Selectors

When you're scraping web pages, the most common task you need to perform is to extract data from the HTML source. There are several libraries available to achieve this:

- BeautifulSoup is a very popular web 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, but it has one drawback: it's slow.
- 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.)

Scrapy comes with its own mechanism for extracting data. They're called selectors because they "select" certain parts of the HTML document specified either by XPath or CSS expressions.

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.

Scrapy selectors are built over the lxml library, which means they're very similar in speed and parsing accuracy.

This page explains how selectors work and describes their API which is very small and simple, unlike the lxml API which is much bigger because the lxml library can be used for many other tasks, besides selecting markup documents.

For a complete reference of the selectors API see Selector reference

Using selectors

Constructing selectors

Scrapy selectors are instances of `Selector` class constructed by passing **text** or `TextResponse` object. It automatically chooses the best parsing rules (XML vs HTML) based on input type:

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
```

Constructing from text:

```
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').extract()
[u'good']
```

Constructing from response:

```
>>> response = HtmlResponse(url='http://example.com', body=body)
>>> Selector(response=response).xpath('//span/text()').extract()
[u'good']
```

For convenience, response objects expose a selector on *.selector* attribute, it's totally OK to use this shortcut when possible:

```
>>> response.selector.xpath('//span/text()').extract()
[u'good']
```

Using selectors

To explain how to use the selectors we'll use the *Scrapy shell* (which provides interactive testing) and an example page located in the Scrapy documentation server:

> http://doc.scrapy.org/en/latest/_static/selectors-sample1.html

Here's its 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>
```

First, let's open the shell:

```
scrapy shell http://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

Then, after the shell loads, you'll have the response available as `response` shell variable, and its attached selector in `response.selector` attribute.

Since we're dealing with HTML, the selector will automatically use an HTML parser.

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

```
>>> response.selector.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
```

Querying responses using XPath and CSS is so common that responses include two convenience shortcuts: `response.xpath()` and `response.css()`:

```
>>> response.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
>>> response.css('title::text')
[<Selector (text) xpath=//title/text()>]
```

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:

```
>>> response.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:

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

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

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

It returns `None` if no element was found:

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

A default return value can be provided as an argument, to be used instead of `None`:

```
>>> response.xpath('//div[@id="not-exists"]/text()').extract_first(default='not-found')
'not-found'
```

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

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

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

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

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

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

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

>>> response.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']

>>> response.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 = response.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:

```
>>> response.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:

```
>>> response.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 `<div>` elements. First, you would get all `<div>` elements:

```
>>> divs = response.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 `<div>` 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.

Variables in XPath expressions

XPath allows you to reference variables in your XPath expressions, using the `$somevariable` syntax. This is somewhat similar to parameterized queries or prepared statements in the SQL world where you replace some arguments in your queries with placeholders like `?`, which are then substituted with values passed with the query.

Here's an example to match an element based on its "id" attribute value, without hard-coding it (that was shown previously):

```
>>> # `$val` used in the expression, a `val` argument needs to be passed
>>> response.xpath('//div[@id=$val]/a/text()', val='images').extract_first()
u'Name: My image 1 '
```

Here's another example, to find the "id" attribute of a `<div>` tag containing five `<a>` children (here we pass the value `5` as an integer):

```
>>> response.xpath('//div[count(a)=$cnt]/@id', cnt=5).extract_first()
u'images'
```

All variable references must have a binding value when calling `.xpath()` (otherwise you'll get a `ValueError: XPath error:` exception). This is done by passing as many named arguments as necessary.

parsel, the library powering Scrapy selectors, has more details and examples on XPath variables.

Using EXSLT extensions

Being built atop lxml, Scrapy selectors also 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 scrapy import Selector
>>> doc = """
... <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, type="html")
>>> 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 = """
... <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 Scrapy selectors, 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 scrapy 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 scrapy 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, Scrapy 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 scrapy import Selector
>>> sel = Selector(text='<div class="hero shout"><time datetime="2014-07-23 19:00">Special
date</time></div>')
>>> 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.

Built-in Selectors reference

Selector objects

class `scrapy.selector.Selector`(*response=None, text=None, type=None*)

An instance of `Selector` is a wrapper over response to select certain parts of its content.

`response` is an `HtmlResponse` or an `XmlResponse` object that will be used for selecting and extracting data.

`text` is a unicode string or utf-8 encoded text for cases when a `response` isn't available. Using `text` and `response` together is undefined behavior.

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

> If `type` is `None`, the selector automatically chooses the best type based on `response` type (see below), or defaults to `"html"` in case it is used together with `text`.
>
> If `type` is `None` and a `response` is passed, the selector type is inferred from the response type as follows:
>
> - `"html"` for `HtmlResponse` type
> - `"xml"` for `XmlResponse` type
> - `"html"` for anything else

Otherwise, if `type` is set, the selector type will be forced and no detection will occur.

`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.

Note

For convenience, this method can be called as `response.xpath()`

`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.

Note

For convenience this method can be called as `response.css()`

`extract`()

Serialize and return the matched nodes as a list of unicode strings. 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)`

Note

Note that `re()` and `re_first()` both decode HTML entities (except `&lt;` and `&amp;`).

`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 examples below.

`remove_namespaces`()

Remove all namespaces, allowing to traverse the document using namespace-less xpaths. See example below.

`__nonzero__`()

Returns `True` if there is any real content selected or `False` otherwise. In other words, the boolean value of a `Selector` is given by the contents it selects.

SelectorList objects

class **`scrapy.selector.SelectorList`**

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

`xpath`(*query*)

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()`

`css`(*query*)

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 in this list and return their results flattened, as a list of unicode strings.

> **re()**

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

Selector examples on HTML response

Here's a couple of `Selector` examples to illustrate several concepts. In all cases, we assume there is already a `Selector` instantiated with a `HtmlResponse` object like this:

```
sel = Selector(html_response)
```

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

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

2. Extract the text of all `<h1>` elements from an HTML response body, 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 response

Here's a couple of examples to illustrate several concepts. In both cases we assume there is already a `Selector` instantiated with an `XmlResponse` object like this:

```
sel = Selector(xml_response)
```

1. Select all `<product>` elements from an XML response body, 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.

First, we open the shell with the url we want to scrape:

```
$ scrapy shell https://github.com/blog.atom
```

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

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

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

```
>>> response.selector.remove_namespaces()
>>> response.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 always called 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 for all documents crawled by Scrapy
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.