
Como buscar recursos da Internet usando o pacote urllib

Release 3.10.19

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

outubro 16, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Introdução	2
2	Acessando URLs	2
2.1	Dados	3
2.2	Cabeçalhos	4
3	Tratamento de exceções	5
3.1	URLError	5
3.2	HTTPError	5
3.3	Resumindo	7
4	info e geturl	8
5	Abridores e tratadores	8
6	Autenticação básica	9
7	Proxies	10
8	Socket e camadas	10
9	Notas de rodapé	11
	Índice	12

Autor Michael Foord

Nota: There is a French translation of an earlier revision of this HOWTO, available at [urllib2 - Le Manuel manquant](#).

1 Introdução

Related Articles

Você também pode achar útil o seguinte artigo na busca de recursos da web com Python:

- [Autenticação Básica](#)

Um tutorial sobre *Autenticação Básica*, com exemplos em Python.

`urllib.request` é um modulo Python para buscar URLs (Uniform Resource Locators). Ele oferece uma interface muito simples, na forma da função `urlopen`. Este é capaz de buscar URLs usando uma variedade de diferentes protocolos. Ele também oferece uma interface um pouco mais complexa para lidar com situações comuns - como autenticação básica, cookies, proxies e assim por diante. Estes são fornecidos por objetos chamados handlers (manipuladores) e openers (abridores).

`urllib.request` suporta o acesso a URLs por meio de vários “esquemas de URL” (identificados pela string antes de “:” na URL - por exemplo “ftp” é o esquema de URL em “ftp://python.org/”) usando o protocolo de rede associado a ele (como FTP e HTTP). Este tutorial foca no caso mais comum, HTTP.

Para situações simples “urlopen” é muito fácil de usar. Mas assim que você se depara com erros ou casos não triviais ao abrir URLs HTTP, você vai precisar entender um pouco mais do HyperText Transfer Protocol. A literatura de referência mais reconhecida e compreensível para o HTTP é [RFC 2616](#). Ela é um documento técnico e não foi feita para ser fácil de ler. Este documento busca ilustrar o uso de `urllib` com detalhes suficientes sobre HTTP para te permitir seguir adiante. Ele não tem a intenção de substituir a documentação do `urllib.request`, mas é suplementar a ela.

2 Acessando URLs

O modo mais simples de usar `urllib.request` é o seguinte:

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
    html = response.read()
```

Se você deseja obter um recurso via URL e guardá-lo em uma localização temporária, você pode fazê-lo com as funções `shutil.copyfileobj()` e `tempfile.NamedTemporaryFile()`:

```
import shutil
import tempfile
import urllib.request

with urllib.request.urlopen('http://python.org/') as response:
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        shutil.copyfileobj(response, tmp_file)

with open(tmp_file.name) as html:
    pass
```

Muitos usos de `urllib` são simples assim (repare que ao invés de uma URL ‘http:’ nós poderíamos ter usado uma string URL começando com ‘ftp:’, ‘file:’, etc.). No entanto, o propósito deste tutorial é explicar casos mais complicados, concentrando em HTTP.

HTTP é baseado em solicitações (requests) e respostas (responses) - o cliente faz solicitações e os servidores mandam respostas. `urllib.request` espelha isto com um objeto `Request` que representa a solicitação HTTP que você está fazendo.

Na sua forma mais simples, você cria um objeto Request que especifica a URL que você quer acessar. Chamar `urlopen` com este objeto Request retorna um objeto de resposta para a URL solicitada. Essa resposta é um objeto arquivo ou similar, o que significa que você pode, por exemplo, chamar `.read()` na resposta:

```
import urllib.request

req = urllib.request.Request('http://www.voidspace.org.uk')
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

Note que `urllib.request` usa a mesma interface Request para tratar todos os esquemas URL. Por exemplo, você pode fazer uma solicitação FTP da seguinte forma:

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

No caso do HTTP, há duas coisas extras que os objetos Request permitem que você faça: primeiro, você pode passar dados a serem enviados ao servidor. Segundo, você pode passar informações extras (“metadados”) *sobre* os dados ou sobre a própria solicitação para o servidor — essas informações são enviadas como “cabeçalhos” HTTP. Vamos analisar cada um deles separadamente.

2.1 Dados

Às vezes, você deseja enviar dados para uma URL (geralmente a URL se refere a um script CGI (Common Gateway Interface) ou outra aplicação web). Com HTTP, isso geralmente é feito usando o que é conhecido como uma solicitação **POST**. Isso geralmente é o que seu navegador faz quando você envia um formulário HTML preenchido na web. Nem todos os POSTs precisam vir de formulários: você pode usar um POST para transmitir dados arbitrários para sua própria aplicação. No caso comum de formulários HTML, os dados precisam ser codificados de forma padrão e, em seguida, passados para o objeto Request como o argumento `data`. A codificação é feita usando uma função da biblioteca `urllib.parse`.

```
import urllib.parse
import urllib.request

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

data = urllib.parse.urlencode(values)
data = data.encode('ascii') # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

Observe que outras codificações às vezes são necessárias (por exemplo, para envio de arquivos de formulários HTML - consulte [HTML Specification, Form Submission](#) para mais detalhes).

Se você não passar o argumento `data`, o `urllib` usará uma requisição **GET**. Uma diferença entre requisições GET e POST é que as requisições POST frequentemente têm “efeitos colaterais”: elas alteram o estado do sistema de alguma forma (por exemplo, ao fazer um pedido ao site para que cem libras de spam enlatado sejam entregues em sua porta). Embora o padrão HTTP deixe claro que os POSTs devem *sempre* causar efeitos colaterais, e as requisições GET *nunca* causar efeitos colaterais, nada impede que uma requisição GET tenha efeitos colaterais, nem que uma requisição POST não tenha efeitos colaterais. Dados também podem ser passados em uma requisição HTTP GET codificando-os na própria URL.

Isso é feito como abaixo:

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

Observe que o URL completo é criado adicionando um ? ao URL, seguido pelos valores codificados.

2.2 Cabeçalhos

Discutiremos aqui um cabeçalho HTTP específico para ilustrar como adicionar cabeçalhos à sua solicitação HTTP.

Alguns sites¹ não gostam de ser navegados por programas ou enviam versões diferentes para navegadores diferentes². Por padrão, urllib se identifica como Python-urllib/x.y (onde x e y são os números de versão principal e secundária da versão do Python, por exemplo, Python-urllib/2.5), o que pode confundir o site ou simplesmente não funcionar. A forma como um navegador se identifica é através do cabeçalho User-Agent³. Ao criar um objeto Request, você pode passar um dicionário de cabeçalhos. O exemplo a seguir faz a mesma solicitação acima, mas se identifica como uma versão do Internet Explorer⁴.

```
import urllib.parse
import urllib.request

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.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

A resposta também possui dois métodos úteis. Veja a seção sobre *info e geturl*, que vem depois de analisarmos o que acontece quando as coisas dão errado.

¹ Google, por exemplo.

² A detecção de navegadores é uma prática muito ruim para o design de sites; construir sites usando padrões web é muito mais sensato. Infelizmente, muitos sites ainda enviam versões diferentes para navegadores diferentes.

³ O user agent para MSIE 6 é 'Mozilla/4.0 (compatível; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)'

⁴ Para obter detalhes sobre mais cabeçalhos de solicitação HTTP, consulte *Referência rápida para cabeçalhos HTTP*.

3 Tratamento de exceções

`urlopen` raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.

As classes de exceção são exportadas do módulo `urllib.error`.

3.1 URLError

Frequentemente, `URLError` é levantada porque não há conexão de rede (nenhuma rota para o servidor especificado) ou o servidor especificado não existe. Nesse caso, a exceção gerada terá um atributo “reason”, que é uma tupla contendo um código de erro e uma mensagem de erro em texto.

Por exemplo

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

3.2 HTTPError

Every HTTP response from the server contains a numeric “status code”. Sometimes the status code indicates that the server is unable to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a “redirection” that requests the client fetch the document from a different URL, `urllib` will handle that for you). For those it can’t handle, `urlopen` will raise an `HTTPError`. Typical errors include ‘404’ (page not found), ‘403’ (request forbidden), and ‘401’ (authentication required).

Veja a seção 10 de [RFC 2616](#) para uma referência sobre todos os códigos de erro HTTP.

The `HTTPError` instance raised will have an integer ‘code’ attribute, which corresponds to the error sent by the server.

Códigos de erro

Como os tratadores padrão controlam redirecionamentos (códigos no intervalo 300) e códigos no intervalo 100-299 indicam sucesso, normalmente você verá apenas códigos de erro no intervalo 400-599.

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by [RFC 2616](#). The dictionary is reproduced here for convenience

```
# 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',
```

(continua na próxima página)

```

    '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',
    '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.'),
}

```

When an error is raised the server responds by returning an HTTP error code *and* an error page. You can use the `HTTPError` instance as a response on the page returned. This means that as well as the code attribute, it also has `read`, `geturl`, and `info`, methods as returned by the `urllib.response` module:

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

3.3 Resumindo

So if you want to be prepared for `HTTPError` *or* `URLError` there are two basic approaches. I prefer the second approach.

Número 1

```
from urllib.request import Request, urlopen
from urllib.error import 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
```

Nota: The `except HTTPError` *must* come first, otherwise `except URLError` will *also* catch an `HTTPError`.

Número 2

```
from urllib.request import Request, urlopen
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
```

(continua na próxima página)

```

    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

```

4 info e geturl

The response returned by `urlopen` (or the `HTTPError` instance) has two useful methods `info()` and `geturl()` and is defined in the module `urllib.response`.

geturl - Isso retorna a URL real da página recuperada. Isso é útil porque `urlopen` (ou o objeto de abertura utilizado) pode ter seguido um redirecionamento. A URL da página recuperada pode não ser a mesma que a URL solicitada.

info - Isso retorna um objeto semelhante a um dicionário que descreve a página recuperada, particularmente os cabeçalhos enviados pelo servidor. Atualmente, é uma instância de `http.client.HTTPMessage`.

Cabeçalhos típicos incluem 'Content-length', 'Content-type' e assim por diante. Consulte a "Referência rápida para cabeçalhos HTTP" <<https://jkorpela.fi/http.html>> para obter uma lista útil de cabeçalhos HTTP com breves explicações sobre seu significado e uso.

5 Abridores e tratadores

Ao buscar uma URL, você usa um abridor (uma instância do talvez confuso nome `urllib.request.OpenerDirector`). Normalmente, usamos o abridor padrão - via `urlopen` -, mas você pode criar abridores personalizados. Os abridores usam manipuladores. Todo o "trabalho pesado" é feito pelos manipuladores. Cada manipulador sabe como abrir URLs para um esquema de URL específico (http, ftp, etc.) ou como lidar com um aspecto da abertura de URL, por exemplo, redirecionamentos HTTP ou cookies HTTP.

Você vai querer criar abridores se quiser buscar URLs com manipuladores específicos instalados, por exemplo, para obter um abridor que manipule cookies ou para obter um abridor que não manipule redirecionamentos.

Para criar um abridor, instancie um `OpenerDirector` e então chame `.add_handler(some_handler_instance)` repetidamente.

Como alternativa, você pode usar `build_opener`, que é uma função conveniente para criar objetos de abertura com uma única chamada de função. `build_opener` adiciona vários tratadores por padrão, mas fornece uma maneira rápida de adicionar mais e/ou substituir os tratadores padrão.

Outros tipos de manipuladores que você pode querer podem lidar com proxies, autenticação e outras situações comuns, mas um pouco especializadas.

`install_opener` pode ser usado para tornar um objeto `opener` o abridor padrão (global). Isso significa que chamadas para `urlopen` usarão o abridor que você instalou.

Objetos abridores têm um método `open`, que pode ser chamado diretamente para buscar URLs da mesma forma que a função `urlopen`: não há necessidade de chamar `install_opener`, exceto por conveniência.

6 Autenticação básica

To illustrate creating and installing a handler we will use the `HTTPBasicAuthHandler`. For a more detailed discussion of this subject – including an explanation of how Basic Authentication works - see the [Basic Authentication Tutorial](#).

Quando a autenticação é necessária, o servidor envia um cabeçalho (e o código de erro 401) solicitando autenticação. Isso especifica o esquema de autenticação e um “domínio”. O cabeçalho se parece com: `WWW-Authenticate: SCHEME realm="REALM"`.

Por exemplo:

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

O cliente deve então tentar a solicitação novamente com o nome e a senha apropriados para o domínio incluídos como cabeçalho na solicitação. Isso é “autenticação básica”. Para simplificar esse processo, podemos criar uma instância de `HTTPBasicAuthHandler` e um `opener` para usar esse manipulador.

O `HTTPBasicAuthHandler` usa um objeto chamado gerenciador de senhas para manipular o mapeamento de URLs e domínios para senhas e nomes de usuário. Se você souber qual é o domínio (a partir do cabeçalho de autenticação enviado pelo servidor), poderá usar um `HTTPPasswordMgr`. Frequentemente, não importa qual seja o domínio. Nesse caso, é conveniente usar `HTTPPasswordMgrWithDefaultRealm`. Isso permite que você especifique um nome de usuário e uma senha padrão para uma URL. Isso será fornecido caso você não forneça uma combinação alternativa para um domínio específico. Indicamos isso fornecendo `None` como argumento de domínio para o método `add_password`.

A URL de nível superior é a primeira URL que requer autenticação. URLs “mais profundas” que a URL que você passa para `.add_password()` também corresponderão.

```
# create a password manager
password_mgr = urllib.request.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 = urllib.request.HTTPBasicAuthHandler(password_mgr)

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

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

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

Nota: In the above example we only supplied our `HTTPBasicAuthHandler` to `build_opener`. By default openers have the handlers for normal situations – `ProxyHandler` (if a proxy setting such as an `http_proxy` environment variable is set), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `DataHandler`, `HTTPErrorProcessor`.

`top_level_url` é, na verdade, *ou* uma URL completa (incluindo o componente do esquema ‘http:’, o nome do host e, opcionalmente, o número da porta), por exemplo, `"http://example.com/"`, *ou* uma “autoridade” (ou seja, o nome do host, incluindo, opcionalmente, o número da porta), por exemplo, `"example.com"` ou `"example.com:8080"`

(este último exemplo inclui um número de porta). A autoridade, se presente, NÃO deve conter o componente “userinfo” - por exemplo, “joe:senha@example.com” não está correto.

7 Proxies

`urllib` detectará automaticamente suas configurações de proxy e as utilizará. Isso ocorre por meio do `ProxyHandler`, que faz parte da cadeia de manipuladores normal quando uma configuração de proxy é detectada. Normalmente, isso é bom, mas há ocasiões em que pode não ser útil⁵. Uma maneira de fazer isso é configurar nosso próprio `ProxyHandler`, sem proxies definidos. Isso é feito seguindo etapas semelhantes à configuração de um manipulador de [autenticação básica](#):

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

Nota: Atualmente, `urllib.request` não oferece suporte à busca de locais `https` por meio de um proxy. No entanto, isso pode ser habilitado estendendo `urllib.request`, conforme mostrado na receita⁶.

Nota: `HTTP_PROXY` será ignorado se uma variável `REQUEST_METHOD` estiver definida; veja a documentação em `getproxies()`.

8 Soceks e camadas

O suporte do Python para buscar recursos web é em camadas. `urllib` usa a biblioteca `http.client`, que por sua vez usa a biblioteca de sockets.

A partir do Python 2.3, você pode especificar quanto tempo um soquete deve aguardar por uma resposta antes de atingir o tempo limite. Isso pode ser útil em aplicações que precisam buscar páginas web. Por padrão, o módulo `socket` não tem tempo limite e pode travar. Atualmente, o tempo limite do soquete não é exposto nos níveis `http.client` ou `urllib.request`. No entanto, você pode definir o tempo limite padrão globalmente para todos os soquetes usando

```
import socket
import urllib.request

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

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

⁵ No meu caso, preciso usar um proxy para acessar a internet no trabalho. Se você tentar buscar URLs *localhost* por meio desse proxy, ele as bloqueia. O IE está configurado para usar o proxy, que o `urllib` detecta. Para testar scripts com um servidor *localhost*, preciso impedir que o `urllib` use o proxy.

⁶ `urllib opener for SSL proxy (CONNECT method)`: [ASPEN Cookbook Recipe](#).

9 Notas de rodapé

Este documento foi revisado e revisado por John Lee.

Índice

H

`http_proxy`, 9

R

RFC

RFC 2616, 2, 5

V

váriavel de ambiente

`http_proxy`, 9