MechanicalSoup Documentation

Release 0.10.0

Feb 04, 2018

Contents

A Python library for automating interaction with websites. MechanicalSoup automatically stores and sends cookies, follows redirects, and can follow links and submit forms. It doesn't do Javascript.

MechanicalSoup was created by M Hickford, who was a fond user of the Mechanize library. Unfortunately, Mechanize is incompatible with Python 3 and its development stalled for several years. MechanicalSoup provides a similar API, built on Python giants Requests (for http sessions) and BeautifulSoup (for document navigation). Since 2017 it is a project actively maintained by a small team including @hemberger and @moy.

Contents:

CHAPTER 1

Introduction

PyPy and PyPy3 are also supported (and tested against).

Find MechanicalSoup on Python Package Index (Pypi) and follow the development on GitHub.

1.1 Installation

Download and install the latest released version from PyPI:

```
pip install MechanicalSoup
```

Download and install the development version from GitHub:

```
pip install git+https://github.com/MechanicalSoup/MechanicalSoup
```

Installing from source (installs the version in the current working directory):

```
git clone https://github.com/MechanicalSoup/MechanicalSoup.git
cd MechanicalSoup
python setup.py install
```

(In all cases, add `--user` to the `install` command to install in the current user's home directory.)

Example code: https://github.com/MechanicalSoup/MechanicalSoup/tree/master/examples/

CHAPTER 2

MechanicalSoup tutorial

2.1 First contact, step by step

As a simple example, we'll browse http://httpbin.org/, a website designed to test tools like MechanicalSoup.

First, let's create a browser object:

```
>>> import mechanicalsoup
>>> browser = mechanicalsoup.StatefulBrowser()
```

To customize the way to build a browser (change the user-agent, the HTML parser to use, the way to react to 404 Not Found errors, ...), see `__init__()`.

Now, open the webpage we want:

```
>>> browser.open("http://httpbin.org/")
<Response [200]>
```

The return value of *`open()`* is an object of type requests.Response. Actually, MechanicalSoup is using the requests library to do the actual requests to the website, so there's no surprise that we're getting such object. In short, it contains the data and meta-data that the server sent us. You see the HTTP response status, 200, which means "OK", but the object also contains the content of the page we just downloaded.

Just like a normal browser's URL bar, the browser remembers which URL it's browsing:

```
>>> browser.get_url()
'http://httpbin.org/'
```

Now, let's follow the link to `/forms/post`:

```
>>> browser.follow_link("forms")
<Response [200]>
>>> browser.get_url()
'http://httpbin.org/forms/post'
```

We passed a regular expression `"forms"` to *`follow_link()`*, who followed the link whose text matched this expression. There are many other ways to call *`follow_link()`*, but we'll get back to it.

We're now visiting http://httpbin.org/forms/post, which contains a form. Let's see the page content:

```
>>> browser.get_current_page()
<!DOCTYPE html>
<html>
...
<form action="/post" method="post">
...
```

Actually, the return type of `get_current_page()` is bs4.BeautifulSoup. BeautifulSoup, aka bs4, is the second library used by Mechanicalsoup: it is an HTML manipulation library. You can now navigate in the tags of the pages using BeautifulSoup. For example, to get all the `<legend>` tags:

```
>>> browser.get_current_page().find_all('legend')
[<legend> Pizza Size </legend>, <legend> Pizza Toppings </legend>]
```

To fill-in a form, we need to tell MechanicalSoup which form we're going to fill-in and submit:

```
>>> browser.select_form('form[action="/post"]')
```

The argument to *`select_form()`* is a CSS selector. Here, we select an HTML tag named `form` having an attribute `action` whose value is `"/post"`. Since there's only one form in the page, `browser.select_form()` would have done the trick too.

Now, give a value to fields in the form. First, what are the available fields? You can print a summary of the currently selected form with *`print_summary()`*:

```
>>> browser.get_current_form().print_summary()
<input name="custname"/>
<input name="custtel" type="tel"/>
<input name="custemail" type="email"/>
<input name="size" type="radio" value="small"/>
<input name="size" type="radio" value="medium"/>
<input name="size" type="radio" value="large"/>
<input name="topping" type="checkbox" value="bacon"/>
<input name="topping" type="checkbox" value="cheese"/>
<input name="topping" type="checkbox" value="onion"/>
<input name="topping" type="checkbox" value="mushroom"/>
<input max="21:00" min="11:00" name="delivery" step="900" type="time"/>
<textarea name="comments"></textarea>
```

For text fields, it's simple: just give a value for `input` element based on their `name` attribute:

```
>>> browser["custname"] = "Me"
>>> browser["custtel"] = "00 00 0001"
>>> browser["custemail"] = "nobody@example.com"
>>> browser["comments"] = "This pizza looks really good :-)"
```

For radio buttons, well, it's simple too: radio buttons have several `input` tag with the same `name` and different values, just select the one you need (`"size"` is the `name` attribute, `"medium"` is the `"value"` attribute of the element we want to tick):

```
>>> browser["size"] = "medium"
```

For checkboxes, one can use the same mechanism to check one box:

```
>>> browser["topping"] = "bacon"
```

But we can also check any number of boxes by assigning a list to the field:

```
>>> browser["topping"] = ("bacon", "cheese")
```

Actually, `browser["..."] = "..."` (i.e. calls to *`__setitem__()`*) is just a helper to fill-in a form, but you can use any tool BeautifulSoup provides to modify the soup object, and MechanicalSoup will take care of submitting the form for you.

Let's see what the filled-in form looks like:

```
>>> browser.launch_browser()
```

`launch_browser()` will launch a real web browser on the current page visited by our `browser` object, including the changes we just made to the form (note that it does not open the real webpage, but creates a temporary file containing the page content, and point your browser to this file). Try changing the boxes ticked and the content of the text field, and re-launch the browser.

This method is very useful in complement with your browser's web development tools. For example, with Firefox, right-click "Inspect Element" on a field will give you everything you need to manipulate this field (in particular the `name` and `value` attributes).

It's also possible to check the content with *`print_summary()`* (that we already used to list the fields):

```
>>> browser.get_current_form().print_summary()
<input name="custname" value="Me"/>
<input name="custtel" type="tel" value="00 00 0001"/>
<input name="custemail" type="email" value="nobody@example.com"/>
<input name="size" type="radio" value="small"/>
<input checked="" name="size" type="radio" value="medium"/>
<input name="size" type="radio" value="large"/>
<input checked="" name="topping" type="checkbox" value="bacon"/>
<input checked="" name="topping" type="checkbox" value="cheese"/>
<input name="topping" type="checkbox" value="onion"/>
<input name="topping" type="checkbox" value="mushroom"/>
<input max="21:00" min="11:00" name="delivery" step="900" type="time"/>
<textarea name="comments">This pizza looks really good :-)</textarea>
```

Assuming we're satisfied with the content of the form, we can submit it (i.e. simulate a click on the submit button):

```
>>> response = browser.submit_selected()
```

The response is not an HTML page, so the browser doesn't parse it to a BeautifulSoup object, but we can still see the text it contains:

```
>>> print(response.text)
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "comments": "This pizza looks really good :-)",
    "custemail": "nobody@example.com",
    "custname": "Me",
    "custtel": "00 00 0001",
    "delivery": "",
    "size": "medium",
```

```
    "topping": [
      "bacon",
      "cheese"
    ]
  },
...
```

To sum up, here is the complete example (examples/expl_httpbin.py):

```
import mechanicalsoup

browser = mechanicalsoup.StatefulBrowser()
browser.open("http://httpbin.org/")

print(browser.get_url())
browser.follow_link("forms")
print(browser.get_url())
print(browser.get_current_page())

browser.select_form('form[action="/post"]')
browser["custname"] = "Me"
browser["custtel"] = "00 00 0001"
browser["custemail"] = "nobody@example.com"
browser["size"] = "medium"
browser["topping"] = "onion"
browser["topping"] = ("bacon", "cheese")
browser["comments"] = "This pizza looks really good :-)"

# Uncomment to launch a real web browser on the current page.
# browser.launch_browser()

# Uncomment to display a summary of the filled-in form
# browser.get_current_form().print_summary()

response = browser.submit_selected()
print(response.text)
```

2.2 A more complete example: logging-in into GitHub

The simplest way to use MechanicalSoup is to use the *StatefulBrowser* class (this example is available as examples/example.py in MechanicalSoup's source code):

```
"""Example app to login to GitHub using the StatefulBrowser class."""

from __future__ import print_function
import argparse
import mechanicalsoup
from getpass import getpass

parser = argparse.ArgumentParser(description="Login to GitHub.")
parser.add_argument("username")
args = parser.parse_args()

args.password = getpass("Please enter your GitHub password: ")
```

```
browser = mechanicalsoup.StatefulBrowser(
    soup_config={'features': 'lxml'},
    raise_on_404=True,
    user_agent='MyBot/0.1: mysite.example.com/bot_info',
)
# Uncomment for a more verbose output:
# browser.set_verbose(2)

browser.open("https://github.com")
browser.follow_link("login")
browser.select_form('#login form')
browser["login"] = args.username
browser["password"] = args.password
resp = browser.submit_selected()

# Uncomment to launch a web browser on the current page:
# browser.launch_browser()

# verify we are now logged in
page = browser.get_current_page()
messages = page.find("div", class_="flash-messages")
if messages:
    print(messages.text)
assert page.select(".logout-form")

print(page.title.text)

# verify we remain logged in (thanks to cookies) as we browse the rest of
# the site
page3 = browser.open("https://github.com/MechanicalSoup/MechanicalSoup")
assert page3.soup.select(".logout-form")
```

Alternatively, one can use the *Browser* class, which doesn't maintain a state from one call to another (i.e. the Browser itself doesn't remember which page you are visiting and what is its content, it's up to the caller to do so). This example is available as examples/example_manual.py in the source:

```
"""Example app to login to GitHub, using the plain Browser class.

See example.py for an example using the more advanced StatefulBrowser."""
import argparse
import mechanicalsoup

parser = argparse.ArgumentParser(description="Login to GitHub.")
parser.add_argument("username")
parser.add_argument("password")
args = parser.parse_args()

browser = mechanicalsoup.Browser(soup_config={'features': 'lxml'})

# request github login page. the result is a requests.Response object
# http://docs.python-requests.org/en/latest/user/quickstart/#response-content
login_page = browser.get("https://github.com/login")

# similar to assert login_page.ok but with full status code in case of
# failure.
login_page.raise_for_status()

# login_page.soup is a BeautifulSoup object
```

```
# http://www.crummy.com/software/BeautifulSoup/bs4/doc/#beautifulsoup
# we grab the login form
login_form = mechanicalsoup.Form(login_page.soup.select_one('#login form'))

# specify username and password
login_form.input({"login": args.username, "password": args.password})

# submit form
page2 = browser.submit(login_form, login_page.url)

# verify we are now logged in
messages = page2.soup.find("div", class_="flash-messages")
if messages:
    print(messages.text)
assert page2.soup.select(".logout-form")

print(page2.soup.title.text)

# verify we remain logged in (thanks to cookies) as we browse the rest of
# the site
page3 = browser.get("https://github.com/MechanicalSoup/MechanicalSoup")
assert page3.soup.select(".logout-form")
```

2.2.1 More examples

For more examples, see the examples directory in MechanicalSoup's source code.

CHAPTER 3

The mechanicalsoup package: API documentation

3.1 StatefulBrowser

class mechanicalsoup.**StatefulBrowser**(**args*, ***kwargs*)

Bases: mechanicalsoup.browser.Browser

An extension of *Browser* that stores the browser's state and provides many convenient functions for interacting with HTML elements. It is the primary tool in MechanicalSoup for interfacing with websites.

Parameters

- **session** – Attach a pre-existing requests Session instead of constructing a new one.
- **soup_config** – Configuration passed to BeautifulSoup to affect the way HTML is parsed. Defaults to `{'features': 'lxml'}`. If overriden, it is highly recommended to specify a parser. Otherwise, BeautifulSoup will issue a warning and pick one for you, but the parser it chooses may be different on different machines.
- **requests_adapters** – Configuration passed to requests, to affect the way HTTP requests are performed.
- **raise_on_404** – If True, raise *LinkNotFoundError* when visiting a page triggers a 404 Not Found error.
- **user_agent** – Set the user agent header to this value.

All arguments are forwarded to *Browser()*.

Examples

```
browser = mechanicalsoup.StatefulBrowser(
    soup_config={'features': 'lxml'},  # Use the lxml HTML parser
    raise_on_404=True,
    user_agent='MyBot/0.1: mysite.example.com/bot_info',
)
browser.open(url)
```

```
# ...
browser.close()
```

Once not used anymore, the browser can be closed using *close()*.

__setitem__(*name*, *value*)
Call item assignment on the currently selected form. See *Form.__setitem__()*.

absolute_url(*url*)
Return the absolute URL made from the current URL and `url`. The current URL is only used to provide any missing components of `url`, as in the .urljoin() method of urllib.parse.

download_link(*link=None*, *file=None*, **args*, ***kwargs*)
Downloads the contents of a link to a file. This function behaves similarly to *follow_link()*, but the browser state will not change when calling this function.

Parameters **file** – Filesystem path where the page contents will be downloaded. If the file already exists, it will be overwritten.

Other arguments are the same as *follow_link()* (`link` can either be a bs4.element.Tag or a URL regex, other arguments are forwarded to *find_link()*).

Returns requests.Response object.

find_link(**args*, ***kwargs*)
Find and return a link, as a bs4.element.Tag object.

The search can be refined by specifying any argument that is accepted by *links()*. If several links match, return the first one found.

If no link is found, raise *LinkNotFoundError*.

follow_link(*link=None*, **args*, ***kwargs*)
Follow a link.

If `link` is a bs4.element.Tag (i.e. from a previous call to *links()* or *find_link()*), then follow the link.

If `link` doesn't have a *href*-attribute or is None, treat `link` as a url_regex and look it up with *find_link()*. Any additional arguments specified are forwarded to this function.

If the link is not found, raise *LinkNotFoundError*. Before raising, if debug is activated, list available links in the page and launch a browser.

Returns Forwarded from *open_relative()*.

get_current_form()
Get the currently selected form as a *Form* object. See *select_form()*.

get_current_page()
Get the current page as a soup object.

get_debug()
Get the debug mode (off by default).

get_url()
Get the URL of the currently visited page.

get_verbose()
Get the verbosity level. See *set_verbose()*.

launch_browser(*soup=None*)
Launch a browser to display a page, for debugging purposes.

Param soup: Page contents to display, supplied as a bs4 soup object. Defaults to the current page of the `StatefulBrowser` instance.

links(*url_regex=None*, *link_text=None*, **args*, ***kwargs*)
Return links in the page, as a list of bs4.element.Tag objects.

To return links matching specific criteria, specify `url_regex` to match the *href*-attribute, or `link_text` to match the *text*-attribute of the Tag. All other arguments are forwarded to the .find_all() method in BeautifulSoup.

list_links(**args*, ***kwargs*)
Display the list of links in the current page. Arguments are forwarded to *links()*.

new_control(*type*, *name*, *value*, ***kwargs*)
Call *Form.new_control()* on the currently selected form.

open(*url*, **args*, ***kwargs*)
Open the URL and store the Browser's state in this object. All arguments are forwarded to *Browser.get()*.

Returns Forwarded from *Browser.get()*.

open_fake_page(*page_text*, *url=None*, *soup_config=None*)
Mock version of *open()*.

Behave as if opening a page whose text is `page_text`, but do not perform any network access. If `url` is set, pretend it is the page's URL. Useful mainly for testing.

open_relative(*url*, **args*, ***kwargs*)
Like *open()*, but `url` can be relative to the currently visited page.

refresh()
Reload the current page with the same request as originally done. Any change (*select_form*, or any value filled-in in the form) made to the current page before refresh is discarded.

Raises **ValueError** – Raised if no refreshable page is loaded, e.g., when using the shallow `Browser` wrapper functions.

Returns Response of the request.

select_form(*selector='form'*, *nr=0*)
Select a form in the current page.

Parameters

- **selector** – CSS selector or a bs4.element.Tag object to identify the form to select. If not specified, `selector` defaults to "form", which is useful if, e.g., there is only one form on the page. For `selector` syntax, see the .select() method in BeautifulSoup.
- **nr** – A zero-based index specifying which form among those that match `selector` will be selected. Useful when one or more forms have the same attributes as the form you want to select, and its position on the page is the only way to uniquely identify it. Default is the first matching form (`nr=0`).

Returns The selected form as a soup object. It can also be retrieved later with *get_current_form()*.

set_debug(*debug*)
Set the debug mode (off by default).

Set to True to enable debug mode. When active, some actions will launch a browser on the current page on failure to let you inspect the page content.

set_verbose(*verbose*)
Set the verbosity level (an integer).

- 0 means no verbose output.
- 1 shows one dot per visited page (looks like a progress bar)
- >= 1 shows each visited URL.

submit_selected(*btnName=None*, **args*, ***kwargs*)
Submit the form that was selected with `select_form()`.

Returns Forwarded from `Browser.submit()`.

If there are multiple submit input/button elements, passes `btnName` to `Form.choose_submit()` on the current form to choose between them. All other arguments are forwarded to `Browser.submit()`.

3.2 Browser

class mechanicalsoup.**Browser**(*session=None*, *soup_config={'features': 'lxml'}*, *requests_adapters=None*, *raise_on_404=False*, *user_agent=None*)
Builds a Browser.

Parameters

- **session** – Attach a pre-existing requests Session instead of constructing a new one.
- **soup_config** – Configuration passed to BeautifulSoup to affect the way HTML is parsed. Defaults to `{'features': 'lxml'}`. If overriden, it is highly recommended to specify a parser. Otherwise, BeautifulSoup will issue a warning and pick one for you, but the parser it chooses may be different on different machines.
- **requests_adapters** – Configuration passed to requests, to affect the way HTTP requests are performed.
- **raise_on_404** – If True, raise `LinkNotFoundError` when visiting a page triggers a 404 Not Found error.
- **user_agent** – Set the user agent header to this value.

See also: `StatefulBrowser()`

static add_soup(*response*, *soup_config*)
Attaches a soup object to a requests response.

close()
Close the current session, if still open.

get(**args*, ***kwargs*)
Straightforward wrapper around requests.Session.get.

Returns requests.Response object with a *soup*-attribute added by `add_soup()`.

get_cookiejar()
Gets the cookiejar from the requests session.

launch_browser(*soup*)
Launch a browser to display a page, for debugging purposes.

Param soup: Page contents to display, supplied as a bs4 soup object.

post(**args*, ***kwargs*)
Straightforward wrapper around requests.Session.post.

Returns requests.Response object with a *soup*-attribute added by *add_soup()*.

request(**args*, ***kwargs*)
Straightforward wrapper around requests.Session.request.

Returns requests.Response object with a *soup*-attribute added by *add_soup()*.

This is a low-level function that should not be called for basic usage (use *get()* or *post()* instead). Use it if you need an HTTP verb that MechanicalSoup doesn't manage (e.g. MKCOL) for example.

set_cookiejar(*cookiejar*)
Replaces the current cookiejar in the requests session. Since the session handles cookies automatically without calling this function, only use this when default cookie handling is insufficient.

Parameters **cookiejar** – Any cookielib.CookieJar compatible object.

set_user_agent(*user_agent*)
Replaces the current user agent in the requests session headers.

submit(*form*, *url=None*, ***kwargs*)
Prepares and sends a form request.

Parameters

- **form** – The filled-out form.
- **url** – URL of the page the form is on. If the form action is a relative path, then this must be specified.
- ****kwargs** – Arguments forwarded to requests.Session.request.

Returns requests.Response object with a *soup*-attribute added by *add_soup()*.

3.3 Form

class mechanicalsoup.**Form**(*form*)
Build a fillable form.

Parameters **form** – A bs4.element.Tag corresponding to an HTML form element.

The Form class is responsible for preparing HTML forms for submission. It handles the following types of elements: input (text, checkbox, radio), select, and textarea.

Each type is set by a method named after the type (e.g. *set_select()*), and then there are convenience methods (e.g. *set()*) that do type-deduction and set the value using the appropriate method.

It also handles submit-type elements using *choose_submit()*.

__setitem__(*name*, *value*)
Forwards arguments to *set()*. For example, `form["name"] = "value"` calls `form.set("name", "value")`.

check(*data*)
For backwards compatibility, this method handles checkboxes and radio buttons in a single call. It will not uncheck any checkboxes unless explicitly specified by `data`, in contrast with the default behavior of *set_checkbox()*.

choose_submit(*submit*)
Selects the input (or button) element to use for form submission.

Parameters **submit** – The bs4.element.Tag (or just its *name*-attribute) that identifies the submit element to use.

To simulate a normal web browser, only one submit element must be sent. Therefore, this does not need to be called if there is only one submit element in the form.

If the element is not found or if multiple elements match, raise a *LinkNotFoundError* exception.

Example:

```
browser = mechanicalsoup.StatefulBrowser()
browser.open(url)
form = browser.select_form()
form.choose_submit('form_name_attr')
browser.submit_selected()
```

new_control(*type*, *name*, *value*, ***kwargs*)
Add a new input element to the form.

The arguments set the attributes of the new element.

print_summary()
Print a summary of the form.

May help finding which fields need to be filled-in.

set(*name*, *value*, *force=False*)
Set a form element identified by `name` to a specified `value`. The type of element (input, textarea, select, ...) does not need to be given; it is inferred by the following methods: *set_checkbox()*, *set_radio()*, *set_input()*, *set_textarea()*, *set_select()*. If none of these methods find a matching element, then if `force` is True, a new element (`<input type="text" ...>`) will be added using *new_control()*.

Example: filling-in a login/password form with EULA checkbox

```
form.set("login", username)
form.set("password", password)
form.set("eula-checkbox", True)
```

Example: uploading a file through a `<input type="file" name="tagname">` field (provide the path to the local file, and its content will be uploaded):

```
form.set("tagname") = path_to_local_file
```

set_checkbox(*data*, *uncheck_other_boxes=True*)
Set the *checked*-attribute of input elements of type "checkbox" specified by `data` (i.e. check boxes).

Parameters

- **data** – Dict of `{name: value, ...}`. In the family of checkboxes whose *name*-attribute is `name`, check the box whose *value*-attribute is `value`. All boxes in the family can be checked (unchecked) if `value` is True (False). To check multiple specific boxes, let `value` be a tuple or list.
- **uncheck_other_boxes** – If True (default), before checking any boxes specified by `data`, uncheck the entire checkbox family. Consider setting to False if some boxes are checked by default when the HTML is served.

set_input(*data*)
Fill-in a set of fields in a form.

Example: filling-in a login/password form

```
form.set_input({"login": username, "password": password})
```

This will find the input element named "login" and give it the value `username`, and the input element named "password" and give it the value `password`.

set_radio(*data*)
Set the *checked*-attribute of input elements of type "radio" specified by `data` (i.e. select radio buttons).

Parameters **data** – Dict of `{name: value, ...}`. In the family of radio buttons whose *name*-attribute is `name`, check the radio button whose *value*-attribute is `value`. Only one radio button in the family can be checked.

set_select(*data*)
Set the *selected*-attribute of the first option element specified by `data` (i.e. select an option from a drop-down).

Parameters **data** – Dict of `{name: value, ...}`. Find the select element whose *name*-attribute is `name`. Then select from among its children the option element whose *value*-attribute is `value`. If the select element's *multiple*-attribute is set, then `value` can be a list or tuple to select multiple options.

set_textarea(*data*)
Set the *string*-attribute of the first textarea element specified by `data` (i.e. set the text of a textarea).

Parameters **data** – Dict of `{name: value, ...}`. The textarea whose *name*-attribute is `name` will have its *string*-attribute set to `value`.

uncheck_all(*name*)
Remove the *checked*-attribute of all input elements with a *name*-attribute given by `name`.

3.4 Exceptions

exception mechanicalsoup.**LinkNotFoundError**
Bases: `exceptions.BaseException`

Exception raised when mechanicalsoup fails to find something.

This happens in situations like (non-exhaustive list):

- *find_link()* is called, but no link is found.
- The browser was configured with raise_on_404=True and a 404 error is triggered while browsing.
- The user tried to fill-in a field which doesn't exist in a form (e.g. browser["name"] = "val" with browser being a StatefulBrowser).

exception mechanicalsoup.**InvalidFormMethod**
Bases: `mechanicalsoup.utils.LinkNotFoundError`

This exception is raised when a method of *Form* is used for an HTML element that is of the wrong type (or is malformed). It is caught within *Form.set()* to perform element type deduction.

It is derived from *LinkNotFoundError* so that a single base class can be used to catch all exceptions specific to this module.

CHAPTER 4

Frequently Asked Questions

4.1 When to use MechanicalSoup?

MechanicalSoup is designed to simulate the behavior of a human using a web browser. Possible use-case include:

- Interacting with a website that doesn't provide a webservice API, out of a browser.
- Testing a website you're developing

There are also situations when you should *not* use MechanicalSoup, like:

- If the website provides a webservice API (e.g. REST), then you should use this API and you don't need MechanicalSoup.
- If the website you're interacting with does not contain HTML pages, then MechanicalSoup won't bring anything compared to requests, so just use requests instead.
- If the website relies on JavaScript, then you probably need a fully-fledged browser. Selenium may help you there, but it's a far heavier solution than MechanicalSoup.
- If the website is specifically designed to interact with humans, please don't go against the will of the website's owner.

4.2 "No parser was explicitly specified"

> UserWarning: No parser was explicitly specified, so I'm using the best available HTML parser for this system ("lxml"). This usually isn't a problem, but if you run this code on another system, or in a different virtual environment, it may use a different parser and behave differently.

Recent versions of BeautifulSoup show a harmless warning to encourage you to specify which HTML parser to use. You can do this in MechanicalSoup:

```
mechanicalsoup.Browser(soup_config={'features':'html.parser'})
```

Or if you have the parser lxml installed:

```
mechanicalsoup.Browser(soup_config={'features':'lxml'})
```

See also https://www.crummy.com/software/BeautifulSoup/bs4/doc/#you-need-a-parser

4.3 "ReferenceError: weakly-referenced object no longer exists"

This error can occur within requests' `session.py` when called by the destructor (`__del__`) of browser. The solution is to call *`close()`* before the end of life of the object.

Alternatively, you may also use the `with` statement which closes the browser for you:

```
def test_with():
    with mechanicalsoup.StatefulBrowser() as browser:
        browser.open(url)
        # ...
    # implicit call to browser.close() here.
```

This problem is fixed in MechanicalSoup 1.0, so this is only required for compatibility with older versions. Code using new versions can let the `browser` variable go out of scope and let the garbage collector close it properly.

4.4 How do I get debug information/logs

To understand what's going on while running a script, you have two options:

- Use *`set_verbose()`* to set the debug level to 1 (show one dot for each page opened, a poor man's progress bar) or 2 (show the URL of each visited page).
- Activate request's logging:

  ```
  import requests
  import logging

  logging.getLogger().setLevel(logging.DEBUG)
  requests_log = logging.getLogger("requests.packages.urllib3")
  requests_log.setLevel(logging.DEBUG)
  requests_log.propagate = True
  ```

  This will display a much more verbose output, including HTTP status code for each page visited. Note that unlike MechanicalSoup's logging system, this includes URL returning a redirect (e.g. HTTP 301), that are dealt with automatically by requests and not visible to MechanicalSoup.

4.5 Should I use Browser or StatefulBrowser?

Short answer: *`mechanicalsoup.StatefulBrowser`*.

`mechanicalsoup.Browser` is historically the first class that was introduced in Mechanicalsoup. Using it is a bit verbose, as the caller needs to store the URL of the currently visited page and manipulate the current form with a separate variable. *`mechanicalsoup.StatefulBrowser`* is essentially a superset of *`mechanicalsoup.Browser`*, it's the one you should use unless you have a good reason to do otherwise.

4.6 How does MechanicalSoup compare to the alternatives?

There are other libraries with the same purpose as MechanicalSoup:

- Mechanize is an ancestor of MechanicalSoup (getting its name from the Perl mechanize module). It was a great tool, but doesn't support Python 3. It was unmaintained for several years but got a new maintainer in 2017. Note that Mechanize is a much bigger piece of code (around 20 times more lines!) than MechanicalSoup, which is small because it delegates most of its work to BeautifulSoup and requests.
- RoboBrowser is very similar to MechanicalSoup. Both are small libraries built on top of requests and BeautifulSoup. Their APIs are very similar. Both have an automated testsuite. As of writing, MechanicalSoup is more actively maintained (only 1 really active developer and no activity the last two years for RoboBrowser).
- Selenium is a much heavier solution: it launches a real web browser (Firefox, Chrome, ...) and controls it with inter-process communication. Selenium is the right solution if you want to test that a website works properly with various browsers (e.g. is the JavaScript code you're writing compatible with all major browsers on the market?), and is generally useful when you need JavaScript support. Though MechanicalSoup does not support JavaScript, it also does not have the overhead of a real web browser, which makes it a simple and efficient solution for basic website interactions.

CHAPTER 5

External Resources

5.1 External libraries

- Requests (HTTP layer): http://docs.python-requests.org/en/master/
- BeautifulSoup (HTML parsing and manipulation): https://www.crummy.com/software/BeautifulSoup/bs4/doc/

5.2 MechanicalSoup on the web

- MechanicalSoup tag on stackoverflow
- MechanicalSoup on Gitter
- News archive:
  - opensource.com blog
  - Hacker News post
  - Reddit discussion

5.3 Projects using MechanicalSoup

These projects use MechanicalSoup for web scraping. You may want to look at their source code for real-life examples.

- Chamilo Tools
- gmusicapi: an unofficial API for Google Play Music
- PatZilla: Patent information research for humans
- *TODO: Add your favorite tool here ...*

CHAPTER 6

Release Notes

6.1 Version 0.10

6.1.1 Main changes:

- Added `StatefulBrowser.refresh()` to reload the current page with the same request. [#188]
- `StatefulBrowser.follow_link`, `StatefulBrowser.submit_selected()` and the new `StatefulBrowser.download_link` now sets the `Referer:` HTTP header to the page from which the link is followed. [#179]
- Added method `StatefulBrowser.download_link`, which will download the contents of a link to a file without changing the state of the browser. [#170]
- The `selector` argument of `Browser.select_form` can now be a bs4.element.Tag in addition to a CSS selector. [#169]
- `Browser.submit` and `StatefulBrowser.submit_selected` accept a larger number of keyword arguments. Arguments are forwarded to requests.Session.request. [#166]

6.1.2 Internal changes:

- `StatefulBrowser.choose_submit` will now ignore input elements that are missing a name-attribute instead of raising a `KeyError`. [#180]
- Private methods `Browser._build_request` and `Browser._prepare_request` have been replaced by a single method `Browser._request`. [#166]

6.2 Version 0.9

6.2.1 Main changes:

- We do not rely on BeautifulSoup's default choice of HTML parser. Instead, we now specify `lxml` as default. As a consequence, the default setting requires `lxml` as a dependency.
- Python 2.6 and 3.3 are no longer supported.
- The GitHub URL moved from https://github.com/hickford/MechanicalSoup/ to https://github.com/MechanicalSoup/MechanicalSoup. @moy and @hemberger are now officially administrators of the project in addition to @hickford, the original author.
- We now have a documentation site: https://mechanicalsoup.readthedocs.io/. The API is now fully documented, and we have included a tutorial, several more code examples, and a FAQ.
- `StatefulBrowser.select_form` can now be called without argument, and defaults to `"form"` in this case. It also has a new argument, `nr` (defaults to 0), which can be used to specify the index of the form to select if multiple forms match the selection criteria.
- We now use requirement files. You can install the dependencies of MechanicalSoup with e.g.:

```
pip install -r requirements.txt -r tests/requirements.txt
```

- The `Form` class was restructured and has a new API. The behavior of existing code is unchanged, but a new collection of methods has been added for clarity and consistency with the `set` method:
  - `set_input` deprecates `input`
  - `set_textarea` deprecates `textarea`
  - `set_select` is new
  - `set_checkbox` and `set_radio` together deprecate `check` (checkboxes are handled differently by default)
- A new `Form.print_summary` method allows you to write `browser.get_current_form().print_summary()` to get a summary of the fields you need to fill-in (and which ones are already filled-in).
- The `Form` class now supports selecting multiple options in a `<select multiple>` element.

6.2.2 Bug fixes

- Checking checkboxes with `browser["name"] = ("val1", "val2")` now unchecks all checkbox except the ones explicitly specified.
- `StatefulBrowser.submit_selected` and `StatefulBrowser.open` now reset __current_page to None when the result is not an HTML page. This fixes a bug where __current_page was still the previous page.
- We don't error out anymore when trying to uncheck a box which doesn't have a `checkbox` attribute.
- `Form.new_control` now correctly overrides existing elements.

6.2.3 Internal changes

- The testsuite has been further improved and reached 100% coverage.
- Tests are now run against the local version of MechanicalSoup, not against the installed version.

- `Browser.add_soup` will now always attach a *soup*-attribute. If the response is not text/html, then soup is set to None.
- `Form.set(force=True)` creates an `<input type=text ...>` element instead of an `<input type=input ...>`.

6.3 Version 0.8

6.3.1 Main changes:

- *Browser* and *StatefulBrowser* can now be configured to raise a *LinkNotFound* exception when encountering a 404 Not Found error. This is activated by passing *raise_on_404=True* to the constructor. It is disabled by default for backward compatibility, but is highly recommended.
- *Browser* now has a *__del__* method that closes the current session when the object is deleted.
- A *Link* object can now be passed to *follow_link*.
- The user agent can now be customized. The default includes *MechanicalSoup* and its version.
- There is now a direct interface to the cookiejar in **Browser* classes (*(set|get)_cookiejar* methods).
- This is the last MechanicalSoup version supporting Python 2.6 and 3.3.

6.3.2 Bug fixes:

- We used to crash on forms without action="..." fields.
- The *choose_submit* method has been fixed, and the *btnName* argument of *StatefulBrowser.submit_selected* is now a shortcut for using *choose_submit*.
- Arguments to *open_relative* were not properly forwarded.

6.3.3 Internal changes:

- The testsuite has been greatly improved. It now uses the pytest API (not only the *pytest* launcher) for more concise code.
- The coverage of the testsuite is now measured with codecov.io. The results can be viewed on: https://codecov.io/gh/hickford/MechanicalSoup
- We now have a requires.io badge to help us tracking issues with dependencies. The report can be viewed on: https://requires.io/github/hickford/MechanicalSoup/requirements/
- The version number now appears in a single place in the source code.

6.4 Version 0.7

see Git history, no changelog sorry.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

F

G

I

L

M

N

O

P

R

S

U