HOWTO Fetch Internet Resources Using The urllib Package

Release 3.13.7

Guido van Rossum and the Python development team

September 19, 2025

Python Software Foundation Email: docs@python.org

Contents

1	Introduction	1
2	Fetching URLs 2.1 Data	2 3 4
3	Handling Exceptions 3.1 URLError	4 4 5
4	info and geturl	6
5	Openers and Handlers	7
6	Basic Authentication	7
7	Proxies	8
8	Sockets and Layers	9
9	Footnotes	9
Inc	Index	

Author

Michael Foord

1 Introduction

Related Articles

You may also find useful the following article on fetching web resources with Python:

• Basic Authentication

A tutorial on Basic Authentication, with examples in Python.

urllib.request is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the *urlopen* function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

urllib.request 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.

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

The simplest way to use urllib.request is as follows:

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

If you wish to retrieve a resource via URL and store it in a temporary location, you can do so via the shutil. copyfileobj() and tempfile.NamedTemporaryFile() functions:

```
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://python.org/')
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 about the 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.

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'
>>> 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 dislike being browsed by programs, or send different versions to different browsers. 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. 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.

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 Handling Exceptions

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.

3.1 URLError

Often, URLError is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute, which is a tuple containing an error code and a text error message.

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')
```

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.

3.2 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).

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 that shows all the response codes used by RFC 2616. An excerpt from the dictionary is shown below

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:

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
```

1 Note

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

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** this returns the real URL of the page fetched. This is useful because urlopen (or the opener object used) may have followed a redirect. The URL of the page fetched may not be the same as the URL requested.
- info this returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an http.client.HTTPMessage instance.

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: <code>WWW-Authenticate: SCHEME realm="REALM"</code>.

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)
```

(continues on next page)

```
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)
```

1 Note

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 helpful⁵. 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:

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

1 Note

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 recipe⁶.

1 Note

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

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.

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

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 Footnotes

This document was reviewed and revised by John Lee.

Index

R

RFC

RFC 2616, 2, 5