華中科技大學

本科生毕业设计（论文）参考文献译文本

译文出处：Michael Foord. HOWTO Fetch Internet Resources Using The urllib Package(Python3). https://docs.python.org/3.1/howto/urllib2.html

院　　系	光学与电子信息学院
专业班级	电子 1404
姓　　名	宁泽骥
学　　号	U201413720
指导教师	郑立新

2018 年　1 月

译文要求

一、　译文内容须与课题（或专业内容）联系，并需在封面注明详细出处。

二、　出处格式为

图书：作者.书名.版本（第×版）.译者.出版地：出版者，出版年.起页～止页

期刊：作者.文章名称.期刊名称，年号，卷号（期号）：起页～止页

三、　译文不少于 5000 汉字（或 2 万印刷符）。

四、　翻译内容用五号宋体字编辑，采用 A4 号纸双面打印，封面与封底采用浅蓝色封面纸（卡纸）打印。要求内容明确，语句通顺。

五、　译文及其相应参考文献一起装订，顺序依次为封面、译文、文献。

六、　翻译应在第七学期完成。

译文评阅

导师评语

应根据学校“译文要求”，对学生译文翻译的准确性、翻译数量以及译文的文字表述情况等做具体的评价后，再评分。

评分：____________________（百分制）　　　　指导教师(签名)：____________________

年　　月　　日

怎样使用 Urllib 库获取网络资源

1. 介绍

urllib.request 是一个获取 URL（Uniform Resource Locators）的 Python 模块，它使用 urlopen 函数的形式，提供了非常简单的接口来使用。urlopen 可以使用各种不同的协议获取 URL。它也能提供常见情况下稍微复杂接口——例如：用户身份验证、cookie、代理等。这些都是由被称之为 handler、opener 的对象提供的。

urllib.request 支持以多种“URL 标准”（一般来说在 URL 字符串的“：”前面——例如：“ftp://python.org” 的“URL 标准”是“ftp”）使用相关的网络协议（例如：ftp、http）来获取 URL。本教程会着重于指导用得最普遍的情况——HTTP。

对于一般的情况来说，urlopen 可以非常方便地去使用。但是当你打开 HTTP URL 遇到一些预料之外的或者说是错误的情况时，你就需要知道一些关于超文本传输协议（HyperText Transfer Protocol）的知识。最全面而权威的 HTTP 参考是：RFC 2616 。它是一个技术性文档并且不容易去阅读。本教程的主要目标是用足够多的例子去登陆 HTTP 网站来展示 urllib 库的用法。但是要注意本教程不能替代 urllib.request 文档，是该文档的补充说明。

2. 获取 URL

以下是 urllib.request 最简单的使用例子：

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

urllib 的许多用法就是这么简单（注意，我们还可以使用一个以“ftp:”或“file:”开头的 URL，而不是“http:”URL）。但是，本教程的目的是解释更复杂的情况，并着重于说明 HTTP。

HTTP 基于请求和响应——客户端发出请求并且服务器送回响应。 urllib.request 用一个 Request 对象来表示你正在做的 HTTP 请求。 以最简单的形式创建一个 Request 对象，该对象指定要获取的 URL。 使用此 Request 对象调用 urlopen 会为请求的 URL 返回一个响应对象。 这个响应是一个文件类对象，这意味着你可以在响应中调用.read()函数。

```
import urllib.request

req = urllib.request.Request('http://www.voidspace.org.uk')
```

```
response = urllib.request.urlopen(req)
the_page = response.read()
```

需要注意的是，urllib.request 使用相同的请求接口来处理所有的 URL 方案。 例如，你可以像这样创建一个 FTP 请求：

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

在使用 HTTP 的情形下，Request 对象允许你去执行两件事：首先，你可以接受要发送到服务器的数据。 其次，你可以将关于数据或关于这次请求本身的额外信息（“元数据”）传递给服务器——此信息作为 HTTP 的“头部”来发送。 让我们依次看看每一种情况。

3. 数据

有时候你想传一些数据到一个 URL（通常这个 URL 将涉及 CGI（Common Gateway Interface 通用网关接口）脚本[1]或其他 web 应用）。对于 HTTP，往往是通过一个被称为 POST 请求来完成的。这也是当你提交 HTML 表单时浏览器所做的。并不是所有的 POST 都来自于表单：你可以用 POST 传送任意数据到你自己的应用程序。HTTP 表单在一般情况下，数据需要用一个标准方法编码（encode），然后作为 data 参数传给 Request 对象。通过 urllib.parse 库中的一个函数完成编码。

```
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)
req = urllib.request.Request(url, data)
response = urllib.request.urlopen(req)
the_page = response.read()
```

注意：其他的编码有时是需要的（如：从 HTML 表单上传文件——参见 HTML

Specification, Form Submission ）。

如果你没有传入 data 参数，urllib 将使用 GET 的请求方式。POST 和 GET 请求方式的不同在于 POST 请求会有‘副作用’：它们在某种程度上会改变整个系统的状态（比如说，你通过向网站下订单，会有一大堆垃圾邮件发送将到你的信箱）。虽然 HTTP 标准明确表明 POST 总是会引起副作用的，GET 请求从不会造成副作用。没有什么能避免一个 GET 请求出现副作用，也没有什么能避免一个 POST 请求不会出现副作用。数据也可以通过在 URL 自身中编码，然后用 HTTP GET 请求方式传送。

如下例所示：

```
>>> 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)
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.open(full_url)
```

提示：对于完整的 URL，编码的数据会附在 URL 之后，以“?”符号来在 URL 中分割和传输。

4. 头部信息

我们将在这讨论一个比较特别的 HTTP 头部信息（header），用来展示如何向你的 HTTP 请求中添加头部信息。

一些网站[2]不太愿意被某些程序所浏览获取到[3]，或者对不同的浏览器发送不同的版本

的网页。默认情况下 urllib 标识其本身为 Python-urllib/x.y（其中 x 和 y 是 Python 的主次版本号，如 Python-urllib/2.5），这可能会让网站感到疑惑，或者直接不传输网页数据。浏览器标识其本身的方式是通过 User-Agent 头部信息[4]。当你创建一个 Request 对象，你可以传一个 dict 类型到头部信息。下面的例子和上面的要求一致，但其自身的标识为 InternetExplorer[5]。

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }
headers = { 'User-Agent' : user_agent }

data = urllib.parse.urlencode(values)
req = urllib.request.Request(url, data, headers)
response = urllib.request.urlopen(req)
the_page = response.read()
```

当然，这里的响应也有两个其他有用的方法。参见另处，在我们查看出现问题时，我们可以看出会发生什么。

5. 处理异常

当 urlopen 不能处理一个响应时，它会抛出 URLError 错误（尽管与普遍情况一样，也会引起 Python API、内置异常如 ValueError , TypeError 等 ）。

HTTPError 是 URLError （在具体到 HTTP URL 的使用场景下） 的子类。

异常类是从 urllib.error 模块导入的。

以下是例子:

```
>>> 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')
```

6. HTTP 异常

每一个来自服务器的 HTTP 响应都包含一个数字“状态码（status code）”。有时状态码表明了服务器无法完成该请求。默认的 handler 会为你处理这些响应中的一部分（例如：如果这个响应是“重定向”，这将请求客户端从其他的 URL 获取文档，urllib 将会为你处理该状态。）对于那些它不能处理的，urlopen 将抛出 HTTPError 错误。典型的错误有：404（未发现网页），403（禁止访问），401（需要身份验证）。

参考 RFC 2616 section 10 ，有所有的 HTTP 错误代码供参考。

抛出 HTTPError 异常里会有一个整型的“code”属性，和服务器发送的错误相一致。

7. 错误代码

因为默认的 handler 会去处理重定向的问题（在 300 之内的代码），并且代码在 100-299 范围内会表示本次请求成功，你一般只会看到在 400-599 范围的错误代码。

http.server.BaseHTTPRequestHandler.responses 是一个有用的字典类用来响应代码，显示了所有被 RFC 2616 使用的响应代码。为了方便来看，将这个字典展示如下：

```
# 此表映射响应代码及错误原因；
# 信息格式{code :(shortmessage,longmessage)}。
responses = {
    100: ('Continue', 'Request received, please continue'), # 100 继续
    101: ('Switching Protocols', 'Switching to new protocol; obey Upgrade header'), # 101 切换协议
    200: ('OK', 'Request fulfilled, document follows'), # 200 成功
    201: ('Created', 'Document created, URL follows'), #201 已创建
    202: ('Accepted', 'Request accepted, processing continues off-line'),# 202 已接受
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),# 203 非授权信息
```

204: ('No Content', 'Request fulfilled, nothing follows'), # 204 无内容

205: ('Reset Content', 'Clear input form for further input.'), # 205 重置内容

206: ('Partial Content', 'Partial content follows.'), # 206 部分内容

300: ('Multiple Choices', 'Object has several resources -- see URI list'), # 300 多种选择

301: ('Moved Permanently', 'Object moved permanently -- see URI list'), # 301 永久移动

302: ('Found', 'Object moved temporarily -- see URI list'), # 302 临时移动

303: ('See Other', 'Object moved -- see Method and URL list'), # 303 查看其他位置

304: ('Not Modified', 'Document has not changed since given time'), # 304 未修改

305: ('Use Proxy', 'You must use proxy specified in Location to access this' 'resource.'), # 305 使用代理

307: ('Temporary Redirect', 'Object moved temporarily -- see URI list'), # 307 临时重定向

400: ('Bad Request', 'Bad request syntax or unsupported method'), # 400 错误请求

401: ('Unauthorized', 'No permission -- see authorization schemes'), # 401 未授权

402: ('Payment Required', 'No payment -- see charging schemes'), # 402 请求支付

403: ('Forbidden', 'Request forbidden -- authorization will not help'),# 403 已禁止

404: ('Not Found', 'Nothing matches the given URI'),# 404 未找到

405: ('Method Not Allowed', 'Specified method is invalid for this server.'), # 405 方法禁用

406: ('Not Acceptable', 'URI not available in preferred format.'), # 406 不接受

407: ('Proxy Authentication Required', 'You must authenticate with ' 'this proxy before proceeding.'), # 407 需要代理授权

408: ('Request Timeout', 'Request timed out; try again later.'), # 408 请求超时

409: ('Conflict', 'Request conflict.'), # 409 冲突

410: ('Gone','URI no longer exists and has been permanently removed.'), # 410 已删除

411: ('Length Required', 'Client must specify Content-Length.'), # 411 需要有效长度

412: ('Precondition Failed', 'Precondition in headers is false.'), # 412 未满足前提条件

413: ('Request Entity Too Large', 'Entity is too large.'), # 413 请求实体过大

414: ('Request-URI Too Long', 'URI is too long.'), # 414 请求 URL 过长

415: ('Unsupported Media Type', 'Entity body in unsupported format.'),# 415 不支持的媒体类型

416: ('Requested Range Not Satisfiable', 'Cannot satisfy request range.'), # 416 请求范围不符合要求

417: ('Expectation Failed', 'Expect condition could not be satisfied.'), # 417 未满足期望值

500: ('Internal Server Error', 'Server got itself in trouble'), # 500 服务器内部错误

501: ('Not Implemented','Server does not support this operation'), # 501 尚未实施

502: ('Bad Gateway', 'Invalid responses from another server/proxy.'), # 502 错误网关

503: ('Service Unavailable', 'The server cannot process the request due to a high load'), # 503 服务不可用

504: ('Gateway Timeout', 'The gateway server did not receive a timely response'),# 504 网关超时

505: ('HTTP Version Not Supported', 'Cannot fulfill request.'), # 505 HTTP 版本不受支持}

当出现一个错误时，服务器通过返回一个 HTTP 错误代码和一个错误页作出响应。你可以使用 HTTPError 实例作为返回错误页的响应。这意味着这个和 code 属性一样，它也有 read、geturl、info 来作为 urllib.response 模块返回的方法：

```
>>> 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
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<?xml-stylesheet href="./css/ht2html.css"
```

```
        type="text/css"?>
<html><head><title>Error 404: File Not Found</title>
...... etc...
```

8. 处理异常

如果你想对可能会出现的 HTTPError 或 URLError 做好防范，有两种基本的方法。我本人更喜欢第二种。

方法 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:
```

注意：except HTTPError 必须是第一个，否则 except URLError 还会抛出一个 HTTPError。

方法 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:
```

9. 信息与 URL 获取

urlopen（或 HTTPError 实例）返回的响应有两个有用的函数 info()和 geturl()，并定义于 urllib.response 模块中。

geturl——返回获取页的真实 URL。这个方法是很有用的，因为 urlopen（或使用的 opener 对象）可能会跟随重定向的请求。这个获取到的 URL 可能和请求的 URL 不同。

info——返回一个描述获取页面的类字典对象。尤其是在服务器发送的头部信息中。目前它是一个 http.client.HTTPMessage 实例。

典型的头部信息（header）包含“Content-length”，“Content-type” 等。参见引用资料。

10. 打开器与处理器

当你获取一个 URL 时，你使用 opener（一个 urllib.request.OpenerDirector 类的实例）。通常我们使用 urlopen 方法，会启用默认的 opener ，但你也可以定制一个 opener。opener 使用 handler。所有的“重活”都是由 handler 来干。每个 handler 都知道怎么打开特定 URL 协议标准的 URL（http，ftp，等），怎么处理 URL 打开时的一些情况，比如 HTTP 重定向或 HTTP cookie。

如果你想要用其他特别的 handler 来获取 URL,你需要去创建一个新的 opener，例如：创建一个 opener 处理 cookie 的情况，或创建一个 opner 不去处理重定向的情况。

为了创建 opener，需要实例化一个 OpenerDirector 类，然后重复调用 add_handler（some_handler_instance）。

另外，你可以使用 build_opener，这是一个使用了单个函数调用创建 opener 对象的便捷函数。build_opener 默认添加了很多 handler，并且提供了一个快捷方法来添加更多或者重载

默认 handler。

其他种类的类的 handler 你可以用来处理代理（proxy）、身份验证（authentication）、和其他常见但稍微专业的情况。

install_opener 能够让一个 opener 对象成为（全局）默认 opener。这意味着当调用 urlopen 时将会使用你定制的 opener。

opener 对象有一个 open 方法，该方法被调用后能够像 urlopen 函数去一样直接获取 URL，而不必调用 install_opener,除非是为了方便调用。

11. 基本的身份验证

为了演示创建和安装 handler 的过程我们将会使用 HTTPBasicAuthHandler。对于这一小节的更多细节讨论——包括基础身份验证是如何工作的解释——可参阅其他教程。

当需要验证（Authentication）时，服务器将会发送一个 header（即 401 的错误代码）请求验证。这个头部信息（header）会指定验证方案还有“realm（域）”。头部信息可类似以下形式：WWW-Authenticate: SCHEME realm=“REALM” 。

例如：

Www-authenticate: Basic realm="cPanel Users"

客户端应该用对该 realm 合适的账号密码重试该请求。这就是“基本身份验证”。为了简化这一过程我们可以创建一个 HTTPBasicAuthHandler 实例和一个使用该 handler 的 opener。

HTTPBasicAuthHandler 里面使用一个被称为密码管理器（Password Manager）的对象去处理 URL 和 realm 到密码和账号的映射关系。如果你知道 realm 是什么（从服务器发送的验证头部信息中），你就可以使用 HTTPPasswordMgr。通常人们并不在乎 realm 是什么。这样做了以后，使用 HTTPPasswordMgrWithDefaultRealm 将会更加方便。它允许你为一个 URL 特别指定一个默认的账号与密码。在比较特殊的 realm 中，当你缺少需要提供的可选组件时它还能使用。我们可以通过将.add_password 方法国的 realm 参数设为 None 来使用。

顶层（top-level）的 URL 是第一个用于验证的 URL。比该 URL“更深”的 URL（你传给.add_password()的）也能进行匹配。

```
# 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)
```

注意：在上面的例子中，我们只向 build_opener 提供了 HTTPBasicAuthHandler 。在正常情况下，默认的 opener 有下列 handler——ProxyHandler，UnknownHandler，HTTPHandler, HTTPDefaultErrorHandler，HTTPRedirectHandler，FTPHandler，FileHandler，DataHandler, HTTPErrorProcessor。

top_level_url 事实上不是一个完整 URL（包括“http”协议、主机、端口号）如：“http://example.com/”或者“authrity”（换言之，主机，可选的端口号）如：“example.com”或“example.com:8080”。这个 URL 如果出现，则不能包含任何“用户信息”组件——比如：”joe:password@example.com”就是不正确的。

12. 代理

urllib 将会自动检测你的代理设置并使用这些设置。这通过 ProxyHandler 来去实现，并且是正常 handler 链的一部分。通常来说这带来很多好处，但有时候却没有帮助[5]。实现

的一种方法是用没有被定义过的代理来设置我们自己的 ProxyHandler ，这个和设置基本身份认证的 handler 的步骤类似：

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

注意：目前 urllib.request 不支持通过代理来获取 https 的站点。但是，在“recipe”[6]里扩展的 urllib.request 是可以支持的。

13. 套接字和层

Python 支持从 web 获取是分层的资源。urllib 使用 http.client 库，而这个库又是使用 socket 库的。

从 Python2.3 开始，你就可以指定一个套接字在时间消耗完之前应该等待多长时间。这个功能对必须获取网页的应用来说很有用。在默认的套接字模块中有“非超时” 和挂断功能。目前，套接字超时（timeout）并没有暴露于 http.client 或 urllib.request 里面。但是，你可以对所有使用的套接字设置为默认的全局超时。

```
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)
```

参考文献

本文件由 John Lee 审阅和修订。

[1]□For an introduction to the CGI protocol see Writing Web Applications in Python.

[2]□Like Google for example. The proper way to use google from a program is to use PyGoogle of course. See Voidspace Google for some examples of using the Google API.

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

[4]□The user agent for MSIE 6 is ‘Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)’

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

[6]□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.

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

参考文献原文

HOWTO Fetch Internet Resources Using The urllib Package

Author:Michael Foord

1. Introduction

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

Many uses of urllib will be that simple (note that instead of an ‘http:’ URL we could have used an 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')
response = urllib.request.urlopen(req)
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.

3.Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script [1] 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)
req = urllib.request.Request(url, data)
response = urllib.request.urlopen(req)
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.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.parse.urlencode(data)
>>> print(url_values)
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.open(full_url)
```

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

4.Headers

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

Some websites [2] dislike being browsed by programs, or send different versions to different browsers [3] . 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 [4]. 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 [5].

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }
headers = { 'User-Agent' : user_agent }
```

```
data = urllib.parse.urlencode(values)

req = urllib.request.Request(url, data, headers)

response = urllib.request.urlopen(req)

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.

5.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.

6.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')
```

7.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.

8.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'),
    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
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<?xml-stylesheet href="./css/ht2html.css"
        type="text/css"?>
<html><head><title>Error 404: File Not Found</title>
...... etc
```

9.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
```

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
```

10.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.

11.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.

12.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)
```

Note In the above example we only supplied our HTTPBasicAuthHandler to build_opener. By default openers have the handlers for normal situations – ProxyHandler, UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, 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.

13.Proxies

urllib will auto-detect your proxy settings and use those. This is through the ProxyHandler which is part of the normal handler chain. Normally that’s a good thing, but there are occasions when it may not be helpful [6]. 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)
```

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 recipe [7].

14.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)
```

15.Footnotes

This document was reviewed and revised by John Lee.

[1]☐For an introduction to the CGI protocol see Writing Web Applications in Python.

[2]☐Like Google for example. The proper way to use google from a program is to use PyGoogle of course. See Voidspace Google for some examples of using the Google API.

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

different browsers.

[4]☐The user agent for MSIE 6 is ‘Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)’

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

[6]☐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.

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