mechanize Documentation

Release 0.4.8

Kovid Goyal

May 16, 2023

Table of Contents:

Stateful programmatic web browsing in Python. Browse pages programmatically with easy HTML form filling and clicking of links.

CHAPTER 1

Frequently Asked Questions

Contents

1.1 General

1.1.1 Which version of Python do I need?

mechanize works on all python versions, python 2 (>= 2.7) and 3 (>= 3.5).

1.1.2 What dependencies does mechanize need?

```
html5lib
```

1.1.3 What license does mechanize use?

mechanize is licensed under the BSD-3-clause license.

1.2 Usage

1.2.1 I'm not getting the HTML page I expected to see?

See *Debugging*.

1.2.2 Is JavaScript supported?

No, sorry. See *JavaScript is messing up my web-scraping. What do I do?*

1.2.3 My HTTP response data is truncated?

mechanize.Browser's response objects support the *.seek()* method, and can still be used after *.close()* has been called. Response data is not fetched until it is needed, so navigation away from a URL before fetching all of the response will truncate it. Call *response.get_data()* before navigation if you don't want that to happen.

1.2.4 Is there any example code?

Look in the *examples/* directory. Note that the examples on the forms page are executable as-is. Contributions of example code would be very welcome!

1.3 Cookies

1.3.1 Which HTTP cookie protocols does mechanize support?

Netscape and RFC 2965. RFC 2965 handling is switched off by default.

1.3.2 What about RFC 2109?

RFC 2109 cookies are currently parsed as Netscape cookies, and treated by default as RFC 2965 cookies thereafter if RFC 2965 handling is enabled, or as Netscape cookies otherwise.

1.3.3 Why don't I have any cookies?

See *Cookies*.

1.3.4 My response claims to be empty, but I know it's not?

Did you call *response.read()* (e.g., in a debug statement), then forget that all the data has already been read? In that case, you may want to use *mechanize.response_seek_wrapper*. *mechanize.Browser* always returns seekable responses, so it's not necessary to use this explicitly in that case.

1.3.5 What's the difference between the *.load()* and *.revert()* methods of *CookieJar*?

.load() appends cookies from a file. *.revert()* discards all existing cookies held by the *CookieJar* first (but it won't lose any existing cookies if the loading fails).

1.3.6 Is it threadsafe?

See *Thread safety*.

1.3.7 How do I do *X*?

Refer to the API documentation in *Browser API*.

1.4 Forms

1.4.1 How do I figure out what control names and values to use?

print(form) is usually all you need. In your code, things like the *HTMLForm.items* attribute of `mechanize.HTMLForm` instances can be useful to inspect forms at runtime. Note that it's possible to use item labels instead of item names, which can be useful — use the *by_label* arguments to the various methods, and the *.get_value_by_label()* / *.set_value_by_label()* methods on *ListControl*.

1.4.2 What do those '*' characters mean in the string representations of list controls?

A * next to an item means that item is selected.

1.4.3 What do those parentheses (round brackets) mean in the string representations of list controls?

Parentheses *(foo)* around an item mean that item is disabled.

1.4.4 Why doesn't *<some control>* turn up in the data returned by *.click*()* when that control has non-*None* value?

Either the control is disabled, or it is not successful for some other reason. 'Successful' (see HTML 4 specification) means that the control will cause data to get sent to the server.

1.4.5 Why does mechanize not follow the HTML 4.0 / RFC 1866 standards for *RADIO* and multiple-selection *SELECT* controls?

Because by default, it follows browser behaviour when setting the initially-selected items in list controls that have no items explicitly selected in the HTML.

1.4.6 Why does *.click()* ing on a button not work for me?

Clicking on a *RESET* button doesn't do anything, by design - this is a library for web automation, not an interactive browser. Even in an interactive browser, clicking on *RESET* sends nothing to the server, so there is little point in having *.click()* do anything special here.

Clicking on a *BUTTON TYPE=BUTTON* doesn't do anything either, also by design. This time, the reason is that that *BUTTON* is only in the HTML standard so that one can attach JavaScript callbacks to its events. Their execution may result in information getting sent back to the server. mechanize, however, knows nothing about these callbacks, so it can't do anything useful with a click on a *BUTTON* whose type is *BUTTON*.

Generally, JavaScript may be messing things up in all kinds of ways. See *JavaScript is messing up my web-scraping. What do I do?*.

1.4.7 How do I change *INPUT TYPE=HIDDEN* field values (for example, to emulate the effect of JavaScript code)?

As with any control, set the control's *readonly* attribute false.

```
form.find_control("foo").readonly = False # allow changing .value of control foo
form.set_all_readonly(False) # allow changing the .value of all controls
```

1.4.8 I'm having trouble debugging my code.

See *Debugging*.

1.4.9 I have a control containing a list of integers. How do I select the one whose value is nearest to the one I want?

```
import bisect
def closest_int_value(form, ctrl_name, value):
    values = map(int, [item.name for item in form.find_control(ctrl_name).items])
    return str(values[bisect.bisect(values, value) - 1])

form["distance"] = [closest_int_value(form, "distance", 23)]
```

1.5 Miscellaneous

1.5.1 I want to see what my web browser is doing?

Use the developer tools for your browser (you may have to install them first). These provide excellent views into all HTTP requests/responses in the browser.

1.5.2 JavaScript is messing up my web-scraping. What do I do?

JavaScript is used in web pages for many purposes – for example: creating content that was not present in the page at load time, submitting or filling in parts of forms in response to user actions, setting cookies, etc. mechanize does not provide any support for JavaScript.

If you come across this in a page you want to automate, you have a few options. Here they are, roughly in order of simplicity:

- Figure out what the JavaScript is doing and emulate it in your Python code. The simplest case is if the JavaScript is setting some cookies. In that case you can inspect the cookies in your browser and emulate setting them in mechanize with `mechanize.Browser.set_simple_cookie()`.
- More complex is to use your browser developer tools to see exactly what requests are sent by the browser and emulate them in mechanize by using `mechanize.Request` to create the request manually and open it with `mechanize.Browser.open()`.
- Third is to use some browser automation framework/library to scrape the site instead of using mechanize. These libraries typically drive a headless version of a full browser that can execute all JavaScript. They are typically much slower than using mechanize and far more resource intensive, but do work as a last resort.

CHAPTER 2

Browser API

API documentation for the mechanize `Browser` object. You can create a mechanize `Browser` instance as:

```
from mechanize import Browser
br = Browser()
```

Contents

2.1 The Browser

class mechanize.**Browser**(*history=None*, *request_class=None*, *content_parser=None*, *factory_class=<class mechanize._html.Factory>*, *allow_xhtml=False*)

Browser-like class with support for history, forms and links.

`BrowserStateError` is raised whenever the browser is in the wrong state to complete the requested operation - e.g., when `back()` is called when the browser history is empty, or when `follow_link()` is called when the current response does not contain HTML data.

Public attributes:

request: current request (`mechanize.Request`)

form: currently selected form (see `select_form()`)

Parameters

- **history** – object implementing the `mechanize.History` interface. Note this interface is still experimental and may change in future. This object is owned by the browser instance and must not be shared among browsers.
- **request_class** – Request class to use. Defaults to `mechanize.Request`
- **content_parser** – A function that is responsible for parsing received html/xhtml content. See the builtin `mechanize._html.content_parser()` function for details on the interface this function must support.
- **factory_class** – HTML Factory class to use. Defaults to `mechanize.Factory`

add_client_certificate(*url*, *key_file*, *cert_file*)

Add an SSL client certificate, for HTTPS client auth.

key_file and cert_file must be filenames of the key and certificate files, in PEM format. You can use e.g. OpenSSL to convert a p12 (PKCS 12) file to PEM format:

openssl pkcs12 -clcerts -nokeys -in cert.p12 -out cert.pem openssl pkcs12 -nocerts -in cert.p12 -out key.pem

Note that client certificate password input is very inflexible ATM. At the moment this seems to be console only, which is presumably the default behaviour of libopenssl. In future mechanize may support third-party libraries that (I assume) allow more options here.

back(*n=1*)

Go back n steps in history, and return response object.

n: go back this number of steps (default 1 step)

click(**args*, ***kwds*)

See `mechanize.HTMLForm.click()` for documentation.

click_link(*link=None*, ***kwds*)

Find a link and return a Request object for it.

Arguments are as for `find_link()`, except that a link may be supplied as the first argument.

cookiejar

Return the current cookiejar (`mechanize.CookieJar`) or None

find_link(*text=None*, *text_regex=None*, *name=None*, *name_regex=None*, *url=None*, *url_regex=None*, *tag=None*, *predicate=None*, *nr=0*)

Find a link in current page.

Links are returned as `mechanize.Link` objects. Examples:

```
# Return third link that .search()-matches the regexp "python" (by
# ".search()-matches", I mean that the regular expression method
# .search() is used, rather than .match()).
find_link(text_regex=re.compile("python"), nr=2)

# Return first http link in the current page that points to
# somewhere on python.org whose link text (after tags have been
# removed) is exactly "monty python".
find_link(text="monty python",
        url_regex=re.compile("http.*python.org"))

# Return first link with exactly three HTML attributes.
find_link(predicate=lambda link: len(link.attrs) == 3)
```

Links include anchors <*a*>, image maps <*area*>, and frames <*iframe*>.

All arguments must be passed by keyword, not position. Zero or more arguments may be supplied. In order to find a link, all arguments supplied must match.

If a matching link is not found, `mechanize.LinkNotFoundError` is raised.

Parameters

- **`text`** – link text between link tags: e.g. <a href="blah">this bit</a> with whitespace compressed.
- **`text_regex`** – link text between tag (as defined above) must match the regular expression object or regular expression string passed as this argument, if supplied
- **`name`** – as for text and text_regex, but matched against the name HTML attribute of the link tag
- **`url`** – as for text and text_regex, but matched against the URL of the link tag (note this matches against Link.url, which is a relative or absolute URL according to how it was written in the HTML)
- **`tag`** – element name of opening tag, e.g. "a"
- **`predicate`** – a function taking a Link object as its single argument, returning a boolean result, indicating whether the links
- **`nr`** – matches the nth link that matches all other criteria (default 0)

`follow_link`(*link=None*, ***kwds*)

Find a link and *`open()`* it.

Arguments are as for *`click_link()`*.

Return value is same as for *`open()`*.

`forms`()

Return iterable over forms.

The returned form objects implement the *`mechanize.HTMLForm`* interface.

`geturl`()

Get URL of current document.

`global_form`()

Return the global form object, or None if the factory implementation did not supply one.

The "global" form object contains all controls that are not descendants of any FORM element.

The returned form object implements the *`mechanize.HTMLForm`* interface.

This is a separate method since the global form is not regarded as part of the sequence of forms in the document – mostly for backwards-compatibility.

`links`(***kwds*)

Return iterable over links (*`mechanize.Link`* objects).

`open`(*url_or_request*, *data=None*, *timeout=<object object>*)

Open a URL. Loads the page so that you can subsequently use *`forms()`*, *`links()`*, etc. on it.

Parameters

- **`url_or_request`** – Either a URL or a *`mechanize.Request`*
- **`data`** (*`dict`*) – data to send with a POST request
- **`timeout`** – Timeout in seconds

Returns A `mechanize.Response` object

open_novisit(*url_or_request*, *data=None*, *timeout=<object object>*)

Open a URL without visiting it.

Browser state (including request, response, history, forms and links) is left unchanged by calling this function.

The interface is the same as for `open()`.

This is useful for things like fetching images.

See also `retrieve()`

reload()

Reload current document, and return response object.

response()

Return a copy of the current response.

The returned object has the same interface as the object returned by `open()`

retrieve(*fullurl*, *filename=None*, *reporthook=None*, *data=None*, *timeout=<object object>*, *open=<built-in function open>*)

Returns (filename, headers).

For remote objects, the default filename will refer to a temporary file. Temporary files are removed when the OpenerDirector.close() method is called.

For file: URLs, at present the returned filename is None. This may change in future.

If the actual number of bytes read is less than indicated by the Content-Length header, raises ContentTooShortError (a URLError subclass). The exception's .result attribute contains the (filename, headers) that would have been returned.

select_form(*name=None*, *predicate=None*, *nr=None*, ***attrs*)

Select an HTML form for input.

This is a bit like giving a form the "input focus" in a browser.

If a form is selected, the Browser object supports the HTMLForm interface, so you can call methods like `set_value()`, `set()`, and `click()`.

Another way to select a form is to assign to the .form attribute. The form assigned should be one of the objects returned by the `forms()` method.

If no matching form is found, `mechanize.FormNotFoundError` is raised.

If *name* is specified, then the form must have the indicated name.

If *predicate* is specified, then the form must match that function. The predicate function is passed the `mechanize.HTMLForm` as its single argument, and should return a boolean value indicating whether the form matched.

nr, if supplied, is the sequence number of the form (where 0 is the first). Note that control 0 is the first form matching all the other arguments (if supplied); it is not necessarily the first control in the form. The "global form" (consisting of all form controls not contained in any FORM element) is considered not to be part of this sequence and to have no name, so will not be matched unless both name and nr are None.

You can also match on any HTML attribute of the *<form>* tag by passing in the attribute name and value as keyword arguments. To convert HTML attributes into syntactically valid python keyword arguments, the following simple rule is used. The python keyword argument name is converted to an HTML attribute name by: Replacing all underscores with hyphens and removing any trailing underscores. You can pass in strings, functions or regular expression objects as the values to match. For example:

```
# Match form with the exact action specified
br.select_form(action='http://foo.com/submit.php')
# Match form with a class attribute that contains 'login'
br.select_form(class_=lambda x: 'login' in x)
# Match form with a data-form-type attribute that matches a regex
br.select_form(data_form_type=re.compile(r'a|b'))
```

set_ca_data(*cafile=None*, *capath=None*, *cadata=None*, *context=None*)
Set the SSL Context used for connecting to SSL servers.

This method accepts the same arguments as the `ssl.SSLContext.load_verify_locations()` method from the Python standard library. You can also pass a pre-built `ssl.SSLContext` via the *context* keyword argument. Note that to use this feature, you must be using Python >= 2.7.9.

set_client_cert_manager(*cert_manager*)
Set a mechanize.HTTPClientCertMgr, or None.

set_cookie(*cookie_string*)
Set a cookie.

Note that it is NOT necessary to call this method under ordinary circumstances: cookie handling is normally entirely automatic. The intended use case is rather to simulate the setting of a cookie by client script in a web page (e.g. JavaScript). In that case, use of this method is necessary because mechanize currently does not support JavaScript, VBScript, etc.

The cookie is added in the same way as if it had arrived with the current response, as a result of the current request. This means that, for example, if it is not appropriate to set the cookie based on the current request, no cookie will be set.

The cookie will be returned automatically with subsequent responses made by the Browser instance whenever that's appropriate.

cookie_string should be a valid value of the Set-Cookie header.

For example:

```
browser.set_cookie(
    "sid=abcdef; expires=Wednesday, 09-Nov-06 23:12:40 GMT")
```

Currently, this method does not allow for adding RFC 2986 cookies. This limitation will be lifted if anybody requests it.

See also *`set_simple_cookie()`* for an easier way to set cookies without needing to create a Set-Cookie header string.

set_cookiejar(*cookiejar*)
Set a mechanize.CookieJar, or None.

set_debug_http(*handle*)
Print HTTP headers to sys.stdout.

set_debug_redirects(*handle*)
Log information about HTTP redirects (including refreshes).

Logging is performed using module logging. The logger name is *"mechanize.http_redirects"*. To actually print some debug output, eg:

```
import sys, logging
logger = logging.getLogger("mechanize.http_redirects")
logger.addHandler(logging.StreamHandler(sys.stdout))
logger.setLevel(logging.INFO)
```

Other logger names relevant to this module:

- *mechanize.http_responses*
- *mechanize.cookies*

To turn on everything:

```
import sys, logging
logger = logging.getLogger("mechanize")
logger.addHandler(logging.StreamHandler(sys.stdout))
logger.setLevel(logging.INFO)
```

set_debug_responses(*handle*)
Log HTTP response bodies.

See `set_debug_redirects()` for details of logging.

Response objects may be .seek()able if this is set (currently returned responses are, raised HTTPError exception responses are not).

set_handle_equiv(*handle*, *head_parser_class=None*)
Set whether to treat HTML http-equiv headers like HTTP headers.

Response objects may be .seek()able if this is set (currently returned responses are, raised HTTPError exception responses are not).

set_handle_gzip(*handle*)
Add header indicating to server that we handle gzip content encoding. Note that if the server sends gzip'ed content, it is handled automatically in any case, regardless of this setting.

set_handle_redirect(*handle*)
Set whether to handle HTTP 30x redirections.

set_handle_referer(*handle*)
Set whether to add Referer header to each request.

set_handle_refresh(*handle*, *max_time=None*, *honor_time=True*)
Set whether to handle HTTP Refresh headers.

set_handle_robots(*handle*)
Set whether to observe rules from robots.txt.

set_handled_schemes(*schemes*)
Set sequence of URL scheme (protocol) strings.

For example: ua.set_handled_schemes(["http", "ftp"])

If this fails (with ValueError) because you've passed an unknown scheme, the set of handled schemes will not be changed.

set_header(*header*, *value=None*)
Convenience method to set a header value in *self.addheaders* so that the header is sent out with all requests automatically.

Parameters

- **header** – The header name, e.g. User-Agent
- **value** – The header value. If set to None the header is removed.

set_html(*html*, *url='http://example.com/'*)
Set the response to dummy with given HTML, and URL if given.

Allows you to then parse that HTML, especially to extract forms information. If no URL was given then the default is "example.com".

set_password_manager(*password_manager*)
Set a mechanize.HTTPPasswordMgrWithDefaultRealm, or None.

set_proxies(*proxies=None*, *proxy_bypass=None*)
Configure proxy settings.

Parameters

- **proxies** – dictionary mapping URL scheme to proxy specification. None means use the default system-specific settings.
- **proxy_bypass** – function taking hostname, returning whether proxy should be used. None means use the default system-specific settings.

The default is to try to obtain proxy settings from the system (see the documentation for urllib.urlopen for information about the system-specific methods used – note that's urllib, not urllib2).

To avoid all use of proxies, pass an empty proxies dict.

```
>>> ua = UserAgentBase()
>>> def proxy_bypass(hostname):
...     return hostname == "noproxy.com"
>>> ua.set_proxies(
...     {"http": "joe:password@myproxy.example.com:3128",
...      "ftp": "proxy.example.com"},
...     proxy_bypass)
```

set_proxy_password_manager(*password_manager*)
Set a mechanize.HTTPProxyPasswordMgr, or None.

set_request_gzip(*handle*)
Add header indicating to server that we handle gzip content encoding. Note that if the server sends gzip'ed content, it is handled automatically in any case, regardless of this setting.

set_response(*response*)
Replace current response with (a copy of) response.

response may be None.

This is intended mostly for HTML-preprocessing.

set_simple_cookie(*name*, *value*, *domain*, *path='/'*)
Similar to `set_cookie()` except that instead of using a cookie string, you simply specify the *name*, *value*, *domain* and optionally the *path*. The created cookie will never expire. For example:

```
browser.set_simple_cookie('some_key', 'some_value', '.example.com',
                          path='/some-page')
```

submit(**args*, ***kwds*)
Submit current form.

Arguments are as for `mechanize.HTMLForm.click()`.

Return value is same as for `open()`.

title()
Return title, or None if there is no title element in the document.

viewing_html()
Return whether the current response contains HTML data.

visit_response(*response*, *request=None*)
Visit the response, as if it had been `open()` ed.

Unlike `set_response()`, this updates history rather than replacing the current response.

2.2 The Request

class mechanize.**Request**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *visit=None*, *timeout=<object object>*, *method=None*)
A request for some network resource. Note that if you specify the method as 'GET' and the data as a dict, then it will be automatically appended to the URL. If you leave method as None, then the method will be auto-set to POST and the data will become part of the POST request.

Parameters

- **url** (`str`) – The URL to request
- **data** – Data to send with this request. Can be either a dictionary which will be encoded and sent as application/x-www-form-urlencoded data or a bytestring which will be sent as is. If you use a bytestring you should also set the Content-Type header appropriately.
- **headers** (`dict`) – Headers to send with this request
- **method** (`str`) – Method to use for HTTP requests. If not specified mechanize will choose GET or POST automatically as appropriate.
- **timeout** (`float`) – Timeout in seconds

The remaining arguments are for internal use.

add_data(*data*)
Set the data (a bytestring) to be sent with this request

add_header(*key*, *val=None*)
Add the specified header, replacing existing one, if needed. If val is None, remove the header.

add_unredirected_header(*key*, *val*)
Same as `add_header()` except that this header will not be sent for redirected requests.

get_data()
The data to be sent with this request

get_header(*header_name*, *default=None*)
Get the value of the specified header. If absent, return *default*

get_method()
The method used for HTTP requests

has_data()
True iff there is some data to be sent with this request

has_header(*header_name*)
Check if the specified header is present

has_proxy()
Private method.

header_items()
Get a copy of all headers for this request as a list of 2-tuples

set_data(*data*)
Set the data (a bytestring) to be sent with this request

2.3 The Response

Response objects in mechanize are *seek()* able `file`-like objects that support some additional methods, depending on the protocol used for the connection. The documentation below is for HTTP(s) responses, as these are the most common.

Additional methods present for HTTP responses:

class `mechanize._mechanize.`**`HTTPResponse`**

`code`
The HTTP status code

`getcode``()`
Return HTTP status code

`geturl``()`
Return the URL of the resource retrieved, commonly used to determine if a redirect was followed

`get_all_header_names``(`*normalize=True*`)`
Return a list of all headers names. When *normalize* is *True*, the case of the header names is normalized.

`get_all_header_values``(`*name*, *normalize=True*`)`
Return a list of all values for the specified header *name* (which is case-insensitive. Since headers in HTTP can be specified multiple times, the returned value is always a list. See `rfc822.Message.getheaders()`.

`info``()`
Return the headers of the response as a `rfc822.Message` instance.

`__getitem__``(`*header_name*`)`
Return the *last* HTTP Header matching the specified name as string. mechanize Response object act like dictionaries for convenient access to header values. For example: `response['Date']`. You can access header values using the header names, case-insensitively. Note that when more than one header with the same name is present, only the value of the last header is returned, use *`get_all_header_values()`* to get the values of all headers.

`get(header_name, default=None):`
Return the header value for the specified *header_name* or *default* if the header is not present. See *`__getitem__()`*.

2.4 Miscellaneous

class `mechanize.`**`Link`**`(`*base_url*, *url*, *text*, *tag*, *attrs*`)`
A link in a HTML document

Variables

- **`absolute_url`** – The absolutized link URL
- **`url`** – The link URL
- **`base_url`** – The base URL against which this link is resolved
- **`text`** – The link text
- **`tag`** – The link tag name
- **`attrs`** – The tag attributes

class mechanize.**History**

Though this will become public, the implied interface is not yet stable.

mechanize._html.**content_parser**(*data*, *url=None*, *response_info=None*, *transport_encoding=None*, *default_encoding='utf-8'*, *is_html=True*)

Parse data (a bytes object) into an etree representation such as `xml.etree.ElementTree` or *lxml.etree*

Parameters

- **data** (*bytes*) – The data to parse
- **url** – The URL of the document being parsed or None
- **response_info** – Information about the document (contains all HTTP headers as `HTTPMessage`)
- **transport_encoding** – The character encoding for the document being parsed as specified in the HTTP headers or None.
- **default_encoding** – The character encoding to use if no encoding could be detected and no transport_encoding is specified
- **is_html** – If the document is to be parsed as HTML.

CHAPTER 3

HTML Forms API

Forms in HTML documents are represented by `mechanize.HTMLForm`. Every form is a collection of controls. The different types of controls are represented by the various classes documented below.

class mechanize.**HTMLForm**(*action*, *method='GET'*, *enctype='application/x-www-form-urlencoded'*, *name=None*, *attrs=None*, *request_class=<class mechanize._request.Request>*, *forms=None*, *labels=None*, *id_to_labels=None*, *encoding=None*)

Represents a single HTML <form> . . . </form> element.

A form consists of a sequence of controls that usually have names, and which can take on various values. The values of the various types of controls represent variously: text, zero-or-one-of-many or many-of-many choices, and files to be uploaded. Some controls can be clicked on to submit the form, and clickable controls' values sometimes include the coordinates of the click.

Forms can be filled in with data to be returned to the server, and then submitted, using the click method to generate a request object suitable for passing to `mechanize.urlopen()` (or the click_request_data or click_pairs methods for integration with third-party code).

Usually, HTMLForm instances are not created directly. Instead, they are automatically created when visting a page with a mechanize Browser. If you do construct HTMLForm objects yourself, however, note that an HTMLForm instance is only properly initialised after the fixup method has been called. See `mechanize.ListControl` for the reason this is required.

Indexing a form (form["control_name"]) returns the named Control's value attribute. Assignment to a form index (form["control_name"] = something) is equivalent to assignment to the named Control's value attribute. If you need to be more specific than just supplying the control's name, use the set_value and get_value methods.

ListControl values are lists of item names (specifically, the names of the items that are selected and not disabled, and hence are "successful" – ie. cause data to be returned to the server). The list item's name is the value of the corresponding HTML element's"value" attribute.

Example:

```
<INPUT type="CHECKBOX" name="cheeses" value="leicester"></INPUT>
<INPUT type="CHECKBOX" name="cheeses" value="cheddar"></INPUT>
```

defines a CHECKBOX control with name "cheeses" which has two items, named "leicester" and "cheddar".

Another example:

```
<SELECT name="more_cheeses">
  <OPTION>1</OPTION>
  <OPTION value="2" label="CHEDDAR">cheddar</OPTION>
</SELECT>
```

defines a SELECT control with name "more_cheeses" which has two items, named "1" and "2" (because the OPTION element's value HTML attribute defaults to the element contents – see `mechanize.SelectControl` for more on these defaulting rules).

To select, deselect or otherwise manipulate individual list items, use the `mechanize.HTMLForm.find_control()` and `mechanize.ListControl.get()` methods. To set the whole value, do as for any other control: use indexing or the *set_value/get_value* methods.

Example:

```
# select *only* the item named "cheddar"
form["cheeses"] = ["cheddar"]
# select "cheddar", leave other items unaffected
form.find_control("cheeses").get("cheddar").selected = True
```

Some controls (RADIO and SELECT without the multiple attribute) can only have zero or one items selected at a time. Some controls (CHECKBOX and SELECT with the multiple attribute) can have multiple items selected at a time. To set the whole value of a ListControl, assign a sequence to a form index:

```
form["cheeses"] = ["cheddar", "leicester"]
```

If the ListControl is not multiple-selection, the assigned list must be of length one.

To check if a control has an item, if an item is selected, or if an item is successful (selected and not disabled), respectively:

```
"cheddar" in [item.name for item in form.find_control("cheeses").items]
"cheddar" in [item.name for item in form.find_control("cheeses").items
                and item.selected]
"cheddar" in form["cheeses"]
# or
"cheddar" in form.get_value("cheeses")
```

Note that some list items may be disabled (see below).

Note the following mistake:

```
form[control_name] = control_value
assert form[control_name] == control_value  # not necessarily true
```

The reason for this is that form[control_name] always gives the list items in the order they were listed in the HTML.

List items (hence list values, too) can be referred to in terms of list item labels rather than list item names using the appropriate label arguments. Note that each item may have several labels.

The question of default values of OPTION contents, labels and values is somewhat complicated: see `mechanize.SelectControl` and `mechanize.ListControl.get_item_attrs()` if you think you need to know.

Controls can be disabled or readonly. In either case, the control's value cannot be changed until you clear those flags (see example below). Disabled is the state typically represented by browsers by 'greying out' a control.

Disabled controls are not 'successful' – they don't cause data to get returned to the server. Readonly controls usually appear in browsers as read-only text boxes. Readonly controls are successful. List items can also be disabled. Attempts to select or deselect disabled items fail with AttributeError.

If a lot of controls are readonly, it can be useful to do this:

```
form.set_all_readonly(False)
```

To clear a control's value attribute, so that it is not successful (until a value is subsequently set):

```
form.clear("cheeses")
```

More examples:

```
control = form.find_control("cheeses")
control.disabled = False
control.readonly = False
control.get("gruyere").disabled = True
control.items[0].selected = True
```

See the various Control classes for further documentation. Many methods take name, type, kind, id, label and nr arguments to specify the control to be operated on: see `mechanize.HTMLForm.find_control()`.

ControlNotFoundError (subclass of ValueError) is raised if the specified control can't be found. This includes occasions where a non-ListControl is found, but the method (set, for example) requires a ListControl. ItemNotFoundError (subclass of ValueError) is raised if a list item can't be found. ItemCountError (subclass of ValueError) is raised if an attempt is made to select more than one item and the control doesn't allow that, or set/get_single are called and the control contains more than one item. AttributeError is raised if a control or item is readonly or disabled and an attempt is made to alter its value.

Security note: Remember that any passwords you store in HTMLForm instances will be saved to disk in the clear if you pickle them (directly or indirectly). The simplest solution to this is to avoid pickling HTMLForm objects. You could also pickle before filling in any password, or just set the password to "" before pickling.

Public attributes:

Variables

- **`action`** – full (absolute URI) form action
- **`method`** – "GET" or "POST"
- **`enctype`** – form transfer encoding MIME type
- **`name`** – name of form (None if no name was specified)
- **`attrs`** – dictionary mapping original HTML form attributes to their values
- **`controls`** – list of Control instances; do not alter this list (instead, call form.new_control to make a Control and add it to the form, or control.add_to_form if you already have a Control instance)

Methods for form filling:

Most of the these methods have very similar arguments. See `mechanize.HTMLForm.find_control()` for details of the name, type, kind, label and nr arguments.

```
def find_control(self,
                 name=None, type=None, kind=None, id=None,
                 predicate=None, nr=None, label=None)

```

(continues on next page)

(continued from previous page)

```
get_value(name=None, type=None, kind=None, id=None, nr=None,
        by_label=False,  # by_label is deprecated
        label=None)
set_value(value,
        name=None, type=None, kind=None, id=None, nr=None,
        by_label=False,  # by_label is deprecated
        label=None)

clear_all()
clear(name=None, type=None, kind=None, id=None, nr=None, label=None)

set_all_readonly(readonly)
```

Method applying only to FileControls:

```
add_file(file_object,
     content_type="application/octet-stream", filename=None,
     name=None, id=None, nr=None, label=None)
```

Methods applying only to clickable controls:

```
click(name=None, type=None, id=None, nr=0, coord=(1,1), label=None)
click_request_data(name=None, type=None, id=None, nr=0, coord=(1,1),
                label=None)
click_pairs(name=None, type=None, id=None, nr=0, coord=(1,1),
                label=None)
```

add_file(*file_object*, *content_type=None*, *filename=None*, *name=None*, *id=None*, *nr=None*, *label=None*)

Add a file to be uploaded.

Parameters

- **file_object** – file-like object (with read method) from which to read data to upload
- **content_type** – MIME content type of data to upload
- **filename** – filename to pass to server

If filename is None, no filename is sent to the server.

If content_type is None, the content type is guessed based on the filename and the data from read from the file object.

At the moment, guessed content type is always application/octet-stream.

Note the following useful HTML attributes of file upload controls (see HTML 4.01 spec, section 17):

- ***accept*: comma-separated list of content types** that the server will handle correctly; you can use this to filter out non-conforming files
- ***size*: XXX IIRC, this is indicative of whether form** wants multiple or single files
- *maxlength*: XXX hint of max content length in bytes?

clear(*name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *label=None*)

Clear the value attribute of a control.

As a result, the affected control will not be successful until a value is subsequently set. AttributeError is raised on readonly controls.

clear_all()

Clear the value attributes of all controls in the form.

See `mechanize.HTMLForm.clear()`

click(*name=None*, *type=None*, *id=None*, *nr=0*, *coord=(1, 1)*, *request_class=<class mechanize._request.Request>*, *label=None*)

Return request that would result from clicking on a control.

The request object is a mechanize.Request instance, which you can pass to mechanize.urlopen.

Only some control types (INPUT/SUBMIT & BUTTON/SUBMIT buttons and IMAGEs) can be clicked.

Will click on the first clickable control, subject to the name, type and nr arguments (as for find_control). If no name, type, id or number is specified and there are no clickable controls, a request will be returned for the form in its current, un-clicked, state.

IndexError is raised if any of name, type, id or nr is specified but no matching control is found. ValueError is raised if the HTMLForm has an enctype attribute that is not recognised.

You can optionally specify a coordinate to click at, which only makes a difference if you clicked on an image.

click_pairs(*name=None*, *type=None*, *id=None*, *nr=0*, *coord=(1, 1)*, *label=None*)

As for click_request_data, but returns a list of (key, value) pairs.

You can use this list as an argument to urllib.urlencode. This is usually only useful if you're using httplib or urllib rather than mechanize. It may also be useful if you want to manually tweak the keys and/or values, but this should not be necessary. Otherwise, use the click method.

Note that this method is only useful for forms of MIME type x-www-form-urlencoded. In particular, it does not return the information required for file upload. If you need file upload and are not using mechanize, use click_request_data.

click_request_data(*name=None*, *type=None*, *id=None*, *nr=0*, *coord=(1, 1)*, *request_class=<class mechanize._request.Request>*, *label=None*)

As for click method, but return a tuple (url, data, headers).

You can use this data to send a request to the server. This is useful if you're using httplib or urllib rather than mechanize. Otherwise, use the click method.

find_control(*name=None*, *type=None*, *kind=None*, *id=None*, *predicate=None*, *nr=None*, *label=None*)

Locate and return some specific control within the form.

At least one of the name, type, kind, predicate and nr arguments must be supplied. If no matching control is found, ControlNotFoundError is raised.

If name is specified, then the control must have the indicated name.

If type is specified then the control must have the specified type (in addition to the types possible for <input> HTML tags: "text", "password", "hidden", "submit", "image", "button", "radio", "checkbox", "file" we also have "reset", "buttonbutton", "submitbutton", "resetbutton", "textarea", "select").

If kind is specified, then the control must fall into the specified group, each of which satisfies a particular interface. The types are "text", "list", "multilist", "singlelist", "clickable" and "file".

If id is specified, then the control must have the indicated id.

If predicate is specified, then the control must match that function. The predicate function is passed the control as its single argument, and should return a boolean value indicating whether the control matched.

nr, if supplied, is the sequence number of the control (where 0 is the first). Note that control 0 is the first control matching all the other arguments (if supplied); it is not necessarily the first control in the form. If no nr is supplied, AmbiguityError is raised if multiple controls match the other arguments.

If label is specified, then the control must have this label. Note that radio controls and checkboxes never have labels: their items do.

fixup()

Normalise form after all controls have been added.

This is usually called by ParseFile and ParseResponse. Don't call it youself unless you're building your own Control instances.

This method should only be called once, after all controls have been added to the form.

get_value(*name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=False*, *label=None*)

Return value of control.

If only name and value arguments are supplied, equivalent to

```
form[name]
```

get_value_by_label(*name=None*, *type=None*, *kind=None*, *id=None*, *label=None*, *nr=None*)

All arguments should be passed by name.

new_control(*type*, *name*, *attrs*, *ignore_unknown=False*, *select_default=False*, *index=None*)

Adds a new control to the form.

This is usually called by mechanize. Don't call it yourself unless you're building your own Control instances.

Note that controls representing lists of items are built up from controls holding only a single list item. See `mechanize.ListControl` for further information.

Parameters

- **type** – type of control (see `mechanize.Control` for a list)
- **attrs** – HTML attributes of control
- **ignore_unknown** – if true, use a dummy Control instance for controls of unknown type; otherwise, use a TextControl
- **select_default** – for RADIO and multiple-selection SELECT controls, pick the first item as the default if no 'selected' HTML attribute is present (this defaulting happens when the HTMLForm.fixup method is called)
- **index** – index of corresponding element in HTML (see MoreFormTests.test_interspersed_controls for motivation)

possible_items(*name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=False*, *label=None*)

Return a list of all values that the specified control can take.

set(*selected*, *item_name*, *name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=False*, *label=None*)

Select / deselect named list item.

Parameters **selected** – boolean selected state

set_single(*selected*, *name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=None*, *label=None*)

Select / deselect list item in a control having only one item.

If the control has multiple list items, ItemCountError is raised.

This is just a convenience method, so you don't need to know the item's name – the item name in these single-item controls is usually something meaningless like "1" or "on".

For example, if a checkbox has a single item named "on", the following two calls are equivalent:

```
control.toggle("on")
control.toggle_single()
```

set_value(*value*, *name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=False*, *label=None*)

Set value of control.

If only name and value arguments are supplied, equivalent to

```
form[name] = value
```

set_value_by_label(*value*, *name=None*, *type=None*, *kind=None*, *id=None*, *label=None*, *nr=None*)

All arguments should be passed by name.

toggle(*item_name*, *name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=False*, *label=None*)

Toggle selected state of named list item.

toggle_single(*name=None*, *type=None*, *kind=None*, *id=None*, *nr=None*, *by_label=None*, *label=None*)

Toggle selected state of list item in control having only one item.

The rest is as for *mechanize.HTMLForm.set_single()*

class mechanize.**Control**(*type*, *name*, *attrs*, *index=None*)

An HTML form control.

An HTMLForm contains a sequence of Controls. The Controls in an HTMLForm are accessed using the HTMLForm.find_control method or the HTMLForm.controls attribute.

Control instances are usually constructed using the ParseFile / ParseResponse functions. If you use those functions, you can ignore the rest of this paragraph. A Control is only properly initialised after the fixup method has been called. In fact, this is only strictly necessary for ListControl instances. This is necessary because ListControls are built up from ListControls each containing only a single item, and their initial value(s) can only be known after the sequence is complete.

The types and values that are acceptable for assignment to the value attribute are defined by subclasses.

If the disabled attribute is true, this represents the state typically represented by browsers by 'greying out' a control. If the disabled attribute is true, the Control will raise AttributeError if an attempt is made to change its value. In addition, the control will not be considered 'successful' as defined by the W3C HTML 4 standard – ie. it will contribute no data to the return value of the HTMLForm.click* methods. To enable a control, set the disabled attribute to a false value.

If the readonly attribute is true, the Control will raise AttributeError if an attempt is made to change its value. To make a control writable, set the readonly attribute to a false value.

All controls have the disabled and readonly attributes, not only those that may have the HTML attributes of the same names.

On assignment to the value attribute, the following exceptions are raised: TypeError, AttributeError (if the value attribute should not be assigned to, because the control is disabled, for example) and ValueError.

If the name or value attributes are None, or the value is an empty list, or if the control is disabled, the control is not successful.

Public attributes:

Variables

- **type** (*str*) – string describing type of control (see the keys of the HTMLForm.type2class dictionary for the allowable values) (readonly)
- **name** (*str*) – name of control (readonly)
- **value** – current value of control (subclasses may allow a single value, a sequence of values, or either)
- **disabled** (*bool*) – disabled state
- **readonly** (*bool*) – readonly state
- **id** (*str*) – value of id HTML attribute

get_labels()

Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

pairs()

Return list of (key, value) pairs suitable for passing to urlencode.

class mechanize.**ScalarControl**(*type*, *name*, *attrs*, *index=None*)

Bases: `mechanize._form_controls.Control`

Control whose value is not restricted to one of a prescribed set.

Some ScalarControls don't accept any value attribute. Otherwise, takes a single value, which must be string-like.

Additional read-only public attribute:

Variables **attrs** (*dict*) – dictionary mapping the names of original HTML attributes of the control to their values

get_labels()

Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

pairs()

Return list of (key, value) pairs suitable for passing to urlencode.

class mechanize.**TextControl**(*type*, *name*, *attrs*, *index=None*)

Bases: `mechanize._form_controls.ScalarControl`

Textual input control.

Covers HTML elements: INPUT/TEXT, INPUT/PASSWORD, INPUT/HIDDEN, TEXTAREA

get_labels()

Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

pairs()

Return list of (key, value) pairs suitable for passing to urlencode.

class mechanize.**FileControl**(*type*, *name*, *attrs*, *index=None*)

Bases: `mechanize._form_controls.ScalarControl`

File upload with INPUT TYPE=FILE.

The value attribute of a FileControl is always None. Use add_file instead.

Additional public method: `add_file()`

`add_file`(*file_object*, *content_type=None*, *filename=None*)
Add data from the specified file to be uploaded. content_type and filename are sent in the HTTP headers if specified.

`get_labels`()
Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

`pairs`()
Return list of (key, value) pairs suitable for passing to urlencode.

class `mechanize.`**`IgnoreControl`**(*type*, *name*, *attrs*, *index=None*)
Bases: `mechanize._form_controls.ScalarControl`

Control that we're not interested in.

Covers html elements: INPUT/RESET, BUTTON/RESET, INPUT/BUTTON, BUTTON/BUTTON

These controls are always unsuccessful, in the terminology of HTML 4 (ie. they never require any information to be returned to the server).

BUTTON/BUTTON is used to generate events for script embedded in HTML.

The value attribute of IgnoreControl is always None.

`get_labels`()
Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

`pairs`()
Return list of (key, value) pairs suitable for passing to urlencode.

class `mechanize.`**`ListControl`**(*type*, *name*, *attrs={}*, *select_default=False*, *called_as_base_class=False*, *index=None*)
Bases: `mechanize._form_controls.Control`

Control representing a sequence of items.

The value attribute of a ListControl represents the successful list items in the control. The successful list items are those that are selected and not disabled.

ListControl implements both list controls that take a length-1 value (single-selection) and those that take length >1 values (multiple-selection).

ListControls accept sequence values only. Some controls only accept sequences of length 0 or 1 (RADIO, and single-selection SELECT). In those cases, ItemCountError is raised if len(sequence) > 1. CHECKBOXes and multiple-selection SELECTs (those having the "multiple" HTML attribute) accept sequences of any length.

Note the following mistake:

```
control.value = some_value
assert control.value == some_value    # not necessarily true
```

The reason for this is that the value attribute always gives the list items in the order they were listed in the HTML.

ListControl items can also be referred to by their labels instead of names. Use the label argument to .get(), and the .set_value_by_label(), .get_value_by_label() methods.

Note that, rather confusingly, though SELECT controls are represented in HTML by SELECT elements (which contain OPTION elements, representing individual list items), CHECKBOXes and RADIOs are not represented by *any* element. Instead, those controls are represented by a collection of INPUT elements. For example, this is a SELECT control, named "control1":

```
<select name="control1">
<option>foo</option>
<option value="1">bar</option>
</select>
```

and this is a CHECKBOX control, named "control2":

```
<input type="checkbox" name="control2" value="foo" id="cbe1">
<input type="checkbox" name="control2" value="bar" id="cbe2">
```

The id attribute of a CHECKBOX or RADIO ListControl is always that of its first element (for example, "cbe1" above).

Additional read-only public attribute: multiple.

fixup()
ListControls are built up from component list items (which are also ListControls) during parsing. This method should be called after all items have been added. See `mechanize.ListControl` for the reason this is required.

get(*name=None*, *label=None*, *id=None*, *nr=None*, *exclude_disabled=False*)
Return item by name or label, disambiguating if necessary with nr.

All arguments must be passed by name, with the exception of 'name', which may be used as a positional argument.

If name is specified, then the item must have the indicated name.

If label is specified, then the item must have a label whose whitespace-compressed, stripped, text substring-matches the indicated label string (e.g. label="please choose" will match " Do please choose an item ").

If id is specified, then the item must have the indicated id.

nr is an optional 0-based index of the items matching the query.

If nr is the default None value and more than item is found, raises AmbiguityError.

If no item is found, or if items are found but nr is specified and not found, raises ItemNotFoundError.

Optionally excludes disabled items.

get_item_attrs(*name*, *by_label=False*, *nr=None*)
Return dictionary of HTML attributes for a single ListControl item.

The HTML element types that describe list items are: OPTION for SELECT controls, INPUT for the rest. These elements have HTML attributes that you may occasionally want to know about – for example, the "alt" HTML attribute gives a text string describing the item (graphical browsers usually display this as a tooltip).

The returned dictionary maps HTML attribute names to values. The names and values are taken from the original HTML.

get_item_disabled(*name*, *by_label=False*, *nr=None*)
Get disabled state of named list item in a ListControl.

get_items(*name=None*, *label=None*, *id=None*, *exclude_disabled=False*)
Return matching items by name or label.

For argument docs, see the docstring for .get()

get_labels()
Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

get_value_by_label()
Return the value of the control as given by normalized labels.

pairs()
Return list of (key, value) pairs suitable for passing to urlencode.

possible_items(*by_label=False*)
Deprecated: return the names or labels of all possible items.

Includes disabled items, which may be misleading for some use cases.

set(*selected*, *name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, set the matching item's selection to the bool value of selected.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

set_all_items_disabled(*disabled*)
Set disabled state of all list items in a ListControl.

Parameters **disabled** – boolean disabled state

set_item_disabled(*disabled*, *name*, *by_label=False*, *nr=None*)
Set disabled state of named list item in a ListControl.

Parameters **disabled** – boolean disabled state

set_single(*selected*, *by_label=None*)
Deprecated: set the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

set_value_by_label(*value*)
Set the value of control by item labels.

value is expected to be an iterable of strings that are substrings of the item labels that should be selected. Before substring matching is performed, the original label text is whitespace-compressed (consecutive whitespace characters are converted to a single space character) and leading and trailing whitespace is stripped. Ambiguous labels: it will not complain as long as all ambiguous labels share the same item name (e.g. OPTION value).

toggle(*name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, toggle the matching item's selection.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

toggle_single(*by_label=None*)

Deprecated: toggle the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

class `mechanize.`**RadioControl**(*type*, *name*, *attrs*, *select_default=False*, *index=None*)

Bases: `mechanize._form_controls.ListControl`

Covers:

INPUT/RADIO

get(*name=None*, *label=None*, *id=None*, *nr=None*, *exclude_disabled=False*)

Return item by name or label, disambiguating if necessary with nr.

All arguments must be passed by name, with the exception of 'name', which may be used as a positional argument.

If name is specified, then the item must have the indicated name.

If label is specified, then the item must have a label whose whitespace-compressed, stripped, text substring-matches the indicated label string (e.g. label="please choose" will match " Do please choose an item ").

If id is specified, then the item must have the indicated id.

nr is an optional 0-based index of the items matching the query.

If nr is the default None value and more than item is found, raises AmbiguityError.

If no item is found, or if items are found but nr is specified and not found, raises ItemNotFoundError.

Optionally excludes disabled items.

get_item_attrs(*name*, *by_label=False*, *nr=None*)

Return dictionary of HTML attributes for a single ListControl item.

The HTML element types that describe list items are: OPTION for SELECT controls, INPUT for the rest. These elements have HTML attributes that you may occasionally want to know about – for example, the "alt" HTML attribute gives a text string describing the item (graphical browsers usually display this as a tooltip).

The returned dictionary maps HTML attribute names to values. The names and values are taken from the original HTML.

get_item_disabled(*name*, *by_label=False*, *nr=None*)

Get disabled state of named list item in a ListControl.

get_items(*name=None*, *label=None*, *id=None*, *exclude_disabled=False*)

Return matching items by name or label.

For argument docs, see the docstring for .get()

get_value_by_label()

Return the value of the control as given by normalized labels.

pairs()

Return list of (key, value) pairs suitable for passing to urlencode.

possible_items(*by_label=False*)

Deprecated: return the names or labels of all possible items.

Includes disabled items, which may be misleading for some use cases.

set(*selected*, *name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, set the matching item's selection to the bool value of selected.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

set_all_items_disabled(*disabled*)
Set disabled state of all list items in a ListControl.

Parameters **disabled** – boolean disabled state

set_item_disabled(*disabled*, *name*, *by_label=False*, *nr=None*)
Set disabled state of named list item in a ListControl.

Parameters **disabled** – boolean disabled state

set_single(*selected*, *by_label=None*)
Deprecated: set the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

set_value_by_label(*value*)
Set the value of control by item labels.

value is expected to be an iterable of strings that are substrings of the item labels that should be selected. Before substring matching is performed, the original label text is whitespace-compressed (consecutive whitespace characters are converted to a single space character) and leading and trailing whitespace is stripped. Ambiguous labels: it will not complain as long as all ambiguous labels share the same item name (e.g. OPTION value).

toggle(*name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, toggle the matching item's selection.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

toggle_single(*by_label=None*)
Deprecated: toggle the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

class mechanize.**CheckboxControl**(*type*, *name*, *attrs*, *select_default=False*, *index=None*)
Bases: `mechanize._form_controls.ListControl`

Covers:

INPUT/CHECKBOX

fixup()
ListControls are built up from component list items (which are also ListControls) during parsing. This method should be called after all items have been added. See `mechanize.ListControl` for the reason this is required.

get(*name=None*, *label=None*, *id=None*, *nr=None*, *exclude_disabled=False*)
Return item by name or label, disambiguating if necessary with nr.

All arguments must be passed by name, with the exception of ‘name’, which may be used as a positional argument.

If name is specified, then the item must have the indicated name.

If label is specified, then the item must have a label whose whitespace-compressed, stripped, text substring-matches the indicated label string (e.g. label=”please choose” will match ” Do please choose an item “).

If id is specified, then the item must have the indicated id.

nr is an optional 0-based index of the items matching the query.

If nr is the default None value and more than item is found, raises AmbiguityError.

If no item is found, or if items are found but nr is specified and not found, raises ItemNotFoundError.

Optionally excludes disabled items.

`get_item_attrs`(*name*, *by_label=False*, *nr=None*)
Return dictionary of HTML attributes for a single ListControl item.

The HTML element types that describe list items are: OPTION for SELECT controls, INPUT for the rest. These elements have HTML attributes that you may occasionally want to know about – for example, the “alt” HTML attribute gives a text string describing the item (graphical browsers usually display this as a tooltip).

The returned dictionary maps HTML attribute names to values. The names and values are taken from the original HTML.

`get_item_disabled`(*name*, *by_label=False*, *nr=None*)
Get disabled state of named list item in a ListControl.

`get_items`(*name=None*, *label=None*, *id=None*, *exclude_disabled=False*)
Return matching items by name or label.

For argument docs, see the docstring for .get()

`get_value_by_label`()
Return the value of the control as given by normalized labels.

`pairs`()
Return list of (key, value) pairs suitable for passing to urlencode.

`possible_items`(*by_label=False*)
Deprecated: return the names or labels of all possible items.

Includes disabled items, which may be misleading for some use cases.

`set`(*selected*, *name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, set the matching item’s selection to the bool value of selected.

Selecting items follows the behavior described in the docstring of the ‘get’ method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

`set_all_items_disabled`(*disabled*)
Set disabled state of all list items in a ListControl.

Parameters **`disabled`** – boolean disabled state

`set_item_disabled`(*disabled*, *name*, *by_label=False*, *nr=None*)
Set disabled state of named list item in a ListControl.

Parameters **`disabled`** – boolean disabled state

set_single(*selected*, *by_label=None*)
Deprecated: set the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

set_value_by_label(*value*)
Set the value of control by item labels.

value is expected to be an iterable of strings that are substrings of the item labels that should be selected. Before substring matching is performed, the original label text is whitespace-compressed (consecutive whitespace characters are converted to a single space character) and leading and trailing whitespace is stripped. Ambiguous labels: it will not complain as long as all ambiguous labels share the same item name (e.g. OPTION value).

toggle(*name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, toggle the matching item's selection.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

toggle_single(*by_label=None*)
Deprecated: toggle the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

class mechanize.**SelectControl**(*type*, *name*, *attrs*, *select_default=False*, *index=None*)
Bases: `mechanize._form_controls.ListControl`

Covers:

SELECT (and OPTION)

OPTION 'values', in HTML parlance, are Item 'names' in mechanize parlance.

SELECT control values and labels are subject to some messy defaulting rules. For example, if the HTML representation of the control is:

```
<SELECT name=year>
    <OPTION value=0 label="2002">current year</OPTION>
    <OPTION value=1>2001</OPTION>
    <OPTION>2000</OPTION>
</SELECT>
```

The items, in order, have labels "2002", "2001" and "2000", whereas their names (the OPTION values) are "0", "1" and "2000" respectively. Note that the value of the last OPTION in this example defaults to its contents, as specified by RFC 1866, as do the labels of the second and third OPTIONs.

The OPTION labels are sometimes more meaningful than the OPTION values, which can make for more maintainable code.

Additional read-only public attribute: attrs

The attrs attribute is a dictionary of the original HTML attributes of the SELECT element. Other ListControls do not have this attribute, because in other cases the control as a whole does not correspond to any single HTML element. control.get(…).attrs may be used as usual to get at the HTML attributes of the HTML elements corresponding to individual list items (for SELECT controls, these are OPTION elements).

Another special case is that the Item.attrs dictionaries have a special key "contents" which does not correspond to any real HTML attribute, but rather contains the contents of the OPTION element:

```
<OPTION>this bit</OPTION>
```

get (*name=None*, *label=None*, *id=None*, *nr=None*, *exclude_disabled=False*)
Return item by name or label, disambiguating if necessary with nr.

All arguments must be passed by name, with the exception of 'name', which may be used as a positional argument.

If name is specified, then the item must have the indicated name.

If label is specified, then the item must have a label whose whitespace-compressed, stripped, text substring-matches the indicated label string (e.g. label="please choose" will match " Do please choose an item ").

If id is specified, then the item must have the indicated id.

nr is an optional 0-based index of the items matching the query.

If nr is the default None value and more than item is found, raises AmbiguityError.

If no item is found, or if items are found but nr is specified and not found, raises ItemNotFoundError.

Optionally excludes disabled items.

get_item_attrs (*name*, *by_label=False*, *nr=None*)
Return dictionary of HTML attributes for a single ListControl item.

The HTML element types that describe list items are: OPTION for SELECT controls, INPUT for the rest. These elements have HTML attributes that you may occasionally want to know about – for example, the "alt" HTML attribute gives a text string describing the item (graphical browsers usually display this as a tooltip).

The returned dictionary maps HTML attribute names to values. The names and values are taken from the original HTML.

get_item_disabled (*name*, *by_label=False*, *nr=None*)
Get disabled state of named list item in a ListControl.

get_items (*name=None*, *label=None*, *id=None*, *exclude_disabled=False*)
Return matching items by name or label.

For argument docs, see the docstring for .get()

get_labels ()
Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by 'for' and 'id', are in the order that appear in the HTML.

get_value_by_label ()
Return the value of the control as given by normalized labels.

pairs ()
Return list of (key, value) pairs suitable for passing to urlencode.

possible_items (*by_label=False*)
Deprecated: return the names or labels of all possible items.

Includes disabled items, which may be misleading for some use cases.

set (*selected*, *name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, set the matching item's selection to the bool value of selected.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

`set_all_items_disabled`(*disabled*)
Set disabled state of all list items in a ListControl.

Parameters **`disabled`** – boolean disabled state

`set_item_disabled`(*disabled*, *name*, *by_label=False*, *nr=None*)
Set disabled state of named list item in a ListControl.

Parameters **`disabled`** – boolean disabled state

`set_single`(*selected*, *by_label=None*)
Deprecated: set the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

`set_value_by_label`(*value*)
Set the value of control by item labels.

value is expected to be an iterable of strings that are substrings of the item labels that should be selected. Before substring matching is performed, the original label text is whitespace-compressed (consecutive whitespace characters are converted to a single space character) and leading and trailing whitespace is stripped. Ambiguous labels: it will not complain as long as all ambiguous labels share the same item name (e.g. OPTION value).

`toggle`(*name*, *by_label=False*, *nr=None*)
Deprecated: given a name or label and optional disambiguating index nr, toggle the matching item's selection.

Selecting items follows the behavior described in the docstring of the 'get' method.

if the item is disabled, or this control is disabled or readonly, raise AttributeError.

`toggle_single`(*by_label=None*)
Deprecated: toggle the selection of the single item in this control.

Raises ItemCountError if the control does not contain only one item.

by_label argument is ignored, and included only for backwards compatibility.

`class` `mechanize.`**`SubmitControl`**(*type*, *name*, *attrs*, *index=None*)
Covers:

INPUT/SUBMIT BUTTON/SUBMIT

Members

Inherited-members

Show-inheritance

`class` `mechanize.`**`ImageControl`**(*type*, *name*, *attrs*, *index=None*)
Bases: `mechanize._form_controls.SubmitControl`

Covers:

INPUT/IMAGE

Coordinates are specified using one of the HTMLForm.click* methods.

get_labels()

Return all labels (Label instances) for this control.

If the control was surrounded by a <label> tag, that will be the first label; all other labels, connected by ‘for’ and ‘id’, are in the order that appear in the HTML.

pairs()

Return list of (key, value) pairs suitable for passing to urlencode.

CHAPTER 4

Advanced topics

4.1 Thread safety

The global `mechanize.urlopen()` and `mechanize.urlretrieve()` functions are thread safe. However, mechanize browser instances **are not** thread safe. If you want to use a mechanize Browser instance in multiple threads, clone it, using *copy.copy(browser_object)* method. The clone will share the same, thread safe cookie jar, and have the same settings/handlers as the original, but all other state is not shared, making the clone safe to use in a different thread.

4.2 Using custom CA certificates

mechanize supports the same mechanism for using custom CA certificates as python >= 2.7.9. To change the certificates a mechanize browser instance uses, call the *`mechanize.Browser.set_ca_data()`* method on it.

4.3 Debugging

Hints for debugging programs that use mechanize.

4.3.1 Cookies

A common mistake is to use `mechanize.urlopen()`, *and* the *.extract_cookies()* and *.add_cookie_header()* methods on a cookie object themselves. If you use *mechanize.urlopen()* (or *OpenerDirector.open()*), the module handles extraction and adding of cookies by itself, so you should not call *.extract_cookies()* or *.add_cookie_header()*.

Are you sure the server is sending you any cookies in the first place? Maybe the server is keeping track of state in some other way (*HIDDEN* HTML form entries (possibly in a separate page referenced by a frame), URL-encoded session keys, IP address, HTTP *Referer* headers)? Perhaps some embedded script in the HTML is setting cookies (see below)? Turn on *Logging*.

When you *.save()* to or *.load()*/*.revert()* from a file, single-session cookies will expire unless you explicitly request otherwise with the *ignore_discard* argument. This may be your problem if you find cookies are going away after saving and loading.

```
import mechanize
cj = mechanize.LWPCookieJar()
opener = mechanize.build_opener(mechanize.HTTPCookieProcessor(cj))
mechanize.install_opener(opener)
r = mechanize.urlopen("http://foobar.com/")
cj.save("/some/file", ignore_discard=True, ignore_expires=True)
```

JavaScript code can set cookies; mechanize does not support this. See *JavaScript is messing up my web-scraping. What do I do?*.

4.3.2 General

Enable *Logging*.

Sometimes, a server wants particular HTTP headers set to the values it expects. For example, the *User-Agent* header may need to be set (`mechanize.Browser.set_header()`) to a value like that of a popular browser.

Check that the browser is able to do manually what you're trying to achieve programmatically. Make sure that what you do manually is *exactly* the same as what you're trying to do from Python – you may simply be hitting a server bug that only gets revealed if you view pages in a particular order, for example.

Try comparing the headers and data that your program sends with those that a browser sends. Often this will give you the clue you need. You can use the developer tools in any browser to see exactly what the browser sends and receives.

If nothing is obviously wrong with the requests your program is sending and you're out of ideas, you can reliably locate the problem by copying the headers that a browser sends, and then changing headers until your program stops working again. Temporarily switch to explicitly sending individual HTTP headers (by calling *.add_header()*, or by using *httplib* directly). Start by sending exactly the headers that Firefox or Chrome send. You may need to make sure that a valid session ID is sent – the one you got from your browser may no longer be valid. If that works, you can begin the tedious process of changing your headers and data until they match what your original code was sending. You should end up with a minimal set of changes. If you think that reveals a bug in mechanize, please report it.

4.3.3 Logging

To enable logging to stdout:

```
import sys, logging
logger = logging.getLogger("mechanize")
logger.addHandler(logging.StreamHandler(sys.stdout))
logger.setLevel(logging.DEBUG)
```

You can reduce the amount of information shown by setting the level to *logging.INFO* instead of *logging.DEBUG*, or by only enabling logging for one of the following logger names instead of *"mechanize"*:

- *"mechanize"*: Everything.
- *"mechanize.cookies"*: Why particular cookies are accepted or rejected and why they are or are not returned. Requires logging enabled at the *DEBUG* level.
- *"mechanize.http_responses"*: HTTP response body data.
- *"mechanize.http_redirects"*: HTTP redirect information.

4.3.4 HTTP headers

An example showing how to enable printing of HTTP headers to stdout, logging of HTTP response bodies, and logging of information about redirections:

```
import sys, logging
import mechanize

logger = logging.getLogger("mechanize")
logger.addHandler(logging.StreamHandler(sys.stdout))
logger.setLevel(logging.DEBUG)

browser = mechanize.Browser()
browser.set_debug_http(True)
browser.set_debug_responses(True)
browser.set_debug_redirects(True)
response = browser.open("http://python.org/")
```

Alternatively, you can examine request and response objects to see what's going on. Note that requests may involve "sub-requests" in cases such as redirection, in which case you will not see everything that's going on just by examining the original request and final response.

CHAPTER 5

Quickstart

The examples below are written for a website that does not exist (*example.com*), so cannot be run.

```python
import re
import mechanize

br = mechanize.Browser()
br.open("http://www.example.com/")
# follow second link with element text matching regular expression
response1 = br.follow_link(text_regex=r"cheese\s*shop", nr=1)
print(br.title())
print(response1.geturl())
print(response1.info())  # headers
print(response1.read())  # body

br.select_form(name="order")
# Browser passes through unknown attributes (including methods)
# to the selected HTMLForm.
br["cheeses"] = ["mozzarella", "caerphilly"]  # (the method here is __setitem__)
# Submit current form.  Browser calls .close() on the current response on
# navigation, so this closes response1
response2 = br.submit()

# print currently selected form (don't call .submit() on this, use br.submit())
print(br.form)

response3 = br.back()  # back to cheese shop (same data as response1)
# the history mechanism returns cached response objects
# we can still use the response, even though it was .close()d
response3.get_data()  # like .seek(0) followed by .read()
response4 = br.reload()  # fetches from server

for form in br.forms():
    print(form)
# .links() optionally accepts the keyword args of .follow_/.find_link()
```

(continues on next page)

(continued from previous page)

```
for link in br.links(url_regex="python.org"):
    print(link)
    br.follow_link(link)  # takes EITHER Link instance OR keyword args
    br.back()
```

You may control the browser's policy by using the methods of *mechanize.Browser*'s base class, *mechanize.UserAgent*. For example:

```
br = mechanize.Browser()
# Explicitly configure proxies (Browser will attempt to set good defaults).
# Note the userinfo ("joe:password@") and port number (":3128") are optional.
br.set_proxies({"http": "joe:password@myproxy.example.com:3128",
                "ftp": "proxy.example.com",
                })
# Add HTTP Basic/Digest auth username and password for HTTP proxy access.
# (equivalent to using "joe:password@..." form above)
br.add_proxy_password("joe", "password")
# Add HTTP Basic/Digest auth username and password for website access.
br.add_password("http://example.com/protected/", "joe", "password")
# Add an extra header to all outgoing requests, you can also
# re-order or remove headers in this function.
br.finalize_request_headers = lambda request, headers: headers.__setitem__(
  'My-Custom-Header', 'Something')
# Don't handle HTTP-EQUIV headers (HTTP headers embedded in HTML).
br.set_handle_equiv(False)
# Ignore robots.txt.  Do not do this without thought and consideration.
br.set_handle_robots(False)
# Don't add Referer (sic) header
br.set_handle_referer(False)
# Don't handle Refresh redirections
br.set_handle_refresh(False)
# Don't handle cookies
br.set_cookiejar()
# Supply your own mechanize.CookieJar (NOTE: cookie handling is ON by
# default: no need to do this unless you have some reason to use a
# particular cookiejar)
br.set_cookiejar(cj)
# Tell the browser to send the Accept-Encoding: gzip header to the server
# to indicate it supports gzip Content-Encoding
br.set_request_gzip(True)
# Do not verify SSL certificates
import ssl
br.set_ca_data(context=ssl._create_unverified_context(cert_reqs=ssl.CERT_NONE))
# Log information about HTTP redirects and Refreshes.
br.set_debug_redirects(True)
# Log HTTP response bodies (i.e. the HTML, most of the time).
br.set_debug_responses(True)
# Print HTTP headers.
br.set_debug_http(True)

# To make sure you're seeing all debug output:
logger = logging.getLogger("mechanize")
logger.addHandler(logging.StreamHandler(sys.stdout))
logger.setLevel(logging.INFO)

# Sometimes it's useful to process bad headers or bad HTML:
```

(continues on next page)

(continued from previous page)

```
response = br.response()  # this is a copy of response
headers = response.info()  # this is a HTTPMessage
headers["Content-type"] = "text/html; charset=utf-8"
response.set_data(response.get_data().replace("<!---", "<!--"))
br.set_response(response)
```

mechanize exports the complete interface of *urllib2*:

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

When using mechanize, anything you would normally import from *urllib2* should be imported from mechanize instead.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index