Parsel Documentation

Release 1.0.3

Scrapy Project

November 22, 2016

Contents

Parsel is a library to extract data from HTML and XML using XPath and CSS selectors

- Free software: BSD license
- Documentation: https://parsel.readthedocs.org.

CHAPTER 1

Features

- Extract text using CSS or XPath selectors
- Regular expression helper methods

Example:

```
>>> from parsel import Selector
>>> sel = Selector(text=u"""<html>
        <body>
            <h1>Hello, Parsel!</h1>
            <ul>
                <li><a href="http://example.com">Link 1</a></li>
                <li><a href="http://scrapy.org">Link 2</a></li>
            </ul
        </body>
        </html>""")
>>>
>>> sel.css('h1::text').extract_first()
u'Hello, Parsel!'
>>>
>>> sel.css('h1::text').re('\w+')
[u'Hello', u'Parsel']
>>>
>>> for e in sel.css('ul > li'):
        print(e.xpath('.//a/@href').extract_first())
http://example.com
http://scrapy.org
```

1.1 Parsel Documentation Contents

Contents:

1.1.1 Installation

To install Parsel, we recommend you to use pip:

```
$ pip install parsel
```

You probably shouldn't, but you can also install it with easy_install:

```
$ easy_install parsel
```

1.1.2 Usage

Getting started

If you already know how to write CSS or XPath expressions, using Parsel is straightforward: you just need to create a *Selector* object for the HTML or XML text you want to parse, and use the available methods for selecting parts from the text and extracting data out of the result.

Creating a `Selector` object is simple:

```
>>> from parsel import Selector
>>> text = u"<html><body><h1>Hello, Parsel!</h1></body></html>"
>>> sel = Selector(text=text)
```

Note: One important thing to note is that if you're using Python 2, make sure to use an *unicode* object for the text argument. *Selector* expects text to be an *unicode* object in Python 2 or an *str* object in Python 3.

Once you have created the Selector object, you can use CSS or XPath expressions to select elements:

```
>>> sel.css('h1')
[<Selector xpath=u'descendant-or-self::h1' data=u'<h1>Hello, Parsel!</h1>'>]
>>> sel.xpath('//h1')  # the same, but now with XPath
[<Selector xpath='//h1' data=u'<h1>Hello, Parsel!</h1>'>]
```

And extract data from those elements:

```
>>> sel.xpath('//h1/text()').extract()
[u'Hello, Parsel!']
>>> sel.css('h1::text').extract_first()
u'Hello, Parsel!'
```

XPath is a language for selecting nodes in XML documents, which can also be used with HTML. CSS is a language for applying styles to HTML documents. It defines selectors to associate those styles with specific HTML elements.

You can use either language you're more comfortable with, though you may find that in some specific cases XPath is more powerful than CSS.

Using selectors

To explain how to use the selectors we'll use the requests library to download an example page located in the Parsel's documentation:

> http://parsel.readthedocs.org/en/latest/_static/selectors-sample1.html

For the sake of completeness, here's its full HTML code:

```
<html>
 <head>
  <base href='http://example.com/' />
  <title>Example website</title>
 </head>
 <body>
  <div id='images'>
```

```
    <a href='image1.html'>Name: My image 1 <br /><img src='image1_thumb.jpg' /></a>
    <a href='image2.html'>Name: My image 2 <br /><img src='image2_thumb.jpg' /></a>
    <a href='image3.html'>Name: My image 3 <br /><img src='image3_thumb.jpg' /></a>
    <a href='image4.html'>Name: My image 4 <br /><img src='image4_thumb.jpg' /></a>
    <a href='image5.html'>Name: My image 5 <br /><img src='image5_thumb.jpg' /></a>
  </div>
 </body>
</html>
```

So, let's download that page and create a selector for it:

```
>>> import requests
>>> from parsel import Selector
>>> url = 'http://parsel.readthedocs.org/en/latest/_static/selectors-sample1.html'
>>> text = requests.get(url).text
>>> selector = Selector(text=text)
```

Since we're dealing with HTML, the default type for Selector, we don't need to specify the *type* argument.

So, by looking at the *HTML code* of that page, let's construct an XPath for selecting the text inside the title tag:

```
>>> selector.xpath('//title/text()')
[<Selector xpath='//title/text()' data=u'Example website'>]
```

You can also ask the same thing using CSS instead:

```
>>> selector.css('title::text')
[<Selector xpath=u'descendant-or-self::title/text()' data=u'Example website'>]
```

As you can see, `.xpath()` and `.css()` methods return a *`SelectorList`* instance, which is a list of new selectors. This API can be used for quickly selecting nested data:

```
>>> selector.css('img').xpath('@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

To actually extract the textual data, you must call the selector `.extract()` method, as follows:

```
>>> selector.xpath('//title/text()').extract()
[u'Example website']
```

If you want to extract only first matched element, you can call the selector `.extract_first()`:

```
>>> selector.xpath('//div[@id="images"]/a/text()').extract_first()
u'Name: My image 1 '
```

It returns `None` if no element was found:

```
>>> selector.xpath('//div[@id="not-exists"]/text()').extract_first() is None
True
```

Notice that CSS selectors can select text or attribute nodes using CSS3 pseudo-elements:

```
>>> selector.css('title::text').extract()
[u'Example website']
```

Now we're going to get the base URL and some image links:

```
>>> selector.xpath('//base/@href').extract()
[u'http://example.com/']

>>> selector.css('base::attr(href)').extract()
[u'http://example.com/']

>>> selector.xpath('//a[contains(@href, "image")]/@href').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> selector.css('a[href*=image]::attr(href)').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> selector.xpath('//a[contains(@href, "image")]/img/@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']

>>> selector.css('a[href*=image] img::attr(src)').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

Nesting selectors

The selection methods (`.xpath()` or `.css()`) return a list of selectors of the same type, so you can call the selection methods for those selectors too. Here's an example:

```
>>> links = selector.xpath('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 u'<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>',
 u'<a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>',
 u'<a href="image4.html">Name: My image 4 <br><img src="image4_thumb.jpg"></a>',
 u'<a href="image5.html">Name: My image 5 <br><img src="image5_thumb.jpg"></a>']

>>> for index, link in enumerate(links):
...     args = (index, link.xpath('@href').extract(), link.xpath('img/@src').extract())
...     print 'Link number %d points to url %s and image %s' % args

Link number 0 points to url [u'image1.html'] and image [u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and image [u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and image [u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and image [u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and image [u'image5_thumb.jpg']
```

Using selectors with regular expressions

Selector also has a `.re()` method for extracting data using regular expressions. However, unlike using `.xpath()` or `.css()` methods, `.re()` returns a list of unicode strings. So you can't construct nested `.re()` calls.

Here's an example used to extract image names from the *HTML code* above:

```
>>> selector.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']
```

There's an additional helper reciprocating `.extract_first()` for `.re()`, named `.re_first()`. Use it to extract just the first matching string:

```
>>> selector.xpath('//a[contains(@href, "image")]/text()').re_first(r'Name:\s*(.*)')
u'My image 1'
```

Working with relative XPaths

Keep in mind that if you are nesting selectors and use an XPath that starts with `/`, that XPath will be absolute to the document and not relative to the `Selector` you're calling it from.

For example, suppose you want to extract all <p> elements inside elements. First, you would get all elements:

```
>>> divs = selector.xpath('//div')
```

At first, you may be tempted to use the following approach, which is wrong, as it actually extracts all <p> elements from the document, not only those inside elements:

```
>>> for p in divs.xpath('//p'):  # this is wrong - gets all <p> from the whole document
...     print p.extract()
```

This is the proper way to do it (note the dot prefixing the `.//p` XPath):

```
>>> for p in divs.xpath('.//p'):  # extracts all <p> inside
...     print p.extract()
```

Another common case would be to extract all direct <p> children:

```
>>> for p in divs.xpath('p'):
...     print p.extract()
```

For more details about relative XPaths see the Location Paths section in the XPath specification.

Using EXSLT extensions

Being built atop lxml, parsel selectors support some EXSLT extensions and come with these pre-registered namespaces to use in XPath expressions:

prefix	namespace	usage
re	http://exslt.org/regular-expressions	regular expressions
set	http://exslt.org/sets	set manipulation

Regular expressions The `test()` function, for example, can prove quite useful when XPath's `starts-with()` or `contains()` are not sufficient.

Example selecting links in list item with a "class" attribute ending with a digit:

```
>>> from parsel import Selector
>>> doc = u"""
... <div>
...     <ul>
...         <li class="item-0"><a href="link1.html">first item</a></li>
...         <li class="item-1"><a href="link2.html">second item</a></li>
...         <li class="item-inactive"><a href="link3.html">third item</a></li>
...         <li class="item-1"><a href="link4.html">fourth item</a></li>
...         <li class="item-0"><a href="link5.html">fifth item</a></li>
...     </ul>
... </div>
... """
>>> sel = Selector(text=doc)
>>> sel.xpath('//li//@href').extract()
[u'link1.html', u'link2.html', u'link3.html', u'link4.html', u'link5.html']
>>> sel.xpath('//li[re:test(@class, "item-\d$")]//@href').extract()
[u'link1.html', u'link2.html', u'link4.html', u'link5.html']
>>>
```

> **Warning:** C library `libxslt` doesn't natively support EXSLT regular expressions so lxml's implementation uses hooks to Python's `re` module. Thus, using regexp functions in your XPath expressions may add a small performance penalty.

Set operations These can be handy for excluding parts of a document tree before extracting text elements for example.

Example extracting microdata (sample content taken from http://schema.org/Product) with groups of itemscopes and corresponding itemprops:

```
>>> doc = u"""
... <div itemscope itemtype="http://schema.org/Product">
...   <span itemprop="name">Kenmore White 17" Microwave</span>
...   <img src="kenmore-microwave-17in.jpg" alt='Kenmore 17" Microwave' />
...   <div itemprop="aggregateRating"
...     itemscope itemtype="http://schema.org/AggregateRating">
...    Rated <span itemprop="ratingValue">3.5</span>/5
...    based on <span itemprop="reviewCount">11</span> customer reviews
...   </div>
...
...   <div itemprop="offers" itemscope itemtype="http://schema.org/Offer">
...     <span itemprop="price">$55.00</span>
...     <link itemprop="availability" href="http://schema.org/InStock" />In stock
...   </div>
...
...   Product description:
...   <span itemprop="description">0.7 cubic feet countertop microwave.
...   Has six preset cooking categories and convenience features like
...   Add-A-Minute and Child Lock.</span>
...
...   Customer reviews:
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
```

```
...      <span itemprop="name">Not a happy camper</span> -
...      by <span itemprop="author">Ellie</span>,
...      <meta itemprop="datePublished" content="2011-04-01">April 1, 2011
...      <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...        <meta itemprop="worstRating" content = "1">
...        <span itemprop="ratingValue">1</span>/
...        <span itemprop="bestRating">5</span>stars
...      </div>
...      <span itemprop="description">The lamp burned out and now I have to replace
...      it. </span>
...    </div>
...
...    <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...      <span itemprop="name">Value purchase</span> -
...      by <span itemprop="author">Lucas</span>,
...      <meta itemprop="datePublished" content="2011-03-25">March 25, 2011
...      <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...        <meta itemprop="worstRating" content = "1"/>
...        <span itemprop="ratingValue">4</span>/
...        <span itemprop="bestRating">5</span>stars
...      </div>
...      <span itemprop="description">Great microwave for the price. It is small and
...      fits in my apartment.</span>
...    </div>
...    ...
... </div>
... """
>>> sel = Selector(text=doc, type="html")
>>> for scope in sel.xpath('//div[@itemscope]'):
...     print "current scope:", scope.xpath('@itemtype').extract()
...     props = scope.xpath('''
...                 set:difference(./descendant::*/@itemprop,
...                                .//*[@itemscope]/*/@itemprop)''')
...     print "    properties:", props.extract()
...     print

current scope: [u'http://schema.org/Product']
    properties: [u'name', u'aggregateRating', u'offers', u'description', u'review', u'review']

current scope: [u'http://schema.org/AggregateRating']
    properties: [u'ratingValue', u'reviewCount']

current scope: [u'http://schema.org/Offer']
    properties: [u'price', u'availability']

current scope: [u'http://schema.org/Review']
    properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description']

current scope: [u'http://schema.org/Rating']
    properties: [u'worstRating', u'ratingValue', u'bestRating']

current scope: [u'http://schema.org/Review']
    properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description']

current scope: [u'http://schema.org/Rating']
    properties: [u'worstRating', u'ratingValue', u'bestRating']

>>>
```

Here we first iterate over `itemscope` elements, and for each one, we look for all `itemprops` elements and exclude those that are themselves inside another `itemscope`.

Some XPath tips

Here are some tips that you may find useful when using XPath with Parsel, based on this post from ScrapingHub's blog. If you are not much familiar with XPath yet, you may want to take a look first at this XPath tutorial.

Using text nodes in a condition When you need to use the text content as argument to an XPath string function, avoid using `.//text()` and use just `.` instead.

This is because the expression `.//text()` yields a collection of text elements – a *node-set*. And when a node-set is converted to a string, which happens when it is passed as argument to a string function like `contains()` or `starts-with()`, it results in the text for the first element only.

Example:

```
>>> from parsel import Selector
>>> sel = Selector(text='<a href="#">Click here to go to the <strong>Next Page</strong></a>')
```

Converting a *node-set* to string:

```
>>> sel.xpath('//a//text()').extract() # take a peek at the node-set
[u'Click here to go to the ', u'Next Page']
>>> sel.xpath("string(//a[1]//text())").extract() # convert it to string
[u'Click here to go to the ']
```

A *node* converted to a string, however, puts together the text of itself plus of all its descendants:

```
>>> sel.xpath("//a[1]").extract() # select the first node
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
>>> sel.xpath("string(//a[1])").extract() # convert it to string
[u'Click here to go to the Next Page']
```

So, using the `.//text()` node-set won't select anything in this case:

```
>>> sel.xpath("//a[contains(.//text(), 'Next Page')]").extract()
[]
```

But using the `.` to mean the node, works:

```
>>> sel.xpath("//a[contains(., 'Next Page')]").extract()
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
```

Beware of the difference between //node[1] and (//node)[1] `//node[1]` selects all the nodes occurring first under their respective parents.

`(//node)[1]` selects all the nodes in the document, and then gets only the first of them.

Example:

```
>>> from parsel import Selector
>>> sel = Selector(text="""
....:     <ul class="list">
....:         <li>1</li>
....:         <li>2</li>
....:         <li>3</li>
....:     </ul>
```

```
....:     <ul class="list">
....:         <li>4</li>
....:         <li>5</li>
....:         <li>6</li>
....:     </ul>""")
>>> xp = lambda x: sel.xpath(x).extract()
```

This gets all first `<li>` elements under whatever it is its parent:

```
>>> xp("//li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

And this gets the first `<li>` element in the whole document:

```
>>> xp("(//li)[1]")
[u'<li>1</li>']
```

This gets all first `<li>` elements under an `<ul>` parent:

```
>>> xp("//ul/li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

And this gets the first `<li>` element under an `<ul>` parent in the whole document:

```
>>> xp("(//ul/li)[1]")
[u'<li>1</li>']
```

When querying by class, consider using CSS Because an element can contain multiple CSS classes, the XPath way to select elements by class is the rather verbose:

```
*[contains(concat(' ', normalize-space(@class), ' '), ' someclass ')]
```

If you use `@class='someclass'` you may end up missing elements that have other classes, and if you just use `contains(@class, 'someclass')` to make up for that you may end up with more elements that you want, if they have a different class name that shares the string `someclass`.

As it turns out, parsel selectors allow you to chain selectors, so most of the time you can just select by class using CSS and then switch to XPath when needed:

```
>>> from parsel import Selector
>>> sel = Selector(text='<div class="hero shout"><time datetime="2014-07-23 19:00">Special date</time
>>> sel.css('.shout').xpath('./time/@datetime').extract()
[u'2014-07-23 19:00']
```

This is cleaner than using the verbose XPath trick shown above. Just remember to use the `.` in the XPath expressions that will follow.

API reference

class `parsel.selector.`**`Selector`**(*text=None*, *type=None*, *namespaces=None*, *root=None*, *base_url=None*, *_expr=None*)

`Selector` allows you to select parts of an XML or HTML text using CSS or XPath expressions and extract data from it.

`text` is a `unicode` object in Python 2 or a `str` object in Python 3

`type` defines the selector type, it can be `"html"`, `"xml"` or `None` (default). If `type` is `None`, the selector defaults to `"html"`.

css(*query*)
Apply the given CSS selector and return a `SelectorList` instance.

`query` is a string containing the CSS selector to apply.

In the background, CSS queries are translated into XPath queries using cssselect library and run `.xpath()` method.

extract()
Serialize and return the matched nodes in a single unicode string. Percent encoded content is unquoted.

re(*regex*)
Apply the given regex and return a list of unicode strings with the matches.

`regex` can be either a compiled regular expression or a string which will be compiled to a regular expression using `re.compile(regex)`

register_namespace(*prefix*, *uri*)
Register the given namespace to be used in this `Selector`. Without registering namespaces you can't select or extract data from non-standard namespaces. See *Selector examples on XML text*.

remove_namespaces()
Remove all namespaces, allowing to traverse the document using namespace-less xpaths. See *Removing namespaces*.

selectorlist_cls
alias of `SelectorList`

xpath(*query*)
Find nodes matching the xpath `query` and return the result as a `SelectorList` instance with all elements flattened. List elements implement `Selector` interface too.

`query` is a string containing the XPATH query to apply.

SelectorList objects

class parsel.selector.**SelectorList**
The `SelectorList` class is a subclass of the builtin `list` class, which provides a few additional methods.

css(*xpath*)
Call the `.css()` method for each element in this list and return their results flattened as another `SelectorList`.

`query` is the same argument as the one in `Selector.css()`

extract()
Call the `.extract()` method for each element is this list and return their results flattened, as a list of unicode strings.

extract_first(*default=None*)
Return the result of `.extract()` for the first element in this list. If the list is empty, return the default value.

re(*regex*)
Call the `.re()` method for each element in this list and return their results flattened, as a list of unicode strings.

re_first(*regex*)
Call the `.re()` method for the first element in this list and return the result in an unicode string.

xpath(*xpath*)
Call the `.xpath()` method for each element in this list and return their results flattened as another `SelectorList`.

`query` is the same argument as the one in `Selector.xpath()`

Selector examples on HTML text Here are some `Selector` examples to illustrate several concepts. In all cases, we assume there is already a `Selector` instantiated with an HTML text like this:

```
sel = Selector(text=html_text)
```

1. Select all `<h1>` elements from an HTML text, returning a list of `Selector` objects (ie. a `SelectorList` object):

```
sel.xpath("//h1")
```

2. Extract the text of all `<h1>` elements from an HTML text, returning a list of unicode strings:

```
sel.xpath("//h1").extract()         # this includes the h1 tag
sel.xpath("//h1/text()").extract()  # this excludes the h1 tag
```

3. Iterate over all `<p>` tags and print their class attribute:

```
for node in sel.xpath("//p"):
    print node.xpath("@class").extract()
```

Selector examples on XML text Here are some examples to illustrate concepts for `Selector` objects instantiated with an XML text like this:

```
sel = Selector(text=xml_text, type='xml')
```

1. Select all `<product>` elements from an XML text, returning a list of `Selector` objects (ie. a `SelectorList` object):

```
sel.xpath("//product")
```

2. Extract all prices from a Google Base XML feed which requires registering a namespace:

```
sel.register_namespace("g", "http://base.google.com/ns/1.0")
sel.xpath("//g:price").extract()
```

Removing namespaces When dealing with scraping projects, it is often quite convenient to get rid of namespaces altogether and just work with element names, to write more simple/convenient XPaths. You can use the `Selector.remove_namespaces()` method for that.

Let's show an example that illustrates this with Github blog atom feed.

Let's download the atom feed using requests and create a selector:

```
>>> import requests
>>> from parsel import Selector
>>> text = requests.get('https://github.com/blog.atom').text
>>> sel = Selector(text=text, type='xml')
```

We can try selecting all `<link>` objects and then see that it doesn't work (because the Atom XML namespace is obfuscating those nodes):

```
>>> sel.xpath("//link")
[]
```

But once we call the `Selector.remove_namespaces()` method, all nodes can be accessed directly by their names:

```
>>> sel.remove_namespaces()
>>> sel.xpath("//link")
[<Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 ...
```

If you wonder why the namespace removal procedure isn't called always by default instead of having to call it manually, this is because of two reasons, which, in order of relevance, are:

1. Removing namespaces requires to iterate and modify all nodes in the document, which is a reasonably expensive operation to perform by default for all documents.
2. There could be some cases where using namespaces is actually required, in case some element names clash between namespaces. These cases are very rare though.

Similar libraries

- BeautifulSoup is a very popular screen scraping library among Python programmers which constructs a Python object based on the structure of the HTML code and also deals with bad markup reasonably well.
- lxml is an XML parsing library (which also parses HTML) with a pythonic API based on ElementTree. (lxml is not part of the Python standard library.). Parsel uses it under-the-hood.
- PyQuery is a library that, like Parsel, uses lxml and cssselect under the hood, but it offers a jQuery-like API to traverse and manipulate XML/HTML documents.

Parsel is built on top of the lxml library, which means they're very similar in speed and parsing accuracy. The advantage of using Parsel over lxml is that Parsel is simpler to use and extend, unlike the lxml API which is much bigger because the lxml library can be used for many other tasks, besides selecting markup documents. Also, Parsel allows you to use CSS, by translating CSS to XPath using the cssselect library.

1.1.3 History

1.0.3 (2016-07-29)

- Add BSD-3-Clause license file
- Re-enable PyPy tests
- Integrate py.test runs with setuptools (needed for Debian packaging)
- Changelog is now called `NEWS`

1.0.2 (2016-04-26)

- Fix bug in exception handling causing original traceback to be lost
- Added docstrings and other doc fixes

1.0.1 (2015-08-24)

- Updated PyPI classifiers
- Added docstrings for csstranslator module and other doc fixes

1.0.0 (2015-08-22)

- Documentation fixes

0.9.6 (2015-08-14)

- Updated documentation
- Extended test coverage

0.9.5 (2015-08-11)

- Support for extending SelectorList

0.9.4 (2015-08-10)

- Try workaround for travis-ci/dpl#253

0.9.3 (2015-08-07)

- Add base_url argument

0.9.2 (2015-08-07)

- Rename module unified -> selector and promoted root attribute
- Add create_root_node function

0.9.1 (2015-08-04)

- Setup Sphinx build and docs structure
- Build universal wheels
- Rename some leftovers from package extraction

0.9.0 (2015-07-30)

- First release on PyPI.

1.2 Indices and tables

- genindex
- modindex
- search

C

E

R

S

X