如何使用 urllib 包获取网络资源

发布 *3.6.9rc1*

Guido van Rossum
and the Python development team

七月 02, 2019

Python Software Foundation
Email: docs@python.org

Contents

作者 Michael Foord

注解： 这份 HOWTO 文档的早期版本有一份法语的译文，可在 urllib2 - Le Manuel manquant 处查阅。

1 概述

Related Articles

关于使用 Python 获取网页资源，你或许还可以找到下列有用的文章：

- 基本的验证

  关于 基本的验证的入门指南，带有一些 Python 的示例。

urllib.request 是一个用于获取 URL（统一资源定位地址）的 Python 模块。它以 *urlopen* 函数的形式提供了一个非常简单的接口。该接口能够使用不同的协议获取 URL。同时它也提供了一个略微复杂的接口来处理常见情形——如：基本验证、cookies、代理等等。这些功能是通过叫做 handlers 和 opener 的对象来提供的。

urllib.request 支持多种 “URL 网址方案”（通过 URL 中 `":"` 之前的字符串加以区分——如 URL 地址 `"ftp://python.org/"` 中的 ``"ftp"``），使用与之相关的网络协议（如：FTP、HTTP）来获取 URL 资源。本指南重点关注最常用的情形——HTTP。

For straightforward situations *urlopen* is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol. The most comprehensive and authoritative reference to HTTP is **RFC 2616**. This is a technical document and not intended to be easy to read. This HOWTO aims to illustrate using *urllib*, with enough detail about HTTP to help you through. It is not intended to replace the `urllib.request` docs, but is supplementary to them.

2 Fetching URLs

下面是使用 urllib.request 最简单的方式：

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

如果你想通过 URL 获取资源并保存某个临时的地方，你可以通过 `shutil.copyfileobj()` 和 `tempfile.NamedTemporaryFile()` 函数：

```
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
```

Many uses of urllib will be that simple (note that instead of an ‘http:’ URL we could have used a URL starting with ‘ftp:’, ‘file:’, etc.). However, it’s the purpose of this tutorial to explain the more complicated cases, concentrating on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. urllib.request mirrors this with a `Request` object which represents the HTTP request you are making. In its simplest form you create a Request object that specifies the URL you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `.read()` on the response:

```
import urllib.request

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

Note that urllib.request makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

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

In the case of HTTP, there are two extra things that Request objects allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information ("metadata") *about* the data or the about request itself, to the server - this information is sent as HTTP "headers". Let's look at each of these in turn.

2.1 Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script or other web application). With HTTP, this is often done using what's known as a **POST** request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the `data` argument. The encoding is done using a function from the `urllib.parse` library.

```
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()
```

Note that other encodings are sometimes required (e.g. for file upload from HTML forms - see HTML Specification, Form Submission for more details).

If you do not pass the `data` argument, urllib uses a **GET** request. One way in which GET and POST requests differ is that POST requests often have "side-effects": they change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door). Though the HTTP standard makes it clear that POSTs are intended to *always* cause side-effects, and GET requests *never* to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself.

This is done as follows:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
```

(下页继续)

(续上页)

```
>>> 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)
```

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

2.2 Headers

We'll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites[1] dislike being browsed by programs, or send different versions to different browsers[2]. By default urllib identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/2.5`), which may confuse the site, or just plain not work. The way a browser identifies itself is through the `User-Agent` header[3]. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer[4].

```
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()
```

The response also has two useful methods. See the section on *info and geturl* which comes after we have a look at what happens when things go wrong.

3 处理异常

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.

The exception classes are exported from the `urllib.error` module.

[1] Google for example.

[2] Browser sniffing is a very bad practice for website design - building sites using web standards is much more sensible. Unfortunately a lot of sites still send different versions to different browsers.

[3] The user agent for MSIE 6 is '*Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)*'

[4] For details of more HTTP request headers, see Quick Reference to HTTP Headers.

3.1 URLError

通常，引发 URLError 的原因是没有网络连接（或者没有到指定服务器的路由），或者指定的服务器不存在。该情况下，将会引发该异常，并带有一个‘reason’属性，该属性是一个包含错误代码和文本错误信息的元组。

e.g.

```
>>> 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

从服务器返回的每个 HTTP 响应都包含一个数字的“状态码”。有时该状态码表明服务器无法完成该请求。默认的处理器（函数?）将会为你处理这其中的一些响应。(例如，如果响应包含了“redirection”，将会要求客户端去向另外的 URL 获取文档，urllib 将会为你处理该情形)。对于那些它无法处理的（状态代码），urlopen 将会引发一个 `HTTPError` 。典型的错误包括：‘404’（页面无法找到)、‘403’（请求遭拒绝）和’ 401 ‘(需要身份验证)。

See section 10 of **RFC 2616** for a reference on all the HTTP error codes.

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

Error Codes

Because the default handlers handle redirects (codes in the 300 range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

`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',
          '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'),
```

(下页继续)

(续上页)

```
    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:
```

(下页继续)

(续上页)

```
...     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 Wrapping it Up

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

Number 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
```

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

Number 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 and geturl

由 urlopen（或者 HTTPError 实例）所返回的响应包含两个有用的方法：info() 和 geturl()，该响应由模块 urllib.response 定义。

geturl - 返回所获取页面的真实 URL。该方法很有用，因为 urlopen（或者所使用的 opener 对象）可能回包括一次重定向。所获取页面的 URL 未必就是所请求的 URL 。

info - 该方法返回一个类似字典的对象，描述了所获取的页面，特别是由服务器送出的头部信息（headers）。目前它是一个 http.client.HTTPMessage 实例。

Typical headers include ‘Content-length’, ‘Content-type’, and so on. See the Quick Reference to HTTP Headers for a useful listing of HTTP headers with brief explanations of their meaning and use.

5 Openers and Handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly-named urllib.request.OpenerDirector). Normally we have been using the default opener - via urlopen - but you can create custom openers. Openers use handlers. All the “heavy lifting” is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirections or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirections.

To create an opener, instantiate an OpenerDirector, and then call .add_handler(some_handler_instance) repeatedly.

Alternatively, you can use build_opener, which is a convenience function for creating opener objects with a single function call. build_opener adds several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to can handle proxies, authentication, and other common but slightly specialised situations.

install_opener can be used to make an opener object the (global) default opener. This means that calls to urlopen will use the opener you have installed.

Opener objects have an open method, which can be called directly to fetch urls in the same way as the urlopen function: there’s no need to call install_opener, except as a convenience.

6 Basic Authentication

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.

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a ‘realm’. The header looks like: WWW-Authenticate: SCHEME realm="REALM".

e.g.

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

The client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is 'basic authentication'. In order to simplify this process we can create an instance of `HTTPBasicAuthHandler` and an opener to use this handler.

The `HTTPBasicAuthHandler` uses an object called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a `HTTPPasswordMgr`. Frequently one doesn't care what the realm is. In that case, it is convenient to use `HTTPPasswordMgrWithDefaultRealm`. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing `None` as the realm argument to the `add_password` method.

The top-level URL is the first URL that requires authentication. URLs "deeper" than the URL you pass to .add_password() will also match.

```
# 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)
```

注解: 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` 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.

7 Proxies

urllib will auto-detect your proxy settings and use those. This is through the `ProxyHandler`, which is part of the normal handler chain when a proxy setting is detected. Normally that's a good thing, but there are occasions when it may not be helpful[5]. One way to do this is to setup our own `ProxyHandler`, with no proxies defined. This is done using similar steps to setting up a *Basic Authentication* handler:

[5] In my case I have to use a proxy to access the internet at work. If you attempt to fetch *localhost* URLs through this proxy it blocks them. IE is set to use the proxy, which urllib picks up on. In order to test scripts with a localhost server, I have to prevent urllib from using the proxy.

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

注解: Currently `urllib.request` *does not* support fetching of `https` locations through a proxy. However, this can be enabled by extending urllib.request as shown in the recipe[6].

注解: `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.

8 Sockets and Layers

The Python support for fetching resources from the web is layered. urllib uses the `http.client` library, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages. By default the socket module has *no timeout* and can hang. Currently, the socket timeout is not exposed at the http.client or urllib.request levels. However, you can set the default timeout globally for all sockets using

```
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 脚注

This document was reviewed and revised by John Lee.

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

索引

非字母

H

R