'''

common XBMC Module

Copyright (C) 2011 t0mm0

This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program. If not, see <http://www.gnu.org/licenses/>.

'''

import cookielib

import gzip

import re

import StringIO

import urllib

import urllib2

import socket

#Set Global timeout - Useful for slow connections and Putlocker.

socket.setdefaulttimeout(60)

class HeadRequest(urllib2.Request):

'''A Request class that sends HEAD requests'''

def get\_method(self):

return 'HEAD'

class Net:

'''

This class wraps :mod:`urllib2` and provides an easy way to make http

requests while taking care of cookies, proxies, gzip compression and

character encoding.

Example::

from t0mm0.common.net import Net

net = Net()

response = net.http\_GET('http://xbmc.org')

print response.content

'''

\_cj = cookielib.LWPCookieJar()

\_proxy = None

\_user\_agent = 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.137 Safari/537.36'

\_http\_debug = False

def \_\_init\_\_(self, cookie\_file='', proxy='', user\_agent='',

http\_debug=False):

'''

Kwargs:

cookie\_file (str): Full path to a file to be used to load and save

cookies to.

proxy (str): Proxy setting (eg.

``'http://user:pass@example.com:1234'``)

user\_agent (str): String to use as the User Agent header. If not

supplied the class will use a default user agent (chrome)

http\_debug (bool): Set ``True`` to have HTTP header info written to

the XBMC log for all requests.

'''

if cookie\_file:

self.set\_cookies(cookie\_file)

if proxy:

self.set\_proxy(proxy)

if user\_agent:

self.set\_user\_agent(user\_agent)

self.\_http\_debug = http\_debug

self.\_update\_opener()

def set\_cookies(self, cookie\_file):

'''

Set the cookie file and try to load cookies from it if it exists.

Args:

cookie\_file (str): Full path to a file to be used to load and save

cookies to.

'''

try:

self.\_cj.load(cookie\_file, ignore\_discard=True)

self.\_update\_opener()

return True

except:

return False

def get\_cookies(self):

'''Returns A dictionary containing all cookie information by domain.'''

return self.\_cj.\_cookies

def save\_cookies(self, cookie\_file):

'''

Saves cookies to a file.

Args:

cookie\_file (str): Full path to a file to save cookies to.

'''

self.\_cj.save(cookie\_file, ignore\_discard=True)

def set\_proxy(self, proxy):

'''

Args:

proxy (str): Proxy setting (eg.

``'http://user:pass@example.com:1234'``)

'''

self.\_proxy = proxy

self.\_update\_opener()

def get\_proxy(self):

'''Returns string containing proxy details.'''

return self.\_proxy

def set\_user\_agent(self, user\_agent):

'''

Args:

user\_agent (str): String to use as the User Agent header.

'''

self.\_user\_agent = user\_agent

def get\_user\_agent(self):

'''Returns user agent string.'''

return self.\_user\_agent

def \_update\_opener(self):

'''

Builds and installs a new opener to be used by all future calls to

:func:`urllib2.urlopen`.

'''

if self.\_http\_debug:

http = urllib2.HTTPHandler(debuglevel=1)

else:

http = urllib2.HTTPHandler()

if self.\_proxy:

opener = urllib2.build\_opener(urllib2.HTTPCookieProcessor(self.\_cj),

urllib2.ProxyHandler({'http':

self.\_proxy}),

urllib2.HTTPBasicAuthHandler(),

http)

else:

opener = urllib2.build\_opener(urllib2.HTTPCookieProcessor(self.\_cj),

urllib2.HTTPBasicAuthHandler(),

http)

urllib2.install\_opener(opener)

def http\_GET(self, url, headers={}, compression=True):

'''

Perform an HTTP GET request.

Args:

url (str): The URL to GET.

Kwargs:

headers (dict): A dictionary describing any headers you would like

to add to the request. (eg. ``{'X-Test': 'testing'}``)

compression (bool): If ``True`` (default), try to use gzip

compression.

Returns:

An :class:`HttpResponse` object containing headers and other

meta-information about the page and the page content.

'''

return self.\_fetch(url, headers=headers, compression=compression)

def http\_POST(self, url, form\_data, headers={}, compression=True):

'''

Perform an HTTP POST request.

Args:

url (str): The URL to POST.

form\_data (dict): A dictionary of form data to POST.

Kwargs:

headers (dict): A dictionary describing any headers you would like

to add to the request. (eg. ``{'X-Test': 'testing'}``)

compression (bool): If ``True`` (default), try to use gzip

compression.

Returns:

An :class:`HttpResponse` object containing headers and other

meta-information about the page and the page content.

'''

return self.\_fetch(url, form\_data, headers=headers,

compression=compression)

def http\_HEAD(self, url, headers={}):

'''

Perform an HTTP HEAD request.

Args:

url (str): The URL to GET.

Kwargs:

headers (dict): A dictionary describing any headers you would like

to add to the request. (eg. ``{'X-Test': 'testing'}``)

Returns:

An :class:`HttpResponse` object containing headers and other

meta-information about the page.

'''

req = HeadRequest(url)

req.add\_header('User-Agent', self.\_user\_agent)

for k, v in headers.items():

req.add\_header(k, v)

response = urllib2.urlopen(req)

return HttpResponse(response)

def \_fetch(self, url, form\_data={}, headers={}, compression=True):

'''

Perform an HTTP GET or POST request.

Args:

url (str): The URL to GET or POST.

form\_data (dict): A dictionary of form data to POST. If empty, the

request will be a GET, if it contains form data it will be a POST.

Kwargs:

headers (dict): A dictionary describing any headers you would like

to add to the request. (eg. ``{'X-Test': 'testing'}``)

compression (bool): If ``True`` (default), try to use gzip

compression.

Returns:

An :class:`HttpResponse` object containing headers and other

meta-information about the page and the page content.

'''

encoding = ''

req = urllib2.Request(url)

if form\_data:

form\_data = urllib.urlencode(form\_data)

req = urllib2.Request(url, form\_data)

req.add\_header('User-Agent', self.\_user\_agent)

for k, v in headers.items():

req.add\_header(k, v)

if compression:

req.add\_header('Accept-Encoding', 'gzip')

response = urllib2.urlopen(req)

return HttpResponse(response)

class HttpResponse:

'''

This class represents a resoponse from an HTTP request.

The content is examined and every attempt is made to properly encode it to

Unicode.

.. seealso::

:meth:`Net.http\_GET`, :meth:`Net.http\_HEAD` and :meth:`Net.http\_POST`

'''

content = ''

'''Unicode encoded string containing the body of the reposne.'''

def \_\_init\_\_(self, response):

'''

Args:

response (:class:`mimetools.Message`): The object returned by a call

to :func:`urllib2.urlopen`.

'''

self.\_response = response

html = response.read()

try:

if response.headers['content-encoding'].lower() == 'gzip':

html = gzip.GzipFile(fileobj=StringIO.StringIO(html)).read()

except:

pass

try:

content\_type = response.headers['content-type']

if 'charset=' in content\_type:

encoding = content\_type.split('charset=')[-1]

except:

pass

r = re.search('<meta\s+http-equiv="Content-Type"\s+content="(?:.+?);' +

'\s+charset=(.+?)"', html, re.IGNORECASE)

if r:

encoding = r.group(1)

try:

html = unicode(html, encoding)

except:

pass

self.content = html

def get\_headers(self):

'''Returns a List of headers returned by the server.'''

return self.\_response.info().headers

def get\_url(self):

'''

Return the URL of the resource retrieved, commonly used to determine if

a redirect was followed.

'''

return self.\_response.geturl()