

---

# **urllib2 を使ってインターネット上のリソースを取得するには**

リリース 2.7.15

**Guido van Rossum  
and the Python development team**

2月16, 2019

Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)

# 目次

第 1 章	はじめに	3
第 2 章	URL を取得する	4
2.1	データ	4
2.2	ヘッダ	5
第 3 章	例外を処理する	7
3.1	URLError	7
3.2	HTTPError	7
3.2.1	エラーコード	7
3.3	エラーをラップする	9
3.3.1	その 1	9
3.3.2	その 2	10
第 4 章	info と geturl	11
第 5 章	Openers と Handlers	12
第 6 章	Basic 認証	13
第 7 章	プロキシ	15
第 8 章	ソケットとレイヤー	16
第 9 章	脚注	17
	索引	18

---

著者 Michael Foord

---

注釈: この HOWTO の前段階の版のフランス語訳が [urllib2 - Le Manuel manquant](#) で入手できます。

---

# 第1章 はじめに

## Related Articles

同じように Python でインターネットリソースを取得するのに以下の記事が役に立ちます:

- [Basic Authentication](#)

*Basic* 認証 についてのチュートリアルで Python の例がついています。

**urllib2** は URLs (Uniform Resource Locators) を取得するための Python モジュールです。このモジュールはとても簡単なインターフェースを *urlopen* 関数の形式で提供しています。また、このモジュールは一般的な状況で利用するためにいくらか複雑なインターフェースも提供しています - basic 認証やクッキー、プロキシ等。これらは handler や opener と呼ばれるオブジェクトとして提供されます。

urllib2 supports fetching URLs for many "URL schemes" (identified by the string before the ":" in URL - for example "ftp" is the URL scheme of "ftp://python.org/") using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.

単純な状況では *urlopen* はとても簡単に使うことができます。しかし HTTP の URL を開くときにエラーが起きたり、特殊なケースに遭遇すると、HyperText Transfer Protocol に関するいくつかのことを理解する必要があります。HTTP に関して最も包括的で信頼できる文献は [RFC 2616](#) です。この文書は技術文書なので簡単には読めません。この HOWTO の目的は *urllib2* の利用法を例示することです、HTTP についてはその助けになるのに十分に詳しく載せています。このドキュメントは *urllib2* のドキュメントの代わりにはなりませんが、補完する役割を持っています。

## 第2章 URL を取得する

urllib2 を利用する最も簡単な方法は以下です:

```
import urllib2
response = urllib2.urlopen('http://python.org/')
html = response.read()
```

多くの urllib2 の利用法はこのように簡単です ('http:' の代わりに URL を 'ftp:' や 'file:' 等で始めればできます。)。しかし、このチュートリアルの目的は、得に HTTP に絞って、より複雑な状況を説明することです。

HTTP はリクエスト (request) とレスポンス (response) が基本となっています - クライアントがリクエストし、サーバーがレスポンスを送ります。urllib2 はこれを真似て、作成する HTTP リクエストを表現する Request オブジェクトを備えています。リクエストオブジェクトを作成する最も簡単な方法は取得したい URL を指定することです。urlopen をこのオブジェクトを使って呼び出すと、リクエストした URL のレスポンスオブジェクトが返されます。このレスポンスはファイルライクオブジェクトで、これはつまりレスポンスに .read() と呼び出せることを意味しています:

```
import urllib2

req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
the_page = response.read()
```

urllib2 は同じリクエストインターフェースを全ての URL スキームに対して利用できるようにしています。例えば、FTP リクエストの場合はこうできます:

```
req = urllib2.Request('ftp://example.com/')
```

HTTP の場合には、リクエストオブジェクトに対して二つの特別な操作ができます: 一つ目はサーバーに送るデータを渡すことができる、二つ目はサーバーに送るデータやリクエスト自身 についての 特別な情報 ("metadata") を渡すことができます - これらの送られる情報は HTTP 「ヘッダ」です。今度はこれらに関してひとつひとつ見ていきましょう。

### 2.1 データ

URL にデータを送りたい場合はよくあります (しばしば、その URL は CGI (Common Gateway Interface) スクリプト<sup>1</sup> や他の web アプリケーションを参照することになります)。これは HTTP では、**POST** リクエストとして知られる方法で行なわれます。これは web 上で HTML フォームを埋めて送信するときにブラウザが行なっていることです。全ての POST がフォームから送られるとは限りません: 自身のアプリケーションに対して任意のデータを POST を使って送ることができます。一般的な HTML フォームの場合、データ

<sup>1</sup> CGI プロトコルの入門については [Writing Web Applications in Python](#) を参照して下さい。

は標準的な方法でエンコードされている必要があり、リクエストオブジェクトに `data` 引数として渡します。エンコーディングは `urllib2` からではなく、`urllib` ライブラリの関数を使って行います:

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
the_page = response.read()
```

他のエンコーディングが必要な場合があることに注意して下さい (例えば、HTML フォームからファイルをアップロードするための詳細については [HTML Specification, Form Submission](#) を見て下さい)。

`data` 引数を渡さない場合、`urllib2` は GET リクエストを利用します。GET と POST リクエストの一つの違いは、POST リクエストにしばしば、「副作用」があることです: POST リクエストはいくつかの方法によってシステムの状態を変化させます (例えば 100 ポンドのスパムの缶詰をドアの前まで配達する注文を web サイトで行う)。とはいえ HTTP 標準で明確にされている内容では、POST は 常に 副作用を持ち、GET リクエストが副作用を持つことを禁止していません、副作用の無い POST リクエストはありません。HTTP の GET リクエストでもデータ自身をエンコーディングすることでデータを渡すことができます。

以下のようにして行います:

```
>>> import urllib2
>>> import urllib
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.urlencode(data)
>>> print url_values # The order may differ.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib2.urlopen(full_url)
```

? を URL に加え、それにエンコードされた値が続くことで、完全な URL が作られていることに注意して下さい。

## 2.2 ヘッダ

ここでは特定の HTTP ヘッダについて議論します、HTTP リクエストにヘッダを追加する方法について例示します。

いくつかの web サイト<sup>2</sup> はプログラムからブラウズされることを嫌っていたり、異なるブラウザに対して異なるバージョンのブラウザに送ります<sup>3</sup>。デフォルトでは `urllib2` は自身の情報を `Python-urllib/x.y`

<sup>2</sup> Google を例題にする。

<sup>3</sup> ブラウザを検知すること (browser sniffing) は web サイトのデザインにおけるとても悪い習慣です - web 標準を利用する方が賢明でしょう。不幸なことに未だに多くの web サイトが異なるブラウザに対して異なるバージョンを返しています。

として扱います (x と y は Python のリリースバージョンのメジャーバージョン、マイナーバージョンです、例えば Python-urllib/2.5 など)、これによって web サイト側が混乱したり、動作しないかもしれません。ブラウザは自身の情報を User-Agent ヘッダ<sup>4</sup>を通して扱っています。リクエストオブジェクトを作るときに、ヘッダに辞書を渡すことができます。以下の例は上の例と同じですが、自身を Internet Explorer<sup>5</sup> のバージョンの一つとして扱っています。

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
```

レスポンスは二つの便利なメソッドも持っています。 *info* と *geturl* の節を見て下さい、この節は後で問題が起きた場合に見ておくべき内容です。

---

<sup>4</sup> MSIE 6 のユーザエージェントは *'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)'* です。

<sup>5</sup> HTTP リクエストヘッダの詳細については、 [Quick Reference to HTTP Headers](#) を参照して下さい。

## 第3章 例外を処理する

`urlopen` はレスポンスを処理できなかった場合、`URLError` を送出します (ふつうの Python API では、組み込み例外の `ValueError`, `TypeError` などが送出されますが)。

`HTTPError` は `URLError` のサブクラスで HTTP URLs の特定の状況で送出されます。

### 3.1 URLError

`URLError` が送出されることはよく起こります、それはネットワーク接続が無い場合や、指定したサーバが無い場合です。この場合、例外は `'reason'` 属性を持っていて、この属性はエラーコードとエラーメッセージのテキストを含むタプルです。

例:

```
>>> req = urllib2.Request('http://www.pretend_server.org')
>>> try: urllib2.urlopen(req)
... except urllib2.URLError as e:
...     print e.reason
...
(4, 'getaddrinfo failed')
```

### 3.2 HTTPError

サーバーからの全ての HTTP レスポンスは「ステータスコード」の数値を持っています。多くの場合ステータスコードはサーバーがリクエストを実現できなかったことを意味します。デフォルトハンドラーはこれらのレスポンスのいくつかを処理してくれます (例えばレスポンスが「リダイ렉션」、つまりクライアントが別の URL を取得するように要求する場合には `urllib2` はこの処理を行ってくれます。) 処理できないものに対しては `urlopen` は `HTTPError` を送出します。典型的なエラーには `'404'` (page not found), `'403'` (request forbidden) と `'401'` (authentication required) が含まれます。

HTTP のエラーコード全てについては RFC 2616 の 10 節を参照して下さい。

送出された `HTTPError` インスタンスは整数の `'code'` 属性を持っていて、サーバーによって送られた応答に対応しています。

#### 3.2.1 エラーコード

デフォルトハンドラーはリダイレクト (コードは 300 番台にあります) を処理し、100-299 番台のコードは成功を意味しているので、たいいてい場合は 400-599 番台のエラーコードのみを見るだけですみます。

BaseHTTPServer.BaseHTTPRequestHandler.responses は RFC2616 で利用されるレスポンスコード全てを示す便利な辞書です。この辞書は便利なのでここに載せておきます:

```
# Table mapping response codes to messages; entries have the
# form {code: (shortmessage, longmessage)}.
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols',
         'Switching to new protocol; obey Upgrade header'),

    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted',
         'Request accepted, processing continues off-line'),
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),
    204: ('No Content', 'Request fulfilled, nothing follows'),
    205: ('Reset Content', 'Clear input form for further input.'),
    206: ('Partial Content', 'Partial content follows.'),

    300: ('Multiple Choices',
         'Object has several resources -- see URI list'),
    301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
    302: ('Found', 'Object moved temporarily -- see URI list'),
    303: ('See Other', 'Object moved -- see Method and URL list'),
    304: ('Not Modified',
         'Document has not changed since given time'),
    305: ('Use Proxy',
         'You must use proxy specified in Location to access this '
         'resource.'),
    307: ('Temporary Redirect',
         'Object moved temporarily -- see URI list'),

    400: ('Bad Request',
         'Bad request syntax or unsupported method'),
    401: ('Unauthorized',
         'No permission -- see authorization schemes'),
    402: ('Payment Required',
         'No payment -- see charging schemes'),
    403: ('Forbidden',
         'Request forbidden -- authorization will not help'),
    404: ('Not Found', 'Nothing matches the given URI'),
    405: ('Method Not Allowed',
         'Specified method is invalid for this server.'),
    406: ('Not Acceptable', 'URI not available in preferred format.'),
    407: ('Proxy Authentication Required', 'You must authenticate with '
         'this proxy before proceeding.'),
    408: ('Request Timeout', 'Request timed out; try again later.'),
    409: ('Conflict', 'Request conflict.'),
    410: ('Gone',
         'URI no longer exists and has been permanently removed.'),
    411: ('Length Required', 'Client must specify Content-Length.'),
    412: ('Precondition Failed', 'Precondition in headers is false.'),
    413: ('Request Entity Too Large', 'Entity is too large.'),
    414: ('Request-URI Too Long', 'URI is too long.'),
    415: ('Unsupported Media Type', 'Entity body in unsupported format.'),
    416: ('Requested Range Not Satisfiable',
         'Cannot satisfy request range.'),
    417: ('Expectation Failed',
         'Expect condition could not be satisfied.'),

    500: ('Internal Server Error', 'Server got itself in trouble'),
    501: ('Not Implemented',
```

(次のページに続く)



(前のページからの続き)

```

        'Server does not support this operation'),
502: ('Bad Gateway', 'Invalid responses from another server/proxy.'),
503: ('Service Unavailable',
      'The server cannot process the request due to a high load'),
504: ('Gateway Timeout',
      'The gateway server did not receive a timely response'),
505: ('HTTP Version Not Supported', 'Cannot fulfill request.'),
    }

```

エラーが起きた場合、サーバーは HTTP エラーコード と エラーページ を返して応答します。返されたページに対する応答として `HTTPError` インスタンスを使うことができます。これは `code` 属性に対しても同様です、これらは `read` も `geturl`, `info` などのメソッドも持っています。:

```

>>> req = urllib2.Request('http://www.python.org/fish.html')
>>> try:
...     urllib2.urlopen(req)
... except urllib2.HTTPError as e:
...     print e.code
...     print e.read()
...
404
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
<title>Page Not Found</title>
...

```

### 3.3 エラーをラップする

`HTTPError` または `URLError` が起きたときのために準備しておきたい場合には、二つの基本的なアプローチがあります。私は二つ目のアプローチを好みます。

#### 3.3.1 その1

```

from urllib2 import Request, urlopen, URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print 'The server couldn\'t fulfill the request.'
    print 'Error code: ', e.code
except URLError as e:
    print 'We failed to reach a server.'
    print 'Reason: ', e.reason
else:
    # everything is fine

```

注釈: `except HTTPError` が必ず最初に来る必要があります、そうしないと `except URLError` も `HTTPError` を捕捉してしまいます。

### 3.3.2 その2

```
from urllib2 import Request, urlopen, URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print 'We failed to reach a server.'
        print 'Reason: ', e.reason
    elif hasattr(e, 'code'):
        print 'The server couldn\'t fulfill the request.'
        print 'Error code: ', e.code
else:
    # everything is fine
```

## 第4章 info と geturl

レスポンスは `urlopen` (または `HTTPError` インスタンス) によって返され、`info()` と `geturl()` の二つの便利なメソッドを持っています。

**geturl** - これは取得したページの実際の URL を返します。 `urlopen` (または利用される `opener` オブジェクト) はリダイレクトに追従するため、有用です。取得したページの URL は要求した URL と同じとは限りません。

**info** - これは取得したページ (特にサーバからヘッダ) を表す辞書風オブジェクトを返します。これは現在では `httplib.HTTPMessage` インスタンスです。

典型的なヘッダは `'Content-length'`, `'Content-type'` 等を含んでいます。HTTP ヘッダその意味と利用法について簡単な説明がつきの便利な一覧 [Quick Reference to HTTP Headers](#) を参照して下さい。

## 第5章 Openers と Handlers

URL を取得する場合、`opener` (混乱を招きやすい名前ですが、`urllib2.OpenerDirector` のインスタンス) を利用します。標準的にはデフォルトの `opener` を `-urlopen` を通して - 利用していますが、カスタムの `opener` を作成することもできます。 `opener` は `handler` を利用します。全ての「一番厄介な仕事」はハンドラによって行なわれます。各 `handler` は特定の URL スキーム (`http`, `ftp`, 等) での URL の開き方を知っていたり、URL を開く局面でどう処理するかを知っています、例えば HTTP リダイレクションや HTTP のクッキーなど。

インストール済みの特定のハンドラで URL を取得したい場合には、`opener` を作成したいと思うでしょう、例えばクッキーを処理する `opener` が得たい場合や、リダイレクションを処理しない `opener` を得たい場合。

`opener` を作成するには、`OpenerDirector` をインスタンス化して、続けて、`.add_handler(some_handler_instance)` を呼び出します。

それに代わる方法として、`build_opener` を利用することもできます、これは `opener` オブジェクトを一回の関数呼び出しで作成できる便利な関数です。`build_opener` はいくつかのハンドラをデフォルトで追加しますが、デフォルトのハンドラに対して追加、継承のどちらかまたは両方を行うのに手っ取り早い方法を提供してくれます。

追加したくなる可能性がある `handler` としては、プロキシ処理、認証など、一般的ですがいくつか特定の状況に限られるものでしょう。

`install_opener` も (グローバルな) デフォルト `opener` オブジェクトの作成に利用できます。つまり、`urlopen` を呼び出すと、インストールした `opener` が利用されます。

`opener` オブジェクトは `open` メソッドを持っていて、`urlopen` 関数と同じ様に、`url` を取得するのに直接呼び出すことができます: 利便性を除けば `install_opener` を使う必要はありません。

## 第6章 Basic 認証

ハンドラの作成とインストールを例示するのに、`HTTPBasicAuthHandler` を利用してみます。この話題についてのより詳しい議論は – Basic 認証がどうやって動作するのかの説明も含んでいる [Basic Authentication Tutorial](#) を参照して下さい。

認証が必要な場合、サーバは認証を要求するヘッダ (401 のエラーコードとともに) を送ります。これによって認証スキームと 'realm' が指定されます。ヘッダはこのようになっています: `WWW-Authenticate: SCHEME realm="REALM"`。

例:

```
WWW-Authenticate: Basic realm="cPanel Users"
```

クライアントはリクエストヘッダに含まれる realm に対して適切な名前とパスワードとともにリクエストを再試行する必要があります。これが 'basic 認証' です。一連の流れを簡単化するために、`HTTPBasicAuthHandler` のインスタンスを作成し、`opener` が `handler` を利用するようにします。

`HTTPBasicAuthHandler` はパスワードマネージャーと呼ばれる、URL と realm をパスワードとユーザ名への対応づけを処理する、オブジェクトを利用します。realm が何なのか (サーバから返される認証ヘッダから) 知りたい場合には、`HTTPPasswordMgr` を利用できます。多くの場合、realm が何なのかについて気にすることはありません。そのような場合には `HTTPPasswordMgrWithDefaultRealm` を使うと便利です。これは URL に対してデフォルトのユーザ名とパスワードを指定できます。これによって特定の realm に対する代替の組み合わせを提供することなしに利用できるようになります。このことは `add_password` メソッドの realm 引数として `None` を与えることで明示することができます。

トップレベルの URL が認証が必要なはじめに URL です。この URL よりも「深い」URL を渡しても `.add_password()` は同様にマッチします。:

```
# create a password manager
password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username, password)

handler = urllib2.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib2.build_opener(handler)

# use the opener to fetch a URL
opener.open(a_url)

# Install the opener.
# Now all calls to urllib2.urlopen use our opener.
urllib2.install_opener(opener)
```

---

注釈: 上の例では `build_opener` に `HTTPBasicAuthHandler` のみを与えましたが。デフォルトの opener は普通の状況に適用するために - `ProxyHandler` (`http.proxy` 環境変数のようなプロキシ設定がセットされていれば), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor` を備えています。

---

`top_level_url` is in fact *either* a full URL (including the 'http:' scheme component and the hostname and optionally the port number) e.g. "`http://example.com/`" or an "authority" (i.e. the hostname, optionally including the port number) e.g. "`example.com`" or "`example.com:8080`" (the latter example includes a port number). The authority, if present, must NOT contain the "userinfo" component - for example "`joe:password@example.com`" is not correct.

## 第7章 プロキシ

`urllib2` は自動でプロキシ設定を認識して使います。プロキシ設定が検出されると、これは通常のハンドラのチェーンに組み入れられている `ProxyHandler` を通して行なわれます。たいていの場合はこれでうまくいきますが、役に立たない場合もあります<sup>6</sup>。この問題に対処する一つの方法はプロキシを定義しない `ProxyHandler` を組み立てることです。この方法は `Basic Authentication handler` を設定したときと同じような流れで行うことができます:

```
>>> proxy_support = urllib2.ProxyHandler({})
>>> opener = urllib2.build_opener(proxy_support)
>>> urllib2.install_opener(opener)
```

---

注釈: 現在 `urllib2` はプロキシ経由で `https` ロケーションを取得する機能をサポートしていません。しかし、`urllib2` をこのレシピ<sup>7</sup> で拡張することで可能にすることができます。

---

---

注釈: 変数 `REQUEST_METHOD` が設定されている場合、`HTTP_PROXY` は無視されます; `getproxies()` のドキュメンテーションを参照してください。

---

---

<sup>6</sup> 私の場合は仕事中にインターネットにアクセスするにはプロキシを利用する必要があります。 `localhost` の URL に対してこのプロキシを経由してアクセスしようとするれば、ブロックされます。IE を proxy を利用するように設定すれば、`urllib2` はその情報を利用します。 `localhost` のサーバでスクリプトをテストしようとする、`urllib2` がプロキシを利用するのを止めなければいけません。

<sup>7</sup> `urllib2 opener for SSL proxy (CONNECT method)`: [ASPN Cookbook Recipe](#).

## 第8章 ソケットとレイヤー

Python はレイヤー化された web 上からリソース取得もサポートしています。urllib2 は httpplib ライブラリを利用します、httpplib はさらに socket ライブラリを利用します。

Python 2.3 ではレスポンスがタイムアウトするまでのソケットの待ち時間を指定することができます。これは web ページを取得する場合に便利に使うことができます。socket モジュールのデフォルトではタイムアウトが無く ハングしてしまうかもしれません。現在では socket のタイムアウトは httpplib や urllib2 のレベルからは隠蔽されています。しかし、以下を利用することで全てのソケットに対してグローバルにデフォルトタイムアウトを設定することができます:

```
import socket
import urllib2

# timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

# this call to urllib2.urlopen now uses the default timeout
# we have set in the socket module
req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
```



## 第9章 脚注

このドキュメントは John Lee によって査読、改訂されました。

# 索引

`http_proxy`, 14

RFC

RFC 2616, 3

環境変数

`http_proxy`, 14