
HOWTO - Cómo obtener recursos de Internet con el paquete urllib

Versión 3.14.0rc2

Guido van Rossum and the Python development team

septiembre 01, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1	Introducción	1
2	Obtención de URLs	2
2.1	Datos	3
2.2	Encabezados (Headers)	4
3	Gestión de excepciones	4
3.1	URLError	4
3.2	HTTPError	5
3.3	Resumiéndolo	6
4	info y geturl	6
5	Objetos de Apertura (Openers) y Gestores (Handlers)	7
6	Autenticación básica	7
7	Proxies	8
8	Sockets y capas	9
9	Notas a pie de página	9
	Índice	10

Autor
Michael Foord

1 Introducción

Related Articles

También puedes encontrar útil el siguiente artículo sobre la obtención de recursos web con Python:

- Basic Authentication

Un tutorial sobre *Autenticación Básica*, con ejemplos en Python.

urllib.request es un módulo Python para acceder y utilizar recursos de internet identificados por URLs (*Uniform Resource Locators*). Ofrece una interfaz muy simple, a través de la función *urlopen*. Esta función es capaz de acceder a URLs usando una variedad de protocolos diferentes. También ofrece una interfaz un poco más compleja para manejar situaciones comunes - como la autenticación básica, cookies y proxies, entre otros. Estos son proporcionados por los llamados objetos de apertura y gestores.

urllib.request soporta la obtención de recursos identificados por URLs para muchos «esquemas de URL» (identificados por la cadena de texto ubicada antes del ":" en el URL - por ejemplo "ftp" es el esquema de URL de "ftp://python.org/") usando sus protocolos de red asociados (por ejemplo FTP, HTTP). Este tutorial se centra en el caso más común, HTTP.

Para situaciones sencillas *urlopen* es muy fácil de usar. Pero tan pronto como encuentres errores o casos no triviales al abrir URLs HTTP, necesitarás entender el Protocolo de Transferencia de Hipertexto. La referencia más completa y autorizada para HTTP es **RFC 2616**. Este es un documento técnico y no pretende ser fácil de leer. Este HOWTO tiene como objetivo ilustrar el uso de la *urllib*, con suficientes detalles sobre HTTP para ayudarte a entenderlo. No pretende reemplazar los documentos *urllib.request*, pero es complementario a ellos.

2 Obtención de URLs

La forma más simple de usar *urllib.request* es la siguiente:

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

Si deseas recuperar un recurso a partir de la URL y almacenarlo en una ubicación temporal, puede hacerlo a través de las funciones *shutil.copyfileobj()* y *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
```

Muchos usos de *urllib* serán así de sencillos (nótese que en lugar de una URL "http:" podríamos haber usado una URL que empezara por "ftp:", "file:", etc.). Sin embargo, el propósito de este tutorial es explicar los casos más complicados, concentrándose en el HTTP.

HTTP se basa en peticiones y respuestas - el cliente hace peticiones y los servidores envían respuestas. *urllib.request* refleja esto con un objeto *Request* que representa la petición HTTP que estás haciendo. En su forma más simple se crea un objeto *Request* que especifica la URL que se quiere obtener. Llamar a *urlopen* con este objeto *Request* retorna un objeto respuesta para la URL solicitada. Esta respuesta es un objeto tipo archivo, lo que significa que puedes por ejemplo llamar a *.read()* en la respuesta:

```
import urllib.request

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

Tenga en cuenta que `urllib.request` utiliza la misma interfaz de `Request` para gestionar todos los esquemas de URL. Por ejemplo, puedes hacer una petición FTP de la siguiente manera:

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

En el caso de HTTP, hay dos cosas adicionales que los objetos `Request` le permiten hacer: Primero, puede pasar datos para enviarlos al servidor. En segundo lugar, puede pasar información adicional («metadatos») *about* los datos o sobre la solicitud en sí, al servidor; esta información se envía como «encabezados» HTTP. Veamos cada uno de estos por turno.

2.1 Datos

A veces quieres enviar datos a una URL (a menudo la URL se referirá a un script CGI (Common Gateway Interface) u otra aplicación web). Con HTTP, esto se hace a menudo usando lo que se conoce como una petición **POST**. Esto es a menudo lo que su navegador hace cuando envías un formulario HTML que has rellenado en la web. No todos los POSTs tienen que provenir de formularios: puedes usar un POST para transmitir datos arbitrarios a tu propia aplicación. En el caso común de los formularios HTML, los datos tienen que ser codificados de forma estándar, y luego pasados al objeto `Request` como el argumento `data`. La codificación se hace usando una función de la 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()
```

Ten en cuenta que a veces se requieren otras codificaciones (por ejemplo, para la carga de archivos desde formularios HTML - ver [HTML Specification, Form Submission](#) para más detalles).

Si no pasas el argumento `data`, `urllib` usa una petición **GET**. Una de las formas en la que las peticiones GET y POST difieren entre sí es que las peticiones POST a menudo tienen «efectos secundarios»: cambian el estado del sistema de alguna manera (por ejemplo, haciendo una petición al sitio para que un centenar de spam chatarra sea entregado a tu dirección). Aunque el estándar HTTP deja claro que las solicitudes POST están *siempre* destinadas a causar efectos secundarios, y las solicitudes GET a *nunca* causar efectos secundarios, nada impide que una solicitud GET tenga efectos secundarios, ni que una solicitud POST no tenga efectos secundarios. Los datos también pueden ser pasados en una solicitud GET HTTP codificándolos en la propia URL.

Esto se hace de la siguiente manera:

```
>>> 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)
```

Nota que la URL completa se crea añadiendo un `?` a la URL, seguido de los valores codificados.

2.2 Encabezados (Headers)

Discutiremos aquí un encabezado HTTP en particular, para ilustrar cómo agregar encabezados a su solicitud HTTP.

A algunos sitios web¹ no les gusta ser navegados por programas, o envían diferentes versiones a diferentes navegadores². Por defecto `urllib` se identifica como `Python-urllib/x.y` (donde `x` y `y` son los números mayor y menor de la versión de Python, por ejemplo `Python-urllib/2.5`), lo que puede confundir el sitio, o simplemente no funcionar. La forma en que un navegador se identifica a sí mismo es a través del encabezado `User-Agent`³. Cuando creas un objeto `Request` puedes pasarle un diccionario de encabezados. El siguiente ejemplo hace la misma petición que arriba, pero se identifica como una versión de 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()
```

La respuesta también tiene dos métodos útiles. Ver la sección de *info y geturl* que viene después de que echemos un vistazo a lo que pasa cuando las cosas van mal.

3 Gestión de excepciones

`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.

Las clases de excepción se exportan desde el módulo `urllib.error`.

3.1 URLError

A menudo, `URLError` se genera porque no hay conexión de red (no se encuentra ruta al servidor especificado), o el servidor especificado no existe. En este caso, la excepción generada tendrá un atributo `reason`, que es una tupla que contiene un código de error y un mensaje de error de texto.

por ejemplo

```
>>> 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')
```

¹ Google por ejemplo.

² El rastreo de navegadores es una práctica muy mala para el diseño de sitios web - construir sitios usando estándares web es mucho más sensato. Desafortunadamente muchos sitios siguen enviando versiones diferentes a diferentes navegadores.

³ El agente de usuario para MSIE 6 es *"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"*

⁴ Para detalles de más encabezados de peticiones HTTP, ver *Quick Reference to HTTP Headers*.

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).

Vea la sección 10 de **RFC 2616** para una referencia sobre todos los códigos de error HTTP.

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

Códigos de error

Debido a que los gestores por defecto gestionan redirecciones (códigos en el rango de 300), y que los códigos en el rango de 100–299 indican éxito, normalmente sólo verás códigos de error en el rango de 400–599.

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes that shows all the response codes used by **RFC 2616**. An excerpt from the dictionary is shown below

```
responses = {
    ...
    <HTTPStatus.OK: 200>: ('OK', 'Request fulfilled, document follows'),
    ...
    <HTTPStatus.FORBIDDEN: 403>: ('Forbidden',
                                   'Request forbidden -- authorization will '
                                   'not help'),
    <HTTPStatus.NOT_FOUND: 404>: ('Not Found',
                                   'Nothing matches the given URI'),
    ...
    <HTTPStatus.IM_A_TEAPOT: 418>: ('I'm a Teapot',
                                   'Server refuses to brew coffee because '
                                   'it is a teapot'),
    ...
    <HTTPStatus.SERVICE_UNAVAILABLE: 503>: ('Service Unavailable',
                                              'The server cannot process the '
                                              'request due to a high load'),
    ...
}
```

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 Resumiéndolo

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.')
        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 y 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** - retorna la verdadera URL de la página obtenida. Esto es útil porque `urlopen` (o el objeto de apertura utilizado) puede haber seguido una redirección. El URL de la página obtenida puede no ser el mismo que el URL solicitado.
- **info** - retorna un objeto parecido a un diccionario que describe la página consultada, particularmente los encabezados enviados por el servidor. Actualmente es una instancia `http.client.HTTPMessage`.

Los encabezados típicos incluyen “Content-length”, “Content-type” y así sucesivamente. Mira la [Quick Reference to HTTP Headers](#) para un listado útil de encabezados de HTTP con breves explicaciones de su significado y uso.

5 Objetos de Apertura (Openers) y Gestores (Handlers)

Cuando obtienes una URL utilizas una apertura (una instancia de la quizás confusamente llamada `urllib.request.OpenerDirector`). Normalmente hemos estado usando la apertura por defecto - a través de `urlopen` - pero puedes crear aperturas personalizadas. Las aperturas utilizan manejadores. Todo el «trabajo pesado» lo hacen los manejadores. Cada manejador sabe cómo abrir URLs para un esquema de URL particular (`http`, `ftp`, etc.), o cómo manejar un aspecto de la apertura de URLs, por ejemplo redirecciones HTTP o cookies HTTP.

Desearás crear objetos de apertura si deseas consultar URLs con gestores específicos instalados, por ejemplo para obtener un objeto de apertura que gestione cookies, o para obtener un objeto de apertura que no gestione redireccionamientos.

Para crear un objeto de apertura, debes instanciar un `OpenerDirector`, y luego llamar a `.add_handler(some_handler_instance)` repetidamente.

Alternativamente, puedes usar `build_opener`, que es una función conveniente para crear objetos de apertura con una sola llamada a la función. `build_opener` añade varios gestores por defecto, pero proporciona una forma rápida de añadir más y/o sobrescribir los gestores por defecto.

Otros tipos de gestores que puedes querer permiten manejar proxies, autenticación, y otras situaciones comunes pero ligeramente especializadas.

`install_opener` puede ser usado para hacer que un objeto `opener` sea el objeto de apertura (global) por defecto. Esto significa que las llamadas a `urlopen` usarán el objeto de apertura que has instalado.

Los objetos de apertura tienen un método `open`, que puede ser llamado directamente para consultar urls de la misma manera que la función «`urlopen`»: no hay necesidad de llamar `install_opener`, excepto por conveniencia.

6 Autenticación 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](#).

Cuando se requiere la autenticación, el servidor envía un encabezado (así como el código de error 401) solicitando la autenticación. Esto especifica el esquema de autenticación y un “realm”. El encabezado tiene el siguiente aspecto: `WWW-Authenticate: SCHEME realm="REALM"`.

por ejemplo.

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

El cliente debe entonces volver a intentar la solicitud con el nombre y la contraseña apropiados para el realm incluido como encabezamiento en la solicitud. Esto es “autenticación básica”. Para simplificar este proceso podemos crear una instancia de `HTTPBasicAuthHandler` y un objeto de apertura para usar este manejador.

El `HTTPBasicAuthHandler` utiliza un objeto llamado administrador de contraseñas para gestionar el mapeo de URLs y realms con contraseñas y nombres de usuario. Si sabes cuál es el realm (por el encabezado de autenticación enviado por el servidor), entonces puedes usar un `HTTPPasswordMgr`. Frecuentemente a uno no le importa cuál es el realm. En ese caso, es conveniente usar «`HTTPPasswordMgrWithDefaultRealm`». Esto te permite especificar un nombre de usuario y una contraseña por defecto para una URL. Esto será suministrado en caso de que no proporciones una combinación alternativa para un realm específico. Lo indicamos proporcionando `None` como el argumento del realm al método `add_password`.

La URL de primer nivel es la primera URL que requiere autenticación. Las URLs «más profundas» que la URL que pasas a `.add_password()` también coincidirán.

```
# 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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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` es de hecho *o* una URL completa (incluyendo el componente del esquema “http:” y el nombre del host y opcionalmente el número de puerto) p.ej. “http://example.com/” *o* una «autoridad» (esto es, el nombre del host, incluyendo opcionalmente el número de puerto) por ejemplo “example.com” *o* “example.com:8080” (este último ejemplo incluye un número de puerto). La autoridad, si está presente, NO debe contener el componente «userinfo» - por ejemplo “joe:password@example.com” no es correcto.

7 Proxies

`urllib` detectará automáticamente tu configuración de proxy y la utilizará. Esto es a través de `ProxyHandler`, que es parte de la cadena de gestores normales cuando se detecta un ajuste de proxy. Normalmente esto es algo bueno, pero hay ocasiones en las que puede no ser útil⁵. Una forma de hacerlo es configurar nuestro propio `ProxyHandler`, sin proxies definidos. Esto se hace usando pasos similares a la configuración de un gestor [Basic Authentication](#):

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

Nota

Actualmente `urllib.request` *no* soporta la consulta de ubicaciones `https` a través de un proxy. Sin embargo, esto puede ser habilitado extendiendo `urllib.request` como se muestra en la receta⁶.

Nota

`HTTP_PROXY` será ignorado si se establece una variable `REQUEST_METHOD`; ver la documentación en `getproxies()`.

⁵ En mi caso tengo que usar un proxy para acceder a internet en el trabajo. Si intentas consultar URLs de *localhost* a través de este proxy, las bloquea. IE está configurado para usar el proxy, que `urllib` recoge. Para poder probar los scripts con un servidor *localhost*, tengo que evitar que `urllib` use el proxy.

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

8 Sockets y capas

El soporte de Python para obtener recursos de la web funciona en capas. `urllib` utiliza la biblioteca `http.client`, que a su vez utiliza la biblioteca de sockets.

A partir de Python 2.3 se puede especificar cuánto tiempo debe esperar un socket para obtener una respuesta antes de que se agote el tiempo de espera. Esto puede ser útil en aplicaciones que tienen que consultar páginas web. Por defecto, el módulo `socket` no tiene *tiempo de espera* y puede colgarse. Actualmente, el tiempo de espera de la conexión no se expone en los niveles `http.client` o `urllib.request`. Sin embargo, puede establecerse el tiempo de espera por defecto de forma global para todas los sockets 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)
```

9 Notas a pie de página

Este documento fue examinado y revisado por John Lee.

Índice

R

RFC

RFC 2616, [2](#), [5](#)