

Re: HTTP – basic authentication example.

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2004-09/3173.html>

From: Michael Foord (fuzzyman_at_gmail.com)

Date: 09/17/04

Date: 17 Sep 2004 05:27:57 -0700

```
"""This example is now updated to show an example at the end that uses
HTTPBasicAuthHandler *and* HTTPPasswordMgrWithDefaultRealm.
This is probably the 'right' way to do it... but is more of a pain
IMHO... (using a password manager either means *already* knowing the
realm... or *never* knowing the realm..)
"""
```

```
#!/usr/bin/python -u
# 16-09-04
# v1.0.0
```

```
# auth_example.py
# A simple CGI 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/atlabibots/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 be protected by 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.

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

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 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 using 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 next example 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) in 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

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

response back. Whether or not the request succeeds the response will still contain headers from the server (or CGI script!!). These we call here 'http response headers'. If there is a problem then this response will include an error code that describes the problem. You will already be familiar with some of these codes - 404 : Page not found, 500 : Internal Server Error etc. If this happens 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 <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. It is rarely just a single page that is protected by authentication but a section - a '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.

HTTP is a 'stateless' protocol. This means that a server using basic authentication won't 'remember' you are logged in and will need to be sent the right header for every protected page you attempt to access. Suppose we 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.

Output from above example :

```
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

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

into your request headers and your request should succeed.

Lets assume you need to access pages which are all in the same realm.

Assuming you have got the username and password from the user, you save the name of the realm from the first access. Then whenever you

get a 401 error in the same realm (from the same server !) you know the username/password to use. So the only detail left, is knowing how

to encode the username/password into the 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, use javascript hashing/encryption and other tricks to authenticate a login. This is much harder to detect and mimic from

python ! 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=6f402501.0409100632.47e403f3%40posti>

for suggestions 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 the 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. When the code has run the contents of the page

we've fetched is saved as a string in the variable 'thepage'. If you are writing an http client of any sort that has to deal with basic

authentication then this example should have everything you need to know - but see the comment below about cookies.

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://www.voidspace.org.uk/atlantibots/recipebook.html#http> - it's a

CGI and there's an online version to try. It shows lot's of the information about http state, the CGI environment etc..

--

In actual fact the 'proper' way to do BASIC authentication with Python

is to install an authentication handler as an 'opener' (along with a password manager) in urllib2. It doesn't show as clearly what is

happening and is less suitable for my needs (within a CGI), where a pickled dictionary is more usefl. See at the bottom of this example

for an alternative example using an auth handler - this was sent to me by Jaime Wyant (and amended by me)...

```
"""
```

```
import urllib2, sys, re, base64
```

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

```
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.'
    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)
thepage = 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
```

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

```
another 401 we can test for an entry in passwdict before having to ask
the user for a username/password
print 'Done successfully - information now stored in passwdict.'
print 'The webpage is stored in thepage.'
"""
```

The proper way of actually doing this is to use an HTTPBasicAuthHandler along with an HTTPPasswordMgr.

The python documentation on this is actually pretty minimal - so below is an example showing how to do this.

The main problem with HTTPPasswordMgr is that you must **already** know the realm - so we're going to use the HTTPPasswordMgrWithDefaultRealm instead !!

```
theurl = 'http://www.someserver.com/highestlevelprotectedpath/somepage.htm'
username = 'johnny'
password = 'XXXXXX'          # a great password
passman = urllib2.HTTPPasswordMgrWithDefaultRealm()      # this
creates a password manager
passman.add_password(None, theurl, username, password)    # because
we have put None at the start it will always use this
username/password combination
authhandler = urllib2.HTTPBasicAuthHandler(passman)      #
create the AuthHandler
opener = urllib2.build_opener(authhandler)
    # build an 'opener' using the handler we've created
# you can use the opener directly to open URLs
# *or* you can install it as the default opener so that all calls to
urllib2.urlopen use this opener
urllib2.install_opener(opener)
# All calls to urllib2.urlopen will now use our handler
ISSUES
CHANGELOG
16-09-04          Version 1.0.0
I think it's ok.
"""
```