dover gestire `Response.raw` in modo diretto. Quando effettuate un download in streaming, questo metodo è da prefersi ed è anzi quello raccomandato.

Header Custom

Se volete aggiungere header HTTP ad una richiesta, semplicemente passate un `dict` come valore del parametro `headers`.

Ad esempio, nel codice precedente non abbiamo specificato il content-type:

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

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

Nota: gli header custom hanno minore priorità rispetto a sorgenti di informazione più specifiche. Ad esempio:

- Gli header Authorization saranno sovrascritti se le credenziali sono passate attraverso il parametro `auth` o sono specificate in un file `.netrc` accessibile nell'ambiente.
- Gli header Authorization saranno rimossi se la risposta è un redirect verso un host diverso.
- Gli header Proxy-Authorization saranno sovrascritti con le credenziali del proxy fornite nell'URL.
- Gli header Content-Length saranno sovrascritti quando è possibile determinare la lunghezza del contenuto.

Inoltre, Requests non modifica in alcun modo il suo comportamento sulla base degli header custom che specificate. Gli header sono semplicemente inseriti nella richiesta conclusiva.

Richieste POST più complesse

Solitamente in POST si invia dati form-encoded - proprio come in un form HTML. Per farlo, basta passare un dizionario come valore dell'argomento `data`. Il vostro dizionario sarà automaticamente form-encoded quando la richiesta sarà lanciata:

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

In altri casi potreste voler inviare dati non form-encoded. Se passate una `string` al posto di un `dict`, i dati saranno POSTati direttamente.

Ad esempio, l'API di GitHub API v3 accetta dati con encoding JSON per le richieste POST/PATCH:

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

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

Postare un file Multipart-Encoded

Requests rende semplice l'invio di file Multipart-encoded:

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

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

Potete specificare esplicitamente il nome del file, il content_type e gli headers:

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

```
>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
```

```
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

Se volete, potete inviare stringhe al posto di file veri e propri:

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

Nel caso in cui dobbiate Postare un file molto grande con una richiesta `multipart/form-data`, dovreste inviare la richiesta in streaming. Di default, `requests` non lo supporta ma esiste un package a parte per questo - `requests-toolbelt`. Per maggiori dettagli su come usarlo, leggete la documentazione di toolbelt .

Per inviare file multipli in una sola richiesta fate riferimento alla sezione *avanzate*.

Status code delle risposte

Possiamo controllare lo status code delle risposte:

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

Requests offre anche un oggetto built-in per fare il lookup veloce degli status code:

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

Se una richiesta non va a buon fine (errore 4XX del client o 5XX del server), possiamo sollevare eccezioni con *`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
```

Ma se lo `status_code` di `r` fosse `200`, invocando `raise_for_status()` otterremmo:

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

e dunque tutto OK.

Header delle risposte

Possiamo visionare gli header delle risposte del server tramite un dizionario Python:

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

Questo dizionario è tuttavia speciale: è fatto apposta per gli header HTTP. Secondo la RFC 7230, i nomi degli header HTTP sono case-insensitive.

Dunque, possiamo accedere agli header usando qualsiasi case:

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

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

Il dizionario è speciale anche perchè il server potrebbe aver inviato lo stesso header più di una volta con valori differenti, ma requests li combina in modo che possano essere rappresentati nel dizionario con un singolo schema di mappatura, così come dice la RFC 7230:

> Un ricevente PUO' combinare più campi header aventi lo stesso nome in una > singola coppia "nome-campo: valore-campo" senza cambiare la semantica del > messaggio, assegnando come valore del campo combinato l'unione di tutti > i valori successivi separati da virgola.

Cookie

Se una risposta contiene dei cookie, potete ottenerli facilmente:

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

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

Per inviare i vostri cookie al server usate il parametro `cookies`:

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

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

```
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

Redirezione e History

Di default Requests effettua la redirezione dell'host per tutti i verbi HTTP ad eccezione di HEAD.

Possiamo usare la property `history` dell'oggetto Risposta per tracciare le redirezioni.

La lista *`Response.history`* contiene gli oggetti di tipo *`Response`* che sono stati creati per completare le richieste. La lista è ordinata a partire dalla risposta meno recente fino a quella più recente.

Per esempio, GitHub redirige tutte le richieste HTTP su HTTPS:

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

Se state usando GET, OPTIONS, POST, PUT, PATCH o DELETE potete disabilitare la redirezione attraverso il parametro‘‘allow_redirects‘‘:

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

Se state usando HEAD potete nello stesso modo abilitare la redirezione:

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

Timeout

Potete fare in modo che Requests smetta di attendere una risposta dopo un certo numero di secondi attraverso il parametro `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)
```

Errori ed Eccezioni

Nel caso di problemi di rete (es: il DNS non risponde, la connessione è rigettata, etc), Requests solleverà un'eccezione di tipo

ConnectionError.

Nel raro caso di una risposta HTTP invalida, Requests solleverà un'eccezione di tipo *HTTPError*.

Se una richiesta va in timeout, viene sollevata un'eccezione di tipo *Timeout*.

Se una richiesta eccede il numero massimo configurato di redirezioni, è sollevata un'eccezione di tipo *TooManyRedirects*.

Tutte le eccezioni esplicitamente sollevate da Requests ereditano dalla classe *requests.exceptions.RequestException*.

Siete pronti ad andare più nel dettaglio? Passate alla sezione *avanzate*.

Uso avanzato di Requests

Questo documento illustra alcune delle caratteristiche avanzate di Requests.

Oggetti Session

L'oggetto Session vi consente di tenere traccia di alcuni parametri tra una richiesta e le successive. Esso memorizza anche i cookie tra le diverse richieste effettuate con esso.

Un oggetto Session ha tutti i metodi dell'API canonica di Requests.

Proviamo a memorizzare dei cookie tra le richieste:

```
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"}}'
```

Le sessioni possono essere usate anche per fornire dati di default ai metodi di richiesta. Basta passare i dati alle property di un oggetto Session:

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

# sia 'x-test' che 'x-test2' sono settati
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

I dizionari che passate a un metodo di richiesta saranno incorporati nei valori specifici della sessione che avete impostato. I parametri a livello di metodo sovrascrivono i corrispondenti parametri a livello di sessione.

Tutti i valori contenuti all'interno di una sessione sono direttamente disponibili. Per saperne di più leggete la *documentazione dell'API delle sessioni*.

Oggetti Request e Response

Quando si invocano `requests.get()` e metodi di richiesta simili accadono due cose importanti. Primo, state costruendo un oggetto `Request` che verrà inviato ad un server per ottenere un qualche genere di risorsa. Secondo, un oggetto `Response` è generato quando `requests` riceve una risposta dal server. L'oggetto Response contiene tutte le informazioni ritornate dal server e contiene anche l'oggetto `Request` creato in origine. Questa è una semplice richiesta che ottiene informazioni molti importanti dai server di Wikipedia:

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

Se volessimo accedere agli header che il server ci ha inviato, useremmo:

```
>>> 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'}
```

Mentre se volessimo accedere agli header che abbiamo inviato noi al server dovremmo accedere prima alla request e quindi agli header della richiesta:

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

Richieste preparate

Ogni volta che ricevete un oggetto *Response* dalla chiamata all'API o a Session, l'attributo `request` contiene la `PreparedRequest` che è stata utilizzata.

In alcuni casi potreste dover manipolare il corpo o gli headers (e a dire il vero qualsiasi altra cosa) di una richiesta prima di inviarla. Un modo semplice per farlo è il seguente:

```
from requests import Request, Session

s = Session()
req = Request('GET', url,
    data=data,
    headers=header
)
prepped = req.prepare()

# fate qualcosa con prepped.body
# fate qualcosa con prepped.headers

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

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

Visto che non dovete fare nulla di speciale con l'oggetto `Request`, lo preparate da subito e modificate l'oggetto `PreparedRequest`. A questo punto inviate questo oggetto insieme gli altri parametri che avreste passato ai metodi `requests.*` o `Session.*`.

Tuttavia, il codice qui sopra perde i vantaggi di avere un oggetto *Session* in Requests. Nello specifico, lo stato a livello di *Session*, come ad esempio i cookie, non viene riportato sulla vostra richiesta. Per ottenere una *PreparedRequest* contenente quello stato, sostituite la chiamata a *Request.prepare()* con la chiamata a *Session.prepare_request()*, in questo modo:

```
from requests import Request, Session

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

prepped = s.prepare_request(req)

# fate qualcosa con prepped.body
# fate qualcosa con prepped.headers

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

print(resp.status_code)
```

Verifica dei certificati SSL

Requests può verificare i certificati SSL per le richieste HTTPS, esattamente come i browser. Per verificare il certificato SSL di un host potete usare l'argomento `verify`:

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

Non ho impostato SSL su quel dominio, per cui la richiesta fallisce. Ottimo. Github tuttavia ha SSL:

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

Potete specificare per `verify` il path ad un file CA_BUNDLE contenente certificati di una Certification Authority di fiducia. La lista di bundle delle CA di fiducia può essere anche secificata attraverso la variabile di ambiente `REQUESTS_CA_BUNDLE`.

Requests può anche ignorare la verifica dei certificati SSL se impostata `verify` a False.

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

Di default, `verify` è True. L'opzione `verify` si applica solo a certi host.

Potete anche specificare un certificato locale da usare come verifica client side, sia sotto forma di file locale (contenente la chiave privata e il certificato) che di tupla contenente i path ad entrambi i file:

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

Se specificate un path errato o un certificato invalido:

```
>>> 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 di lettura del corpo delle risposte

Di default, quando lanciate una richiesta il corpo della risposta è scaricato immediatamente. Potete modificare questo comportamento e deferire il download del corpo della risposta fino a quando non leggete il valore dell'attributo *`Response.content`* utilizzando il parametro

`stream`:

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

A questo punto solo gli header sono stati scaricati per la risposta e la connessione rimane aperta, dunque lasciando il recupero del contenuto alla nostra volontà:

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

Potete ancora di più controllare il workflow usando i metodi *`Response.iter_content`* e *`Response.iter_lines`*. In alternativa, potete leggere il corpo della risposta dalla sottostante urllib3 `urllib3.HTTPResponse` disponibile invocando *`Response.raw`*.

Se impostate `stream` a `True` quando inviate una richiesta, Requests non può liberare la connessione in modo che ritorni nel pool fino a che non fruite dei dati o non chiamate *`Response.close`*. Questo comportamento può portare ad un uso inefficiente delle connessioni. Se vi capita sovente di leggere solo parzialmente il corpo delle richieste (o di non leggerlo per nulla) quando `stream=True`, dovreste prendere in considerazione l'uso di `contextlib.closing` (documentato qui), in questo modo:

```
from contextlib import closing

with closing(requests.get('http://httpbin.org/get', stream=True)) as r:
    # Usate la risposta
```

Keep-Alive

Grandi notizie — grazie a urllib3, il keep-alive delle connessioni è al 100% automatico all'interno di una sessione! Ogni richiesta che inviate all'interno di una sessione userà automaticamente la stessa connessione!

Notate che le connessioni sono rilasciate e mess nel pool per il riuso solo quando tutti i dati del corpo delle richieste sono stati letti; dunque accertatevi o di impostare `stream` a `False` o di leggere il `content` dei vostri oggetti `Response`.

Upload in streaming

Requests supporta gli upload in streaming, il che consente di inviare grossi flussi di dati o grandi file senza doverli leggere in memoria. Per lanciare un upload in streaming, vi basta fornire un file-like object come corpo della richiesta:

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

Richieste Chunk-Encoded

Requests supporta anche il trasferimento in chunks sia per le richieste in uscita che per le risposte in ingresso. Per inviare una richiesta chunk-encoded, dovete semplicemente fornire un generatore (o qualsiasi iteratore di lunghezza indefinita) come corpo della richiesta:

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

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

POST-are più file Multipart-Encoded

Potete inviare più file in una singola richiesta. Ad esempio, immaginate di voler uploadare dei file immagine da un form HTML con un campo file multiplo di nome 'images':

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

Per fare lo stesso con Requests vi basta inserire i file in una lista di tuple nella forma (nome_campo_del_form, tupla_con_info_sul_file):

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

Hook per gli eventi

Requests ha un sistema di hook che potete usare per manipolare le fasi del processo di richiesta o gestire eventi di segnalazione.

Hook disponibili:

response: La risposta generata a partire da una richiesta.

Potete impostare una funzione callback ad ogni singola richiesta, passando un dizionario `{hook_name: callback_function}` al parametro `hooks` della richiesta:

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

La funzione `callback_function` riceverà come primo argomento la richiesta.

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

Se accade un errore durante l'esecuzione della vostra callback, viene sollevato un warning. Se la callback ritorna un valore, il contratto implicito è usare questo valore per rimpiazzare i dati che sono stati passati come argomento. Se la callback non ritorna nulla, nessuna azione è intrapresa.

Printiamo a runtime alcuni metodi di richiesta:

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

Autenticazione custom

Requests vi permette di specificare un meccanismo di autenticazione custom.

Ogni callable che passerete all'argomento `auth` di un metodo di richiesta avrà la possibilità di modificare la richiesta prima che questa venga inviata.

Le implementazioni del meccanismo di autenticazione devono essere sottoclassi di `requests.auth.AuthBase`, e sono semplici da realizzare. Requests mette a disposizione in `requests.auth` due schemi di autenticazione di uso comune: `HTTPBasicAuth` e `HTTPDigestAuth`.

Mettiamoci nell'ipotetico caso di avere un web service che risponde solamente se l'header `X-Pizza` contiene una username. Abbastanza improbabile, ma andiamo avanti.

```
from requests.auth import AuthBase

class PizzaAuth(AuthBase):
    """Aggiunge l'Autenticazione HTTP Pizza a questa istanza di Request."""
    def __init__(self, username):
        # setup any auth-related data here
        self.username = username

    def __call__(self, r):
        # modifico e ritorno la richiesta
        r.headers['X-Pizza'] = self.username
        return r
```

A questo punto possiamo fare richieste usando la nostra classe Pizza Auth:

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

Richieste in streaming

Con *requests.Response.iter_lines()* potete iterare in maniera semplice su API in streaming come l'API Streaming di Twitter. Basta impostare `stream` a `True` e iterare sulla risposta con *iter_lines()*:

```
import json
import requests

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

for line in r.iter_lines():

    # printa le nuove righe in streaming
    if line:
        print(json.loads(line))
```

..caveat:

```
:class:`~requests.Response.iter_lines()` non è un metodo rientrante.
Invocarlo più volte provoca la perdita di parte dei dati ricevuti. Nel caso
in cui serva invocarlo da più punti del vostro codice, usate piuttosto
l'iteratore che risulta dalla sua invocazione::

    lines = r.iter_lines()
    # Memorizza la prima riga per dopo o skippala
    first_line = next(lines)
    for line in lines:
        print(line)
```

Proxy

Se dovete utilizzare un proxy, potete configurare ogni singolo metodo di richiesta con l'argomento `proxies`:

```
import requests

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

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

Potete anche configurare i proxy attraverso le variabili di ambiente `HTTP_PROXY` e `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")
```

Per usare la HTTP Basic Authentication con il vostro proxy, servitevi della sintassi *http://user:password@host/*

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

Notate che gli URL dei proxy devono includere lo schema.

Conformità

Requests è pensato per essere conforme con tutte le specifiche e le RFC rilevanti laddove tale conformità non causi difficoltà d'utilizzo per gli utenti. Questa attenzione alla specifica può portare ad un comportamento che potrebbe sembrare inusuale a coloro che non sono familiari con le specifiche stesse.

Encoding

Quando ricevete una risposta, Requests cerca di capire l'encoding da usare per decodificarla quando accedete l'attributo *Response.text*. Requests dapprima cercherà un encoding specifico negli header HTTP, e se non ne trova, allora si servirà chardet per cercare di indovindare l'encoding.

L'unica situazione in cui Requests non seguirà questa procedura è quando non è specificato un charset esplicito negli header HTTP **e** l'header `Content-Type` contiene `text`. In tale situazione, la RFC 2616 specifica che il charset di default deve essere `ISO-8859-1`. Requests si conforma alla specifica in questo caso. Se avete bisogno di un encoding diverso, potete settare manualmente la property *Response.encoding* o usare l'oggetto *Response.content* raw.

Verbi HTTP

Requests dà accesso a quasi tutto il range di verbi HTTP: GET, OPTIONS, HEAD, POST, PUT, PATCH e DELETE. Di seguito vengono illustrati esempi dettagliati di come usare questi verbi in Requests, usando l'API di GitHub.

Cominceremo con il verbo di più comune utilizzo: GET. HTTP GET è un metodo idempotente che ritorna una risorsa da un URL specifico. Dunque è il verbo che dovete usare quando cercate di ottenere dati da un indirizzo web. Un esempio d'uso potrebbe essere recuperare informazioni su una specifica commit da GitHub. Immaginate di volere la commit `a050faf` con Requests. La potremmo recuperare così:

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

Dovremmo controllare se GitHub ha risposto correttamente. Se sì, wogliamo capire quale tipo di contenuto ha ritornato. Facciamo così:

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

Dunque GitHub ritorna JSON. Grandioso, possiamo usare il metodo *r.json* per convertirlo in un oggetto Python.

```
>>> 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
```

Finora tutto semplice. Ora investighiamo l'API di GitHub un po' più nel dettaglio. Potremmo guardare la documentazione, ma sarebbe più divertente usare Requests. Possiamo servirci del verbo OPTIONS per vedere quali metodi HTTP sono consentiti sull'URL che abbiamo appena usato.

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

Cosa? Questo non ci serve! Sembra che GitHub, come molti provider di API, non implementi il metodo OPTIONS. E' una cosa noiosa, ma OK, possiamo sempre annoiarci a leggere la documentazione. Se GitHub avesse implementato OPTIONS, avrebbe ritornato negli header i metodi consentiti. As esempio

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

Se leggiamo la documentazione, notiamo che l'unico altro metodo consentito sulle commit è POST, che crea una nuova commit. Dato che stiamo usando il repo di Requests, dovremmo evitare di creare a mano delle nuovo commit. Usiamo invece le Issue di GitHub.

Il documento che state leggendo è stato aggiunto in risposta alla Issue #482. Dal momento che tale issue esiste già, la useremo come esempio. Cominciamo con il leggerla.

```
>>> 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
```

Grande, abbiamo tre commenti. Diamo un'occhiata all'ultimo.

```
>>> 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
```

Bè, non sembra la migliore delle sezioni! Postiamo un nuovo commento per dire all'autore del commento che è un cretino. Ma... chi è l'autore?

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

OK, diciamo a questo tizio di nome Kenneth che crediamo che l'esempio debba andare nella sezione quickstart. La documentazione dell'API di GitHub, il modo giusto è POST-are su un thread. Facciamolo

```
>>> 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
```

Mmmm, strano. Probabilmente dobbiamo autenticarci. Sarà un'impresa, vero? No, sbagliato. Requests rende facile usare varie forme di autenticazione, inclusa la comunissima HTTP Basic Authentication.

```
>>> 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.
```

Fico! No, aspettate un attimo! Volevamo anche aggiungere che ci metteremo un po' perchè dobbiamo dare da mangiare al nostro gatto. Se solo potessimo modificare quel commento! Con piacere, GitHub ci permette di usare un altro verbo HTTP, PATCH, per modificare quel commento. Facciamolo.

```
>>> 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
```

Eccellente. Ora, per continuare a torturare questo tizio di nome Kenneth, decidiamo di lasciarlo all'oscuro del fatto che stiamo lavorando sulla issue. Questo significa che vogliamo cancellare il commento usando il metodo DELETE, dal nome incredibilmente azzeccato. Buttiamo via il commento.

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

Perfetto. E' sparito. L'ultima cosa che vorremmo sapere è quanto siamo vicini al numero limite di chiamate all'API. Vediamolo. GitHub invia questa informazione negli headers, dunque al posto di scaricare un'intera pagina manderemo una richiesta di tipo HEAD su di essa per recuperare solo gli headers.

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

Eccellente. E' il momento di scrivere un programma Python che abusi dell'API di GitHub in un sacco di modi divertenti, 4995 altre volte.

Header Link

Molte API web usano header di tipo Link. Questi rendono le API più esplicite da comprendere ed esplorabili.

L'API di GitHub li usa per la paginazione dei dati, ad esempio:

```
>>> 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 leggerà automaticamente gli header Link e li renderà facilmente usabili:

```
>>> 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'}
```

Adapter di Trasporto

Dalla versione v1.0.0, Requests ha adottato un design interno modulare. Uno dei motivi alla base di questo è l'implementazione di Adapter di Trasporto, in origine **'descritti qui'_**. Gli Adapter di Trasporto forniscono un meccanismo per definire i metodi di interazione con un servizio HTTP. Nello specifico, permettono di utilizzare una configurazione specifica per ogni servizio.

Requests contiene un singolo Adapter di Trasporto, *HTTPAdapter*. Questo adapter implementa l'interazione di default di Requests con HTTP e HTTPS servendosi della poderosa libreria urllib3 library. Ogni volta che una *Session* è inizializzata, un adapter è allegato all'oggetto *Session* per HTTP e un secondo adapter per HTTPS.

Requests consente agli utenti di creare e usare i propri Adapter di Trasporto per esporre funzionalità custom. Una volta creato, un Adapter di Trasporto può essere montato su un oggetto Session insieme all'indicazione di quali servizi web si dovrebbe applicare.

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

Questa chiamata registra un'istanza specifica di un Adapter di Trasporto ad un prefisso. Una volta montato, ogni richiesta HTTP fatta usando la sessione il cui URL inizia con il prefisso specificato si servirà dell'Adapter specificato.

Molti dei dettagli implementativi di un Adapter di Trasporto sono oltre lo scopo di questa documentazione, ma date un'occhiata al prossimo esempio per un semplice caso d'uso con SSL. Se volete andare ancora oltre, dovreste creare una sottoclasse di `requests.adapters.BaseAdapter`.

Esempio: usare una versione specifica di SSL

Il team di Requests ha fatto la scelta specifica di usare qualsiasi versione di SSL si trovi di default nella sottostante libreria (urllib3). Ci potrebbero essere volte in cui dovrete connettervi ad un servizio web che usa una versione di SSL non compatibile con la vostra di default.

In questo caso potete usare gli Adapter di Trasporto riutilizzando la maggior parte dell'implementazione di un HTTPAdapter e aggiungendo un parametro *ssl_version* che viene passato ad *urllib3*. Creiamo un Adapter che dice alla libreria di usare SSLv3:

```
import ssl

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

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

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

Bloccante o Non-Bloccante?

Quando usa l'Adapter di Trasporto di default, Requests non provvede alcun tipo di IO non-bloccante. La property `Response.content` si bloccherà fincho a che l'intera risposta non è stata scaricata. Se vi serve pià granularità, le potenzialità di streaming della libreria (si veda *Richieste in streaming*) vi consentiranno di recuperare piccoli pezzi della risposta uno dopo l'altro. Tuttavia, queste chiamate sono sempre bloccanti.

Se intendete usare un paradigma di IO non-bloccante, ci sono molti progetti che combinano Requests con uno dei framework asincroni per Python. Due esempi d'eccellenza sono grequests e requests-futures.

Timeout

La maggior parte delle richieste a server esterni dovrebbero contenere un timeout nel caso in cui il server non risponda con le tempistiche che ci si attende. Senza un timeout il vostro codice potrebbe rimanere in attesa per minuti o anche più.

Il timeout di **connessione** è il numero di secondi che Requests attenderà che il vostro client stabilisca una connessione alla macchina remota (corrisponde alla chiamata connect()) sul socket. E' buona pratica settare i timeout di connessione poco oltre un multiplo di 3, che è la finestra di default di TCP per la ritrasmissione dei pacchetti.

Quando il client è connesso al server e ha inviato la richiesta HTTP, il timeout **di lettura** è il numero di secondi che il client attenderà che il server invii una rispostsa (nello specifico, è il numero di secondi che il client attende *tra* ogni byte inviato dal server. Nel 99.9% dei casi, questo è il tempo che passa prima che il server invii il primo byte).

Se specificate un valore singolo per il timeout, ad esempio così:

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

Il valore di timeout sarà usato per entrambi i timeout di `connect` e `read`. Specificate una tupla se volete impostare i valori separatamente:

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

Se il server remoto è molto lento, potete istruire Requests di attendere indefinitamente per la risposta, passando None come valore di timeout e munendovi di una tazza di caffè.

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

Certificati delle CA

Di default Requests contiene una raccolta di certificati root delle Certification Authority che considera fidate, provenienti dal Mozilla trust store. Tuttavia questi certificati sono aggiornati solo ad ogni nuova versione di Requests. Ciò significa che se utilizzate per molto tempo una specifica versione di Requests, i certificati possono divenire molto obsoleti.

Dalla versione 2.4.0 in poi, Requests cerca di utilizzare i certificati da certifi se questo è presente sul sistema. Ciò consente agli utenti di aggiornare i loro certificati di fiducia senza dover modificare il codice che gira sui loro sistemi.

A beneficio della sicurezza, vi raccomandiamo di aggiornare certifi frequentemente!

Autenticazione

Questo documento discute l'uso dei diversi tipi di autenticazione con Requests.

Molti servizi web richiedono autenticazione e ce ne sono varie tipologie. Illustreremo le varie forme di autenticazione disponibili in Requests, dalla più semplice alla più complessa.

Autenticazione Basic

Molto servizi web che richiedono autenticazione accettano l'HTTP Basic Auth. E' la più semplice tipologia di autenticazione e Requests la supporta di default.

Lanciare richieste con HHTTP Basic Auth è molto semplice:

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

L'uso di HTTP Basic Auth è così comune che Request fornisce una scorciatoia molto comoda per usarla:

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

Fornire le credenziali in una tupla in questo modo funziona esattamente come l'esempio sopra con `HTTPBasicAuth`.

Autenticazione con netrc

Se nessun metodo di autenticazione è specificato per il parametro `auth`, Requests tenta di recuperare le credenziali di autenticazione per l'hostname dell'URL dal file netrc dell'utente.

Se vengono trovate delle credenziali per l'hostname, la richiesta è inviata con HTTP Basic Auth.

Autenticazione con Digest

Un'altra tipologia molto popolare di autenticazione HTTP è quella con Digest, e Requests supporta anche questa di default:

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

Autenticazione con OAuth 1

Un'altra forma comune di autenticazione per dicerse API web è OAuth. La libreria `requests-oauthlib` consente agli utenti di Requests di fare richieste con 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]>
```

Per maggiori informazioni sul workflow di autenticazione OAuth, fate riferimento al sito werb officiale di OAuth. Per esempi e documentazione su requests-oauthlib, consultate il repository GitHub requests_oauthlib

Altre forme di Autenticazione

Requests è strutturato per essere agganciabile in maniera semplice con altri metodi di autenticazione.

La community open-source scrive di frequente nuovi handler di autenticazione per forme di autenticazione più complesse o usate meno comunemente. Alcuni dei migliori sono stati riuniti su GitHub sotto l'organizzazione Requests, tra cui:

- Kerberos
- NTLM

Se volete usare queste forme di autenticazione, visitate la loro pagina GitHub e seguite le istruzioni relative.

Nuove forme di autenticazione

Se non riuscite a trovare un'implementazione di una forma di autenticazione che vi soddisfi, potete crearla voi stessi. Requests rende facile l'aggiunta di una forma di autenticazione custom.

Per farlo, create una sottoclasse di *AuthBase* e implementate il metodo `__call__()`:

```
>>> import requests
>>> class MyAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Implementate la vostra autenticazione
...         return r
...
>>> url = 'http://httpbin.org/get'
>>> requests.get(url, auth=MyAuth())
<Response [200]>
```

Quando un handler di autenticazione è incluso in una richiesta, viene invocato durante il setup della richiesta. Il metodo `__call__` deve quindi fare tutto ciò che è possibile per fare funzionare l'autenticazione. Alcune forme di autenticazione aggiungeranno hook addizionali per mettere a disposizione ulteriori funzionalità.

Potete trovare altri esempi su GitHub sotto l'organizzazione Requests e nel file `auth.py`.

CAPITOLO 4

Documentazione per la community

Questa parte della documentazione, che è per lo più in forma di prosa, spiega i dettagli dell'ecosistema e della community attorno a Requests.

Frequently Asked Questions

Questa parte della documentazione cerca di rispondere alle domande più comuni su Requests

Encoding dei dati?

Requests decomprime automaticamente le risposte compresse gzip, e fa del suo meglio per decodificare il contenuto delle risposte in Unicode quando è possibile.

Se serve, potete anche accedere al contenuto raw delle risposte (e pure al socket sottostante)

Custom User-Agents?

Requests vi consente di sovrascrivere in maniera semplice l'header User-Agent, così come ogni altro header HTTP.

Perchè Requests e non Httplib2?

Chris Adams ha scritto un'eccellente nota a riguardo su Hacker News:

> httplib2 è parte del motivo per cui dovreste usare requests: è assai rispettabile come client ma non è documentata sufficienza e richiede ancora troppo codice per effettuare anche operazioni di base. Apprezzo quello che httplib2 cerca di fare, che la costruzione di una libreria HTTP moderna richiede miriadi di scocciature in codice di basso livello, ma in maniera franca vi dico: usate requests. Kenneth Reitz è molto motivato e ha un'idea precisa di quanto le cose più usate debbano essere semplici, mentre httplib2 sembra più un esercizio accademico che un qualcosa che la gente può usare per costruire sistemi reali[1].

Vi rivelo una cosa: sono elencato nel file AUTHORS di requests ma posso accreditarmi solo, oh, circa lo 0.0001% della sua magnificenza.

1. http://code.google.com/p/httplib2/issues/detail?id=96 è un ottimo esempio: un bug rognoso che ha infastidito tante persone, una fix è rimasta disponibile per mesi e ha funzionato alla grande quando l'ho inclusa in una fork e ho scaricato un paio di TB con essa, ma ci ha messo più di un anno per essere inclusa nel trunk e ancora più tempo ad arrivare su PyPI, dove tutti gli altri progetti che richiedevano "httplib2" la attendevano.

Supporto per Python 3?

Si! Ecco una lista di versioni Python che sono ufficialmente supportate:

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- Python 3.4
- PyPy 1.9
- PyPy 2.2

Cosa significano gli errori "hostname doesn't match"?

Questi errori accadono quando la *verifica del certificato SSL* fallisce: non c'è matching tra il certificato inviato in risposta dal server e l'hostname che Requests crede di contattare. Se siete sicuri che il setup SSL del server è corretto (per sempio, perchè riuscite a visitare il sito con il vostro browser) e state utilizzando Python 2.6 o 2.7, una soluzione possibile è abilitare la Server-Name-Indication.

Server-Name-Indication, o SNI, è un'estensione ufficiale di SSL che prevede che i client dicano al server quale hostname stanno contattando. Questo è importante se i server usano il Virtual Hosting. Infatti quando questi server ospitano più di un sito su SSL devono essere in grado di ritornare il certificato SSL giusto sulla base dell'hostname a cui i client si connettono.

Il modulo SSL di Python3 include il supporto nativo per SNI. Questa feature non è stata riportata su Python2. Per maggiori dettagli sull'uso di SNI con Requests su Python2 si faccia riferimento a questa **'risposta su Stack Overflow'_**.

Package ed Estensioni raccomandate

Requests dispone di una grande varietà di utili estensioni di terze party. Questa pagina offre una visione di insieme delle migliori.

Certifi CA Bundle

Certifi è una collezione scelta di Root Certificates per validare la fiducia dei certificati SSL mentre si verifica l'identità degli host TLS. E' nata dal progetto Requests.

CacheControl

CacheControl è un'estensione che aggiunge piene capacità di caching HTTP a Requests. Questo rende le vostre richieste HTTP di gran lunga più efficienti, e dovrebbe essere usato quando effettuate un gran numero di richieste.

Requests-Toolbelt

Requests-Toolbelt è una collezione di moduli che gli utenti di Requests possono trovare utili, ma non fanno parte di Requests. Questa libreria è attivamente mantenuta da membri del core team di Requests e riflette le funzionalità più richieste dagli utenti nella community.

Requests-OAuthlib

requests-oauthlib rende possibile il workflow di autenticazione OAuth da Requests in modo automatico. Questo è utile per il gran numero di siti web che usano OAuth per fornire servizi di autenticazione. Vengono anche messi a disposizione molti tweaks per gestire l'autenticazione su provider specifici che non aderiscono alle specifiche OAuth standard.

Betamax

Betamax traccia le tue interazioni HTTP al posto dell'NSA. E' un'imitazione di VCR imitation pensata specificamente per Python-Requests.

Integrazioni

ScraperWiki

ScraperWiki è un servizio eccellente che consente di eseguire script di scraping in Pyhon, Ruby e PHP sul web. Requests v0.6.1 è attualmente disponibile per l'utilizzo nei vostri scrapers!

Per provarlo, basta:

```
import requests
```

Python per iOS

Requests è incluso nel grandioso runtime Python for iOS !

Per provarlo, basta:

```
import requests
```

Articoli & Talk

- Python per il Web spiega come

usare Python per interagire con il web usando Requests. - Recensione di Daniel Greenfeld su Requests - **'Il mio talk 'Python per gli esseri umani'<http://python-for-humans.heroku.com>'_** (audio) - Talk di Issac Kelly 'Utilizzare Web API' - Post di un blog che spiega come installare Requests con Yum - Post di un blog russo che introduce a Requests - Inviare JSON con Requests

Supporto

Se avete domande o problemi riguardo a Requests, avete diverse opzioni:

StackOverflow

Se la vostra domanda non contiene informazioni sensibili o può essere anonimizzata senza problemi, chiedete aiuto su StackOverflow e usate il tag `python-requests`.

Tweet

Se la vostra domanda è contenibile in 140 caratteri, inviate un tweet a @kennethreitz, @sigmavirus24, o @lukasaoz.

Create una issue

Se vi accorgete che Requests si comporta in maniera anomala, o volete richiedere lo sviluppo di una nuova feature, aprite una issue su GitHub.

E-mail

Sono più che contento di rispondere ad ogni domanda personale o tecnica circa Requests. Siate liberi di inviarmi una e-mail a requests@kennethreitz.com.

IRC

Il canale Freenode per Requests è #python-requests

Gli sviluppatori del core team di Requests sono su IRC durante la giornata. Li potete trovare in `#python-requests` sotto questi nomi:

- kennethreitz
- lukasa
- sigmavirus24

Rendere esplicite le vulnerabilità

Se pensate di aver individuato una potenziale falla di sicurezza i requests, siete pregati di comunicarla direttamente via e-mail a sigmavirus24 e Lukasa . **Non aprite alcuna issue pubblica a riguardo.**

Le nostre PGP Key fingerprints sono:

- 0161 BB7E B208 B5E0 4FDC 9F81 D9DA 0A04 9113 F853 (@sigmavirus24)

- 90DC AE40 FEA7 4B14 9B70 662D F25F 2144 EEC1 373D (@lukasa)

Se la vostra madrelingua non è l'inglese, perfavore provate a descrivere il problema e il suo impatto al meglio delle vostre capacità linguistiche. Per entrare nel dettaglio, utilizzate la vostra lingua madre e cercheremo di tradurla al meglio usando dei servizi online.

Siete pregati anche di includere il codice che avete usato per identificare il problema e la minima porzione di codice necessaria per riprodurlo.

Perfavore non rivelate il problema a nessun altro. Recupereremo un identificatore CVE se necessario e vi daremo la piena paternità della scoperta sotto qualsiasi nome o alias vorrete fornirci. Vi chiederemo un modo per identificarvi solo quando avremo trovato una soluzione al problema e potremo pubblicarla in una release.

Rispetteremo la vostra privacy e renderemo pubblico il vostro coinvolgimento solo se ci darete il permesso di farlo.

Processo

Le seguenti informazioni riguardano il processo che il progetto requests seguirà in risposta alla rivelazione di una vulnerabilità. Se state per comunicare una vulnerabilità, questa parte della documentazione vi spiega come reagiremo alla vostra segnalazione.

Timeline

Quando riportate un problema, uno dei membri del progetto vi risponderà entro *al massimo* due giorni. Nella maggior parte dei casi le risposte saranno più rapide, di solito entro le 12 ore. Questa prima risposta servirà almeno da conferma della ricezione del vostro report.

Se riusciremo a riprodurre velocemente il problema, la risposta iniziale conterrà anche una conferma dell'esistenza della issue. Se non ci riusciremo, probabilmente chiederemo più informazioni per ripodurre lo scenario.

Il nostro obiettivo è di rilasciare una fix per ogni vulnerabilità entro due settimane dalla notifica della sua esistenza. Questo può potenzialmente portare ad una release ad intermi che disabilita la funzionalità mentre una soluzione definitiva è in sviluppo, ma generalmente ci sarà una versione completa rilasciata il prima possibile.

Lungo il processo di risoluzione vi terremo aggiornati sul progresso della fix. Una volta che la fix è pronta, ve lo faremo sapere. Spesso vi richiederemo di confermare che la fix risolve il problema nel vostro ambiente di lavoro, in special modo se non siamo pienamente convinti della sua efficacia nel nostro ambiente di lavoro.

Giunti a tal momento, ci prepareremo per la release. Se sarà necessario, richiederemo un identificativo CVE, assegnandovi la piena attribuzione della scoperta. Decideremo anche una data per la release e vi faremo sapere quando sarà. Questa data di release sarà *sempre* un giorno feriale.

Quindi, contatteremo i principali downstream packagers per informarli di una imminente patch di sicurezza di modo che si possano organizzare. In più, questi packagers riceveranno la patch per tempo, così che possano rilasciare in tempo i loro packages dipendenti. Al momento, la lista delle persone che contattiamo attivamente *prima di ciascuna public release* è:

- Ralph Bean, Red Hat (@ralphbean)
- Daniele Tricoli, Debian (@eriol)

Informeremo queste persone almeno una settimana prima della nostra release pianificata per assicurarci che abbiano tempo a sufficienza per prepararsi. Se pensate di dover essere inclusi nella lista, siete pregati di informare uno dei maintainer agli indirizzi e-mail provvisti all'inizio del documento.

Il giorno della release, pusheremo la patch sul nostro repository pubblico, aggiornando il changelog con la descrizione del problema e attribuendovi il merito della scoperta. Faremo in seguito una release su PyPI contenente la patch.

Infine, pubblicizzeremo la release. Ciò avverrà attraverso e-mail alle mailing list, tweet e tutti gli altri mezzi di comunicazione disponibili al core team.

Menzioneremo esplicitamente quali commit contengono la fix in modo che sia facile per i distributori e gli utenti creare patch nelle proprie versioni di requests se l'aggiornamento alla nuova release non è per loro cosa fattibile.

Storico delle CVEs

- Risolto nella versione 2.6.0
  - CVE 2015-2296, segnalato da Matthew Daley di BugFuzz.
- Risolto nella versione 2.3.0
  - CVE 2014-1829
  - CVE 2014-1830

Aggiornamenti dalla community

Se volete restare aggiornati sulla community e lo stato di sviluppo di Requests, avete varie opzioni:

GitHub

Il modo migliore per seguire lo sviluppo di Requests è sul repo GitHub.

Twitter

L'autore, Kenneth Reitz, twitta spesso su nuove feature e release di Requests.

Seguite @kennethreitz per avere aggiornamenti.

Storico delle release e delle versioni

Storico delle release

2.7.0 (2015-05-03)

Questa è la prima release che segue il nuovo processo di release. Per maggiori dettagli si veda la nostra documentazione.

Fix di bachi

- Aggiornata urllib3 a 1.10.4, risolvendo vari bachi sull'encoding nei trasferimenti chunked e sul framing delle risposte.

2.6.2 (2015-04-23)

Fix di bachi

- Fix di una regressione dove dati compressi inviati in maniera chunked non erano decompressi correttamente. (#2561)

2.6.1 (2015-04-22)

Fix di bachi

- Rimossa il meccanismo di VendorAlias sugli import introdotto in v2.5.2.
- Semplificata l'API di PreparedRequest.prepare: non si richiede più che l'utente passi una lista vuota al keyword argument hooks (c.f. #2552)
- Risolve redirects ora riceve e inoltra tutti gli argomenti originali all'adapter. (#2503)
- Gestione degli UnicodeDecodeErrors quando si cerca di trattare un URL Unicode che non puà essere ASCII-endoded. (#2540)
- Durante l'autenticazione Digest viene popolato il path del campo URI (#2426)
- Si può copiare in maniera più affidabile il CookieJar di una PreparedRequest quando questo non è un'istanza di RequestsCookieJar. (#2527)

2.6.0 (2015-03-14)

Fix di bachi

- CVE-2015-2296: Fix della gestione dei cookie sulle redirezioni. In precedenza, un cookie senza il valore host avrebbe usato l'hostname dell'URL di redirezione esponendo così gli utenti di Requests ad attacchi di fixing delle sessioni e al potenziale furto dei cookie. Questo è stato comunicato privatamente da Matthew Daley di BugFuzz. Questo baco riguarda tutte le versioni di Requests dalla v2.1.0 alla v2.5.3 (estremi inclusi).
- Fix dell'errore che accade quando Requests è una dipendenza `install_requires` e si esegue `python setup.py test`. (#2462)
- Fix dell'errore che si ha quando urllib3 non è inclusa nella distribuzione e Requests continua ad utilizzare la locazione della versione bundled.
- Fix alla gestione degli header da parte di `urllib3`.
- Ora il modo con cui Requests gestisce le dipendenze non incluse è più restrittivo

Feature e Miglioramenti

- Supporto al passaggio di bytearrays come parametri nell'argomento''files''. (#2468)
- Evitata duplicazione dei dati quando si crea una richiesta con valori di tipo `str`, `bytes``o ``bytearray` per l'argomento `files`.

2.5.3 (2015-02-24)

Fix di bachi

- Rollback della modifica al bundle dei certificati. Per più informazioni si vedano (#2455, #2456, and http://bugs.python.org/issue23476)

2.5.2 (2015-02-23)

Feature e Miglioramenti

- Aggiunto il sopporto alle checksum sha256. (shazow/urllib3#540)
- Migliorata la performance degli header. (shazow/urllib3#544)

Fix di bachi

- Copiato il meccanismo degli import di pip. Quando i redistributori downstream rimuovono requests.packages.urllib3 il meccanismo di import farà sì che gli stessi simboli continuino a funzionare. Gli esempi d'uso della documentazione di Requests e librerie di terze parti che si basano su copie redistribuite di urllib3 continueranno a funzionare senza dover passare alla urllib3 di sistema.
- Tentativo di fare quoting delle parti di un URL sulle redirezioni, se l'unquoting e in seguito il quoting falliscono. (#2356)
- Fix del check sul tipo di nome del file sugli upload di dati multipart form. (#2411)
- Viene correttamente gestito il caso in cui un server che invia challenges per autenticazione Digest fornisce sia valori-qop auth che auth-int. (#2408)
- Fix del leak di un socket. (shazow/urllib3#549)
- Fix di svariati header `Set-Cookie`. (shazow/urllib3#534)
- Disabilitata la verifica built-in dell'hostname. (shazow/urllib3#526)
- Fix del decoding di uno stream dati vuoto. (shazow/urllib3#535)

Sicurezza

- Pullato un `cacert.pem` aggiornato.
- Rimossa RC4 dalla lista degli algoritmi di cifratura di default. (shazow/urllib3#551)

2.5.1 (2014-12-23)

Modifiche al comportamento

- In raise_for_status vengono catturati solo gli HTTPErrors (#2382)

Fix di bachi

- Gestione di LocationParseError di urllib3 (#2344)
- Gestione dei nomi dei file-like object che non sono stringhe (#2379)
- Nell'handler di HTTPDigestAuth handler viene consentito di negoziare nuovi nonce (#2389)

2.5.0 (2014-12-01)

Miglioramenti

- Consentito l'uso dell'oggetto Retry di urllib3 negli HTTPAdapters (#2216)
- Il metodo `iter_lines` su una risposta ora accetta un delimitatore con il quale splittare il contenuto (#2295)

Modifiche al comportamento

- Aggiunto warning di deprecazione alle funzioni di requests.utils che saranno rimosse in in 3.0 (#2309)
- Le Sessioni usate dall'API funzionale vengono sempre chiuse (#2326)
- Le richieste sono state ristrette ai soli HTTP/1.1 e HTTP/1.0 (non più HTTP/0.9) (#2323)

Fix di bachi

- Gli URL vengono parsati una volta sola (#2353)
- L'header Content-Length può essere sempre sovrascritto (#2332)
- Gestione corretta dei file handle nell'autenticazione HTTP Digest (#2333)
- Limitazione della dimensione di redirect_cache per prevenire abusi di memoria (#2299)

- Fix della gestione delle redirezioni dopo autenticazione corretta con HTTP Digest (#2253)
- Fix del crash quando si danno parametri custom Session.request (#2317)
- Fix su come gli header Link sono parsati usando la libreria per le espressioni regolari (#2271)

Documentazione

- Aggiunte più riferimenti per l'interlinking (#2348)
- Aggiornato il CSS del tema (#2290)
- Aggiornata la larghezza dei pulsanti e della sidebar (#2289)
- Sostituiti i riferimenti a Gittip con quelli a Gratipay (#2282)
- Aggiunto nella sidebar un link al changelog (#2273)

2.4.3 (2014-10-06)

Fix di bachi

- Miglioramenti agli URL Unicode per Python2.
- Re-ordinamento del parametro JSON per retrocompatibilità.
- Deframmentazione automatica degli schemi di autenticazione dagli URI con host/password. (#2249)

2.4.2 (2014-10-05)

Miglioramenti

- FINALMENTE! Aggiunto il parametro json parameter per gli upload! (#2258)
- Supporto per i bytestring URL su Python 3.x (#2238)

Fix di bachi

- Rimossa situazione di loop infinito (#2244)
- Varie chiamate a iter* fallivano con un errore non autodescrittivo. (#2240, #2241)

Documentazione

- Corretta l'introduzione alla redirezione (#2245)
- Aggiunto esempio di come inviare più file in una sola richiesta (#2227)
- Spiegato meglio come passare un set di certificati custom (#2248)

2.4.1 (2014-09-09)

- Ora c'è un set di extras di nome "security", `$ pip install requests[security]`
- Requests ora usa Certifi se è disponibile.
- Viene catturato e ri-sollevato l'errore ProtocolError di urllib3
- Fix al baco per cui alcune cercano di redirigere a se stesse all'infinito (ma che c...)

2.4.0 (2014-08-29)

Modifiche al comportamento

- L'header `Connection: keep-alive` ora è inviato in maniera automatica.

Miglioramenti

- Supporto per il timeout delle connessioni! Timeout ora accetta una tupla di forma (connect, read) usata per settare i timeout individuali di connessione e lettura.
- E' consentito compiare una PreparedRequests senza headers/cookies.
- Aggiornata la dipendenza a urllib3.
- Refactoring nel caricamneto dei setting dall'ambiente: introdotto *Session.merge_environment_settings*.
- Gestione degli errori dei socket dentro iter_content.

2.3.0 (2014-05-16)

Modifiche all'API

- Nuova property `is_redirect` per `Response`: è true quando la libreria avrebbe potuto processare la risposta come una redirezione (che lo abbia fatto o meno).
- Il parametro `timeout` ora impatta le richieste sia con `stream=True` che `stream=False` senza distinzione.
- Rollback della modifica fatta in v2.0.0 che richiedeva esplicitazione di URL scheme per i proxy. Questi ora defaultano a `http://`.
- Il `CaseInsensitiveDict` usato per gli header HTTP ora si comporta come un normale dizionario quando si riferisce a stringhe o viene visto tramite l'interprete.

Fix di bachi

- Gli header Authorization e Proxy-Authorization non sono più esposti in caso di redirezioni. Fix CVE-2014-1829 e CVE-2014-1830 rispettivamente.
- L'autorizzazione è ri-effettuata ad ogni redirezione.
- Sulle redirezioni, gli URL sono passati come stringhe native.
- Fall-back sull'encoding auto-rilevato per il JSON quando la rilevazione Unicode fallisce.
- Gli headers di valore `None` nelle `Session` non vengono inviati.
- Viene onorata in modo corretto `decode_unicode` anche se non era usata in precedenza nella stessa risposta
- Il Content-Encoding `compress` non è più supportato.
- Il parametro `Response.history` ora è sempre una lista.
- Tanti, tanti fix a bachi di `urllib3`.

2.2.1 (2014-01-23)

Fix di bachi

- Fix del parsing scorretto delle credenziali proxy che contengono un carattere '#' letterale o encoded.
- Vari fix a urllib3.

2.2.0 (2014-01-09)

Modifiche all'API

- Nuova eccezione: `ContentDecodingError`. Sollevato al posto dell'eccezione `DecodeError` di `urllib3`.

Fix di bachi

- Evitato il sollevamento di un sacco di eccezioni sulla debole implementazione di `proxy_bypass` su OS X per Python 2.6.
- Evitato il crashing mentre si cerca di ottenere credenziali di autenticazione da ~/.netrc se si impersona un utente senza una home directory.
- Uso della dimensione corretta per i pool di connessioni ai proxy.
- Fix sull'iterazione degli oggetti `CookieJar`.
- Ora i cookie sono salvati durante i redirect.
- Ritorniamo ad usare chardet, dal momento che si è fuso con charade.

2.1.0 (2013-12-05)

- Ovviamente, aggiornato il bundle dei certificati.
- I cookie impostati su singole richieste attraverso una `Session` (es: con `Session.get()`) non sono più salvati sulla `Session`.
- Non c'è più leak della connessioni quando si verificano problemi sugli upload chunked.
- Le connessioni vengono ritornate nel pool quando un upload chunked va a buon fine.
- Implementate le recommendation HTTPbis per le redirezioni HTTP 301.
- Rimossa l'attesa indefinita sugli upload in streaming con autenticazione Digest e un 401 è ricevuto.
- I valori degli header impostati da Requests sono ora tipi stringa nativi.
- Fix: il supporto SNI era rotto.
- Fix: accesso ai proxy HTTP usando l'autenticazione ai proxy
- Decodifica degli username e password HTTP Basic estratti dagli URL.
- Supporto ai range di indirizzi IP nella variabile d'ambiente no_proxy.
- Corretto il parsing degli header quando gli utenti sovrascrivono l'header `Host:` di default.
- Nessun URL-munging in caso di server case-sensitive.
- Gestione degli URL più rilassata per gli URL non-HTTP/HTTPS.
- I metodi Unicode sono accettati in Python 2.6 e 2.7.
- Gestione cookie più robusta agli errori.
- Gli oggetti `Response` sono serializzabili con Pickle.
- Ora sono stati davvero (diversamente dalla volta scorsa) aggiunte le sessioni MD5 all'autenticazione Digest.
- Aggiornata la dipendenza a urllib3.
- Fix: la mancanza di senso estetico di @Lukasa.

2.0.1 (2013-10-24)

- Aggiornato il bundle dei certificati con nuovi provider parzialmente fidati e un processo automatico
- Aggiunte sessioni MD5 all'autenticazione Digest
- Vengono accettati headers per per ogni singolo file nelle POST di file multipli
- Fix: non veniva inviato l'URL intero sulle CONNECT
- Fix: lo URL schema nei redirect viene messo correttamente in lowercase
- Fix: i cookie impostati attraverso l'API funzionale non erano salvati
- Fix: tradotto l'errore di ProxyError urllib3 in un errore ProxyError di Requests derivato da ConnectionError.
- Aggiornate le dipendenze a urllib3 e chardet.

2.0.0 (2013-09-24)

Cambiamenti all'API:

- Le chiavi nel dizionario Headers sono stringhe native in tutte le versioni di Python, es: bytestrings su Python 2, Unicode su Python 3.
- Gli URL dei Proxy ora *devono* avere uno schema esplicito. In caso contrario, un'eccezione `MissingSchema` è solelvata.
- I timeout ora si applicano al tempo di lettura dei dati se `Stream=False`.
- `RequestException` è ora sottoclasse di `IOError`, non `RuntimeError`.
- Aggiunto nuovo metodo agli oggetti `PreparedRequest`: `PreparedRequest.copy()`.
- Aggiunto nuovo metodo agli oggetti `Session`: `Session.update_request()`. Questo metodo aggiorna un oggetto `Request` con i dati (es: cookie) salvati sulla `Session`.
- Aggiunto nuovo metodo agli oggetti `Session`: `Session.prepare_request()`. Questo metodo aggiorna e prepara un oggetto `Request` e ritorna il corrispondente oggetto `PreparedRequest`.
- Aggiunto nuovo metodo agli oggetti `HTTPAdapter`: `HTTPAdapter.proxy_headers()`. Non dovrebbe essere invocato direttamente, ma migliora l'interfaccia delle sottoclassi.
- Le eccezioni `httplib.IncompleteRead` causate da un encoding dei chunk ora sollevano un'eccezione `ChunkedEncodingError` di Requests.
- Sequenze di percent-escape invalide causano ora un'eccezione `InvalidURL` di Requests.
- HTTP 208 non usa più il messaggio di spiegazione `"im_used"`. Usa invece correttamente `"already_reported"`.
- Aggiunto messaggio di spiegazione ad HTTP 226 (`"im_used"`).

Fix di bachi:

- Migliorato sensibilmente il supporto ai proxy, incluso il verbo CONNECT. Un ringraziamento speciale ai tanti collaboratori che hanno lavorato su questa miglioria.
- I cookies ora sono gestiti bene quando si ricevono risposte 401.
- Fix all'encoding chunked.
- Supporto per schemi URL mixed-case.
- Migliorata la gestione dei download in streaming.

- Vengono recuperati i proxy ambientali da più locazioni.
- Fix di minore entità sui cookie.
- Migliorato il comportamento di redirezione.
- Migliorato il comportamento dello streaming, in particolare con dati compressi
- Fix su vari piccoli bachi sul text encoding su Python 3.
- `.netrc` non sovrascrive più le forme di autenticazione esplicite.
- I cookie impostati dagli hook vengono ora correttamente salvati nelle sessioni
- Fix di un baco sui cookie per cui si specificava il numero di porta nel loro campo host
- `BytesIO` può essere usato per fare upload in streaming
- Parsing più generoso della variabile di ambiente `no_proxy`.
- E' possibile passare oggetti diversi da stringhe come dati insieme ai file.

1.2.3 (2013-05-25)

- Semplice fix sul packaging

1.2.2 (2013-05-23)

- Semplice fix sul packaging

1.2.1 (2013-05-20)

- Le redirezioni 301 e 302 cambiano tutti i verbi - non solo POST - in GET, migliorando la compatibilità con i browser
- Compatibilità con Python 3.3.2
- Gli header Location sono sempre percent-encoded
- Fix: i primi connection adapter ad essere matchati sono i più specifici
- nuovo argomento per il connection adapter di default per passare un argomento block
- quando non ci sono header Link non viene sollevato un KeyError

1.2.0 (2013-03-31)

- Fix sui cookie durante le sessioni e sulle richieste
- Pesantemente modificato il modo in cui gli hoook sono invocati - ora gli hook ricevono tutti gli argomenti specificati dall'utente quando lancia una richiesta così gli hook possono lanciare una richiesta secondaria con gli stessi parametri. Questo è particolarmente necessario per gli autori di handler di autenticazione.
- Il supporto a certifi è stato rimosso
- Fix al baco per cui non venivano inviati dati usando OAuth1 con il corpo `signature_type`
- Grosso lavoro sui proxy grazie a @Lukasa incluso il parsing dei dati di autenticazione a partire dall'URL del proxy
- Fix baco sulla gestione di troppo 401 con autenticazione DigestAuth

- Aggiornata la urllib3 per includere fix di bachi SSL
- Ora i keyword arguments possono essere passati a `json.loads()` attraverso il metodo `Response.json()`
- Di default non viene inviato l'header `Content-Length` sulle richieste `GET` o `HEAD`
- Aggiunto l'attributo `elapsed` agli oggetti `Response` per misurare quanto tempo ha impiegato una richiesta ad essere evasa
- Fix a `RequestsCookieJar`
- Sessioni e Adapter ora sono serializzabili con Pickle, es: possono essere usati con la libreria multiprocessing
- Aggiornato charade alla versione 1.0.3

La modifica nel modo in cui gli hoook sono invocati molto probabilmente causerà un gran numero di issues.

1.1.0 (2013-01-10)

- RICHIESTE CHUNKED
- Supporto per corpo delle risposte iterabili
- Assunzione che i server memorizzino i parametri di redirect
- Ora è possibile specificare dei content type espliciti per i file
- Durante il lookup delle chiavi, merge_kwargs è case-insensitive

1.0.3 (2012-12-18)

- Fix baco sull'encoding durante upload di file
- Fix sul comportamento dei cookie

1.0.2 (2012-12-17)

- Fix sul proxy per HTTPAdapter.

1.0.1 (2012-12-17)

- Fix baco di verifica dei certificati.
- Fix sul proxy per HTTPAdapter.

1.0.0 (2012-12-17)

- Un micchio di refactoring e semplificazione
- Adozione della licenza Apache 2.0
- Connection Adapters sostituibili
- Connection Adapters montabili sulle sessioni
- Catena delle ProcessedRequest è mutabile
- /s/prefetch/stream
- Rimozione di tutta la configurazione

- Logging attraverso la Standard library
- Ora Response.json() è una callable, non una property
- Utilizzo del nuovo progetto charade project, che fornisce la chardet simultanea per python 2 e 3
- Rimozione di tutti gli hook ad eccezione di'response'
- Rimozione di tutti gli helper di autenticazione (OAuth, Kerberos)

Questa release non è retrocompatibile.

0.14.2 (2012-10-27)

- Migliorata la gestione del JSON mime-compatibile
- Fix sui Proxy
- Fix sull'hacking dei path
- Headers Content-Encoding sono ora case-insensitive
- Supporto per i parametri CJK nel POST-ing dei form

0.14.1 (2012-10-01)

- Compatibilità con Python 3.3
- Semplice valore di default per accept-encoding
- Fix di bachi

0.14.0 (2012-09-02)

- iter_content non dà più errori se il contenuto è stato già scaricato.

0.13.9 (2012-08-25)

- Fix su OAuth + POSTs
- Rimosso occultamento delle eccezioni da dispatch_hook
- Fix di bachi

0.13.8 (2012-08-21)

- Supporto pazzesco agli header Link :)

0.13.7 (2012-08-19)

- Supporto per liste in formato (key, value) dovunque.
- Miglioramenti sull'autenticazione Digest.
- Ora l'esclusione dei proxy funziona correttamente.
- Eccezioni UnicodeError più chiare.

- Casting automatico degli URL a stringhe
- Fix di bachi.

0.13.6 (2012-08-06)

- Fix atteso da lungo tempo sulle connessioni in stallo!

0.13.5 (2012-07-27)

- Fix sul packaging

0.13.4 (2012-07-27)

- Autenticazione GSSAPI/Kerberos!
- Fix all'App Engine 2.7!
- Fix baco sui leak delle connessioni (segue dall'update di urllib3)
- Fix dell'hacking sui path in OAuthlib
- Fix dei parametri degli URL in OAuthlib.

0.13.3 (2012-07-12)

- Uso di simplejson se disponibile.
- Non nascondere gli SSLErrors dietro ai Timeouts.
- Gestione dei parametri fissi con URL che contengono fragments.
- Migliorato sensibilmente il contenuto di User Agent.
- I certificati dei client sono ignorati quando verify=False

0.13.2 (2012-06-28)

- Nessuna dipendenza (di nuovo)!
- Nuovo: Response.reason
- Firma dei parametri di query in OAuth 1.0
- I certificati dei client non vengono più ignorati quando verify=False
- Aggiunto supporto ai certificati openSUSE

0.13.1 (2012-06-07)

- E' possibile passare un file o un oggetto file-like come dati.
- Gli hook possono ritornare risposte che indicano errori.
- Fix su Response.text e Response.json per risposte senza corpo.

0.13.0 (2012-05-29)

- Rimozione di Requests.async in favore di grequests
- E' ora possibile disabilitare la persistenza dei cookie.
- Nuova immplementazione di safe_mode
- cookies.get ora supporta argomenti di default
- I cookie di sessione non sono salvati quando Session.request è invocata con return_response=False
- Variabili di ambiente: supporto a no_proxy.
- Miglioramenti a RequestsCookieJar.
- Fix a vari bachi.

0.12.1 (2012-05-08)

- Nuova propery `Response.json`.
- Possibilità di aggiungere upload di file sotto forma di stringhe.
- Fix baco out-of-range su iter_lines.
- Fix sulla dimensione di default di iter_content.
- Fix baco su redirezioni POST che contengono file.

0.12.0 (2012-05-02)

- SUPPORTO SPERIMANTALE A OAUTH!
- Migliorata interfaccia (dict-like) con i cookie CookieJar.
- Fix baco su lentezza dei chunk di contenuto che non vengono iterati.
- Spostato `pre_request` in una locazione più usabile.
- Nuovo hook `pre_send`.
- Encoding lazy di dati, parametri, files.
- Caricamento del bundle di certificati di sistema se `certify` non è disponibile.
- Pulizia del codice, fix.

0.11.2 (2012-04-22)

- Tentativo di utilizzo del bundle di certificati del sistema operativo se `certifi` non è disponibile.
- Fix baco su redirezione infinita su autenticazione Digest.
- Miglioramenti nell'upload di file Multi-part.
- Fix baco su decoding degli %encodings invalidi sugli URL.
- Se non c'è contenuto in una risposta non viene sollevato un errore la seconda volta che si prova a leggere il contenuto.
- Upload di dati nelle redirezioni.

0.11.1 (2012-03-30)

- Le redirect su POST ora non seguono la RFC e fanno come i browser: proseguono con una GET.
- Nuova configurazione `strict_mode` per disabilitare il nuovo comportamento di redirezione.

0.11.0 (2012-03-14)

- Supporto ai certificati privati su SSL
- Rimosso select.poll dal monkeypatching di Gevent
- Rimosso un generatore ridondante nell'encoding dei trasferimenti chunked
- Fix: Response.ok sollevava Timeout Exception in safe_mode

0.10.8 (2012-03-09)

- Fix sulla generazione di ValueError chunked
- Configurazione dei Proxy tramite variabili d'ambiente
- Semplificazione di iter_lines.
- Nuova configurazione *trust_env* per disabilitare i suggerimenti di sistema/ambiente.
- Soppressione degli errori sui cookie.

0.10.7 (2012-03-07)

- *encode_uri* = False

0.10.6 (2012-02-25)

- '=' è consentito nei cookies.

0.10.5 (2012-02-25)

- Fix baco su corpo delle risposte con 0 content-length.
- Nuovo async.imap.
- Fix crash su utilizzo di netrc.

0.10.4 (2012-02-20)

- Viene utilizzato netrc.

0.10.3 (2012-02-20)

- Le richieste HEAD non seguono più le redirect.
- raise_for_status() non solleva più gli errori 3xx.
- Gli oggetti Session sono serializzabili con Pickle.
- ValueError per gli URL con schema invalido.

0.10.2 (2012-01-15)

- Profondo miglioramento al quoting degli URL.
- Consentiti più valori per le chiavi dei cookie.
- Tentativo di fix per l'errore "Too many open files"
- Sostituzione degli errori Unicode alla prima passata, secondo passata non più necessario.
- Concatenamento di '/' agli URL con solo dominio prima dell'inserimento della query.
- Ora le Eccezioni ereditano da RuntimeError.
- Fix su upload binari uploads e autenticazione.
- Fix di bachi.

0.10.1 (2012-01-23)

- SUPPORTO A PYTHON 3!
- Abbandonato supporto a Python 2.5. (*Non retrocompatibile*)

0.10.0 (2012-01-21)

- `Response.content` ritorna ora solo bytes. (*Non retrocompatibile*)
- `Response.text` ora ritorna solo Unicode.
- Se non è specificato un `Response.encoding` e `chardet` è disponibile, `Response.text` tenta di indovinare l'encoding.
- Default sull'encoding ISO-8859-1 (Western) per i sottotipi di "text".
- Rimozione di *decode_unicode*. (*Non retrocompatibile*)
- Nuovo sistema ad hoook multipli.
- Nuovo metodo `Response.register_hook` per registrare hook all'interno della pipeline.
- `Response.url` ora ritorna Unicode.

0.9.3 (2012-01-18)

- Fix baco su verify=False di SSL (apparente sulle macchine Windows).

0.9.2 (2012-01-18)

- Metodo async.send è ora asincrono.
- Supporto per la corretta delimitazione degli stream di chunks.
- Argomento session per le classi Session.
- Stampa delle intere traceback, non solo dell'istanza dell'eccezione
- Fix response.iter_lines quando è in attesa delle prossima linea.
- Fix baco sull'autenticazione HTTP-digest con URI con query string.
- Fix nella sezione Hook degli Eventi.
- Aggiornamento di Urllib3.

0.9.1 (2012-01-06)

- danger_mode quando Response.raise_for_status() è automatico
- Refactoring di Response.iter_lines

0.9.0 (2011-12-28)

- La verifica SSL è fatta di default.

0.8.9 (2011-12-28)

- Fix sul packaging.

0.8.8 (2011-12-28)

- VERIFICA DEI CERTIFICATI SSL!
- Release di Certifi: la lista di certificati di Mozilla.
- Nuovo argoento 'verify' per le richieste SSL.
- Aggiornamento di Urllib3.

0.8.7 (2011-12-24)

- Fix sul troncamento dell'ultima riga con iter_lines
- Viene forzato safe_mode per le richieste asincrone
- Gestione più consistente delle eccezioni in safe_mode
- Fix sull'iterazione delle risposte nulle in safe_mode

0.8.6 (2011-12-18)

- Fix sui timeout a livello socket.
- Supporto all'autorizzazione per i Proxy.

0.8.5 (2011-12-14)

- Response.iter_lines!

0.8.4 (2011-12-11)

- Fix baco sul Prefetch.
- Aggiunta licenza per la versione installata.

0.8.3 (2011-11-27)

- Semplificazione del sistema di autenticazione per l'uso di oggetti callable.
- Nuovo parametro session per i metodi dell'API
- Visualizzazine dell'URl intero nei log.

0.8.2 (2011-11-19)

- Nuovo sistema di decoding Unicode, basato su *Response.encoding*, che è overridable
- Gestione corretta del quoting degli slash negli URL.
- I cookie contenenti `[`, `]`, e _ sono ora consentiti.

0.8.1 (2011-11-15)

- UFix sul path dell'URL nelle richieste
- Fix sui Proxy.
- Fix sui Timeouts.

0.8.0 (2011-11-13)

- Supporto al Keep-alive!
- Rimozione completa di Urllib2
- Rimozione completa di Poster
- Rimozione completa di CookieJars
- Nuovo modo di sollevare ConnectionError
- Safe_mode per il catching degli errori
- Prefetch dei parametri per i metodi di richiesta
- Supporto a OPTION
- Tuning asincrono delle dimensioni del pool
- Gli upload dei file inviano nomi reali
- Inserita dipendenza a urllib3

0.7.6 (2011-11-07)

- Fix baco su autenticazione Digest (concatenamento dei dati di query al path)

0.7.5 (2011-11-04)

- Response.content = None se c'è stata una risposta invalida.
- Gestione della redirezione in sede di autenticazione.

0.7.4 (2011-10-26)

- Fix sugli hook delle sessioni.

0.7.3 (2011-10-23)

- Fix sull'autenticazione Digest.

0.7.2 (2011-10-23)

- Fix su PATCH.

0.7.1 (2011-10-23)

- L'handling delle autenticazionidi urllib2 non è più usato.
- Rimozione completa di AuthManager, AuthObject, etc.
- Nuovo sistema di autenticazione basato su tuple e esecuzione di callback.

0.7.0 (2011-10-22)

- Le sessioni sono ora l'interfaccia primaria.
- InvalidMethodException è ora deprecata.
- Fix su PATCH.
- Nuovo sistema di configurazione (non si usano più setting globali)

0.6.6 (2011-10-19)

- Fix baco sui parametri di sessione (merging dei parametri).

0.6.5 (2011-10-18)

- Suite di test offline (veloce).
- Merging degli argomenti dei dizionari di sessione.

0.6.4 (2011-10-13)

- Decoding automatico di Unicode, sulla base degli header HTTP.
- Nuovo setting `decode_unicode`.
- Rimozione dei metodi `r.read/close`.
- Nuova interfaccia `r.faw` per un uso avanzato delle risposte.
- Espansione automatica degli header parametrizzati.

0.6.3 (2011-10-13)

- modulo `requests.async`, per inviare richieste asincrone con gevent.

0.6.2 (2011-10-09)

- GET/HEAD onorano allow_redirects=False.

0.6.1 (2011-08-20)

- Migliorata l'esperienza d'uso degli status code `\o/`
- Specifica del numero massimo di redirezioni (`settings.max_redirects`)
- Supporto completo agli URL Unicode
- Supporto alle redirezioni protocol-less.
- E' possibile inviare tipologie arbitrarie di richiesta.
- Fix di bachi

0.6.0 (2011-08-17)

- Nuovo sistema per l'hooking delle callback
- Nuovi oggetti sessioni permanenti e nuovo context manager
- Gestione trasparente Dict-cookie
- Oggetto per il riferimento agli status code
- Rimosso Response.cached
- Aggiunto Response.request
- Tutti gli argomenti sono kwargs
- Supporto alle redirezioni relative
- Miglioramenti alla gestione degli HTTPError
- Migliorato il testing di HTTPS
- Fix di bachi

0.5.1 (2011-07-23)

- Supporto ai Nomi a Dominio Internazionali!
- Accesso agli headers senza dover recuperare l'intero corpo (`read()`)
- Uso di liste come dicts per i parametri
- Aggiunta autenticazione Basic Forzata
- L'autenticazione Basic Forzata è ora quella di default
- `python-requests.org` è lo User-Agent header di default
- caching lower-case di CaseInsensitiveDict
- Fix baco su Response.history

0.5.0 (2011-06-21)

- Supporto a PATCH
- Support ai Proxy
- Suite di test con HTTPBin
- Fix sulle redirezioni
- Stream in scrittura con settings.verbose
- Querystrings per tutti i metodi
- Gli URLErrors (Connection Refused, Timeout, Invalid URLs) sono trattati come se fossero esplicitamente sollevati dalla libreria `r.requests.get('hwe://blah'); r.raise_for_status()`

0.4.1 (2011-05-22)

- Migliorata la gestione delle redirezioni
- Nuovo parametro 'allow_redirects' per seguire le redirezioni non-GET/HEAD
- Refactoring del modulo dei settings

0.4.0 (2011-05-15)

- Response.history: lista di risposte in seguito a redirezioni
- I dizionari degli header ora sono case-insensitive!
- URL Unicode

0.3.4 (2011-05-14)

- Fix baco di ricorsione nella HTTPAuthentication di Urllib2 (Basic/Digest)
- Refactoring interno
- Fix baco di upload di bytes

0.3.3 (2011-05-12)

- Timeout sulle richieste
- Dati url-encoded con Unicode
- Gestore e modulo per la configurazione del contesto

0.3.2 (2011-04-15)

- Decompressione automatica del contenuto GZip
- Supporto per AutoAuth Support per l'autenticazione HTTP tramite tupla

0.3.1 (2011-04-01)

- Modifiche ai cookie
- Response.read()
- Fix su Poster

0.3.0 (2011-02-25)

- Cambiamento automatico dell'API di Autenticazione
- Parametrizzazione più intelligente delle URL query
- E' possibile uploadare file e POST-are dati contemporaneamente
- **Nuovo sistema di gestione dell'autenticazione**
  - Sistema Basic HTTP più semplice
  - Supporta tutti i meccanismi builtin di autenticazione in urllib2
  - Possibilità di usare handler di autenticazione custom

0.2.4 (2011-02-19)

- Supporto per Python 2.5
- Supporto per PyPy-c v1.4
- Test per Auto-Autenticazione
- Migliorato il costruttore degli oggetti di tipo Request

0.2.3 (2011-02-15)

- **Nuovi metodi per HTTPHandling**
  - Response.__nonzero__ (false se ho HTTP Status di errore)
  - Response.ok (True se ho HTTP Status OK)
  - Response.error (Logga un HTTPError se ho HTTP Status di errore)
  - Response.raise_for_status() (Solleva un HTTPError archiviato)

0.2.2 (2011-02-14)

- Le richieste vengono comunque gestite in caso di un HTTPError. (Issue #2)
- Supporto al monkeypatching con Eventlet e Gevent.
- Supporto ai Cookie (Issue #1)

0.2.1 (2011-02-14)

- Aggiunto l'attributo file alle richieste POST e PUT per upload di file multipart-encoded.
- Aggiunto l'attributo Request.url per il contesto e le redirect

0.2.0 (2011-02-14)

- Nascita!

0.0.1 (2011-02-13)

- Frustrazione
- Idea iniziale

Processo e regole di release

Nuovo nella versione v2.6.2.

Dalla prima release successiva alla `v2.6.2`, le regole che seguono verranno usate per aiutare il core team di Requests a produrre una nuova release.

Major Release

Una major release include dei cambiamenti non retrocompatibili. Quando ad essa viene data una versione, il suo formato è `vX.0.0`. Ad esempio, se la release precedente fosse la `v10.2.7`, la successiva sarebbe la `v11.0.0`.

I cambiamenti non retrocompatibili distruggono la compatibilità con le versioni precedenti. Se nel progetto si modificasse l'attributo `text` di una `Response` o di un metodo, questo sarebbe possibile solamente in una major release.

Le major release possono anche includere risoluzioni a vari bachi e aggiornamenti a package di vendor esterni. Gli sviluppatori del core team di Requests si impegnano per offrire una buona user experience: ciò significa che ci impegnamo anche a mantenere la retrocompatibilità il più possibile. Le major release saranno poco frequenti e richiederanno delle ottime giustificazioni prima di essere considerate.

Minor Release

Una minor release è priva di modifiche non retrocompatibili ma può includere risoluzioni a vari bachi e aggiornamenti a package di vendor esterni. Ad esempio, se la release precedente fosse la `v10.2.7`, una minor release sarebbe versionata come `v10.3.0`.

Le minor release saranno retrocompatibili con le releease che hanno lo stesso numero di versione major. Altrimenti detto, tutte le versioni che iniziano con `v10.` dovrebbero essere compatibili tra di loro.

Hotfix Release

Una hotfix release include solo risoluzioni a bachi che non erano state incluse quando è stata rilasciata la versione precedente del progetto. Se la versione precedente di Requests fosse la `v10.2.7`, una hotfix release sarebbe versionata come `v10.2.8`.

Dalla versione `v2.6.2`, le hotfix release **non** includono aggiornamenti a package di vendor esterni.

Ragionamento

Nelle serie di release 2.5 e 2.6, il core team di Requests ha aggiornato le dipendenze esterne, causando a se stesso e agli utenti di bei mal di testa. Per evitare questo genere di situazioni, stiamo costruendo un corpo di regole per stabilire concretamente le aspettative sulle release.

CAPITOLO 5

Documentazione dell'API

Se cerchi informazioni su una funzione, una classe o un metodo in particolare, questa parte della documentazione è il posto giusto.

API per gli sviluppatori

Questa parte della documentazione copre tutte le interfacce di Requests. Le interfacce con librerie esterne sono documentate a grandi linee e vengono forniti link alla loro documentazione canonica.

Interfaccia principale

Tutte le funzionalità di Requests sono accessibili da questi 7 metodi. Tutti ritornano un'istanza dell'oggetto *Response*.

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

Parametri

- **method** – method for the new *Request* object.
- **url** – URL for the new *Request* object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the *Request*.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json data to send in the body of the *Request*.
- **headers** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **cookies** – (optional) Dict or CookieJar object to send with the *Request*.
- **files** – (optional) Dictionary of `'name': file-like-objects` (or `{'name': ('filename', fileobj)}`) for multipart encoding upload.

- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) 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.

Ritorna *Response* object

Tipo di ritorno *requests.Response*

Usage:

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

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

Parametri

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

Ritorna *Response* object

Tipo di ritorno *requests.Response*

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

Parametri

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

Ritorna *Response* object

Tipo di ritorno *requests.Response*

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

Parametri

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

Ritorna `Response` object

Tipo di ritorno *requests.Response*

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

Parametri

- **`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.

Ritorna `Response` object

Tipo di ritorno *requests.Response*

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

Parametri

- **`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.

Ritorna `Response` object

Tipo di ritorno *requests.Response*

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

Parametri

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

Ritorna `Response` object

Tipo di ritorno *requests.Response*

Classi di livello più basso

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

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

Parametri

- **`method`** – HTTP method to use.
- **`url`** – URL to send.
- **`headers`** – dictionary of headers to send.
- **`files`** – dictionary of {filename: fileobject} files to multipart upload.
- **`data`** – the body to attach to the request. If a dictionary is provided, form-encoding will take place.

- **json** – json for the body to attach to the request (if data is not specified).
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

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

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

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

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

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

apparent_encoding
The apparent encoding, provided by the chardet library

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

Note: Should not normally need to be called explicitly.

content
Content of the response, in bytes.

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

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

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

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

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

is_permanent_redirect
True if this Response one of the permanant versions of redirect

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

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

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

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

Nota: This method is not reentrant safe.

---

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

Parametri ****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). Use of `raw` requires that `stream=True` be set on the request.

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

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

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

text
Content of the response, in unicode.

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

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

url = None
Final URL location of Response.

Sessioni

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

Provides cookie persistence, connection-pooling, and configuration.

Basic Usage:

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

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

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

Parametri

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

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

Parametri

- **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.

Parametri

- **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.

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

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.

Parametri

- **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.

Parametri

- **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*, *json=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parametri

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

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

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

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.

Parametri

- **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.

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

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

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

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

Parametri

- **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*.
- **json** – (optional) json to send in the body of the *Request*.
- **headers** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **cookies** – (optional) Dict or CookieJar object to send with the *Request*.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol 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*, ***adapter_kwargs*)
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.

Parametri

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

Usage:

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

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

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

Parametri

- **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*

Parametri

- **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*.

Parametri

- **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*.

Parametri

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

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

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

Parametri

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

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

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

Parametri

- **proxies** – The url of the proxy being used for this request.
- **kwargs** – Optional additional keyword arguments.

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

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

Parametri

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

Ritorna ProxyManager

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

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

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

Parametri

- **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.

Parametri

- **request** – The *PreparedRequest* being sent.
- **stream** – (optional) Whether to stream the request content.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **verify** – (optional) Whether to verify SSL certificates.
- **cert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

Autenticazione

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

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

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

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

Eccezioni

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

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

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

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

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

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

Requests that produced this error are safe to retry.

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

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

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

Ricerca degli status code

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

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

>>> requests.codes.teapot
418

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

Cookie

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

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

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

Parametri

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

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

Parametri

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

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

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

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

Unlike a regular CookieJar, this class is pickleable.

Avvertimento: dictionary operations that are normally O(1) may be O(n).

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

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

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

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

Raises KeyError if no matching cookie exists.

clear_expired_cookies()
Discard all expired cookies.

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

clear_session_cookies()
Discard all session cookies.

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

copy()
Return a copy of this RequestsCookieJar.

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

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

Avvertimento: operation is O(n), not O(1).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Encoding

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

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

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

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

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

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

Tried:

1.charset from content-type

2.fall back and replace all unicode characters

Classi

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

apparent_encoding
The apparent encoding, provided by the chardet library

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

Note: Should not normally need to be called explicitly.

content
Content of the response, in bytes.

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

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

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

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

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

is_permanent_redirect
True if this Response one of the permanant versions of redirect

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

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

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

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

Nota: This method is not reentrant safe.

---

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

Parametri ****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). Use of `raw` requires that `stream=True` be set on the request.

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

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

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

text
Content of the response, in unicode.

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

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

url = None
Final URL location of Response.

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

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

Parametri

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

Usage:

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

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

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

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

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

Generated from either a *Request* object or manually.

Usage:

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

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

body = None
request body to send to the server.

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

headers = None
dictionary of HTTP headers.

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

method = None
HTTP verb to send to the server.

path_url
Build the path URL to use.

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

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

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

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

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

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

`prepare_hooks`(*hooks*)
Prepares the given hooks.

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

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

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

`url` = None
HTTP URL to send the request to.

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

Provides cookie persistence, 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

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

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

Parametri

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

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

Parametri

- **`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.

Parametri

- **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.

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

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.

Parametri

- **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.

Parametri

- **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*, *json=None*, ***kwargs*)
Sends a POST request. Returns *Response* object.

Parametri

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.

- **json** – (optional) json to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

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

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

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.

Parametri

- **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.

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

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

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

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

Parametri

- **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*.
- **json** – (optional) json to send in the body of the *Request*.
- **headers** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **cookies** – (optional) Dict or CookieJar object to send with the *Request*.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.

- **proxies** – (optional) Dictionary mapping protocol 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*, ***adapter_kwargs*)
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.

Parametri

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

Usage:

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

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

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

Parametri

- **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*

Parametri

- **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*.

Parametri

- **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*.

Parametri

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

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

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

Parametri

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

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

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

Parametri

- **proxies** – The url of the proxy being used for this request.
- **kwargs** – Optional additional keyword arguments.

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

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

Parametri

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

Ritorna ProxyManager

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

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

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

Parametri

- **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.

Parametri

- **request** – The *PreparedRequest* being sent.
- **stream** – (optional) Whether to stream the request content.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **verify** – (optional) Whether to verify SSL certificates.
- **cert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

Migrazione alla versione 1.x

Questa sezione illustra le principali differenze tra le versioni 0.x e 1.x e il suo scopo è facilitare l'upgrading.

Cambiamenti nell'API

- `Response.json` ora è una callable e non più una property di una response.

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()   # Questa *chiamata* lancia un'eccezione se il decoding del JSON␣
↪fallisce
```

- La `Session` API è cambiata. Gli oggetti Sessione non accettano più parametri. `Session` ora è indicata con la lettera maiuscola ma può ancora essere istanziata tramite `session` con l'iniziale minuscola per retrocompatibilità.

```
s = requests.Session()    # prima, le sessioni accettavano parametri
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- Sono stati rimossi tutti gli hook delle richieste tranne 'response'.
- Le funzioni di supporto all'autenticazione sono state spostate in moduli separati. Si vedano requests-oauthlib e requests-kerberos.
- Il nome del parametro per le richieste streaming è cambiato da `prefetch` a `stream` e la logica è stata invertita. In più, `stream` viene ora richiesto per la lettura delle risposte raw.

```
# nella versione 0.x, il passaggio di prefetch=False avrebbe dato lo stesso␣
↪risultato
r = requests.get('https://github.com/timeline.json', stream=True)
for chunk in r.iter_content(8192):
    ...
```

- Il parametro `config` della funzione requests è stato rimosso. Alcune delle opzioni vengono ora configurate su una `Session` (es: keep-alive e numero massimo di redirezioni). L'opzione di verbosità dovrebbe essere gestita tramite configurazione del logging.

```
import requests
import logging

# queste due righe abilitano il debugging a livello di httplib (requests->urllib3-
↪>httplib)
# si vedranno la REQUEST, con HEADERS e DATA, e la RESPONSE con gli HEADERS ma␣
↪senza DATA.
# l'unico elemento mancante è il response.body che non viene loggato.
import httplib
httplib.HTTPConnection.debuglevel = 1

logging.basicConfig() # occorre inizializzare il logging, altrimenti non si vedrà␣
↪nulla delle richieste
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')
```

Licenza

Una differenza fondamentale che non impatta l'API è il cambio di licenza da ISC ad Apache 2.0 . La licenza Apache 2.0 garantisce che i contributi a Requests siano anch'essi licenziati con Apache 2.0.

Migrazione alla versione 2.x

Rispetto alla release 1.0, ci sono state relativamente poche modifiche non retrocompatibili ma ci sono ancora alcuni problemi di questa major release di cui è bene sapere.

Per ulteriore dettaglio sulle modifiche in questa release, comprese nuove APIs, link alle issues relative su GitHub e il fixing di alcuni bachi, si legga il blog di Cory.

Modifiche nell'API

- Ci sono stati un paio di modifiche sul modo con cui Requests gestisce le eccezioni. `RequestException` ora è una sottoclasse di `IOError` invece che di `RuntimeError` perchè così viene meglio indicato il tipo di situazione erronea. In più, ora una sequenza di escaping degli URL non valida solleva l'istanza di una sottoclasse di `RequestException` piuttosto che di `ValueError`.

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

  Infine, le eccezioni di tipo `httplib.IncompleteRead` causate da un encoding scorretto sui chunks sollevano ora un'istanza di `ChunkedEncodingError` di Requests.

- L'API per i proxy è stata modificata lievemente. Viene ora richiesto di specificare lo schema per l'URL dei proxy.

```
proxies = {
  "http": "10.10.1.10:3128",    # ora va utilizzato http://10.10.1.10:3128
}

# Nelle versioni 1.x di requests, questo codice era eseguito senza problemi mentre
# nelle versioni 2.x solleva un'eccezione requests.exceptions.MissingSchema
requests.get("http://example.org", proxies=proxies)
```

Modifiche al comportamento

- Le chiavi nel dizionario `headers` sono ora stringhe native in tutte le versioni di Python es. bytestrings in Python 2 e unicode in Python 3. Se le chiavi non sono stringhe native (unicode in Python2 oppure bytestrings in Python 3), queste vengono convertite in stringhe native utilizzando UTF-8 come encoding.

CAPITOLO 6

Documentazione per i collaboratori

Se desiderate contribuire al progetto, questa parte della documentazione è il punto di partenza.

Guida per i collaboratori

Se state leggendo questa pagina probabilmente siete interessati a contribuire a Requests. Come prima cosa, vi diciamo: grazie! I progetti open source vivono e muoiono sul supporto che ricevono dalle altre persone, e il fatto stesso che stiate considerando di supportare Requests è molto generoso da parte vostra.

Questo documento individua alcune linee guida e alcuni consigli per contribuire a Requests. Se state considerando di contribuire, cominciate a leggere bene questo documento e a farvi un'idea di come si fa a contribuire al progetto. Per ogni domanda, sentitevi liberi di contattare Ian Cordasco o Cory Benfield, i maintainer principali.

Questa guida è divisa in sezioni sulla tipologia di contributo che state considerando di fare, con una sezione che tratta le linee guida generali per tutti i collaboratori.

Per tutti i tipi di contributi

Siate cordiali

Siate cordiali o state lontani.

Requests ha una regola importante che governa tutte le tipologie di contributo, inclusi i bug report o le richieste di nuove feature. Questa regola d'oro è siate cordiali o state lontani. **Tutti i contributi sono bene accetti**, fintanto che tutte le persone coinvolte vengono trattate con rispetto.

Cercate un feedback rapido

Se contribuite, non mantenete privato il vostro lavoro fino a che non è perfetto e completo. Se cercate di avere feedback sul vostro lavoro il prima possible, date una mano a tutti. Inviare una versione draft e incompleta del vostro lavoro

per ottenere feedback non pregiudicherà in alcun modo le vostre chance di ottenere che esso venga accettato, anzi può evitarvi di fare un mucchio di lavoro su un contributo che in realtà potrebbe non essere adatto al progetto.

Idoneità dei contributi

Il project maintainer ha l'ultima parola sull'idoneità di un contributo al progetto Requests. Tutti i contributi saranno considerati, ma certe volte capiterà che alcuni contributi saranno rigettati perchè non sono adatti al progetto.

Se il vostro contributo viene rigettato, non disperate! Se avete seguito queste linee guida avrete la miglior chance di vedere accettato il vostro prossimo contributo.

Contributi al codice

Passi

Se contribuite al codice, servitevi di questa checklist:

1. Forkate il repository su GitHub.
2. Fate girare i test per stabilire se passano tutti nel vostro ambiente. In caso negativo, cercate di capire perchè falliscono. Se non riuscite a capirlo da soli, compilate un bug report seguendo i passi descritti in questo documento: *Bug Report*.
3. Scrivete dei test che dimostrino l'esistenza del baco o la mancanza della nuova feature. Accertatevi che questi test falliscano.
4. Apportate le vostre modifiche al codice.
5. Fate girare di nuovo l'intera test suite, avendo conferma che tutti i test passano *inclusi quelli che avete aggiunto in precedenza*.
6. Inviate una Pull Request su GitHub sulla `master` branch del repository principale. Le Pull Requests di GitHub sono il metodo che usiamo nel progetto per inviare i contributi di codice.

Le prossime sottosezioni entrano più in dettaglio su alcuni dei punti sopra illustrati.

Revisione del codice

I contributi non saranno mergiati nella codebase finchè il codice non sarà rivisto. Dovreste implementare ogni feedback che ricevete a livello di revisione del codice, a meno che non siate in forte disaccordo con essi. In tale caso, dovreste spiegare il perchè in maniera chiara e pacata. Se, anche dopo averlo fatto, il feedback non viene modificate, potete o implementare il feedback oppure ritirare il vostro contributo.

Nuovi collaboratori

Se non vi siete mai accostati all'Open Source o siete relativamente nuovi, benvenuti! Requests vuol essere un'introduzione gentile al mondo dell'Open Source. Se siete indecisi su come contribuire al meglio, perfavore prendete in considerazione di mandare una e-mail ad uno dei maintainer (riportati sopra) e chiedere aiuto.

Leggete anche la sezione *Cercate un feedback rapido*.

Contributi alla documentazione

Le migliorie alla documentazione sono sempre benvenute! I file della documentazione risiedono nella cartella `docs/` della codebase. Sono scritti in reStructuredText, e utilizziamo Sphinx per generare l'intero corpo della documentazione.

Quando contribuite alla documentazione, siete pregati di seguirne lo stile. Ciò significa un limite di 79 colonne nei file di testo e uno stile di prosa semi-formale.

Bug Report

I bug report sono di vitale importanza! Prima di notificare un bug, perfavore leggete le issues di GitHub, **sia quelle aperte che quelle chiuse**, per avere conferma che il baco non sia già stato segnalato in passato. I bug report duplicati sono una grossa perdita di tempo per gli altri collabortori e dovrebbero essere assolutamente evitati.

Richiesta di nuove feature

Requests è in un perpetuo stato di congelamento delle feature. I maintainers credono che la libreria contenga tutte le principali feature utili alla stragrande maggioranza degli utenti.

Se credete che manchi una feature, siate liberi di inviare una richiesta ma sappiate che con ogni probabilità la richiesta non sarà accettata.

Filosofia di sviluppo

Requests è una libreria open ma con una propria visione, creata da uno sviluppatore open ma con una propria visione.

Stile di gestione del progetto

Kenneth Reitz è il BDFL. Ha l'ultima parola su ogni decisione relativa al progetto Requests. Kenneth è responsabile della direzione e della forma che la libreria prende. Oltre a prendere decisioni sulla base dei suoi meriti tecnici, è responsabile delle decisioni sulla filosofia di sviluppo di Requests. Solo Kenneth può mergiare codice in Requests.

Ian Cordasco e Cory Benfield sono i collaboratori fondamentali. Sono responsabili della valutazione della prorità dei bug report, della revisione delle pull request e di mantenere Kenneth aggiornato con gli sviluppi sulla libreria. La gestione giornaliera del progetto è fatta da loro. Sono responsabili di giudicare se la richiesta di una nuova feature ha la possibilità di essere accettata da Kenneth. Non hanno l'autorità di cambiare il codice o di mergiare modifiche, anche se possono modificare la documentazione. La loro opinione non è quella definitiva.

Valori

- La semplicità è sempre meglio della funzionalità.
- Date ascolto a tutti, e poi ignorateli.
- L'API è l'unica cosa che conta. Tutto il resto è secondario.
- Realizzate il caso d'uso che accontenta il 90% degli utenti. Ignorate i piagnucoloni.

Semantic Versioning

Per molti anni la community dell'open source è stata flagellata dalla distonia sui numeri di versione. Il significato dei numeri di versione varia così tanto da progetto a progetto da farne in pratica perdere il significato.

Requests adotta il Semantic Versioning. Questo manifesto cerca di porre fine alla follia del versioning attraverso un piccolo insieme di linee guida che voi ed i vostri colleghi potete usare nei vostri prossimi progetti.

Libreria Standard?

Requests non rientra in alcun piano *attivo* per l'inclusione nella libreria standard Python. Questa decisione è stata discussa al tempo con Guido ed alcuni sviluppatori del core team.

Sostanzialmente, la libreria standard Python sarebbe la morte della libreria Requests. Sarebbe opportuno includere un modulo nella libreria standard solo quando uno sviluppo attivo non è più necessario.

Requests ha da poco raggiunto la versione v1.0.0. Questa grande milestone ha segnato uno step significativo nella giusta direzione.

Pacchetti per le distribuzioni Linux

Sono state create distribuzioni di Requests per molti repository Linux, tra cui: Ubuntu, Debian, RHEL e Arch.

Queste distribuzioni a volte sono fork divergenti, oppure non sono mantenute aggiornate con il codice e i bugfix più recenti. PyPI (e i suoi mirror) e GitHub sono i canali ufficiali di distribuzione; le alternative non sono supportate dal progetto Requests.

Come contribuire

Requests è un progetto attivamente mantenuto, e i contributi sono bene accetti!

1. Verificate se ci sono issue aperte o apritene una nuova per iniziare la discussione su un baco. Le issue marcata con il tag Contributor Friendly sono l'ideale per chi non è ancora molto familiare con la codebase.
2. Forkate il repository su GitHub e iniziate ad apportare le vostre modifiche su una nuova branch.
3. Scrivete un test che dimostra che il baco è stato risolto.
4. Inviate una pull request e rompete l'anima ai maintainer finchè questa non viene mergiata e resa pubblica. :) Non scordatevi di aggiungere il vostro nome al file AUTHORS.

Congelamento delle feature

Dalla versione v1.0.0, le feature di Requests sono state congelate. Le richieste per nuove feature e le Pull Request che le implementano non saranno più accettate.

Dipendenze per lo sviluppo

Dovrete installare py.test per far girare la test suite di Requests:

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

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

Ambienti di runtime

Requests al momento supporta le seguenti versioni di Python:

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

Il supporto per Python 3.1 e 3.2 può essere abbandonato in futuro.

Google App Engine non sarà mai supportato ufficialmente. Le Pull Requests di compatibilità saranno accettate solo se non complicheranno la codebase.

Siete fuori di testa?

- Il supporto a SPDY sarebbe fantastico. Nessuna estensione C però.

Repackaging Downstream

Se redistribuite Requests nei vostri pacchetti, ricordate che dovete anche redistribuire il file `cacerts.pem` affinchè SSL funzioni correttamente.

Autori

Requests è scritto e mantenuto da Kenneth Reitz e vari collaboratori:

I Guardiani dei Tre Cristalli

- Kenneth Reitz <me@kennethreitz.org> @kennethreitz, Guardiano del Cristallo Principale.
- Cory Benfield <cory@lukasa.co.uk> @lukasa
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24

Urllib3

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

Patch e Suggerimenti

- Diversi membri di 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)
- Matt Giuca
- Adam Tauber
- Honza Javorek
- Brendan Maguire <maguire.brendan@gmail.com>
- Chris Dary
- Danver Braganza <danverbraganza@gmail.com>
- Max Countryman

- Nick Chadwick
- Jonathan Drosdeck
- Jiri Machalek
- Steve Pulec
- Michael Kelly
- Michael Newman <newmaniese@gmail.com>
- Jonty Wareing <jonty@jonty.co.uk>
- Shivaram Lingamneni
- Miguel Turner
- Rohan Jain (@crodjer)
- Justin Barber <barber.justin@gmail.com>
- Roman Haritonov (@reclosedev)
- Josh Imhoff <joshimhoff13@gmail.com>
- Arup Malakar <amalakar@gmail.com>
- Danilo Bargen (@dbrgn)
- Torsten Landschoff
- Michael Holler (@apotheos)
- Timnit Gebru
- Sarah Gonzalez
- Victoria Mo
- Leila Muhtasib
- Matthias Rahlf <matthias@webding.de>
- Jakub Roztocil <jakub@roztocil.name>
- Rhys Elsmore
- André Graf (@dergraf)
- Stephen Zhuang (@everbird)
- Martijn Pieters
- Jonatan Heyman
- David Bonner <dbonner@gmail.com> (@rascalking)
- Vinod Chandru
- Johnny Goodnow <j.goodnow29@gmail.com>
- Denis Ryzhkov <denisr@denisr.com>
- Wilfred Hughes <me@wilfred.me.uk>
- Dmitry Medvinsky <me@dmedvinsky.name>
- Bryce Boe <bbzbryce@gmail.com> (@bboe)
- Colin Dunklau <colin.dunklau@gmail.com> (@cdunklau)

- Bob Carroll <bob.carroll@alum.rit.edu> (@rcarz)
- Hugo Osvaldo Barrera <hugo@osvaldobarrera.com.ar> (@hobarrera)
- Łukasz Langa <lukasz@langa.pl>
- Dave Shawley <daveshawley@gmail.com>
- James Clarke (@jam)
- Kevin Burke <kev@inburke.com>
- Flavio Curella
- David Pursehouse <david.pursehouse@gmail.com> (@dpursehouse)
- Jon Parise
- Alexander Karpinsky (@homm86)
- Marc Schlaich (@schlamar)
- Park Ilsu <daftonshady@gmail.com> (@daftshady)
- Matt Spitz (@mattspitz)
- Vikram Oberoi (@voberoi)
- Can Ibanoglu <can.ibanoglu@gmail.com> (@canibanoglu)
- Thomas Weißschuh <thomas@t-8ch.de> (@t-8ch)
- Jayson Vantuyl <jayson@aggressive.ly>
- Pengfei.X <pengphy@gmail.com>
- Kamil Madac <kamil.madac@gmail.com>
- Michael Becker <mike@beckerfuffle.com> (@beckerfuffle)
- Erik Wickstrom <erik@erikwickstrom.com> (@erikwickstrom)
- (@podshumok)
- Ben Bass (@codedstructure)
- Jonathan Wong <evolutionace@gmail.com> (@ContinuousFunction)
- Martin Jul (@mjul)
- Joe Alcorn (@buttscicles)
- Syed Suhail Ahmed <ssuhail.ahmed93@gmail.com> (@syedsuhail)
- Scott Sadler (@ssadler)
- Arthur Darcet (@arthurdarcet)
- Ulrich Petri (@ulope)
- Muhammad Yasoob Ullah Khalid <yasoob.khld@gmail.com> (@yasoob)
- Paul van der Linden (@pvanderlinden)
- Colin Dickson (@colindickson)
- Claudio Sparpaglione (@csparpa)

Indice del modulo Python

r

Indice

T

U

V