
urllib 패키지를 사용하여 인터넷 리소스를 가져오는 방법

릴리스 3.11.10

Guido van Rossum and the Python development team

9월 09, 2024

Contents

1	소개	2
2	URL을 가져오기	2
2.1	데이터	3
2.2	헤더	4
3	예외 처리	5
3.1	URLError	5
3.2	HTTPError	5
3.3	마무리	7
4	info와 geturl	8
5	오프너와 처리기	8
6	기본 인증	9
7	프락시	10
8	소켓과 계층	10
9	각주	11
	색인	12

저자
Michael Foord

1 소개

Related Articles

파이썬으로 웹 리소스를 가져오는 방법에 대한 다음 기사도 유용합니다:

- [Basic Authentication](#)

파이썬 예제가 있는 기본 인증(*Basic Authentication*)에 대한 자습서.

`urllib.request`는 URL(Uniform Resource Locator)을 가져오기 위한 파이썬 모듈입니다. `urlopen` 함수의 형태로, 매우 간단한 인터페이스를 제공합니다. 다양한 프로토콜을 사용하여 URL을 가져올 수 있습니다. 또한 기본 인증(basic authentication), 쿠키, 프락시 등과 같은 일반적인 상황을 처리하기 위한 약간 더 복잡한 인터페이스도 제공합니다. 이들은 처리기와 오프너라는 객체에 의해 제공됩니다.

`urllib.request`는 관련 네트워크 프로토콜(예를 들어 FTP, HTTP)을 사용하여 많은 “URL 스킴(scheme)” (URL에서 “:” 앞의 문자열로 식별됩니다 - 예를 들어 “ftp”는 “ftp://python.org/”의 URL 스킴입니다)에 대해 URL을 가져오는 것을 지원합니다. 이 자습서는 가장 흔한 경우인 HTTP에 초점을 맞춥니다.

간단한 상황에서 `urlopen`은 사용하기가 매우 쉽습니다. 그러나 HTTP URL을 열 때 예러나 사소하지 않은 사례를 만나자마자, HTTP(HyperText Transfer Protocol)에 대한 이해가 필요합니다. HTTP에 대한 가장 포괄적이고 권위 있는 레퍼런스는 [RFC 2616](#)입니다. 이것은 기술 문서이며 읽기 쉽지 않습니다. 이 HOWTO에서는 `urllib`를 사용하는 방법을 설명하고, HTTP에 대해 충분히 자세하게 설명합니다. `urllib.request` 문서를 대체하려는 것이 아니라, 보조하려는 것입니다.

2 URL을 가져오기

`urllib.request`를 사용하는 가장 간단한 방법은 다음과 같습니다:

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

URL을 통해 리소스를 가져와서 임시 위치에 저장하려면, `shutil.copyfileobj()`와 `tempfile.NamedTemporaryFile()` 함수를 통해 수행할 수 있습니다:

```
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
```

`urllib`의 많은 용도는 이렇게 간단합니다 (‘http:’ URL 대신 ‘ftp:’, ‘file:’ 등으로 시작하는 URL을 사용할 수 있음에 유의하십시오). 그러나, 이 자습서의 목적은 HTTP에 집중하여 더 복잡한 경우를 설명하는 것입니다.

HTTP는 요청과 응답을 기반으로 합니다 - 클라이언트는 요청하고 서버는 응답을 보냅니다. `urllib.request`는 HTTP 요청을 나타내는 `Request` 객체로 이것을 반영합니다. 가장 간단한 형식에서 가져오려는 URL을 지정하는 `Request` 객체를 만듭니다. 이 `Request` 객체로 `urlopen`을 호출하면 요청된 URL에 대한 응답 객체를 반환합니다. 이 응답은 파일류 객체입니다, 응답에서 예를 들어 `.read()`를 호출할 수 있다는 뜻입니다:

```
import urllib.request

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

urllib.request는 모든 URL 스킴을 처리하기 위해 같은 Request 인터페이스를 사용합니다. 예를 들어, 다음과 같이 FTP 요청을 할 수 있습니다:

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

```

HTTP의 경우, Request 객체로 수행할 수 있는 추가 작업이 두 가지 있습니다: 첫째, 서버로 보낼 데이터를 전달할 수 있습니다. 둘째, 데이터나 요청 자체에 관한 추가 정보(“메타 데이터”)를 서버에 전달할 수 있습니다 - 이 정보는 HTTP “헤더”로 전송됩니다. 이들을 차례로 살펴봅시다.

2.1 데이터

URL로 데이터를 보내려고 할 때도 있습니다(종종 URL은 CGI (Common Gateway Interface) 스크립트나 다른 웹 응용 프로그램을 가리킵니다). HTTP에서, 이것은 종종 **POST** 요청이라고 알려진 것을 사용하여 수행됩니다. 이것은 종종 웹에서 채워 넣은 HTML 폼(form)을 제출할 때 브라우저가 수행하는 것입니다. 모든 POST가 폼에서 비롯될 필요는 없습니다: POST를 사용하여 임의의 데이터를 여러분 자신의 응용 프로그램으로 전송할 수 있습니다. 일반적인 HTML 폼의 경우, 데이터를 표준 방식으로 인코딩할 필요가 있고, 그런 다음 data 인자로 Request 객체에 전달합니다. 인코딩은 urllib.parse 라이브러리의 함수를 사용하여 수행됩니다.

```
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()
```

다른 인코딩이 필요한 경우도 있음에 유의하십시오(예를 들어 HTML 폼에서 파일을 업로드하는 경우 자세한 내용은 [HTML Specification, Form Submission](#)을 참조하십시오).

data 인자를 전달하지 않으면, urllib는 **GET** 요청을 사용합니다. GET과 POST 요청이 다른 한 가지는 POST 요청에 종종 “부작용”이 있다는 것입니다: 어떤 방식으로든 시스템의 상태를 변경합니다(예를 들어 캔에 담긴 스팸이 여러분의 문 앞에 배달되도록 주문을 넣습니다). HTTP 표준이 POST는 항상 부작용을 일으키려는 것이고, GET은 절대 부작용을 일으키지 않는다고 분명히 하고 있지만, GET 요청이 부작용을 일으키거나 POST 요청에 부작용이 없는 것을 막을 수는 없습니다. URL 자체에 인코딩하여 HTTP GET 요청에 데이터를 전달할 수도 있습니다.

다음과 같이 수행됩니다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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)
```

전체 URL은 URL에 ?를 추가한 다음 인코딩된 값을 추가하여 만들어짐에 유의하십시오.

2.2 헤더

여기서는 HTTP 요청에 헤더를 추가하는 방법을 설명하기 위해 한 가지 특정 HTTP 헤더에 관해 설명합니다.

일부 웹 사이트는¹ 프로그램이 브라우저하는 것을 싫어하거나, 브라우저에 따라 다른 버전을 보냅니다². 기본적으로, `urllib`는 자신을 `Python-urllib/x.y`(여기서 `x`와 `y`는 파이썬 배포의 주 버전과 부 버전 번호입니다, 예를 들어 `Python-urllib/2.5`)로 식별하는데, 이는 사이트를 혼동시키거나, 단지 작동하지 않을 수 있습니다. 브라우저가 자신을 식별하는 방식은 `User-Agent` 헤더를³ 통하는 것입니다. `Request` 객체를 만들 때 헤더가 담긴 딕셔너리를 전달할 수 있습니다. 다음 예제는 위와 같은 요청을 하지만, 자신을 `Internet Explorer`의 한 버전으로 식별합니다⁴.

```
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()
```

응답에는 두 가지 유용한 메서드도 있습니다. 문제가 발생했을 때 어떤 일이 발생했는지 살펴본 후에 나오는 `info`와 `geturl` 섹션을 참조하십시오.

¹ 예를 들어 구글.

² 브라우저 스니핑(browser sniffing)은 웹 사이트 디자인에 매우 나쁜 습관입니다 - 웹 표준을 사용하여 사이트를 구축하는 것이 훨씬 합리적입니다. 불행히도 많은 사이트가 여전히 브라우저마다 다른 버전을 보냅니다.

³ MSIE 6의 사용자 에이전트(user agent)는 `'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)'` 입니다.

⁴ 더 많은 HTTP 요청 헤더에 대한 자세한 내용은 [Quick Reference to HTTP Headers](#)를 참조하십시오.

3 예외 처리

`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.

예외 클래스는 `urllib.error` 모듈이 내보냅니다.

3.1 URLError

종종, 네트워크 연결이 없거나 (지정된 서버로의 경로가 없거나), 지정된 서버가 없기 때문에 `URLError`가 발생합니다. 이 경우, 발생한 예외에는 'reason' 어트리뷰트가 있으며, 이는 에러 코드와 텍스트 에러 메시지를 포함하는 튜플입니다.

예를 들어

```
>>> 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).

모든 HTTP 에러 코드에 대한 레퍼런스는 [RFC 2616](#)의 섹션 10을 참조하십시오.

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

에러 코드

기본 처리기는 리디렉션(300 범위의 코드)을 처리하고, 100-299 범위의 코드는 성공을 나타내므로, 보통 400-599 범위의 에러 코드만 보게 됩니다.

`http.server.BaseHTTPRequestHandler.responses`는 [RFC 2616](#)가 사용하는 모든 응답 코드를 표시하는 유용한 응답 코드 딕셔너리입니다. 편의를 위해 딕셔너리를 여기에 재현합니다

```
# 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',
```

(다음 페이지에 계속)

```

    '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 마무리

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

1번

```
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

```

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

2번

```
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.')

```

(다음 페이지에 계속)

```

    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 info와 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** - 가져온 페이지의 실제 URL을 반환합니다. 이는 `urlopen`(또는 사용된 오프너 객체)이 리디렉션을 수행했을 수 있기 때문에 유용합니다. 가져온 페이지의 URL은 요청한 URL과 같지 않을 수 있습니다.
- **info** - 가져온 페이지를 설명하는 딕셔너리 객체, 특히 서버가 보낸 헤더. 현재 `http.client.HTTPMessage` 인스턴스입니다.

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 오프너와 처리기

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.

특정 처리기가 설치된 상태로 URL을 가져오려면 오프너를 만듭니다, 예를 들면 쿠키를 처리하는 오프너를 얻거나, 리디렉션을 처리하지 않는 오프너를 얻는 것이 있습니다.

오프너를 만들려면, `OpenerDirector`를 인스턴스화 한 다음, `.add_handler(some_handler_instance)`를 반복적으로 호출합니다.

또는, 단일 함수 호출로 오프너 객체를 만드는 편의 함수인 `build_opener`를 사용할 수 있습니다. `build_opener`는 기본적으로 여러 처리기를 추가하지만, 더 추가하거나 기본 처리기를 재정의하는 빠른 방법을 제공합니다.

여러분이 원할 수도 있는 다른 유형의 처리기는 프락시, 인증 및 다른 혼하지만 약간 특수한 상황을 처리할 수 있습니다.

`install_opener`를 사용하여 opener 객체를 (전역) 기본 오프너로 만들 수 있습니다. 즉, `urlopen`을 호출하면 설치한 오프너가 사용됩니다.

오프너 객체에는 `urlopen` 함수와 같은 방식으로 URL을 가져오기 위해 직접 호출할 수 있는 `open` 메서드가 있습니다: 편의 이외에, `install_opener`를 호출할 필요는 없습니다.

6 기본 인증

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](#).

인증이 필요할 때, 서버는 (401 에러 코드와 함께) 인증을 요청하는 헤더를 보냅니다. 이것은 인증 스킴과 ‘영역 (realm)’ 을 지정합니다. 헤더는 이렇게 생겼습니다: `WWW-Authenticate: SCHEME realm="REALM"`.

예를 들어

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

그러면 클라이언트는 영역에 적절한 이름과 비밀번호를 요청의 헤더로 포함해 요청을 다시 시도해야 합니다. 이것이 ‘기본 인증 (basic authentication)’ 입니다. 이 프로세스를 단순화하기 위해 `HTTPBasicAuthHandler` 인스턴스와 이 처리기를 사용할 오프너를 만들 수 있습니다.

`HTTPBasicAuthHandler`는 비밀번호 관리자 (password manager) 라는 객체를 사용하여 URL과 영역에서 비밀번호 (password)와 사용자 이름 (username)으로의 매핑을 처리합니다. (서버가 보낸 인증 헤더로부터) 영역이 무엇인지 안다면, `HTTPPasswordMgr`를 사용할 수 있습니다. 종종 영역이 무엇인지 상관하지 않습니다. 이 경우, `HTTPPasswordMgrWithDefaultRealm`를 사용하는 것이 편리합니다. 이것은 URL의 기본 사용자 이름과 비밀번호를 지정할 수 있습니다. 특정 영역에 대한 대체 조합을 제공하지 않으면 이것이 제공됩니다. `None`을 `add_password` 메서드에 대한 `realm` 인자로 제공하여 이를 나타냅니다.

최상위 URL은 인증이 필요한 첫 번째 URL입니다. `.add_password()`에 전달한 URL보다 “더 깊은” URL도 일치합니다.

```
# 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)
```

참고: 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`은 실제로 (‘http:’ 스킴 구성 요소와 호스트 이름 및 선택적인 포트 번호를 포함하는) 전체 URL, 예를 들어 `"http://example.com/"` 이거나 “주체 (authority)” (즉, 선택적으로 포트 번호를 포함하는 호스트명), 예를 들어 `"example.com"`이나 `"example.com:8080"` (후자의 예는 포트 번

호를 포함합니다)입니다. 주체가 있다면 “userinfo” 구성 요소를 포함하지 않아야 합니다 - 예를 들어 “joe:password@example.com”은 올바르지 않습니다.

7 프락시

`urllib`는 프락시 설정을 자동 감지하여 사용합니다. 이는 프락시 설정이 감지될 때 일반 처리기 체인의 일부가 되는 `ProxyHandler`를 통해 이루어집니다. 일반적으로 좋은 일이지만, 도움이 되지 않는 경우가 있습니다⁵. 이를 위한 한 가지 방법은 프락시가 정의되지 않은 자체 `ProxyHandler`를 설정하는 것입니다. 이것은 `Basic Authentication` 처리기 설정과 비슷한 단계를 사용하여 수행됩니다:

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

참고: 현재 `urllib.request`는 프락시를 통한 `https` 위치를 가져오는 것을 지원하지 않습니다. 그러나, 조리법에 표시된 대로 `urllib.request`를 확장하여 활성화할 수 있습니다⁶.

참고: 변수 `REQUEST_METHOD`가 설정되면 `HTTP_PROXY`는 무시됩니다; `getproxies()`의 설명서를 참조하십시오.

8 소켓과 계층

웹에서 리소스를 가져오기 위한 파이썬 지원은 계층화되어 있습니다. `urllib`는 `http.client` 라이브러리를 사용하고, 이것은 다시 `socket` 라이브러리를 사용합니다.

파이썬 2.3부터 시간제한으로 중단되기 전에 소켓이 응답을 기다리는 시간을 지정할 수 있습니다. 웹 페이지를 가져와야 하는 응용 프로그램에서 유용 할 수 있습니다. 기본적으로 소켓 모듈에는 시간제한이 없고 멈출(hang) 수 있습니다. 현재, 소켓 시간제한은 `http.client`나 `urllib.request` 수준에서 노출되지 않습니다. 그러나, 다음과 같이 모든 소켓에 대해 기본 시간제한을 전역적으로 설정할 수 있습니다

```
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 제 경우에는 직장에서 인터넷에 액세스하려면 프락시를 사용해야 합니다. 이 프락시를 통해 `localhost` URL을 가져오려고 시도하면 차단됩니다. IE는 프락시를 사용하도록 설정되어 있고, `urllib`는 이것을 선택합니다. `localhost` 서버로 스크립트를 테스트하려면, `urllib`가 프락시를 사용하지 못하게 해야 합니다.

⁶ SSL 프락시용 `urllib` 오픈너 (CONNECT 메서드): [ASPN Cookbook Recipe](#).

9 각주

이 문서는 John Lee가 검토하고 수정했습니다.

색인

R

RFC

RFC 2616, 2, 5