mechanize — Documentation

This documentation is in need of reorganisation!

This page is the old ClientCookie documentation. It deals with operation on the level of `urllib2 Handler` objects, and also with adding headers, debugging, and cookie handling. See the **front page** for more typical use.

Examples

```
import mechanize
response = mechanize.urlopen("http://example.com/")
```

This function behaves identically to `urllib2.urlopen()`, except that it deals with cookies automatically.

Here is a more complicated example, involving `Request` objects (useful if you want to pass `Requests` around, add headers to them, etc.):

```
import mechanize
```

```
request = mechanize.Request("http://example.com/")
# note we're using the urlopen from mechanize, not urllib2
response = mechanize.urlopen(request)
# let's say this next request requires a cookie that was set
# in response
request2 = mechanize.Request("http://example.com/spam.html")
response2 = mechanize.urlopen(request2)

print response2.geturl()
print response2.info()  # headers
print response2.read()  # body (readline and readlines work too)
```

In these examples, the workings are hidden inside the `mechanize.urlopen()` function, which is an extension of `urllib2.urlopen()`. Redirects, proxies and cookies are handled automatically by this function (note that you may need a bit of configuration to get your proxies correctly set up: see `urllib2` documentation).

There is also a `urlretrieve()` function, which works like `urllib.urlretrieve()`.

An example at a slightly lower level shows how the module processes cookies more clearly:

```
# Don't copy this blindly!  You probably want to follow the examples
# above, not this one.
import mechanize

# Build an opener that *doesn't* automatically call .add_cookie_header()
# and .extract_cookies(), so we can do it manually without interference.
class NullCookieProcessor(mechanize.HTTPCookieProcessor):
    def http_request(self, request): return request
    def http_response(self, request, response): return response
opener = mechanize.build_opener(NullCookieProcessor)

request = mechanize.Request("http://example.com/")
response = mechanize.urlopen(request)
cj = mechanize.CookieJar()
cj.extract_cookies(response, request)
# let's say this next request requires a cookie that was set in response
request2 = mechanize.Request("http://example.com/spam.html")
cj.add_cookie_header(request2)
response2 = mechanize.urlopen(request2)
```

The `CookieJar` class does all the work. There are essentially two operations: `.extract_cookies()` extracts HTTP cookies from `Set-Cookie` (the original **Netscape cookie standard**) and `Set-Cookie2` (**RFC 2965**) headers from a response if and only if they should be set given the request, and `.add_cookie_header()` adds `Cookie` headers if and only if they are appropriate for a particular HTTP request. Incoming cookies are checked for acceptability based on the host name, etc. Cookies are only set on outgoing requests if they match the request's host name, path, etc.

Note that if you're using `mechanize.urlopen()` (or if you're using `mechanize.HTTPCookieProcessor` by some other means), you don't need to call `.extract_cookies()` or `.add_cookie_header()` yourself. If, on the other hand, you want to use mechanize to provide cookie handling for an HTTP client other than mechanize itself, you will need to use this pair of methods. You can make your own `request` and `response` objects, which must support the interfaces described in the docstrings of `.extract_cookies()` and `.add_cookie_header()`.

There are also some `CookieJar` subclasses which can store cookies in files and databases. `FileCookieJar` is the abstract class for `CookieJar`s that can store cookies in disk files. `LWPCookieJar` saves cookies in a format compatible with the libwww-perl library. This class is convenient if you want to store cookies in a human-readable file:

```
import mechanize
cj = mechanize.LWPCookieJar()
cj.revert("cookie3.txt")
opener = mechanize.build_opener(mechanize.HTTPCookieProcessor(cj))
r = opener.open("http://foobar.com/")
cj.save("cookie3.txt")
```

The `.revert()` method discards all existing cookies held by the `CookieJar` (it won't lose any existing cookies if the load fails). The `.load()` method, on the other hand, adds the loaded cookies to existing cookies held in the `CookieJar` (old cookies are kept unless overwritten by newly loaded ones).

`MozillaCookieJar` can load and save to the Mozilla/Netscape/lynx-compatible `'cookies.txt'` format. This format loses some information (unusual and nonstandard cookie attributes such as comment, and also information specific to RFC 2965 cookies). The subclass `MSIECookieJar` can load (but not save) from Microsoft Internet Explorer's cookie files on Windows.

Important note

Only use names you can import directly from the `mechanize` package, and that don't start with a single underscore. Everything else is subject to change or disappearance without notice.

Cooperating with Browsers

Firefox since version 3 persists cookies in an sqlite database, which is not supported by MozillaCookieJar.

The subclass `MozillaCookieJar` differs from `CookieJar` only in storing cookies using a different, Firefox 2/Mozilla/Netscape-compatible, file format known as “cookies.txt”. The lynx browser also uses this format. This file format can’t store RFC 2965 cookies, so they are downgraded to Netscape cookies on saving. `LWPCookieJar` itself uses a libwww-perl specific format (`Set-Cookie3’) — see the example above. Python and your browser should be able to share a cookies file (note that the file location here will differ on non-unix OSes):

WARNING: you may want to back up your browser’s cookies file if you use `MozillaCookieJar` to save cookies. I *think* it works, but there have been bugs in the past!

```
import os, mechanize
cookies = mechanize.MozillaCookieJar()
cookies.load(os.path.join(os.environ["HOME"], "/.netscape/cookies.txt"))
# see also the save and revert methods
```

Note that cookies saved while Mozilla is running will get clobbered by Mozilla — see `MozillaCookieJar.__doc__`.

`MSIECookieJar` does the same for Microsoft Internet Explorer (MSIE) 5.x and 6.x on Windows, but does not allow saving cookies in this format. In future, the Windows API calls might be used to load and save (though the index has to be read directly, since there is no API for that, AFAIK; there’s also an unfinished `MSIEDBCookieJar`, which uses (reads and writes) the Windows MSIE cookie database directly, rather than storing copies of cookies as `MSIECookieJar` does).

```
import mechanize
```

```
cj = mechanize.MSIECookieJar(delayload=True)
cj.load_from_registry()  # finds cookie index file from registry
```

A true `delayload` argument speeds things up.

On Windows 9x (win 95, win 98, win ME), you need to supply a username to the `.load_from_registry()` method:

```
cj.load_from_registry(username="jbloggs")
```

Konqueror/Safari and Opera use different file formats, which aren't yet supported.

Saving cookies in a file

If you have no need to co-operate with a browser, the most convenient way to save cookies on disk between sessions in human-readable form is to use `LWPCookieJar`. This class uses a libwww-perl specific format (`Set-Cookie3'). Unlike `MozilliaCookieJar`, this file format doesn't lose information.

Supplying a CookieJar

You might want to do this to **use your browser's cookies**, to customize `CookieJar`'s behaviour by passing constructor arguments, or to be able to get at the cookies it will hold (for example, for saving cookies between sessions and for debugging).

If you're using the higher-level `urllib2`-like interface (`urlopen()`, etc), you'll have to let it know what `CookieJar` it should use:

```
import mechanize
cookies = mechanize.CookieJar()
# build_opener() adds standard handlers (such as HTTPHandler and
# HTTPCookieProcessor) by default.  The cookie processor we supply
# will replace the default one.
opener = mechanize.build_opener(mechanize.HTTPCookieProcessor(cookies))

r = opener.open("http://example.com/")  # GET
r = opener.open("http://example.com/", data)  # POST
```

The `urlopen()` function uses a global `OpenerDirector` instance to do its work, so if you want to use `urlopen()` with your own `CookieJar`, install the `OpenerDirector` you built with `build_opener()` using the `mechanize.install_opener()` function, then proceed as usual:

```
mechanize.install_opener(opener)
r = mechanize.urlopen("http://example.com/")
```

Of course, everyone using `urlopen` is using the same global `CookieJar` instance!

You can set a policy object (must satisfy the interface defined by `mechanize.CookiePolicy`), which determines which cookies are allowed to be set and returned. Use the `policy` argument to the `CookieJar` constructor, or use the `.set\_policy()` method. The default implementation has some useful switches:

```
from mechanize import CookieJar, DefaultCookiePolicy as Policy
cookies = CookieJar()
# turn on RFC 2965 cookies, be more strict about domains when setting and
# returning Netscape cookies, and block some domains from setting cookies
```

```
 # or having them returned (read the DefaultCookiePolicy docstring for the
 # domain matching rules here)
 policy = Policy(rfc2965=True, strict_ns_domain=Policy.DomainStrict,
                 blocked_domains=["ads.net", ".ads.net"])
 cookies.set_policy(policy)
```

Additional Handlers

The following handlers are provided in addition to those provided by `urllib2`:

`HTTPRobotRulesProcessor`

> WWW Robots (also called wanderers or spiders) are programs that traverse many pages in the World Wide Web by recursively retrieving linked pages. This kind of program can place significant loads on web servers, so there is a **standard** for a `robots.txt` file by which web site operators can request robots to keep out of their site, or out of particular areas of it. This handler uses the standard Python library's `robotparser` module. It raises `mechanize.RobotExclusionError` (subclass of `mechanize.HTTPError`) if an attempt is made to open a URL prohibited by `robots.txt`.

`HTTPEquivProcessor`

> The `<META HTTP-EQUIV>` tag is a way of including data in HTML to be treated as if it were part of the HTTP headers. mechanize can automatically read these tags and add the `HTTP-EQUIV` headers to the response object's real HTTP headers. The HTML is left unchanged.

`HTTPRefreshProcessor`

The `Refresh` HTTP header is a non-standard header which is widely used. It requests that the user-agent follow a URL after a specified time delay. mechanize can treat these headers (which may have been set in `<META HTTP-EQUIV>` tags) as if they were 302 redirections. Exactly when and how `Refresh` headers are handled is configurable using the constructor arguments.

`HTTPRefererProcessor`

The `Referer` HTTP header lets the server know which URL you've just visited. Some servers use this header as state information, and don't like it if this is not present. It's a chore to add this header by hand every time you make a request. This adds it automatically. **NOTE**: this only makes sense if you use each handler for a single chain of HTTP requests (so, for example, if you use a single HTTPRefererProcessor to fetch a series of URLs extracted from a single page, **this will break**). **mechanize.Browser** does this properly.

Example:

```
import mechanize
cookies = mechanize.CookieJar()

opener = mechanize.build_opener(mechanize.HTTPRefererProcessor,
                                mechanize.HTTPEquivProcessor,
                                mechanize.HTTPRefreshProcessor,
                                )
opener.open("http://www.rhubarb.com/")
```

Seekable responses

Response objects returned from (or raised as exceptions by) `mechanize.SeekableResponseOpener`, `mechanize.UserAgent` (if `.set_seekable_responses(True)` has been called) and `mechanize.Browser()` have `.seek()`, `.get_data()` and `.set_data()` methods:

```
import mechanize
opener = mechanize.OpenerFactory(mechanize.SeekableResponseOpener).build_opener()
response = opener.open("http://example.com/")
# same return value as .read(), but without affecting seek position
total_nr_bytes = len(response.get_data())
assert len(response.read()) == total_nr_bytes
assert len(response.read()) == 0  # we've already read the data
response.seek(0)
assert len(response.read()) == total_nr_bytes
response.set_data("blah\n")
assert response.get_data() == "blah\n"
...
```

This caching behaviour can be avoided by using `mechanize.OpenerDirector`. It can also be avoided with `mechanize.UserAgent`. Note that `HTTPEquivProcessor` and `HTTPResponseDebugProcessor` require seekable responses and so are not compatible with `mechanize.OpenerDirector` and `mechanize.UserAgent`.

```
import mechanize
ua = mechanize.UserAgent()
ua.set_seekable_responses(False)
ua.set_handle_equiv(False)
ua.set_debug_responses(False)
```

Note that if you turn on features that use seekable responses (currently: HTTP-EQUIV handling and response body debug printing), returned responses *may* be seekable as a side-effect of these features. However, this is not guaranteed (currently, in these cases, returned response objects are seekable, but raised respose objects — `mechanize.HTTPError` instances — are not seekable). This applies regardless of whether you use `mechanize.UserAgent` or `mechanize.OpenerDirector`. If you explicitly request seekable responses by calling `.set_seekable_responses(True)` on a `mechanize.UserAgent` instance, or by using `mechanize.Browser` or `mechanize.SeekableResponseOpener`, which always return seekable responses, then both returned and raised responses are guaranteed to be seekable.

Handlers should call `response = mechanize.seek_wrapped_response(response)` if they require the `.seek()`, `.get_data()` or `.set_data()` methods.

Request object lifetime

Note that handlers may create new `Request` instances (for example when performing redirects) rather than adding headers to existing `Request` objects.

Adding headers

Adding headers is done like so:

```
import mechanize
req = mechanize.Request("http://foobar.com/")
req.add_header("Referer", "http://wwwsearch.sourceforge.net/mechanize/")
r = mechanize.urlopen(req)
```

You can also use the `headers` argument to the `mechanize.Request` constructor.

mechanize adds some headers to `Request` objects automatically — see the next section for details.

Automatically-added headers

`OpenerDirector` automatically adds a `User-Agent` header to every `Request`.

To change this and/or add similar headers, use your own `OpenerDirector`:

```
import mechanize
cookies = mechanize.CookieJar()
opener = mechanize.build_opener(mechanize.HTTPCookieProcessor(cookies))
opener.addheaders = [("User-agent", "Mozilla/5.0 (compatible; MyProgram/0.1)"),
                     ("From", "responsible.person@example.com")]
```

Again, to use `urlopen()`, install your `OpenerDirector` globally:

```
mechanize.install_opener(opener)
r = mechanize.urlopen("http://example.com/")
```

Also, a few standard headers (`Content-Length`, `Content-Type` and `Host`) are added when the `Request` is passed to `urlopen()` (or `OpenerDirector.open()`). You shouldn't need to change these headers, but since this is done by `AbstractHTTPHandler`, you can change the way it works by passing a subclass of that handler to `build_opener()` (or, as always, by constructing an opener yourself and calling `.add_handler()`).

Initiating unverifiable transactions

This section is only of interest for correct handling of third-party HTTP cookies. See **below** for an explanation of 'third-party'.

First, some terminology.

An *unverifiable request* (defined fully by (**RFC 2965**) is one whose URL the user did not have the option to approve. For example, a transaction is unverifiable if the request is for an image in an HTML document, and the user had no option to approve the fetching of the image from a particular URL.

The *request-host of the origin transaction* (defined fully by RFC 2965) is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this is the request-host of the request for the page containing the image.

mechanize knows that redirected transactions are unverifiable, and will handle that on its own (ie. you don't need to think about the origin request-host or verifiability yourself).

If you want to initiate an unverifiable transaction yourself (which you should if, for example, you're downloading the images from a page, and 'the user' hasn't explicitly OKed those URLs):

```
request = Request(origin_req_host="www.example.com", unverifiable=True)
```

RFC 2965 support

Support for the RFC 2965 protocol is switched off by default, because few browsers implement it, so the RFC 2965 protocol is essentially never seen on the internet. To switch it on, see **here**.

Parsing HTTP dates

A function named `str2time` is provided by the package, which may be useful for parsing dates in HTTP headers. `str2time` is intended to be liberal, since HTTP date/time formats are poorly standardised in practice. There is no need to use this function in normal operations: `CookieJar` instances keep track of cookie lifetimes automatically. This function will stay around in some form, though the supported date/time formats may change.

Dealing with bad HTML

XXX Intro

XXX Test me

Note about cookie standards

There are several standards relevant to HTTP cookies.

The Netscape protocol is the only standard supported by most web browsers (including Internet Explorer and Firefox). This is a *de facto* standard defined by the behaviour of popular browsers, and neither the **cookie_spec.html** document that was published by Netscape, nor the RFCs that were published later, describe the Netscape protocol accurately or completely. Netscape protocol cookies are also known as V0 cookies, to distinguish them from RFC 2109 or RFC 2965 cookies, which have a version cookie-attribute with a value of 1.

RFC 2109 was introduced to fix some problems identified with the Netscape protocol, while still keeping the same HTTP headers (`Cookie` and `Set-Cookie`). The most prominent of these problems is the ‘third-party’ cookie issue, which was an accidental feature of the Netscape protocol. Some features defined by RFC2109 (such as the port and max-age cookie attributes) are now part of the de facto Netscape protocol, but the RFC was never implemented fully by browsers, because of differences in behaviour between the Netscape and Internet Explorer browsers of the time.

RFC 2965 attempted to fix the compatibility problem by introducing two new headers, `Set-Cookie2` and `Cookie2`. Unlike the `Cookie` header, `Cookie2` does *not* carry cookies to the server — rather, it simply advertises to the server that RFC 2965 is understood. `Set-Cookie2` *does* carry cookies, from server to client: the new header means that both IE and Netscape ignore these cookies. This preserves backwards compatibility, but popular browsers did not implement the RFC, so it was never widely adopted. One confusing point to note about RFC 2965 is that it uses the same value (1) of the Version attribute in HTTP headers as does RFC 2109. See also **RFC 2964**, which discusses use of the protocol.

Because Netscape cookies are so poorly specified, the general philosophy of the module's Netscape protocol implementation is to start with RFC 2965 and open holes where required for Netscape protocol-compatibility. RFC 2965 cookies are *always* treated as RFC 2965 requires, of course.

There is more information about the history of HTTP cookies in **this paper by David Kristol**.

Recently (2011), **an IETF effort has started** to specify the syntax and semantics of the `Cookie` and `Set-Cookie` headers as they are actually used on the internet.

I prefer questions and comments to be sent to the **mailing list** rather than direct to me.

John J. Lee, March 2011.