parslepy Documentation

Release 0.3.0

Paul Tremberth

Jun 07, 2017

Contents

CHAPTER 1

Introduction

parslepy lets you extract content from HTML and XML documents **using rules defined in a JSON object** (or a Python `dict`). The object keys mean the names you want to assign for the data in each document section and the values are CSS selectors or XPath expressions that will match the document parts (elements or attributes).

Here is an example for extracting questions in StackOverflow first page:

```
{
    "first_page_questions(//div[contains(@class,'question-summary')])": [{
        "title": ".//h3/a",
        "tags": "div.tags",
        "votes": "div.votes div.mini-counts",
        "views": "div.views div.mini-counts",
        "answers": "div.status div.mini-counts"
    }]
}
```

Some details

parslepy is a Python implementation (built on top of lxml and cssselect) of the Parsley DSL for extraction content from structured documents, defined by Kyle Maxwell and Andrew Cantino (see the parsley wiki for more details and original C implementation).

The default behavior for the selectors is:

- selectors for elements will output their matching textual content (children elements' content is also included)
- selectors matching element attributes will output the attribute's value

So, if you use `//h1/a` in a selector, *parslepy* will extract the text inside of the `a` element and its children, and if you use `//h1/a/@href` it will extract the value for `href`, i.e., the address the link is pointing to.

You can also nest objects, generate lists of objects, and mix CSS and XPath – although not in the same selector.

parslepy understands what lxml and cssselect understand, which is roughly CSS3 Selectors and XPath 1.0.

Syntax summary

Here is a quick description of the rules format:

```
output key (mandatory)
    |
    |  optionality operator (optional)
    |   |
    |   |   scope, always within brackets (optional)
    |   |      |
    v   v      v
"somekey?(someselector)":   "someCSSSelector"

or         //           :   "someXPathExpression"

or         //           :   ["someXPathOrCSSExpression"]

or         //           :    { ...some other rules... }

or         //           :    [{ ...some other rules... }]
```

A collection of extraction rules (also called a *parselet*, or *Parsley script*) looks like this:

```
{
    "somekey": "#someID .someclass",                        # using a CSS selector
    "anotherkey": "//sometag[@someattribute='somevalue']",  # using an XPath␣
↪expression
    "nestedkey(.somelistclass)": [{                         # CSS selector for␣
↪multiple elements (scope selector)
        "somenestedkey": "somenestedtag/@someattribute"     # XPath expression for an␣
↪attribbute
   }]
}
```

And the output would be something like:

```
{
    "somekey": "some value inside the first element matching the CSS selector",
    "anotherkey": "some value inside the first element matching the XPath expression",
    "nestedkey: [
        {"somenestedkey": "attribute of 1st nested element"},
        {"somenestedkey": "attribute of 2nd nested element"},
        ...
        {"somenestedkey": "attribute of last nested element"}
    ]
}
```

CHAPTER 2

Quickstart

Install

From PyPI

You can install *parslepy* from PyPI:

```
sudo pip install parslepy
```

From latest source

You can also install from source code (make sure you have the `lxml` and `cssselect` libraries already installed):

```
git clone https://github.com/redapple/parslepy.git
sudo python setup.py install
```

You probably want also to make sure the tests passes:

```
sudo pip install nosetests # only needed if you don't have nosetests installed
nosetests -v tests
```

Usage

Here are some examples on how to use parslepy. You can also check out the examples and tutorials at parsley's wiki at GitHub.

Extract the questions from StackOverflow first page:

```
>>> import parslepy, urllib2
>>> rules = {"questions(//div[contains(@class,'question-summary')])": [{"title": ".//
↪h3/a", "votes": "div.votes div.mini-counts"}]}
>>> parslepy.Parselet(rules).parse(urllib2.urlopen('http://stackoverflow.com'))
{'questions': [{'title': u'node.js RSS memory grows over time despite fairly␣
↪consistent heap sizes',
    'votes': u'0'},
    {'title': u'SQL query for count of predicate applied on rows of subquery',
    'votes': u'3'},
    ...
}
```

Extract a page heading and a list of item links from a string containing HTML:

```
>>> import lxml.etree
>>> import parslepy
>>> import pprint
>>> html = """
... <!DOCTYPE html>
... <html>
... <head>
...     <title>Sample document to test parslepy</title>
...     <meta http-equiv="content-type" content="text/html;charset=utf-8" />
... </head>
... <body>
... <h1 id="main">What&rsquo;s new</h1>
... <ul>
...     <li class="newsitem"><a href="/article-001.html">This is the first article</a>
↪</li>
...     <li class="newsitem"><a href="/article-002.html">A second report on something
↪</a></li>
...     <li class="newsitem"><a href="/article-003.html">Python is great!</a> <span␣
↪class="fresh">New!</span></li>
... </ul>
... </body>
... </html>"""
>>> rules = {
...      "heading": "h1#main",
...      "news(li.newsitem)": [{
...          "title": ".",
...          "url": "a/@href",
...          "fresh": ".fresh"
...      }],
... }
>>> p = parslepy.Parselet(rules)
>>> extracted = p.parse_fromstring(html)
>>> pprint.pprint(extracted)
{'heading': u'What\u2019s new',
 'news': [{'title': u'This is the first article', 'url': '/article-001.html'},
          {'title': u'A second report on something',
           'url': '/article-002.html'},
          {'fresh': u'New!',
           'title': u'Python is great! New!',
           'url': '/article-003.html'}]}
>>>
```

Extract using the rules in a JSON file (from *parslepy*'s `examples/` directory):

```
# Parselet file containing CSS selectors
$ cat examples/engadget_css.let.json
{
    "sections(nav#nav-main > ul li)": [{
        "title": ".",
        "url": "a.item @href",
    }]
}
$ python run_parslepy.py --script examples/engadget_css.let.json --url http://www.
↪engadget.com
{u'sections': [{u'title': u'News', u'url': '/'},
            {u'title': u'Reviews', u'url': '/reviews/'},
            {u'title': u'Features', u'url': '/features/'},
            {u'title': u'Galleries', u'url': '/galleries/'},
            {u'title': u'Videos', u'url': '/videos/'},
            {u'title': u'Events', u'url': '/events/'},
            {u'title': u'Podcasts',
                u'url': '/podcasts/the-engadget-podcast/'},
            {u'title': u'Engadget Show', u'url': '/videos/show/'},
            {u'title': u'Topics', u'url': '#nav-topics'}]}
```

You may want to check out the other examples given in the `examples/` directory. You can run them using the `run_parslepy.py` script like shown above.

Selector syntax

parslepy understands CSS3 Selectors and XPath 1.0 expressions.

Select elements attributes by name

It also accepts Parsley DSL's `@attributename` at the end of CSS selectors, to get the attribute(s) of the preceding selected element(s). *parslepy* supports Scrapy's `::attr(attributename)` functional pseudo element extension to CCS3, which gets attributes by `attributename`.

See the two syntax variants in use:

```
>>> import parslepy
>>> import pprint
>>>
>>> html = """
... <!DOCTYPE html>
... <html>
... <head>
...     <title>Sample document to test parslepy</title>
...     <meta http-equiv="content-type" content="text/html;charset=utf-8" />
... </head>
... <body>
... <div>
... <a class="first" href="http://www.example.com/first">First link</a>
... <a class="second" href="http://www.example.com/second">Second link</a>
... </div>
... </body>
... </html>"""
>>> rules = {
...      "links": {
```

```
...         "first_class": ["a.first::attr(href)"],
...         "second_class": ["a.second @href"],
...     }
... }
>>> p = parslepy.Parselet(rules)
>>> extracted = p.parse_fromstring(html)
>>> pprint.pprint(extracted)
{'links': {'first_class': ['http://www.example.com/first'],
           'second_class': ['http://www.example.com/second']}}
>>>
```

Select text and comments nodes

Borrowing from Scrapy's extension to CCS3 selectors, *parslepy* supports `::text` and `::comment` pseudo elements (resp. get text nodes of an element, and extract comments in XML/HTML elements)

```
>>> import parslepy
>>> import pprint
>>>
>>> html = """
... <!DOCTYPE html>
... <html>
... <head>
...     <title>Sample document to test parslepy</title>
...     <meta http-equiv="content-type" content="text/html;charset=utf-8" />
... </head>
... <body>
... <h1 id="main">News</h1>
... <!-- this is a comment -->
... <div>
... <p>Something to say</p>
... <!-- this is another comment -->
... </div>
... </body>
... </html>"""
>>> rules = {
...     "comments": {
...         "all": ["::comment"],
...         "inside_div": "div::comment"
...     }
... }
>>> p = parslepy.Parselet(rules)
>>> extracted = p.parse_fromstring(html)
>>> pprint.pprint(extracted)
{'comments': {'all': [u'this is a comment', u'this is another comment'],
              'inside_div': u'this is another comment'}}
>>>
```

CHAPTER 3

Dependencies

The current dependencies of the master branch are:

- lxml>=2.3 (http://lxml.de/, https://pypi.python.org/pypi/lxml)
- cssselect (https://github.com/SimonSapin/cssselect/, https://pypi.python.org/pypi/cssselect) (for lxml>=3)

CHAPTER 4

API

Parselet is the main class for extracting content from documents with *parslepy*.

Instantiate it with a Parsley script, containing a mapping of name keys, and selectors (CSS or XPath) to apply on documents, or document parts.

Then, run the extraction rules by passing an HTML or XML document to *extract()* or *parse()*

The output will be a `dict` containing the same keys as in your Parsley script, and, depending on your selectors, values will be:

- text serialization of matching elements
- element attributes
- nested lists of extraction content

class `parslepy.base.`**`Parselet`** (*parselet*, *selector_handler=None*, *strict=False*, *debug=False*)

Take a parselet and optional selector_handler and build an abstract representation of the Parsley extraction logic.

Two helper class methods can be used to instantiate a Parselet from JSON rules: *from_jsonstring()*, *from_jsonfile()*.

Parameters

- **`parselet`** (*dict*) – Parsley script as a Python dict object
- **`strict`** (*boolean*) – Set to *True* is you want to enforce that missing required keys raise an Exception; default is False (i.e. lenient/non-strict mode)
- **`selector_handler`** – an instance of `selectors.SelectorHandler` optional selector handler instance; defaults to an instance of `selectors.DefaultSelectorHandler`

Raises *InvalidKeySyntax*

Example:

```
>>> import parslepy
>>> rules = {
```

```
...     "heading": "h1#main",
...     "news(li.newsitem)": [{
...         "title": ".",
...         "url": "a/@href"
...     }],
... }
>>> p = parslepy.Parselet(rules)
>>> type(p)
<class 'parslepy.base.Parselet'>
```

Use `extract()` or `parse()` to get extracted content from documents.

extract (*document*, *context=None*)

Extract values as a dict object following the structure of the Parsley script (recursive)

Parameters

- **document** – lxml-parsed document
- **context** – user-supplied context that will be passed to custom XPath extensions (as first argument)

Return type Python *dict* object with mapped extracted content

Raises *NonMatchingNonOptionalKey*

```
>>> import lxml.etree
>>> import parslepy
>>> html = '''
... <!DOCTYPE html>
... <html>
... <head>
...     <title>Sample document to test parslepy</title>
...     <meta http-equiv="content-type" content="text/html;charset=utf-8" />
... </head>
... <body>
... <h1 id="main">What&rsquo;s new</h1>
... <ul>
...     <li class="newsitem"><a href="/article-001.html">This is the first␣
↪article</a></li>
...     <li class="newsitem"><a href="/article-002.html">A second report on␣
↪something</a></li>
...     <li class="newsitem"><a href="/article-003.html">Python is great!</a>
↪<span class="fresh">New!</span></li>
... </ul>
... </body>
... </html>
... '''
>>> html_parser = lxml.etree.HTMLParser()
>>> doc = lxml.etree.fromstring(html, parser=html_parser)
>>> doc
<Element html at 0x7f5fb1fce9b0>
>>> rules = {
...     "headingcss": "#main",
...     "headingxpath": "//h1[@id='main']"
... }
>>> p = parslepy.Parselet(rules)
>>> p.extract(doc)
{'headingcss': u'What's new', 'headingxpath': u'What's new'}
```

classmethod from_jsonfile (*fp*, *selector_handler=None*, *strict=False*, *debug=False*)
Create a Parselet instance from a file containing the Parsley script as a JSON object

```
>>> import parslepy
>>> with open('parselet.json') as fp:
...     parslepy.Parselet.from_jsonfile(fp)
...
<parslepy.base.Parselet object at 0x2014e50>
```

Parameters **fp** (*file*) – an open file-like pointer containing the Parsley script

Return type *Parselet*

Other arguments: same as for *Parselet* contructor

classmethod from_jsonstring (*s*, *selector_handler=None*, *strict=False*, *debug=False*)
Create a Parselet instance from s (str) containing the Parsley script as JSON

```
>>> import parslepy
>>> parsley_string = '{ "title": "h1", "link": "a @href"}'
>>> p = parslepy.Parselet.from_jsonstring(parsley_string)
>>> type(p)
<class 'parslepy.base.Parselet'>
>>>
```

Parameters **s** (*string*) – a Parsley script as a JSON string

Return type *Parselet*

Other arguments: same as for *Parselet* contructor

keys ()
Return a list of 1st level keys of the output data model

```
>>> import parslepy
>>> rules = {
...     "headingcss": "#main",
...     "headingxpath": "//h1[@id='main']"
... }
>>> p = parslepy.Parselet(rules)
>>> sorted(p.keys())
['headingcss', 'headingxpath']
```

parse (*fp*, *parser=None*, *context=None*)
Parse an HTML or XML document and return the extacted object following the Parsley rules give at instantiation.

Parameters

- **fp** – file-like object containing an HTML or XML document, or URL or filename
- **parser** – *lxml.etree._FeedParser* instance (optional); defaults to lxml.etree.HTMLParser()
- **context** – user-supplied context that will be passed to custom XPath extensions (as first argument)

Return type Python `dict` object with mapped extracted content

Raises *NonMatchingNonOptionalKey*

To parse from a string, use the `parse_fromstring()` method instead.

Note that the fp paramater is passed directly to lxml.etree.parse, so you can also give it an URL, and lxml will download it for you. (Also see http://lxml.de/tutorial.html#the-parse-function.)

parse_fromstring(*s*, *parser=None*, *context=None*)

Parse an HTML or XML document and return the extacted object following the Parsley rules give at instantiation.

Parameters

- **s** (*string*) – an HTML or XML document as a string
- **parser** – *lxml.etree._FeedParser* instance (optional); defaults to lxml.etree.HTMLParser()
- **context** – user-supplied context that will be passed to custom XPath extensions (as first argument)

Return type Python `dict` object with mapped extracted content

Raises *NonMatchingNonOptionalKey*

CHAPTER 5

Customizing

You can use a `Parselet` directly with it's default configuration, which should work fine for HTML documents when the content you want to extract can be accessed by regular CSS3 selectors or XPath 1.0 expressions.

But you can also customize how selectors are interpreted by sub-classing `SelectorHandler` and passing an instance of your selector handler to the Parselet constructor.

class `parslepy.selectors.`**`Selector`**(*selector*)

Class of objects returned by `SelectorHandler` instances' (and subclasses) `make()` method.

class `parslepy.selectors.`**`SelectorHandler`**(*debug=False*)

Called when building abstract Parsley trees and when etracting object values during the actual parsing of documents

This should be subclassed to implement the selector processing logic you need for your Parsley handling.

All 3 methods, `make()`, `select()` and `extract()` MUST be overridden

`extract`(*document*, *selector*)

Apply the selector on the document and return a value for the matching elements (text content or element attributes)

Parameters

- **`document`** – lxml-parsed document
- **`selector`** – input `Selector` to apply on the document

Return type depends on the selector (string, boolean value, ..)

Return value can be single- or multi-valued.

`make`(*selection_string*)

Interpret a selection_string as a selector for elements or element attributes in a (semi-)structured document. In case of XPath selectors, this can also be a function call.

Parameters **`selection_string`** – a string representing a selector

Return type `Selector`

select (*document*, *selector*)
Apply the selector on the document

Parameters

- **document** – lxml-parsed document
- **selector** – input `Selector` to apply on the document

Return type lxml.etree.Element list

class `parslepy.selectors.`**XPathSelectorHandler** (*namespaces=None*, *extensions=None*, *context=None*, *debug=False*)

This selector only accepts XPath selectors.

It understands what lxml.etree.XPath understands, that is XPath 1.0 expressions

Parameters

- **namespaces** – namespace mapping as `dict`
- **extensions** – extension `dict`
- **context** – user-context passed to XPath extension functions

namespaces and *extensions* dicts should have the same format as for lxml: see http://lxml.de/xpathxslt.html#namespaces-and-prefixes and http://lxml.de/extensions.html#xpath-extension-functions

Extension functions have a slightly different signature than pure-lxml extension functions: they must expect a user-context as first argument; all other arguments are the same as for *lxml* extensions.

context will be passed as first argument to extension functions registered through *extensions*. Alternative: user-context can also be passed to `parslepy.base.Parselet.parse()`

class `parslepy.selectors.`**DefaultSelectorHandler** (*namespaces=None*, *extensions=None*, *context=None*, *debug=False*)

Default selector logic, loosely based on the original *Parsley* implementation.

This handler understands what cssselect and lxml.etree.XPath understands, that is (roughly) XPath 1.0 and CSS3 for things that dont need browser context

Parameters

- **namespaces** – namespace mapping as `dict`
- **extensions** – extension `dict`
- **context** – user-context passed to XPath extension functions

namespaces and *extensions* dicts should have the same format as for lxml: see http://lxml.de/xpathxslt.html#namespaces-and-prefixes and http://lxml.de/extensions.html#xpath-extension-functions

Extension functions have a slightly different signature than pure-lxml extension functions: they must expect a user-context as first argument; all other arguments are the same as for *lxml* extensions.

context will be passed as first argument to extension functions registered through *extensions*. Alternative: user-context can also be passed to `parslepy.base.Parselet.parse()`

Example with iTunes RSS feed:

```
>>> import lxml.etree
>>> xml_parser = lxml.etree.XMLParser()
>>> url = 'http://itunes.apple.com/us/rss/topalbums/limit=10/explicit=true/xml'
>>>
>>> # register Atom and iTunes namespaces with prefixes "atom" and "im"
... # with a custom SelectorHandler
```

```
... xsh = parslepy.XPathSelectorHandler(
...     namespaces={
...         'atom': 'http://www.w3.org/2005/Atom',
...         'im': 'http://itunes.apple.com/rss'
...     })
>>>
>>> # use prefixes to target elements in the XML document
>>> rules = {
...     "entries(//atom:feed/atom:entry)": [
...         {
...             "title": "atom:title",
...             "name": "im:name",
...             "id": "atom:id/@im:id",
...             "artist(im:artist)": {
...                 "name": ".",
...                 "href": "@href"
...             },
...             "images(im:image)": [{
...                 "height": "@height",
...                 "url": "."
...             }],
...             "releasedate": "im:releaseDate"
...         }
...     ]
... }
>>> parselet = parslepy.Parselet(rules, selector_handler=xsh)
>>> parselet.parse(url, parser=xml_parser)
{'entries': [{'name': u'Born Sinner (Deluxe Version)', ...
```

CHAPTER 6

Exceptions

exception `parslepy.base.`**`InvalidKeySyntax`**

Raised when the input Parsley script's syntax is invalid

```
>>> import parslepy
>>> try:
...     p = parslepy.Parselet({"heading@": "#main"})
... except parslepy.base.InvalidKeySyntax as e:
...     print e
Key heading@ is not valid
```

exception `parslepy.base.`**`NonMatchingNonOptionalKey`**

Raised by a *`Parselet`* instance while extracting content in strict mode, when a required key does not yield any content.

```
>>> import parslepy
>>> html = '''
... <!DOCTYPE html>
... <html>
... <head>
...     <title>Sample document to test parslepy</title>
...     <meta http-equiv="content-type" content="text/html;charset=utf-8" />
... </head>
... <body>
... <h1 id="main">What&rsquo;s new</h1>
... <ul>
...     <li class="newsitem"><a href="/article-001.html">This is the first article
↪</a></li>
...     <li class="newsitem"><a href="/article-002.html">A second report on␣
↪something</a></li>
...     <li class="newsitem"><a href="/article-003.html">Python is great!</a>
↪<span class="fresh">New!</span></li>
... </ul>
... </body>
... </html>
... '''
```

```
>>> rules = {
...     "heading1": "h1#main",
...     "heading2": "h2#main",
... }
>>> p = parslepy.Parselet(rules, strict=True)
>>> try:
...     p.parse_fromstring(html)
... except parslepy.base.NonMatchingNonOptionalKey as e:
...     print "Missing mandatory key"
Missing mandatory key
```

CHAPTER 7

Extension functions

parslepy extends XPath 1.0 functions through lxml's XPath extensions. See http://lxml.de/extensions.html for details.

Built-in extensions

parslepy comes with a few XPath extension functions. These functions are available by default when you use `XPathSelectorHandler` or `DefaultSelectorHandler`.

- `parslepy:text(xpath_expression[, false()])`: returns the text content for elements matching *xpath_expression*. The optional boolean second parameter indicates whether *tail* content should be included or not. (Internally, this calls *lxml.etree.tostring(..., method="text", encoding=unicode)*.) Use *true()* and *false()* XPath functions, not only *true* or *false*, (or 1 or 0). Defaults to *true()*.

```
>>> import parslepy
>>> doc = """<!DOCTYPE html>
... <html>
... <head>
...     <title>Some page title</title>
... </head>
...
... <body>
...     <h1>Some heading</h1>
...
...     Some text
...
...     <p>
...     Some paragraph
...     </p>
... </body>
...
... </html>"""
>>> rules = {"heading": "h1"}
>>>
>>> # default text extraction includes tail text
```

```
... parslepy.Parselet(rules).parse_fromstring(doc)
{'heading': u'Some heading Some text'}
>>>
>>> # 2nd argument to false means without tail text
... rules = {"heading": "parslepy:text(//h1, false())"}
>>> parslepy.Parselet(rules).parse_fromstring(doc)
{'heading': 'Some heading'}
>>>
>>> # 2nd argument to true is equivalent to default text extraction
>>> rules = {"heading": "parslepy:text(//h1, true())"}
>>> parslepy.Parselet(rules).parse_fromstring(doc)
{'heading': 'Some heading Some text'}
>>>
```

See http://lxml.de/tutorial.html#elements-contain-text for details on how lxml handles text and tail element properties

- `parslepy:textnl(xpath_expression)`: similar to `parslepy:text()` but appends *n* characters to HTML block elements such as *
*, *<hr>*, * *

```
>>> import parslepy
>>> doc = """<!DOCTYPE html>
... <html>
... <head>
...     <title>Some page title</title>
... </head>
... <body>
... <h1>Some heading</h1><p>Some paragraph<div>with some span inside</div>ending␣
↪now.</p>
... </body>
... </html>
... """
>>> parslepy.Parselet({"heading": "parslepy:text(//body)"}).parse_fromstring(doc)
{'heading': 'Some headingSome paragraphwith some span insideending now.'}
>>>
>>> parslepy.Parselet({"heading": "parslepy:textnl(//body)"}).parse_
↪fromstring(doc)
{'heading': 'Some heading\nSome paragraph\nwith some span inside\nending now.'}
>>>
```

- `parslepy:html(xpath_expression)` returns the HTML content for elements matching *xpath_expression*. Internally, this calls *lxml.html.tostring(element)*.

```
>>> import parslepy
>>> doc = """<!DOCTYPE html>
... <html>
... <head>
...     <title>Some page title</title>
... </head>
... <body>
... <h1>(Some heading)</h1>
... <h2>[some sub-heading]</h2>
... </body>
... </html>
... """
>>> parslepy.Parselet({"heading": "parslepy:html(//h1)"}).parse_fromstring(doc)
{'heading': '<h1>(Some heading)</h1>'}
>>> parslepy.Parselet({"heading": "parslepy:html(//body)"}).parse_fromstring(doc)
```

```
{'heading': '<body>\n<h1>(Some heading)</h1>\n<h2>[some sub-heading]</h2>\n</body>
↪'}
>>>
```

- `parslepy:xml(xpath_expression)` returns the XML content for elements matching *xpath_expression*. Internally, this calls *lxml.etree.tostring(element)*.
- `parslepy:strip(xpath_expression[, chars])` behaves like Python's *strip()* method for strings but for the text content of elements matching *xpath_expression*. See http://docs.python.org/2/library/string.html#string.strip

```
>>> import parslepy
>>> doc = """<!DOCTYPE html>
... <html>
... <head>
...     <title>Some page title</title>
... </head>
... <body>
... <h1>(Some heading)</h1>
... <h2>[some sub-heading]</h2>
... </body>
... </html>
... """
>>> parslepy.Parselet({"heading": "parslepy:strip(//h2, '[')"}).parse_
↪fromstring(doc)
{'heading': 'some sub-heading]'}
>>> parslepy.Parselet({"heading": "parslepy:strip(//h2, ']')"}).parse_
↪fromstring(doc)
{'heading': '[some sub-heading'}
>>> parslepy.Parselet({"heading": "parslepy:strip(//h2, '[]')"}).parse_
↪fromstring(doc)
{'heading': 'some sub-heading'}
>>> parslepy.Parselet({"heading": "parslepy:strip(//h1, '()')"}).parse_
↪fromstring(doc)
{'heading': 'Some heading'}
>>>
```

- `parslepy:attrname(xpath_expression_matching_attribute)` returns name of an attribute. This works with the catch-all-attributes @* expression or a specific attribute expression like @*class*. It may sound like a useless extension but it can be useful when combined with the simple @* selector like in the example below:

```
>>> img_attributes = {
...     "images(img)": [{
...         "attr_names": ["parslepy:attrname(@*)"],
...         "attr_vals": ["@*"],
...     }]
... }
>>> extracted = parslepy.Parselet(img_attributes).parse('http://www.python.org')
>>> for r in extracted["images"]:
...:     print dict(zip(r.get("attr_names"), r.get("attr_vals")))
...:
{'src': '/images/python-logo.gif', 'alt': 'homepage', 'border': '0', 'id': 'logo'}
{'src': '/images/trans.gif', 'alt': 'skip to navigation', 'border': '0', 'id':
↪'skiptonav'}
{'src': '/images/trans.gif', 'alt': 'skip to content', 'border': '0', 'id':
↪'skiptocontent'}
{'width': '116', 'alt': '', 'src': '/images/donate.png', 'title': '', 'height':
↪'42'}
```

```
{'width': '94', 'style': 'align:center', 'src': '/images/worldmap.jpg', 'alt':
↪'[Python resources in languages other than English]', 'height': '46'}
{'src': '/images/success/Carmanah.png', 'alt': 'success story photo', 'class':
↪'success'}
```

User-defined extensions

parslepy also lets you define your own XPath extensions, just like lxml does, except the function you register must accept a user-supplied context object passed as first argument, subsequent arguments to your extension function will be the same as for lxml extensions, i.e. an XPath context, followed by matching elements and whatever additional parameters your XPath call passes.

The user-supplied context should be passed to `parslepy.base.Parselet.parse()`, or globally to a XPathSelectorHandler subclass instance passed to instantiate a Parselet.

Let's illustrate this with a custom extension to make <*img*> @src attributes "absolute".

Suppose we already have an extraction rule that outputs the @*src* attributes from <*img*> tags on the Python.org homepage:

```
>>> import parslepy
>>> import pprint
>>> parselet = parslepy.Parselet({"img_abslinks": ["//img/@src"]})
>>> pprint.pprint(parselet.parse('http://www.python.org'))
{'img_abslinks': ['/images/python-logo.gif',
                  '/images/trans.gif',
                  '/images/trans.gif',
                  '/images/donate.png',
                  '/images/worldmap.jpg',
                  '/images/success/afnic.fr.png']}
```

We now want to generate full URLs for these images, not relative to http://www.python.org.

First we need to define our extension function as a Python function:

parslepy's extension functions must accept a user-context as first argument, then should expect an XPath context, followed by elements or strings matching the XPath expression, and finally whatever other parameters are passed to the function call in extraction rules.

In our example, we expect @*src* attribute values as input from XPath, and combine them with a base URL (via *urlparse.urljoin()*), the URL from which the HTML document was fetched. The base URL will be passed as user-context, and we will receive it as first argument. So the Python extension function may look like this:

```
>>> import urlparse
>>> def absurl(ctx, xpctx, attributes, *args):
...         # user-context "ctx" will be the URL of the page
...         return [urlparse.urljoin(ctx, u) for u in attributes]
...
```

Then, we need to register this function with parslepy through a custom selector handler, with a custom namespace and its prefix:

```
>>> # choose a prefix and namespace, e.g. "myext" and "local-extensions"
... mynamespaces = {
...         "myext": "local-extensions"
...     }
>>> myextensions = {
```

```
...         ("local-extensions", "absurl"): absurl,
...     }
>>>
>>> import parslepy
>>> sh = parslepy.DefaultSelectorHandler(
...         namespaces=mynamespaces,
...         extensions=myextensions)
>>>
```

Now we can use this **absurl()** XPath extension within *parslepy* rules, with the "myext" prefix (**do not forget to pass your selector handler** to your Parselet instance):

```
>>> rules = {"img_abslinks": ["myext:absurl(//img/@src)"]}
>>> parselet = parslepy.Parselet(rules, selector_handler=sh)
```

And finally, run the extraction rules on Python.org's homepage again, with a context argument set to the URL

```
>>> import pprint
>>> pprint.pprint(parselet.parse('http://www.python.org',
...         context='http://www.python.org'))
{'img_abslinks': ['http://www.python.org/images/python-logo.gif',
                  'http://www.python.org/images/trans.gif',
                  'http://www.python.org/images/trans.gif',
                  'http://www.python.org/images/donate.png',
                  'http://www.python.org/images/worldmap.jpg',
                  'http://www.python.org/images/success/afnic.fr.png']}
>>>
```

In this case, it may feel odd to have to pass the URL *twice*, but parse(*URL*) does not store the URL anywhere, it processes only the HTML stream from the page.

CHAPTER 8

More examples

Check out more examples and tutorials at parsley's wiki at GitHub.

CHAPTER 9

Changelog

Version 0.3.0 - March 3., 2015

- **Improvements:**
  - **CSS selectors extensions:**
    - pseudo-elements `::text` (borrowed from Scrapy) and `::comment`
    - functional pseudo-element `::attr(name)`
  - Cleaned up documentation (thanks @eliasdorneles!)
  - New `keys()` method for `Parselet` nodes

Version 0.2.0 - August 5., 2013

- **Improvements:**
  - Support XPath namespace prefixes (`namespace:element`) and CSS namespace prefixes (`namespace|element`) in DefaultSelectorHandler
  - new built-in extension function `parslepy:strip()` mapped to Python's strip() for strings
  - new built-in extension function `parslepy:attrname()` that takes an attribute selector and returns the attribute's name
  - support for user-defined extension functions, which take an additional context parameter when called (context is passed either at selector handler instantiation or when calling Parselet.parse())
  - use `smart_strings=False` for XPath compiled expressions, except for user-defined extensions and some built-in extensions (see http://lxml.de/xpathxslt.html#xpath-return-values)
- **Bug fixes:**
  - #2: XPath namespace prefixes raise cssselect.xpath.ExpressionError with DefaultSelectorHandler

  - #3: Docs suggest using *.js files when they are JSON documents
  - #4: The example usage should not have both url_css and url_xpath
  - #5: In example usage, skip lines between "configuration" and "execution"
  - #6: add underscore to _version__
  - #7: Empty result set on boolean or numerical selectors

Version 0.1.2 - July 9, 2013

- **Bug fixes:**
  - #1: headingxpath rule does not seem to work as expected

Version 0.1.1 - July 3, 2013

- Docstrings added to main classes and methods.
- Added parse_fromstring() method to Parselet
- Added tests for Parselet.parse() and Parselet.parse_fromstring()

Version 0.1 - June 30, 2013

Initial release

Index