Requests Documentation

Release 0.13.9

Kenneth Reitz

October 16, 2014

Contents

Release v0.13.9. (*Installation*)

Requests est une librairie HTTP *sous licence ISC*, écrite en Python, pour les êtres humains.

Le module **urllib2** de la librairie standard fournit toutes les fonctionnalités dont vous avez besoin, mais son API est complètement **moisie**. Il a été crée dans une autre époque - lorsque le web était autre chose, et demande une *énorme* quantité de travail (voire des remplacements de méthodes) pour achever les plus simples tâches.

Ca ne devrait pas se passer comme ca. Pas en 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, ...}
```

Voir le même code, sans Requests.

Requests reprend tout les travaux autour de Python HTTP/1.1 - et rend l'intégration avec des webservices très facile. Pas besoin d'ajouter des querystrings à vos URLs manuellement, ou d'encoder vous-même vos datas pour les POST. Les Keep-alive et le groupement des connexions HTTP sont 100% automatisés, grâce à urllib3, qui est directement intégré à Requests.

CHAPTER 1

Témoignages

Kippt, Heroku, PayPal, Transifex, Native Instruments, The Washington Post, Twitter, Inc, Readability, and Federal US Institutions utilisent Requests en interne. Le module a été installé plus de 100,000 fois via PyPI.

Armin Ronacher Requests est l'exemple parfait de ce que peut être une belle API grâce à la bonne dose d'abstraction.

Matt DeBoard Je vais me tatouer le module Python requests de @kennethreitz sur le corps, d'une façon ou d'une autre. partout.

Daniel Greenfeld Je viens de passer de 1200 lignes de code spaghetti à 10 lignes de code grâce au module requests de @kennethreitz. Aujourd'hui est un grand jour.

Kenny Meyers HTTP avec Python: si vous avez un doute, ou pas, utilisez Requests. Beau, simple, pythonic.

CHAPTER 2

Fonctionnalités

Requests est prêt pour le web d'aujourd'hui

- Gestion domaines et URLS internationales
- Keep-Alive & Groupement de connections (Pooling)
- Sessions et Cookies persistants
- Verification SSL
- Authentifications Basic/Digest ou personnalisées
- Gestion élégante des Cookies clé/valeur
- Décompression automatique
- Corps des réponses en unicode
- Upload de fichiers multiparts
- Timeouts de connexion
- supprt de `.netrc`
- Thread-safe.

CHAPTER 3

Guide utilisateur

Cette partie de la documentation commence avec des informations de fond sur Requests, puis nous présentons étape par étape les instructions pour tirer le meilleur parti de Requests.

3.1 Introduction

3.1.1 Philosophie

Requests a été développé en gardant en tête les idiomes de la **PEP 20**

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

Toutes les contributions à Requests doivent suivre ces règles importantes.

3.1.2 License ISC

Beaucoup de projets open source que vous pouvez trouver aujourd'hui sont distribués sous Licence GPL. Alors que la GPL a eu ses heures de gloire, ce n'est pas toujours la licence idéale pour les projets open source.

Un projet distribué sous licence GPL ne peut pas être utilisé dans des produits commerciaux sans que le produit lui-même soit également open-source.

Les licences MIT, BSD, ISC, et Apache2 sont de très bonnes alternatives à la GPL qui permettent qu'un projet open source soit utilisé librement dans des projets propriétaires et closed-source.

Requests est distribué sous les termes de la License ISC.

3.1.3 Licence Requests

3.2 Installation

Cette partie de la documentation traite de l'installation de Requests. La première étape pour utiliser une librairie est de l'installer correctement.

3.2.1 Distribute & Pip

Requests s'installe simplement avec pip:

```
$ pip install requests
```

ou, avec easy_install:

```
$ easy_install requests
```

Mais, ce n'est pas conseillé.

3.2.2 Miroir Cheeseshop

Si Cheeseshop n'est pas accessible, vous pouvez également installer Requests depuis le miroir personnel Cheeseshop de Kenneth Reitz:

```
$ pip install -i http://pip.kennethreitz.com/simple requests
```

3.2.3 Obtenir le code

Requests est activement développé sur GitHub, ou le code est toujours disponible.

Vous pouvez cloner le dépot public:

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

Télécharger le tarball:

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

Ou, télécharger le zipball:

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

Une fois que vous avez une copie de la source, vous pouvez l'intégrer dans votre package Python, ou l'installer facilement dans votre dossier site-packages:

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

3.3 Quickstart

Impatient de commencer? Cette page vous donne une bonne introduction pour démarrer avec Requests. Ceci implique que vous ayez déja Requests installé. Si ce n'est pas la cas, suivez la section *Installation*.

Premièrement, vérifier que:

- Requests est *installé*
- Requests est *à jour*

Commençons avec des exemples et des cas simples.

3.3.1 Créer une requête

Créer une requête standard avec Request est très simple.

Commençons par import le module Requests:

```
>>> import requests
```

Maintenant, essayons de récupérer une page web. Pour cette exemple, récupérons la timeline publique de github:

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

Nous récupérons alors un objet `Response` appelé `r`. Celui-ci contient toutes les informations dont nous avons besoin.

L'API simple de Requests permet d'effectuer toute sorte de requête HTTP très simplement. Par exemple, pour faire une requete HTTP POST:

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

Pratique, non?

Et pour les autres types de requêtes: PUT, DELETE, HEAD et OPTIONS ?

C'est tout aussi simple:

```
>>> r = requests.put("http://httpbin.org/put")
>>> r = requests.delete("http://httpbin.org/delete")
>>> r = requests.head("http://httpbin.org/get")
>>> r = requests.options("http://httpbin.org/get")
```

Jusqu'ici tout va bien, et c'est juste un petit aperçu de ce que Requests peut faire.

3.3.2 Passer des paramètres dans les URLs

Il est fréquent d'avoir besoin de passer des données dans les URLs sous forme de paramètres. En construisant l'URL à la main, ces données devraient être fournies sous forme de clé/valeur dans l'URL après un point d'interrogation, par exemple `httpbin.org/get?key=val`. Requests vous permet de fournir ces arguments sous forme de dictionnaire, en utilisant l'argument `params`. Par exemple, si vous souhaitez passer `key1=value1` et `key2=value2` à `httpbin.org/get`, vous pouvez utiliser le code suivant:

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

Vous pouvez constater que l'URL a été correctement encodée en imprimant l'URL:

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

3.3.3 Contenu de la réponse

Nous pouvons lire le contenu de la réponse du serveur. Pour reprendre l'exemple de la timeline GitHub:

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

Lorsque vous effectuez une requête, Requests devine l'encodage de la réponse en fonction des en-têtes HTTP. Le texte décodé selon cet encodage est alors accesible via `r.text`. Pour consulter l'encoding que Requests a utilisé, et le modifier, utilisez la propriété `r.encoding`:

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

Lorsque vous modifiez cette propriété, Requests l'utilise la nouvelle valeur quand vous accédez à `r.text`.

Requests sait également utiliser des encodages personnalisés si jamais vous en avez besoin. Si vous avez crée votre propre encodage et l'avez enregistré avec le module `codecs`, utilisez simplement le nom du codec comme valeur de `r.encoding` et Requests gérera le décodage pour vous.

3.3.4 Réponse binaire

Pour les requêtes non-texte (fichiers binaires), vous pouvez accéder au contenu de la réponse sous forme de bytes:

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

Les réponse avec l'en-tête transfer-encoding à `gzip` et `deflate` sont automatiquement décodés pour vous.

Par exemple, pour créer une image à partir de données recues par une requête, vous pouvez utiliser le code suivant:

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

3.3.5 Réponse JSON

Si vous devez travailler avec des données JSON, Requests dispose dun décodeur intégré:

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

Si jamais le décodage échoue, `r.json` renverra simplement `None`.

3.3.6 Réponse brute

Dans de rares cas, si vous avez besoin d'accéder au contenu brut de la réponse du serveur, vous pouvez y accéder directement via `r.raw`:

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

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

3.3.7 En-têtes personnalisées

Si vous souhaitez ajouter des headers HTTP à une requête, passez simpement un object de type `dict` au pramètre `headers`.

Par exemple, pour spécifier un content-type dans l'exemple précédent:

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

3.3.8 Requêtes POST avancées

Typiquement, vous avec besoin d'envoyer des données encodées comme par exemple un formulaire HTML. Pour cela, on passe simplement un dictionnaire avec l'argument *data*. Votre dictionnaire de données sera automatiquement encodé comme un formulaire au moment de la requête:

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

Dans certains cas, vous ne souhaitez pas que les données soit encodées. Si vous passez une chaîne de caractères `string` à la place d'un objet `dict`, les données seront postées directement.

Par exemple, l'API GitHub v3 accepte les requêtes POST/PATCH avec des données JSON:

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

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

3.3.9 POST de fichiers Multipart

Requests simplifie l'upload de fichiers encodés en MultiPart:

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

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

Pour forcer le nom du fichier explicitement:

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

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

Vous pouvez également envoyer des chaînes de caractères en tant que fichier

```
>>> 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
{
  // ...snip... //
  "files": {
    "file": "some,data,to,send\\nanother,row,to,send\\n"
  },
  // ...snip... //
}
```

3.3.10 Codes de retour des réponses (status)

Nous pouvons vérifier le code de retour d'une réponse:

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

Requests fournit également un code de statut interne pour faciliter les vérifications :

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

Si nous faisons une mauvaise requête (code de retour autre que 200), nous pouvons lever une exception avec `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
```

Mais comme notre `status_code` pour `r` était `200`, lorsque l'on appele `raise_for_status()` nous obtenons:

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

Tout va bien.

3.3.11 En-têtes des réponses

On peut accéder aux en-têtes HTTP (headers) de la réponse du serveur via une simple dictionnaire Python:

```
>>> r.headers
{
    'status': '200 OK',
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json; charset=utf-8'
}
```

Ce dictionnaire est cependant particulier : Il est spécifique aux en-têtes HTTP. En effet, selon la RFC 2616, les en-têtes HTTP ne doivent pas être sensibles à la casse.

Donc, nous pouvons accéder aux en-têtes quelque soit la casse utilisée:

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

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

Si l'en-tête n'existe pas dans la Response, la valeur par défaut est `None`:

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

3.3.12 Cookies

Si la résponse contient des Cookies, vous pouvez y accéder rapidement:

```
>>> url = 'http://httpbin.org/cookies/set/requests-is/formidable'
>>> r = requests.get(url)

>>> r.cookies['requests-is']
'formidable'
```

Pour envoyer vos propres cookies au serveur, vous pouvez utiliser le paramètre `cookies`:

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

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

3.3.13 Authentification basique

La plupart des services web nécessitent une authentification. Il y a différents types d'authentification, mais la plus commune est l'authentification HTTP basique.

Utiliser l'authentification basique avec Requests est extrêmement simple:

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

Comme l'authentification HTTP basique est le standard le plus répandu, Requests fournit un raccourci pour cette méthode d'authentification:

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

Fournir de cette manière un tuple d'authentification au paramètre *auth* équivaut à utiliser l'exemple `HTTPBasicAuth` ci-dessus.

3.3.14 Authentification Digest

Une autre forme populaire de protection des web services est l'authentification Digest:

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

3.3.15 Authentification OAuth

Le projet requests-oauth de Miguel Araujo fournit une interface simple pour établir des connexions OAuth. La documentation et des exemples peuvent être trouvées sur git repository.

3.3.16 Redirections et Historique

Requests effectue automatiquement les redirections lorsque vous utilisez les méthodes GET et OPTIONS.

GutHub par exemple redirige tout le traffic HTTP vers HTTPS. Nous pouvons utiliser la méthode `history` de l'object Response pour tracker les redirections. Regardons ce qu'il se passe pour Github:

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

```
200
>>> r.history
[<Response [301]>]
```

La liste `Response.history` contient la liste des objets `Request` qui ont été crées pour compléter la requête. Cette liste est triée de la plus ancienne à la plus récente.

Si vous utilisez les methodes GET ou OPTIONS, vous pouvez désactiver la gestion des redirections avec le paramètre `allow_redirects`:

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

Si vous utilisez POST, PUT, PATCH, DELETE ou HEAD vous pouvez également autoriser explicitement les redirections:

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

3.3.17 Timeouts

Vous pouvez demander à Requests d'arrêter d'attendre après un certains nombre de secondes avec le paramètre `timeout`:

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

Note:

`timeout` affecte uniquement le délai de connection, pas le temps de téléchargment des réponses.

3.3.18 Erreurs et Exceptions

Dans le cas de problèmes de réseau (e.g. erreurs DNS, connexions refusées, etc), Requests lévera une exception `ConnectionError`.

Dans les rares cas de réponses HTTP invalides, Requests lévera une exception `HTTPError`.

Si une requête dépasse le temps d'attente, une exception `Timeout` est levée.

Si une requête dépasse le nombre maximum de redirections possibles configuré, une exception `TooManyRedirects` est levée.

Toutes les exceptions levées par Requests héritent de `requests.exceptions.RequestException`.

Vous pouvez vous référer à *Configuration API Docs* si vous souhaitez toujours lever des exceptions `HTTPError` avec l'option `danger_mode`, ou laisser Requests attraper la majorité des `requests.exceptions.RequestException` avec l'option `safe_mode`.

Prêt pour aller plus loin ? Visitez la section *avancée*.

3.4 Utilisation avancée

Ce document traite de quelques fonctionnalités avancées de Requests.

3.4.1 Objets Session

L'objet Session vous permet de conserver des paramètres entre plusieurs requêtes. Il permet également de conserver les cookies entre toutes les requêtes de la même instance Session.

Un objet Session a toutes les methodes de l'API Requests principale.

Pour conserver des cookies entre les requêtes:

```
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"}}'
```

Les Sessions peuvent aussi être utilisées pour fournir des valeurs par défaut aux requêtes:

```
headers = {'x-test': 'true'}
auth = ('user', 'pass')

with requests.session(auth=auth, headers=headers) as c:

    # 'x-test' et 'x-test2' sont envoyés
    c.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

Tous les dictionnaires que vous passez aux methodes de requête sont fusionnés avec les valeur de session déja définies. Les paramètres de la méthode surchargent les paramètres de session.

Supprimer une valeur d'un paramètre

Parfois vous voudrez supprimer des paramètres de session lors de vos requêtes. Pour cela, il suffit d'envoyer lors de l'appel de la méthode un dictionnaire dont les clés seraient les paramètres a supprimer et les valeurs seraient `None`. Ces paramètres seront alors automatiquement omis.

Toutes les valeurs contenues dans la session sont directement accessibles. Pour en savoir plus, cf *Session API Docs*.

3.4.2 Objets Request et Response

Lorsqu'un appel à requests.* est effectué, vous faites deux choses. Premièrement, vous construisez un object `Request` qui va être envoyé au serveur pour récupérer ou intérroger des ressources. Dès que l'objet `requests` reçoit une réponse du serveur, un object de type `Response` est généré. L'objet `Response``contient toutes les informations retournées par le serveur mais aussi l'object ``Request` que vous avz crée initialement. Voici une requête simple pour obtenir des informations depuis les serveurs Wikipedia:

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

Si nous voulons accéder aux en-têtes renvoyées par le serveur, nous faisons:

```
>>> response.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'}
```

Toutefois, si nous souhaitons récupérer les en-têtes que nous avions envoyées au serveur, nous accédons simplement à la requête, et aux en-têtes de la requête:

```
>>> response.request.headers
{'Accept-Encoding': 'identity, deflate, compress, gzip',
'Accept': '*/*', 'User-Agent': 'python-requests/0.13.1'}
```

3.4.3 Verifications certificats SSL

Requests peut vérifier les certificats SSL sur les requêtes HTTPS, comme n'importe quel navigateur web. Pour vérifier le certificat d'un serveur, vous pouvez utiliser l'argument `verify`:

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

SSL n'est pas configuré sur ce domaine, donc cela génère une erreur. Parfait. Par contre, GitHub en a un:

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

Vous pouvez aussi passer au paramètre `verify` le chemin vers un fichier `CA_BUNDLE` pour les certificats privés. Vous pouvez également définir la variable d'environnement `REQUESTS_CA_BUNDLE`.

Requests peut aussi ignorer les verifications SSL en mettant `verify` à False.

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

Par défautl, `verify` est True. L'option `verify` s'applique uniquement aux certificats des hôtes.

Vous pouvez également spcéifier un certificat local, comme un chemin de fichier ou une paire clé/valeur:

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

Si vous spécifiez un mauvais chemin ou un certificate invalide:

```
>>> 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 l
```

3.4.4 Process d'accès au contenu des réponses

Par défaut, lorsque vous effectuez une requête, le corps de la réponse n'est pas téléchargé automatiquement. Les en-têtes sont téléchargés, mais le contenu lui-même n'est téléchargé que lorsque vous accédez à l'attribut `Response.content`.

Exemple:

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

La requête a été effectuée, et la connexion est toujours ouverte. Le corps de la réponse n'est pas encore été téléchargé.:

```
r.content
```

Le contenu est téléchargé et mis en cache à ce moment-là.

Vous pouvez modifier ce comportement par défaut avec le paramètre `prefetch`:

```
r = requests.get(tarball_url, prefetch=True)
# Appel bloquant jusqu'à reception du corps de la réponse
```

3.4.5 Configurer Requests

Vous pouvez avoir envie de configurer une requête pour personnaliser son comportement. Pour faire cela vous pouvez passer un dictionnaire `config` à une requête ou une session. Pour en savoir plus, cf *Configuration API Docs* to learn more.

3.4.6 Keep-Alive

Bonne nouvelle - grâce à urllib3, le keep-alive est 100% automatique pendant une session! Toutes les requêtes que vous ferez à travers une session réutiliseront automatiquement la connexion appropriée!

A noter que les connexions ne sont libérées pour réutilisation seulement lorsque les données ont été lues. Faites attention à bien mettre `prefetch` à `True` ou toujours accéder à la propriété `content` de l'object `Response`.

Si vous souhaitez désactiver le keep-alive, vous pouvez définir l'attribut de configuration `keep_alive` à `False`:

```
s = requests.session()
s.config['keep_alive'] = False
```

3.4.7 Requêtes asynchrones

`requests.async` a été supprimé de requests et dispose maintenant de son propre repository nommé GRequests.

3.4.8 Hooks d'évenements

Requests dispose d'un système de 'hooks' que vous pouvez utiliser pour manipuler des portions du processus de requêtage ou signaler des évènements.

Hooks disponibles:

`args`: Un dictionnaire d'arguments prêts à être envoyés à Request().

`pre_request`: L'objet Request, juste avant d'être envoyé.

`post_request`: L'objet Request, juste après avoir été envoyé.

`response`: La réponse générée après une requête.

Vous pouvez assigner une fonction de hook par requête, en passant au paramètre `hooks` de la Request un dictionnaire de hooks `{hook_name: callback_function}`:

```
hooks=dict(args=print_url)
```

La fonction `callback_function` recevra un bloc de données en premier argument.

```
def print_url(args):
    print args['url']
```

Si une exception apparait lors de l'éxecution du callback, un warning est affiché.

Si le callback renvoie une valeur, on suppose que cela remplace les données qui lui ont été passées. Si la fonction ne renvoie rien, alors rien n'est affecté.

Affichons quelques arguments a la volée:

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

Cette fois-ci, modifions les arguments avec un nouveau callback:

```
def hack_headers(args):
    if args.get('headers') is None:
        args['headers'] = dict()

    args['headers'].update({'X-Testing': 'True'})

    return args

hooks = dict(args=hack_headers)
headers = dict(yo=dawg)
```

Et essayons:

```
>>> requests.get('http://httpbin.org/headers', hooks=hooks, headers=headers)
{
    "headers": {
        "Content-Length": "",
        "Accept-Encoding": "gzip",
        "Yo": "dawg",
        "X-Forwarded-For": "::ffff:24.127.96.129",
        "Connection": "close",
        "User-Agent": "python-requests.org",
        "Host": "httpbin.org",
        "X-Testing": "True",
        "X-Forwarded-Protocol": "",
        "Content-Type": ""
    }
}
```

3.4.9 Authentification personnalisée

Requests vous permet de spécifier vos propres mécanismes d'authentification.

N'importe quel 'callable' à qui l'on passe l'argument `auth` pour une méthode de requête a l'opportunité de modifier la requête avant de la dispatcher.

Les implémentations d'authentification doivent hériter de la classe `requests.auth.AuthBase`, et sont très faciles à définir. Request fournit deux modèles communs d'authentification dans `requests.auth`: `HTTPBasicAuth` et `HTTPDigestAuth`.

Admettons que nous ayons un webservice qui réponde uniquement si le header `X-Pizza` est présent et défini avec un certain mot de passe. Peu de chance que cela arrive, mais voyons voir ce que cela pourrait donner.

```
from requests.auth import AuthBase
class PizzaAuth(AuthBase):
    """Attache l'authentification HTTP Pizza à un object Request."""
    def __init__(self, username):
        # setup any auth-related data here
        self.username = username

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Pizza'] = self.username
        return r
```

On peut alors faire une requête qui utilise notre authentification Pizza:

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

3.4.10 Requête en streaming

Avec la méthode `requests.Response.iter_lines()` vous pouvez facilement itérer sur des API en streaming comme par exemple la Twitter Streaming API.

Pour utiliser la Twitter Streaming API et pister le mot-clé "requests":

```
import requests
import json

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

for line in r.iter_lines():
    if line: # filtre les lignes vides (keep-alive)
        print json.loads(line)
```

3.4.11 Logging verbeux

Si vous voulez avoir une bonne vision des requêtes HTTP qui sont envoyées par votre application, vous pouvez activer le logging verbeux.

Pour cela, configurez Requests avec un stream où ecrire les logs:

```
>>> my_config = {'verbose': sys.stderr}
>>> requests.get('http://httpbin.org/headers', config=my_config)
2011-08-17T03:04:23.380175   GET   http://httpbin.org/headers
<Response [200]>
```

3.4.12 Proxys

Si vous avez besoin d'utiliser un proxy, vous pouvez configurer individuellement les requêtes avec l'argument `proxies` dans toutes les méthodes:

```
import requests

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

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

Vous pouvez aussi définir des proxys avec les variables d'environnement `HTTP_PROXY` et `HTTPS_PROXY`.

```
$ export HTTP_PROXY="10.10.1.10:3128"
$ export HTTPS_PROXY="10.10.1.10:1080"
$ python
>>> import requests
>>> requests.get("http://example.org")
```

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

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

3.4.13 Compatibilité

Requests est destiné à être conforme avec toutes les spécifications et RFC pertinentes, tant que cela ne cause pas de difficultés pour l'utilisateur. Cette attention aux spécifications peut mener à des comportements qui peuvent sembler inhabituels pour ceux qui n'en sont pas familiers.

Encodages

Lorsque vous recevez une réponse, Requests devine l'encodage à utiliser pour décoder la réponse quand vous accéder à `Response.text`. Requests commence par vérifier l'encodage dans l'en-tête HTTP, et si aucun n'est présent, Request utilisera le module chardet pour tenter de deviner l'encodage.

Le seul cas ou Requests ne suivra pas cette méthode est quand l'en-tête charset n'est pas présent et l'en-tête `Content-Type` contient `text`. Dans ce cas, la RFC 2616 spécifie que le jeu de caractères par défaut doit être `ISO-8859-1`. Requests suit donc les spécifications dans ce cas. Si vous avez besoin d'un encodage différent, vous pouvez définir manuellement la propriété `Response.encoding` ou utiliser la réponse brute avec `Request.content`.

3.4.14 Methodes (verbes) HTTP

Requests fournit l'accès à toute la gamme des verbes HTTP: GET, OPTIONS, HEAD, POST, PUT, PATCH et DELETE. Vous trouverez ci dessous divers exemples d'utilisation de ces verbes avec Requests, en utilisant l'API GitHub.

Nous commençons avec les verbes les plus utilisé : GET. La methode HTTP GET est une méthode idempotente qui retourne une ressource pour une URL donnée. C'est donc ce verbe que vous allez utiliser pour tenter de récupérer des données depuis le web. Un exemple d'usage serait de récupérer les informations d'un commit spécifique sur GitHub. Admettons que nous souhaitions récupérer le commit `a050faf` de Requests. On peut le récupérer de cette façon:

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

On devrait confirmer que GitHub a répondu correctement. Si c'est le cas on peut alors travailler avec le contenu reçu. Voici comment faire:

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

Donc, GitHub renvoie du JSON. C'est super, on peut alors utiliser le module JSON pour convertir le résultat en object Python. Comme GitHub renvoie de l'UTF-8, nous devons accéder à `r.text` et pas `r.content`. `r.content` renvoie un bytestring, alors que ‘‘r.text‘‘renvoie une chaîne encodée en unicode.

```
>>> import json
>>> commit_data = json.loads(r.text)
>>> 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
```

Tout simple. Poussons un peu plus loin sur l'API GitHub. Maintenant, nous pouvons regarder la documentation, mais ce serait plus fun d'utiliser Requests directement. Nous pouvons tirer profit du verbe HTTP OPTIONS pour consulter quelles sont les methodes HTTP supportées sur une URL.

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

Comment ça? Cela ne nous aide pas du tout. Il se trouve que GitHubn comme beaucoup de fournisseurs d'API n'implémente pas la méthode HTTP OPTIONS. C'est assez embétant mais ca va aller, on peut encore consulter la documentation. Si GitHub avait correctement implémenté la méhode OPTIONS, elle retournerait la liste des méthodes autorisées dans les en-têtes, par exemple.

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

En regardant la documentation, on découvre que la seule autre méthode HTTP autorisée est POST, pour créer un nouveau commit. Comme nous utilisons le repository Requests, nous devrions éviter d'envoyer des requêtes assemblées manuellement. Nous allons plutôt jouter avec les Issues de GitHub.

Cette documentation a été ajotuée en réponse à l'issue #482. Sachant que cette issue existe encore, nous allons l'utiliser en exemple. Commençons par la récupérer.

```
>>> r = requests.get('https://api.github.com/repos/kennethreitz/requests/issues/482')
>>> r.status_code
200
>>> issue = json.loads(r.text)
>>> print issue[u'title']
Feature any http verb in docs
>>> print issue[u'comments']
3
```

Cool, nous avons 3 commentaires. Regardons le dernier.

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

```
>>> 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
```

Bon, le commentaire à l'air stupide. Ajoutons un commentaire pour en informer son auteur. D'ailleurs, qui est-il ?

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

OK, donc disons à ce Kenneth que l'on pense que cet exemple devrait plutôt aller dans la section quickstart. D'après la doc de l'API GitHub, il faut utiliser la méthode POST pour ajouter un commentaire. allons-y.

```
>>> 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
```

Mince, c'est bizarre. On doit avoir besoin d'une authentification. Ca va pas être simple, hein ? Non. Requests rend très simple tout sortes d'authentification, comme la très classique Basic Auth.

```
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth('fake@example.com', 'not_a_real_password')
>>> r = requests.post(url=url, data=body, auth=auth)
>>> r.status_code
201
>>> content = json.loads(r.text)
>>> print content[u'body']
Sounds great! I'll get right on it.
```

Parfait. Hum, en fait non! j'aimerai modifier mon commentaire. Si seulement je pouvais l'éditer! Heureusement, GitHub nous permet d'utiliser un autre verbe, PATCH, pour éditer ce commentaire. Essayons.

```
>>> print content[u"id"]
5804413
>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it once I feed my cat."})
>>> url = u"https://api.github.com/repos/kennethreitz/requests/issues/comments/5804413"
>>> r = requests.patch(url=url, data=body, auth=auth)
>>> r.status_code
200
```

Excellent. Bon finalement, juste pour embéter ce Kenneth, j'ai décidé de le laisser attendre et de ne pas lui dire que je travaille sur le problème. Donc je veux supprimer ce commentaire. GitHub nous permet de supprimer des commentaire unqiuement avec le verbe bien nommé DELETE. Allons-y.

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

Parfait. Plus rien. La dernière chose que je voudrais savoir c'est combien j'ai consommé de mon taux de requêtes autorisées. GitHub envoie cette information dans les en-têtes HTTP, donc au lieu de télécharger toute la page, on peut simplement envoyer une requête HEAD pour récupérer uniquement les en-têtes.

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

```
'x-ratelimit-limit': '5000'
...
```

Excellent. Il est temps d'écrire un programme Python qui abuse de l'API GitHub de toutes les façons possibles, encore 4995 fois :)

3.4.15 Liens dans les en-têtes

De nombreuses APIs HTTP fournissent des liens dans les en-têtes (Link headers). Ceci rend les APIs plus auto-descriptives et détéctables.

GitHub les utilise dans son API pour la pagination, par exemple:

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

Requests analyse automatiquement ces liens d'entête et les rends facilement utilisables:

```
>>> r.links['next']
'https://api.github.com/users/kennethreitz/repos?page=2&per_page=10'

>>> r.links['last']
'https://api.github.com/users/kennethreitz/repos?page=6&per_page=10'
```

CHAPTER 4

Guide de la communauté

Cette partie de la documentation détaille l'ecosystème de Requests et de sa communauté

4.1 Frequently Asked Questions

This part of the documentation answers common questions about Requests.

4.1.1 Encoded Data?

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

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

4.1.2 Custom User-Agents?

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

4.1.3 Why not Httplib2?

Chris Adams gave an excellent summary on Hacker News:

> httplib2 is part of why you should use requests: it's far more respectable as a client but not as well documented and it still takes way too much code for basic operations. I appreciate what httplib2 is trying to do, that there's a ton of hard low-level annoyances in building a modern HTTP client, but really, just use requests instead. Kenneth Reitz is very motivated and he gets the degree to which simple things should be simple whereas httplib2 feels more like an academic exercise than something people should use to build production systems[1].
>
> Disclosure: I'm listed in the requests AUTHORS file but can claim credit for, oh, about 0.0001% of the awesomeness.
>
> 1. http://code.google.com/p/httplib2/issues/detail?id=96 is a good example: an annoying bug which affect many people, there was a fix available for months, which worked great when I applied it in a fork and pounded a couple TB of data through it, but it took over a year to make it into trunk and even longer to make it onto PyPI where any other project which required " httplib2" would get the working version.

4.1.4 Python 3 Support?

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

- cPython 2.6
- cPython 2.7
- cPython 3.1
- cPython 3.2
- PyPy-c 1.4
- PyPy-c 1.5
- PyPy-c 1.6
- PyPy-c 1.7

4.1.5 Keep-alive Support?

Yep!

4.1.6 Proxy Support?

You bet!

4.1.7 SSL Verification?

Absolutely.

4.2 Modules

- requests-oauth, adds OAuth support to Requests.
- rauth, an alternative to requests-oauth, supports OAuth versions 1.0 and 2.0.
- FacePy, a Python wrapper to the Facebook API.
- robotframework-requests, a Robot Framework API wrapper.
- fullerene, a Graphite Dashboard.
- urbanairship-python, a fork of the Urban Airship API wrapper.
- WhitespaceBot, a project that automatically forks repos, strips trailing whitespace, and sends a pull request.
- python-rexster, Rexter client that provides a simple interface for graph databases.
- daikon, a CLI for ElasticSearch.

4.3 Articles & Talks

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

4.4 Integrations

4.4.1 ScraperWiki

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

To give it a try, simply:

```
import requests
```

4.5 Managed Packages

Requests is available in a number of popular package formats. Of course, the ideal way to install Requests is via The Cheeseshop.

4.5.1 Ubuntu & Debian

Requests is available installed as a Debian package! Debian Etch Ubuntu, since Oneiric:

```
$ apt-get install python-requests
```

4.5.2 Fedora and RedHat

You can easily install Requests v0.6.1 with yum on rpm-based systems:

```
$ yum install python-requests
```

4.6 Support

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

4.6.1 Send a Tweet

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

4.6.2 File an Issue

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

4.6.3 E-mail

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

4.6.4 IRC

The official Freenode channel for Requests is #python-requests

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

4.7 Updates

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

4.7.1 GitHub

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

4.7.2 Twitter

I often tweet about new features and releases of Requests.

Follow @kennethreitz for updates.

4.7.3 Mailing List

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

CHAPTER 5

Documentation de l'API

Si vous cherchez des informations sur une fonction, une classe ou une méthode spécifique, c'est dans cette partie de la documentation que vous devez regarder.

5.1 API

Cette partie de la documentation présente toutes les interfaces possibles de Requests. Pour certaines parties, Requests dépend de librairies externes, nous documentons les plus importantes ici et fournissons des liens vers les documentations externes.

5.1.1 Interface Principale

Toutes les fonctionnalités de Requests sont accessibles via ces 7 méthodes. Elles retournent toutes une instance de l'objet `Response`.

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

---

class `requests.`**`Response`**

The core `Response` object. All `Request` objects contain a `response` attribute, which is an instance of this class.

`config` = **None**
Dictionary of configurations for this request.

`content`
Content of the response, in bytes.

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

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

`error` = **None**
Resulting `HTTPError` of request, if one occurred.

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

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

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

iter_lines (*chunk_size=10240*, *decode_unicode=None*)
Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

json
Returns the json-encoded content of a response, if any.

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

raise_for_status (*allow_redirects=True*)
Raises stored HTTPError or URLError, if one occurred.

raw = None
File-like object representation of response (for advanced usage).

reason
The HTTP Reason for the response.

request = None
The Request that created the Response.

status_code = None
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

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

url = None
Final URL location of Response.

---

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

`requests.`**`session`**(***kwargs*)
Returns a `Session` for context-management.

Exceptions

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

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

exception `requests.`**`HTTPError`**
An HTTP error occurred.

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

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

5.1.2 Configurations

requests.defaults

This module provides the Requests configuration defaults.

Configurations:

base_headers Default HTTP headers.

verbose Stream to write request logging to.

max_redirects Maximum number of redirects allowed within a request.s

keep_alive Reuse HTTP Connections?

max_retries The number of times a request should be retried in the event of a connection failure.

danger_mode If true, Requests will raise errors immediately.

safe_mode If true, Requests will catch all errors.

strict_mode If true, Requests will do its best to follow RFCs (e.g. POST redirects).

pool_maxsize The maximium size of an HTTP connection pool.

pool_connections The number of active HTTP connection pools to use.

encode_uri If true, URIs will automatically be percent-encoded.

trust_env If true, the surrouding environment will be trusted (environ, netrc).

param store_cookies If false, the received cookies as part of the HTTP response would be ignored.

5.1.3 Async

5.1.4 Utilitaires

Ces fonctions sont utilisées en internes, mais peuvent être utiles en dehors de Requests.

Status Code Lookup

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

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

>>> requests.codes.teapot
418

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

Cookies

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

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

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

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

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

Parameters

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

Encodages

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

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

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

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

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

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

Tried:

1.charset from content-type

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

3.fall back and replace all unicode characters

5.1.5 Internes

Ces éléments sont des composants internes de Requests, et ne doivent jamais être vus par l'utilisateur final (développeur). Cette partie de la documentation existe uniquement pour ceux qui étendent les fonctionnalités de Requests.

Classes

class requests.**Response**
The core `Response` object. All `Request` objects contain a `response` attribute, which is an instance of this class.

config = None
Dictionary of configurations for this request.

content
Content of the response, in bytes.

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

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

error = None
Resulting `HTTPError` of request, if one occurred.

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

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

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

iter_lines(*chunk_size=10240*, *decode_unicode=None*)
Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

json
Returns the json-encoded content of a response, if any.

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

raise_for_status(*allow_redirects=True*)
Raises stored `HTTPError` or `URLError`, if one occurred.

raw = None
File-like object representation of response (for advanced usage).

reason
The HTTP Reason for the response.

request = None
The `Request` that created the Response.

status_code = None
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

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

url = None
Final URL location of Response.

class `requests.`**Request**(*url=None*, *headers={}*, *files=None*, *method=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *timeout=None*, *redirect=False*, *allow_redirects=False*, *proxies=None*, *hooks=None*, *config=None*, *prefetch=True*, *_poolmanager=None*, *verify=None*, *session=None*, *cert=None*)
The `Request` object. It carries out all functionality of Requests. Recommended interface is with the Requests functions.

allow_redirects = None
Set to True if full redirects are allowed (e.g. re-POST-ing of data at new `Location`)

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

cert = None
SSL Certificate

config = None
Dictionary of configurations for this request.

data = None
Dictionary, bytes or file stream of request body data to attach to the `Request`.

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

files = None
Dictionary of files to multipart upload (`{filename: content}`).

full_url
Build the actual URL to use.

headers = None
Dictionary of HTTP Headers to attach to the `Request`.

hooks = None
Event-handling hooks.

method = None
HTTP Method to use.

params = None
Dictionary or byte of querystring data to attach to the `Request`. The dictionary values can be lists for representing multivalued query parameters.

path_url
Build the path URL to use.

prefetch = None
Prefetch response content

redirect = None
True if `Request` is part of a redirect chain (disables history and HTTPError storage).

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

response = None
`Response` instance, containing content and metadata of HTTP Response, once `sent`.

send(*anyway=False*, *prefetch=None*)
Sends the request. Returns True if successful, False if not. If there was an HTTPError during transmission, self.response.status_code will contain the HTTPError code.

Once a request is successfully sent, *sent* will equal True.

Parameters **anyway** – If True, request will be sent, even if it has

already been sent.

Parameters **prefetch** – If not None, will override the request's own setting

for prefetch.

sent = None
True if Request has been sent.

session = None
Session.

timeout = None
Float describes the timeout of the request.

verify = None
SSL Verification.

class requests.**Session**(*headers=None*, *cookies=None*, *auth=None*, *timeout=None*, *proxies=None*, *hooks=None*, *params=None*, *config=None*, *prefetch=True*, *verify=True*, *cert=None*)
A Requests session.

close()
Dispose of any internal state.

Currently, this just closes the PoolManager, which closes pooled connections.

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

- **method** – method for the new `Request` object.
- **url** – URL for the new `Request` object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.
- **data** – (optional) Dictionary or bytes to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of 'filename': file-like-objects for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **return_response** – (optional) If False, an un-sent Request object will returned.
- **config** – (optional) A configuration dictionary. See `request.defaults` for allowed keys and their default values.
- **prefetch** – (optional) whether to immediately download the response content. Defaults to `True`.
- **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.

CHAPTER 6

Guide du développeur

Si vous voulez contribuer au projet, cette partie de la documentation est pour vous

6.1 How to Help

Requests is under active development, and contributions are more than welcome!

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug. There is a Contributor Friendly tag for issues that should be ideal for people who are not very familiar with the codebase yet.
2. Fork the repository on Github to start making your changes to the **develop** branch (or branch off of it).
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to AUTHORS.

6.1.1 Development dependencies

You'll need to install `gunicorn` and `httpbin` and various other dependencies in order to run requests' test suite:

```
$ virtualenv env
$ . env/bin/activate
$ make
$ make test
```

The `Makefile` has various useful targets for testing.

6.1.2 What Needs to be Done

- Documentation needs a roadmap.

6.2 Authors

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

6.2.1 Development Lead

- Kenneth Reitz <me@kennethreitz.com>

6.2.2 Urllib3

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

6.2.3 Patches and Suggestions

- Various Pocoo Members
- Chris Adams
- Flavio Percoco Premoli
- Dj Gilcrease
- Justin Murphy
- Rob Madole
- Aram Dulyan
- Johannes Gorset
- (Megane Murayama)
- James Rowe
- Daniel Schauenberg
- Zbigniew Siciarz
- Daniele Tricoli 'Eriol'
- Richard Boulton
- Miguel Olivares <miguel@moliware.com>
- Alberto Paro
- Jérémy Bethmont
- (Xu Pan)
- Tamás Gulácsi
- Rubén Abad
- Peter Manser
- Jeremy Selier
- Jens Diemer
- Alex <@alopatin>
- Tom Hogans <tomhsx@gmail.com>
- Armin Ronacher
- Shrikant Sharat Kandula
- Mikko Ohtamaa

- Den Shabalin
- Daniel Miller <danielm@vs-networks.com>
- Alejandro Giacometti
- Rick Mak
- Johan Bergström
- Josselin Jacquard
- Travis N. Vaught
- Fredrik Möllerstrand
- Daniel Hengeveld
- Dan Head
- Bruno Renié
- David Fischer
- Joseph McCullough
- Juergen Brendel
- Juan Riaza
- Ryan Kelly
- Rolando Espinoza La fuente
- Robert Gieseke
- Idan Gazit
- Ed Summers
- Chris Van Horne
- Christopher Davis
- Ori Livneh
- Jason Emerick
- Bryan Helmig
- Jonas Obrist
- Lucian Ursu
- Tom Moertel
- Frank Kumro Jr
- Chase Sterling
- Marty Alchin
- takluyver
- Ben Toews (mastahyeti)
- David Kemp
- Brendon Crawford
- Denis (Telofy)

- Cory Benfield (Lukasa)
- Matt Giuca
- Adam Tauber
- Honza Javorek
- Brendan Maguire <maguire.brendan@gmail.com>
- Chris Dary
- Danver Braganza <danverbraganza@gmail.com>
- Max Countryman
- Nick Chadwick
- Jonathan Drosdeck
- Jiri Machalek
- Steve Pulec
- Michael Kelly
- Michael Newman <newmaniese@gmail.com>
- Jonty Wareing <jonty@jonty.co.uk>
- Shivaram Lingamneni
- Miguel Turner
- Rohan Jain (crodjer)
- Justin Barber <barber.justin@gmail.com>
- Roman Haritonov <@reclosedev>
- Josh Imhoff <joshimhoff13@gmail.com>
- Arup Malakar <amalakar@gmail.com>
- Danilo Bargen (gwrtheyrn)
- Torsten Landschoff
- Michael Holler (apotheos)
- Timnit Gebru
- Sarah Gonzalez
- Victoria Mo
- Leila Muhtasib
- Matthias Rahlf <matthias@webding.de>
- Jakub Roztocil <jakub@roztocil.name>
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24
- Rhys Elsmore

r

S

T

U

V