Démarrage rapide

Envie de vous lancer ? Cette page donne une bonne introduction sur la façon de démarrer avec les demandes.

Tout d'abord, assurez-vous que :

- Les requêtes sont installées
- Les demandes sont à jour

Commençons par quelques exemples simples.

Faire une demande

Faire une demande avec Requests est très simple.

Commencez par importer le module Requêtes :

```
>>> import requests
```

Maintenant, essayons d'obtenir une page Web. Pour cet exemple, obtenons la chronologie publique de GitHub :

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

Maintenant, nous avons un **Response**objet appelé `r`. Nous pouvons obtenir toutes les informations dont nous avons besoin à partir de cet objet.

L'API simple de Requests signifie que toutes les formes de requête HTTP sont aussi évidentes. Par exemple, voici comment faire une requête HTTP POST :

```
>>> r = requests.post('https://httpbin.org/post', data = {'key':'value'})
```

Bonne droite? Qu'en est-il des autres types de requêtes HTTP : PUT, DELETE, HEAD et OPTIONS ? Tout cela est tout aussi simple :

```
>>> r = requests.put('https://httpbin.org/put', data = {'key':'value'})
>>> r = requests.delete('https://httpbin.org/delete')
>>> r = requests.head('https://httpbin.org/get')
>>> r = requests.options('https://httpbin.org/get')
```

C'est bien beau, mais ce n'est aussi que le début de ce que les requêtes peuvent faire.

Passer des paramètres dans les URL

Vous souhaitez souvent envoyer une sorte de données dans la chaîne de requête de l'URL. Si vous construisiez l'URL à la main, ces données seraient données sous forme de paires 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 de chaînes, en utilisant le `params`mot - clé argument. Par exemple, si vous vouliez passer `key1=value1`et `key2=value2`à `httpbin.org/get`, vous utiliseriez le code suivant :

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


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

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

Notez que toute clé de dictionnaire dont la valeur est `None`ne sera pas ajoutée à la chaîne de requête de l'URL.

Vous pouvez également passer une liste d'éléments en tant que valeur :

```
>>> payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

>>> r = requests.get('https://httpbin.org/get', params=payload)
>>> print(r.url)
https://httpbin.org/get?key1=value1&key2=value2&key2=value3
```

Contenu de la réponse

Nous pouvons lire le contenu de la réponse du serveur. Considérez à nouveau la chronologie de GitHub :

```
>>> import requests

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

Les requêtes décoderont automatiquement le contenu du serveur. La plupart des jeux de caractères Unicode sont décodés de manière transparente.

Lorsque vous faites une demande, Requests fait des suppositions éclairées sur l'encodage de la réponse en fonction des en-têtes HTTP. L'encodage de texte deviné par Requests est utilisé lorsque vous accédez à `r.text`. Vous pouvez découvrir l'encodage utilisé par Requests et le modifier à l'aide de la `r.encoding`propriété :

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

Si vous modifiez l'encodage, Requests utilisera la nouvelle valeur de `r.encoding` chaque fois que vous appelez `r.text`. Vous voudrez peut-être le faire dans n'importe quelle situation où vous pouvez appliquer une logique spéciale pour déterminer quel sera l'encodage du contenu. Par exemple, HTML et XML ont la possibilité de spécifier leur encodage dans leur corps. Dans des situations comme celle-ci, vous devez utiliser `r.content`pour rechercher l'encodage, puis définir `r.encoding`. Cela vous permettra d'utiliser `r.text`avec le bon encodage.

Les demandes utiliseront également des encodages personnalisés au cas où vous en auriez besoin. Si vous avez créé votre propre encodage et l'avez enregistré avec le `codecs` module, vous pouvez simplement utiliser le nom du codec comme valeur de `r.encoding`et Requests se chargera du décodage pour vous.

Contenu de la réponse binaire

Vous pouvez également accéder au corps de la réponse sous forme d'octets, pour les requêtes non textuelles :

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


Les codages de transfert `gzip`et `deflate`sont automatiquement décodés pour vous.

L' `br` encodage de transfert est automatiquement décodé pour vous si une bibliothèque Brotli comme brotli ou brotlicffi est installée.

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

```
>>> from PIL import Image
>>> from io import BytesIO

>>> i = Image.open(BytesIO(r.content))
```

Contenu de la réponse JSON

Il existe également un décodeur JSON intégré, au cas où vous auriez affaire à des données JSON :

```
>>> import requests

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

En cas d'échec du décodage JSON, `r.json()` lève une exception. Par exemple, si la réponse obtient un 204 (Pas de contenu), ou si la réponse contient un JSON non valide, la tentative `r.json()` augmente `simplejson.JSONDecodeError` si simplejson est installé ou augmente sur Python 2 ou Python 3.`ValueError: No JSON object could be decoded` `json.JSONDecodeError`

Il convient de noter que le succès de l'appel à `r.json()` n'indique **pas** le succès de la réponse. Certains serveurs peuvent renvoyer un objet JSON dans une réponse ayant échoué (par exemple, les détails de l'erreur avec HTTP 500). Ce JSON sera décodé et renvoyé. Pour vérifier qu'une requête est réussie, utilisez `r.raise_for_status()` ou vérifiez `r.status_code` ce que vous attendez.

Contenu brut de la réponse

Dans les rares cas où vous souhaiteriez obtenir la réponse brute du socket du serveur, vous pouvez accéder à `r.raw`. Si vous souhaitez le faire, assurez-vous de le définir `stream=True` dans votre demande initiale. Une fois que vous l'avez fait, vous pouvez le faire :

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

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

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

En général, cependant, vous devez utiliser un modèle comme celui-ci pour enregistrer ce qui est diffusé dans un fichier :

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size=128):
        fd.write(chunk)
```

L'utilisation `Response.iter_content` gérera une grande partie de ce que vous auriez autrement à gérer lors de l'utilisation `Response.raw` directe. Lors de la diffusion en continu d'un téléchargement, ce qui précède est le moyen préféré et recommandé pour récupérer le contenu. Notez que `chunk_size` peut être librement ajusté à un nombre qui peut mieux s'adapter à vos cas d'utilisation.

Noter:

Une note importante sur l'utilisation de `Response.iter_content`versus `Response.raw`. `Response.iter_content`décodera automatiquement les `gzip`et les `deflate` codages de transfert. `Response.raw`est un flux brut d'octets – il ne transforme pas le contenu de la réponse. Si vous avez vraiment besoin d'accéder aux octets tels qu'ils ont été renvoyés, utilisez `Response.raw`.

En-têtes personnalisés

Si vous souhaitez ajouter des en-têtes HTTP à une requête, transmettez simplement a `dict`au `headers`paramètre.

Par exemple, nous n'avons pas spécifié notre user-agent dans l'exemple précédent :

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

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

Remarque : Les en-têtes personnalisés ont moins de priorité que les sources d'informations plus spécifiques. Par exemple:

- Les en-têtes d'autorisation définis avec *headers=* seront remplacés si les informations d'identification sont spécifiées dans `.netrc`, qui à leur tour seront remplacées par le `auth=` paramètre. Les requêtes rechercheront le fichier netrc dans *~/.netrc* , *~/_netrc* ou dans le chemin spécifié par la variable d'environnement *NETRC* .
- Les en-têtes d'autorisation seront supprimés si vous êtes redirigé hors hôte.
- Les en-têtes Proxy-Authorization seront remplacés par les informations d'identification de proxy fournies dans l'URL.
- Les en-têtes Content-Length seront remplacés lorsque nous pourrons déterminer la longueur du contenu.

De plus, Requests ne modifie pas du tout son comportement en fonction des en-têtes personnalisés spécifiés. Les en-têtes sont simplement transmis dans la requête finale.

Remarque : Toutes les valeurs d'en-tête doivent être une `string`chaîne d'octets ou un unicode. Bien que cela soit autorisé, il est conseillé d'éviter de transmettre des valeurs d'en-tête Unicode.

Demandes POST plus compliquées

En règle générale, vous souhaitez envoyer des données codées sous forme de formulaire, un peu comme un formulaire HTML. Pour ce faire, il suffit de passer un dictionnaire à l' `data`argument. Votre dictionnaire de données sera automatiquement encodé sous forme de formulaire lors de la demande :

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}

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

L' `data`argument peut également avoir plusieurs valeurs pour chaque clé. Cela peut être fait en créant `data`une liste de tuples ou un dictionnaire avec des listes comme valeurs. Ceci est particulièrement utile lorsque le formulaire comporte plusieurs éléments qui utilisent la même clé :

```
>>> payload_tuples = [('key1', 'value1'), ('key1', 'value2')]
>>> r1 = requests.post('https://httpbin.org/post', data=payload_tuples)
>>> payload_dict = {'key1': ['value1', 'value2']}
>>> r2 = requests.post('https://httpbin.org/post', data=payload_dict)
>>> print(r1.text)
{
  ...
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
>>> r1.text == r2.text
True
```

Il peut arriver que vous souhaitiez envoyer des données qui ne sont pas codées sous forme de formulaire. Si vous transmettez un `string`au lieu d'un `dict`, ces données seront publiées directement.

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

```
>>> import json

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

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

Au lieu d'encoder le `dict`vous - même, vous pouvez aussi le passer directement en utilisant le `json`paramètre (ajouté dans la version 2.4.2) et il sera encodé automatiquement :

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

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

Notez que le `json`paramètre est ignoré si l'un `data`ou l' autre `files`est transmis.

L'utilisation du `json`paramètre dans la requête changera le `Content-Type`dans l'en-tête en `application/json`.

POST un fichier codé en plusieurs

Requests simplifie le téléchargement de fichiers codés en plusieurs parties :

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

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

```
  ...
}
```

Vous pouvez définir explicitement le nom de fichier, le type de contenu et les en-têtes :

```
>>> url = 'https://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-exce

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

Si vous le souhaitez, vous pouvez envoyer des chaînes à recevoir sous forme de fichiers :

```
>>> url = 'https://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"
  },
  ...
}
```

Si vous publiez un fichier très volumineux en tant que `multipart/form-data` demande, vous souhaiterez peut-être diffuser la demande. Par défaut, `requests`cela ne prend pas en charge cela, mais il existe un package distinct qui le fait - `requests-toolbelt`. Vous devriez lire la documentation de la ceinture à outils pour plus de détails sur son utilisation.

Pour envoyer plusieurs fichiers en une seule requête, reportez-vous à la section avancée .

> **Avertissement:**
>
> Il est fortement recommandé d'ouvrir les fichiers en mode binaire . Cela est dû au fait que Requests peut tenter de vous fournir l'en- `Content-Length`tête, et si c'est le cas, cette valeur sera définie sur le nombre d' *octets* du fichier. Des erreurs peuvent se produire si vous ouvrez le fichier en *mode texte* .

Codes d'état de réponse

Nous pouvons vérifier le code d'état de la réponse :

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

Les requêtes sont également livrées avec un objet de recherche de code d'état intégré pour une référence facile :

v : maître

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

Si nous avons fait une mauvaise requête (une erreur client 4XX ou une réponse d'erreur serveur 5XX), nous pouvons la relancer avec **`Response.raise_for_status()`**:

```
>>> bad_r = requests.get('https://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, puisque notre `status_code`for rétait `200`, lorsque nous appelons, `raise_for_status()`nous obtenons :

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

Tout est bien.

En-têtes de réponse

Nous pouvons afficher les en-têtes de réponse du serveur à l'aide d'un dictionnaire 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'
}
```

Le dictionnaire est cependant spécial : il est conçu uniquement pour les en-têtes HTTP. Selon RFC 7230 , les noms d'en-tête HTTP ne sont pas sensibles à la casse.

Ainsi, nous pouvons accéder aux en-têtes en utilisant n'importe quelle majuscule que nous voulons :

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

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

Il est également particulier que le serveur aurait pu envoyer plusieurs fois le même en-tête avec des valeurs différentes, mais les requêtes les combinent afin qu'elles puissent être représentées dans le dictionnaire au sein d'un seul mappage, conformément à la RFC 7230 :

> Un destinataire PEUT combiner plusieurs champs d'en-tête avec le même nom de champ en une paire « nom de champ : valeur de champ », sans changer la sémantique du message, en ajoutant chaque valeur de champ suivante à la valeur de champ combinée dans l'ordre, séparés par un virgule.

cookies

v : maître

Si une réponse contient des Cookies, vous pouvez y accéder rapidement :

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

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

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

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

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

Les cookies sont renvoyés dans un **`RequestsCookieJar`**, qui agit comme un `dict`mais offre également une interface plus complète, adaptée à une utilisation sur plusieurs domaines ou chemins. Les pots de cookies peuvent également être transmis aux requêtes :

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('tasty_cookie', 'yum', domain='httpbin.org', path='/cookies')
>>> jar.set('gross_cookie', 'blech', domain='httpbin.org', path='/elsewhere')
>>> url = 'https://httpbin.org/cookies'
>>> r = requests.get(url, cookies=jar)
>>> r.text
'{"cookies": {"tasty_cookie": "yum"}}'
```

Redirection et historique

Par défaut, les requêtes effectueront une redirection de localisation pour tous les verbes à l'exception de HEAD.

Nous pouvons utiliser la `history`propriété de l'objet Response pour suivre la redirection.

La **`Response.history`**liste contient les **`Response`**objets qui ont été créés afin de compléter la demande. La liste est triée de la réponse la plus ancienne à la plus récente.

Par exemple, GitHub redirige toutes les requêtes HTTP vers HTTPS :

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

>>> r.url
'https://github.com/'

>>> r.status_code
200

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

Si vous utilisez GET, OPTIONS, POST, PUT, PATCH ou DELETE, vous pouvez désactiver la gestion de la redirection avec le `allow_redirects`paramètre :

```
>>> r = requests.get('http://github.com/', allow_redirects=False)

>>> r.status_code
301

>>> r.history
[]
```

Si vous utilisez HEAD, vous pouvez également activer la redirection :

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

>>> r.url
'https://github.com/'

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

Délais d'attente

Vous pouvez dire à Requests d'arrêter d'attendre une réponse après un certain nombre de secondes avec le `timeout`paramètre. Presque tout le code de production doit utiliser ce paramètre dans presque toutes les requêtes. Si vous ne le faites pas, votre programme peut se bloquer indéfiniment :

```
>>> requests.get('https://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 t
```

> ### Noter:
>
> `timeout`n'est pas une limite de temps sur l'intégralité du téléchargement de la réponse ; plutôt, une exception est levée si le serveur n'a pas émis de réponse pendant des `timeout`secondes (plus précisément, si aucun octet n'a été reçu sur le socket sous-jacent pendant des `timeout`secondes). Si aucun délai d'expiration n'est spécifié explicitement, les demandes n'expirent pas.

Erreurs et exceptions

En cas de problème de réseau (par exemple, panne DNS, connexion refusée, etc.), les requêtes généreront une **`ConnectionError`**exception.

`Response.raise_for_status()`lèvera un **`HTTPError`**si la requête HTTP a renvoyé un code d'état d'échec.

Si une requête expire, une **`Timeout`**exception est levée.

Si une demande dépasse le nombre configuré de redirections maximales, une **`TooManyRedirects`**exception est déclenchée.

Toutes les exceptions que Requests lève explicitement héritent de **`requests.exceptions.RequestException`**.

Prêt pour plus? Consultez la section avancée .

Si vous êtes sur le marché du travail, pensez à répondre à ce quiz sur la programmation . Un don substantiel sera fait à ce projet, si vous trouvez un emploi grâce à cette plateforme.

v : maître