

HTTP – basic authentication example.

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2004-09/2807.html>

From: Michael Foord (fuzzyman_at_gmail.com)

Date: 09/15/04

Date: 15 Sep 2004 08:37:12 -0700

```
#!/usr/bin/python -u
# 15-09-04
# v1.0.0

# auth_example.py
# A simple script manually demonstrating basic authentication.

# Copyright Michael Foord
# Free to use, modify and relicense.
# No warranty express or implied for the accuracy, fitness to purpose
or otherwise for this code....
# Use at your own risk !!!

# E-mail or michael AT foord DOT me DOT uk
# Maintained at www.voidspace.org.uk/atlantibots/pythonutils.html

"""
There is a system for requiring a username/password before a client
can visit a webpage. This is called authentication and is implemented
by the server – it actually allows for a whole set of pages (called a
realm) to require authentication. This scheme (or schemes) are
actually defined by the HTTP spec, and so whilst python supports
authentication it doesn't document it very well. The HTTP
documentation is in the form of RFCs
(http://www.faqs.org/rfcs/rfc2617.html for basic and digest
authentication) which are technical documents and so not the most
readable !!
```

When I searched the web for details on authentication with python I found lots of people asking questions, but a lack of clear answers. This document and example code shows how to manually do basic authentication with python. It is an example for performing a simple operation rather than a technical document.

I am doing it manually rather than using an auth handler because my script is a CGI which runs once for each page access. I have to store the username/passwords between each access. A 'manual' explanation

comp.lang.python: HTTP – basic authentication example.

also shows more clearly what is happening.

I've seen references to three authentication schemes, BASIC, NTLM and DIGEST. It's possible there are more – but BASIC authentication is overwhelmingly the most common. This tutorial/example only covers BASIC authentication although some of the details may be applicable to the other schemes.

--

In all these examples we will be the python standard library urllib2 to fetch web pages.

A client is any program that makes requests over the internet. It could be a browser – or it could be a python program. When a client requests a web page it sends a request to the server. That request consists of headers with certain information about the request. Here we are calling these headers 'http request headers'. If the request fails to reach a server (the server name doesn't exist or there is no internet connection for example) then the request will just fail. If the request is made by python then an exception will be raised. This exception will have a 'reason' attribute that is a tuple describing the error.

The example below shows us creating a urllib2 request object, adding a fake 'User-Agent' request header and making a request. The resulting error shows what happens if you try to fetch a webpage without an internet connection. The User-Agent request header tells the server what program is asking for the web page – some sites (e.g. google) won't allow requests from anything other than a browser... so we might have to pretend to be a browser. (Which is generally considered bad client behaviour ? so I might get my knuckles rapped for including it here?).

```
>>> import urllib2
>>> req = urllib2.Request('http://www.google.co.uk')
>>> req.add_header('User-Agent', 'Mozilla/4.0 (compatible; MSIE 5.5;
Windows NT)')
>>> try:
    handle = urllib2.urlopen(req)
except IOError, e:
    if hasattr(e, 'reason'):
        print 'Reason : '
        print e.reason
    else:
        print handle.read()          # if we had a connection this
would print the page
```

```
Reason :
(7, 'getaddrinfo failed')
>>>
```

The actual exception is an URLError which is a subclass of IOError – the one tested for above in the try?except block. If you did a dir(e) on the above example then you would see all the attributes of the exception object.

If however the request reaches a server then the server will send a response. Whether or not the request succeeds the response will contain headers from the server (or CGI script!!). These we call here 'http response headers'. If there is a problem then the response will include an error code – you are familiar with some of them 404 : Page not found, 500 : Internal Server Error etc. In this case an exception will still be raised by urllib2, but instead of a 'reason' attribute it will have a code attribute. The code attribute is an integer that corresponds to the http error code. (see

comp.lang.python: HTTP – basic authentication example.

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> for a *full* list of codes).

If a page requires authentication then the error code is 401. Included in the response headers is a 'WWW-authenticate' header that tells you what authentication scheme the server is using for this page *and* also something called a realm. As you know it is rarely just a single page that is protected by authentication but a whole 'realm' of a website. The name of the realm is included in this header line. If the client *already knows* the username/password for this realm then it can encode them into the request headers and try again. If the username/password combination are correct, then the request will succeed as normal. If the client doesn't know the username/password it should ask the user. This means that if you enter a protected 'realm' the client effectively has to request each page twice. The first time it will get an error code and be told what realm it is attempting to access ? the client can then get the right username/password for that realm (on that server) and repeat the request.

Suppose we are attempt to fetch a webpage protected by basic authentication.

```
>>> theurl = 'http://www.someserver.com/somepath/someprotectedpage.html'  
>>> try:
```

```
    handle = urllib2.urlopen(theurl)  
except IOError, e:  
    if hasattr(e, 'code'):  
        if e.code != 401:  
            print 'We got another error'  
            print e.code  
        else:  
            print e.headers  
            print e.headers['www-authenticate']  
            # print e.headers.get('www-authenticate', '') might be a safer way
```

of doing this

Note the following things. We accessed the page directly from the url instead of creating a request object. If the exception has a 'code' attribute it also has an attribute called 'headers'. This is a dictionary like object with all the headers in ? but you can also print it to display all the headers. See the last line that displays the 'www-authenticate' header line which ought to be present whenever you get a 401 error.

```
WWW-Authenticate: Basic realm="cPanel"  
Connection: close  
Set-Cookie: cprelogin=no; path=/  
Server: cpsrvd/9.4.2  
Content-type: text/html  
Basic realm="cPanel"
```

You can see the authentication scheme and the 'realm' part of the 'www-authenticate' header. Assuming you know the username and password you can then navigate around that website ? whenever you get a 401 error with *the same realm* you can just encode the username/password into your request headers and your request should succeed.

Lets assume you need to access two pages which are likely to be in same realm. Lets also assume that you know the username and password. You can save the realm information from when you make the first access, and whenever you get a 401 and the same realm (assuming your request is from the same server) you know you can use the same username/password. So the only detail left is knowing how to encode the username/password into request header. This is done by encoding it as a base 64 string. It doesn't actually look like clear text ? but it is only the most vaguest of 'encryption'. This means basic authentication is just that ? basic. Anyone sniffing your traffic who sees an authentication request header will be able to extract your username and password from it. Many websites (like yahoo or ebay) may

comp.lang.python: HTTP – basic authentication example.

use javascript hashing/encryption to authenticate a login, which is much harder to detect and mimic. You may need to use a proxy client server and see what information your browser is actually sending to the website (See <http://groups.google.co.uk/groups?hl=en&lr=&ie=UTF-8&threadm=6f402501.0409100632> for a suggestion of several proxy servers that can do this).

There is a very simple recipe on the Activestate Python Cookbook (It's actually in the comments of this page

<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/267197>) showing how to encode a username/password into a request header. It looks like this :

```
import base64
base64string = base64.encodestring('%s:%s' % (data['name'],
data['pass']))[:-1]
req.add_header("Authorization", "Basic %s" % base64string)
```

Where req is our request object like in the first example.

Let's wrap all this up with an example that shows accessing a page, doing the authentication and saving the realm. I use a regular expression to pull the scheme and realm out of the authentication response header. I use urlparse to get the server part of a url. If we store the username/password (or the whole request header line) then we can re-use that information automatically if we come across another page in that realm.

Some websites may also use cookies with authentication. Luckily there is a library that will allow you to have automatic cookie management without thinking about it. This is ClientCookie

(<http://wwwsearch.sourceforge.net/ClientCookie/>). In python 2.4 it becomes part of the python standard library as clientcookie. See my cookbook example of how to use it

(<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/302930>).

I've done a bigger example that displays a lot of http information including cookies, headers, environment variables (the CGI environment) and all this authentication stuff. You can find it at (<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/298336>).

```
"""
import urllib2, sys, re, base64
from urlparse import urlparse
theurl = 'http://www.someserver.com/somepath/somepage.html'
# if you want to run this example you'll need to supply a protected
page with your username and password
username = 'johnny'
password = 'XXXXXX'          # a very bad password
req = urllib2.Request(theurl)
try:
    handle = urllib2.urlopen(req)
except IOError, e:           # here we are assuming we fail
    pass
else:                        # If we don't fail then the page
isn't protected
    print "This page isn't protected by authentication."
    sys.exit(1)

if not hasattr(e, 'code') or e.code != 401:           # we got
an error - but not a 401 error
    print "This page isn't protected by authentication."
    print 'But we failed for another reason.'
    sys.exit(1)
authline = e.headers.get('www-authenticate', '')      # this
gets the www-authenticat line from the headers - which has the
authentication scheme and realm in it
if not authline:
    print 'A 401 error without an authentication response header -
very weird.'
```

comp.lang.python: HTTP – basic authentication example.

```
sys.exit(1)

authobj = re.compile(r'' '(?:\s*www-authenticate\s*:)?\s*(\w*)\s+realm=["](\w+)["]' ''',
re.IGNORECASE)      # this regular expression is used to extract
scheme and realm
matchobj = authobj.match(authline)
if not matchobj:      # if the
authline isn't matched by the regular expression then something is
wrong
    print 'The authentication line is badly formed.'
    sys.exit(1)
scheme = matchobj.group(1)
realm = matchobj.group(2)
if scheme.lower() != 'basic':
    print 'This example only works with BASIC authentication.'
    sys.exit(1)
base64string = base64.encodestring('%s:%s' % (username,
password))[:-1]
authheader = "Basic %s" % base64string
req.add_header("Authorization", authheader)
try:
    handle = urllib2.urlopen(req)
except IOError, e:      # here we shouldn't fail if the
username/password is right
    print "It looks like the username or password is wrong."
    sys.exit(1)
thefirstpage = handle.read()

server = urlparse(theurl)[1].lower()      # server names are
case insensitive, so we will convert to lower case
test = server.find(':')
if test != -1: server = server[:test]      # remove the :port
information if present, we're working on the principle that realm
names per server are likely to be unique...
passdict = {(server, realm) : authheader }      # now if we get
another 401 we can test for an entry in passdict before having to ask
the user for a username/password
print 'Done successfully - information now stored in passdict.'
"""
ISSUES
CHANGELOG
15-09-04      Version 1.0.0
I think it's ok - a few references in the documentation to find.
"""
```