21.5. `urllib` — Open arbitrary resources by URL¶

Note

The `urllib` module has been split into parts and renamed in Python 3.0 to `urllib.request`, `urllib.parse`, and `urllib.error`. The *2to3* tool will automatically adapt imports when converting your sources to 3.0. Also note that the `urllib.urlopen()` function has been removed in Python 3.0 in favor of `urllib2.urlopen()`.

This module provides a high-level interface for fetching data across the World Wide Web. In particular, the `urlopen()` function is similar to the built-in function `open()`, but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available.

21.5.1. High-level interface¶

`urllib.urlopen`(*url*[, *data*[, *proxies*]])¶

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has `file:` as its scheme identifier, this opens a local file (without universal newlines); otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made the `IOError` exception is raised. If all went well, a file-like object is returned. This supports the following methods: `read()`, `readline()`, `readlines()`, `fileno()`, `close()`, `info()`, `getcode()` and `geturl()`. It also has proper support for the *iterator* protocol. One caveat: the `read()` method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case.

Except for the `info()`, `getcode()` and `geturl()` methods, these methods have the same interface as for file objects — see section *File Objects* in this manual. (It is not a built-in file object, however, so it can't be used at those few places where a true built-in file object is required.)

The `info()` method returns an instance of the class `httplib.HTTPMessage` containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in response to the FTP retrieval request. A Content-Type header will be present if the MIME type can be guessed. When the method is local-file, returned headers will include a Date representing the file's last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file's type. See also the description of the `mimetools` module.

The `geturl()` method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The `urlopen()` function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The `geturl()` method can be used to get at this redirected URL.

The `getcode()` method returns the HTTP status code that was sent with the response, or `None` if the URL is no HTTP URL.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be in standard *application/x-www-form-urlencoded* format; see the `urlencode()` function below.

The `urlopen()` function works transparently with proxies which do not require authentication. In a Unix or Windows environment, set the **http_proxy**, or **ftp_proxy** environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the `'%'` is the command prompt):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

The **no_proxy** environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

In a Windows environment, if no proxy environment variables are set, proxy settings are obtained from the registry's Internet Settings section.

In a Mac OS X environment, `urlopen()` will retrieve proxy information from the OS X System Configuration Framework, which can be managed with Network System Preferences panel.

Alternatively, the optional *proxies* argument may be used to explicitly specify proxies. It must be a dictionary mapping scheme names to proxy URLs, where an empty dictionary causes no proxies to be used, and `None` (the default value) causes environmental proxy settings to be used as discussed above. For example:

```
# Use http://www.someproxy.com:3128 for http proxying
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

Changed in version 2.3: Added the *proxies* support.

Changed in version 2.6: Added `getcode()` to returned object and support for the **no_proxy** environment variable.

Deprecated since version 2.6: The `urlopen()` function has been removed in Python 3.0 in favor of `urllib2.urlopen()`.

`urllib.urlretrieve`(*url*[, *filename*[, *reporthook*[, *data*]]])¶

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object, possibly cached). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the `urlencode()` function below.

Changed in version 2.5: `urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib._urlopener`¶

The public functions `urlopen()` and `urlretrieve()` create an instance of the `FancyURLopener` class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of `URLopener` or `FancyURLopener`, then assign an instance of that class to the `urllib._urlopener` variable before calling the desired function. For example, applications may want to specify a different *User-Agent* header than `URLopener` defines. This can be accomplished with the following code:

```
import urllib

class AppURLopener(urllib.FancyURLopener):
   version = "App/1.7"

urllib._urlopener = AppURLopener()
```

`urllib.urlcleanup`()¶
Clear the cache that may have been built up by previous calls to `urlretrieve()`.

21.5.2. Utility functions¶

`urllib.quote`(*string*[, *safe*])¶

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-'` are never quoted. By default, this function is intended for quoting the path section of the URL.The optional *safe* parameter specifies additional characters that should not be quoted — its default value is `'/'`.

Example: `quote('/~connolly/')` yields `'/%7econnolly/'`.

`urllib.quote_plus`(*string*[, *safe*])¶
Like `quote()`, but also replaces spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

`urllib.unquote`(*string*)¶

Replace %xx escapes by their single-character equivalent.

Example: unquote('/%7Econnolly/') yields '/~connolly/'.

urllib.unquote_plus(*string*)¶
Like unquote(), but also replaces plus signs by spaces, as required for unquoting HTML form values.

urllib.urlencode(*query*[, *doseq*])¶
Convert a mapping object or a sequence of two-element tuples to a "url-encoded" string, suitable to pass to urlopen() above as the optional *data* argument. This is useful to pass a dictionary of form fields to a POST request. The resulting string is a series of key=value pairs separated by '&' characters, where both *key* and *value* are quoted using quote_plus() above. If the optional parameter *doseq* is present and evaluates to true, individual key=value pairs are generated for each element of the sequence. When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The order of parameters in the encoded string will match the order of parameter tuples in the sequence. The urlparse module provides the functions parse_qs() and parse_qsl() which are used to parse query strings into Python data structures.

urllib.pathname2url(*path*)¶
Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the quote() function.

urllib.url2pathname(*path*)¶
Convert the path component *path* from an encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses unquote() to decode *path*.

urllib.getproxies()¶
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named <scheme>_proxy for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows.

21.5.3. URL Opener objects¶

class urllib.URLopener([*proxies*[, ***x509*]])¶

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than http:, ftp:, or file:, you probably want to use FancyURLopener.

By default, the URLopener class sends a *User-Agent* header of urllib/*VVV*, where *VVV* is the urllib version number. Applications can define their own *User-Agent* header by subclassing URLopener or FancyURLopener and setting the class attribute version to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is None, in which case environmental proxy settings will be used if present, as discussed in the definition of urlopen(), above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the https: scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an IOError exception if the server returns an error code.

open(*fullurl*[, *data*])¶
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, open_unknown() is called. The *data* argument has the same meaning as the *data* argument of urlopen().

open_unknown(*fullurl*[, *data*])¶
Overridable interface to open unknown URL types.

retrieve(*url*[, *filename*[, *reporthook*[, *data*]]])¶

Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either a mimetools.Message object containing the response headers (for remote URLs) or None (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of tempfile.mktemp() with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the http: scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the urlencode() function below.

version¶
Variable that specifies the user agent of the opener object. To get urllib to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.FancyURLopener(...)¶

FancyURLopener subclasses URLopener providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

Note

According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and `urllib` reproduces this behaviour.

The parameters to the constructor are the same as those for `URLopener`.

Note

When performing basic authentication, a `FancyURLopener` instance calls its `prompt_user_passwd()` method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The `FancyURLopener` class offers one additional method that should be overloaded to provide the appropriate behavior:

`prompt_user_passwd`(*host*, *realm*)¶

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, `(user, password)`, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

exception `urllib.ContentTooShortError`(*msg*[, *content*])¶

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header). The `content` attribute stores the downloaded (and supposedly truncated) data.

New in version 2.5.

21.5.4. `urllib` Restrictions¶

Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), FTP, and local files.

The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper processing of Expiration time headers.

There should be a function to query whether a particular URL is in the cache.

For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.

The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module `htmllib` to parse it.

The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module, subclassing `FancyURLOpener`, or changing *_urlopener* to meet your needs.

This module does not support the use of proxies which require authentication. This may be implemented in the future.

Although the `urllib` module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module `urlparse`.

21.5.5. Examples¶

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

The following example uses the `POST` method instead:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```


Previous topic

21.4. `wsgiref` — WSGI Utilities and Reference Implementation

Next topic

21.6. `urllib2` — extensible library for opening URLs

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 21. Internet Protocols and Support »