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, such as:

- 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 :mod:`~xml.etree.ElementTree`. (lxml is not part of the Python standard library.)

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 16); *backlink***
>
> Unknown interpreted text role "mod".

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.

> **Note**
>
> Scrapy Selectors is a thin wrapper around parsel library; the purpose of this wrapper is to provide better integration with Scrapy Response objects.
>
> parsel is a stand-alone web scraping library which can be used without Scrapy. It uses lxml library under the hood, and implements an easy API on top of lxml API. It means Scrapy selectors are very similar in speed and parsing accuracy to lxml.

Using selectors

Constructing selectors

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 49)**
>
> Unknown directive type "highlight".
>
> ```
> .. highlight:: python
> ```

Response objects expose a :class:`~scrapy.Selector` instance on `.selector` attribute:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 51); *backlink***
>
> Unknown interpreted text role "class".

```
>>> response.selector.xpath('//span/text()').get()
'good'
```

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

```
>>> response.xpath('//span/text()').get()
'good'
>>> response.css('span::text').get()
'good'
```

Scrapy selectors are instances of :class:`~scrapy.Selector` class constructed by passing either :class:`~scrapy.http.TextResponse` object or markup as a string (in `text` argument).

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 65); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 65); *backlink***

Unknown interpreted text role "class".

Usually there is no need to construct Scrapy selectors manually: `response` object is available in Spider callbacks, so in most cases it is more convenient to use `response.css()` and `response.xpath()` shortcuts. By using `response.selector` or one of these shortcuts you can also ensure the response body is parsed only once.

But if required, it is possible to use `Selector` directly. Constructing from text:

```
>>> from scrapy.selector import Selector
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').get()
'good'
```

Constructing from response - :class:`~scrapy.http.HtmlResponse` is one of :class:`~scrapy.http.TextResponse` subclasses:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 83); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 83); *backlink*

Unknown interpreted text role "class".

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
>>> response = HtmlResponse(url='http://example.com', body=body)
>>> Selector(response=response).xpath('//span/text()').get()
'good'
```

`Selector` automatically chooses the best parsing rules (XML vs HTML) based on input type.

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:

> https://docs.scrapy.org/en/latest/_static/selectors-sample1.html

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 108)

Unknown directive type "literalinclude".

```
.. literalinclude:: ../_static/selectors-sample1.html
   :language: html
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 111)

Unknown directive type "highlight".

```
.. highlight:: sh
```

First, let's open the shell:

```
scrapy shell https://docs.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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 122)

Unknown directive type "highlight".

```
.. highlight:: python
```

So, by looking at the :ref:`HTML code <topics-selectors-htmlcode>` of that page, let's construct an XPath for selecting the text inside the title tag:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 124); *backlink*

Unknown interpreted text role "ref".

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

To actually extract the textual data, you must call the selector `.get()` or `.getall()` methods, as follows:

```
>>> response.xpath('//title/text()').getall()
['Example website']
>>> response.xpath('//title/text()').get()
'Example website'
```

`.get()` always returns a single result; if there are several matches, content of a first match is returned; if there are no matches, None is returned. `.getall()` returns a list with all results.

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

```
>>> response.css('title::text').get()
'Example website'
```

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 148); *backlink*

Unknown interpreted text role "class".

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

If you want to extract only the first matched element, you can call the selector `.get()` (or its alias `.extract_first()` commonly used in previous Scrapy versions):

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

It returns `None` if no element was found:

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

Instead of using e.g. `'@src'` XPath it is possible to query for attributes using `.attrib` property of a :class:`~scrapy.Selector`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 177); *backlink*

Unknown interpreted text role "class".

```
>>> [img.attrib['src'] for img in response.css('img')]
['image1_thumb.jpg',
 'image2_thumb.jpg',
 'image3_thumb.jpg',
 'image4_thumb.jpg',
 'image5_thumb.jpg']
```

As a shortcut, `.attrib` is also available on SelectorList directly; it returns attributes for the first matching element:

```
>>> response.css('img').attrib['src']
'image1_thumb.jpg'
```

This is most useful when only a single result is expected, e.g. when selecting by id, or selecting unique elements on a web page:

```
>>> response.css('base').attrib['href']
'http://example.com/'
```

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

```
>>> response.xpath('//base/@href').get()
'http://example.com/'
```

```
>>> response.css('base::attr(href)').get()
'http://example.com/'
```

```
>>> response.css('base').attrib['href']
'http://example.com/'
```

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

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

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

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

Extensions to CSS Selectors

Per W3C standards, CSS selectors do not support selecting text nodes or attribute values. But selecting these is so essential in a web scraping context that Scrapy (parsel) implements a couple of **non-standard pseudo-elements**:

- to select text nodes, use `::text`
- to select attribute values, use `::attr(name)` where *name* is the name of the attribute that you want the value of

Warning

These pseudo-elements are Scrapy-/Parsel-specific. They will most probably not work with other libraries like lxml or PyQuery.

Examples:

- `title::text` selects children text nodes of a descendant `<title>` element:

```
>>> response.css('title::text').get()
'Example website'
```

- `*::text` selects all descendant text nodes of the current selector context:

```
>>> response.css('#images *::text').getall()
['\n   ',
 'Name: My image 1 ',
 '\n   ',
 'Name: My image 2 ',
 '\n   ',
 'Name: My image 3 ',
 '\n   ',
 'Name: My image 4 ',
 '\n   ',
 'Name: My image 5 ',
 '\n  ']
```

- `foo::text` returns no results if `foo` element exists, but contains no text (i.e. text is empty):

```
>>> response.css('img::text').getall()
[]
```

This means `.css('foo::text').get()` could return None even if an element exists. Use `default=''` if you always want a string:

```
>>> response.css('img::text').get()
>>> response.css('img::text').get(default='')
''
```

- `a::attr(href)` selects the *href* attribute value of descendant links:

```
>>> response.css('a::attr(href)').getall()
['image1.html',
 'image2.html',
 'image3.html',
 'image4.html',
 'image5.html']
```

> **Note**
>
> See also: :ref:`selecting-attributes`.
>
> > **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 304); *backlink***
> >
> > Unknown interpreted text role "ref".

> **Note**
>
> You cannot chain these pseudo-elements. But in practice it would not make much sense: text nodes do not have attributes, and attribute values are string values already and do not have children nodes.

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.getall()
['<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>']
```

```
>>> for index, link in enumerate(links):
...     href_xpath = link.xpath('@href').get()
...     img_xpath = link.xpath('img/@src').get()
...     print(f'Link number {index} points to url {href_xpath!r} and image {img_xpath!r}')
Link number 0 points to url 'image1.html' and image 'image1_thumb.jpg'
Link number 1 points to url 'image2.html' and image 'image2_thumb.jpg'
Link number 2 points to url 'image3.html' and image 'image3_thumb.jpg'
Link number 3 points to url 'image4.html' and image 'image4_thumb.jpg'
Link number 4 points to url 'image5.html' and image 'image5_thumb.jpg'
```

Selecting element attributes

There are several ways to get a value of an attribute. First, one can use XPath syntax:

```
>>> response.xpath("//a/@href").getall()
['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
```

XPath syntax has a few advantages: it is a standard XPath feature, and `@attributes` can be used in other parts of an XPath expression - e.g. it is possible to filter by attribute value.

Scrapy also provides an extension to CSS selectors (`::attr(...)`) which allows to get attribute values:

```
>>> response.css('a::attr(href)').getall()
['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
```

In addition to that, there is a `.attrib` property of Selector. You can use it if you prefer to lookup attributes in Python code, without using XPaths or CSS extensions:

```
>>> [a.attrib['href'] for a in response.css('a')]
['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
```

This property is also available on SelectorList; it returns a dictionary with attributes of a first matching element. It is convenient to use when a selector is expected to give a single result (e.g. when selecting by element ID, or when selecting an unique element on a page):

```
>>> response.css('base').attrib
{'href': 'http://example.com/'}
>>> response.css('base').attrib['href']
'http://example.com/'
```

`.attrib` property of an empty SelectorList is empty:

```
>>> response.css('foo').attrib
```

```
{}
```

Using selectors with regular expressions

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

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 386); *backlink***
>
> Unknown interpreted text role "class".

Here's an example used to extract image names from the :ref:`HTML code <topics-selectors-htmlcode>` above:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 391); *backlink***
>
> Unknown interpreted text role "ref".

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

There's an additional helper reciprocating `.get()` (and its alias `.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*(.*)')
'My image 1'
```

extract() and extract_first()

If you're a long-time Scrapy user, you're probably familiar with `.extract()` and `.extract_first()` selector methods. Many blog posts and tutorials are using them as well. These methods are still supported by Scrapy, there are **no plans** to deprecate them.

However, Scrapy usage docs are now written using `.get()` and `.getall()` methods. We feel that these new methods result in a more concise and readable code.

The following examples show how these methods map to each other.

1. `SelectorList.get()` is the same as `SelectorList.extract_first()`:

   ```
   >>> response.css('a::attr(href)').get()
   'image1.html'
   >>> response.css('a::attr(href)').extract_first()
   'image1.html'
   ```

2. `SelectorList.getall()` is the same as `SelectorList.extract()`:

   ```
   >>> response.css('a::attr(href)').getall()
   ['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
   >>> response.css('a::attr(href)').extract()
   ['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
   ```

3. `Selector.get()` is the same as `Selector.extract()`:

   ```
   >>> response.css('a::attr(href)')[0].get()
   'image1.html'
   >>> response.css('a::attr(href)')[0].extract()
   'image1.html'
   ```

4. For consistency, there is also `Selector.getall()`, which returns a list:

   ```
   >>> response.css('a::attr(href)')[0].getall()
   ['image1.html']
   ```

So, the main difference is that output of `.get()` and `.getall()` methods is more predictable: `.get()` always returns a single result, `.getall()` always returns a list of all extracted results. With `.extract()` method it was not always obvious if a result is a list or not; to get a single result either `.extract()` or `.extract_first()` should be called.

Working with XPaths

Here are some tips which may help you to use XPath with Scrapy selectors effectively. If you are not much familiar with XPath yet, you may want to take a look first at this XPath tutorial.

> **Note**

Some of the tips are based on this post from Zyte's blog.

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.get())
```

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.get())
```

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

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

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

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></di
>>> sel.css('.shout').xpath('./time/@datetime').getall()
['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.

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).getall()
```

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

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

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

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

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

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

```
['<li>1</li>', '<li>4</li>']
```

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

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

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()').getall() # take a peek at the node-set
['Click here to go to the ', 'Next Page']
>>> sel.xpath("string(//a[1]//text())").getall() # convert it to string
['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]").getall() # select the first node
['<a href="#">Click here to go to the <strong>Next Page</strong></a>']
>>> sel.xpath("string(//a[1])").getall() # convert it to string
['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')]").getall()
[]
```

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

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

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').get()
'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).get()
'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.

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 :meth:`Selector.remove_namespaces` method for that.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 656); *backlink*

Unknown interpreted text role "meth".

Let's show an example that illustrates this with the Python Insider blog atom feed.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 663)

Unknown directive type "highlight".

```
.. highlight:: sh
```

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

```
$ scrapy shell https://feeds.feedburner.com/PythonInsider
```

This is how the file starts:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet ...
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:openSearch="http://a9.com/-/spec/opensearchrss/1.0/"
      xmlns:blogger="http://schemas.google.com/blogger/2008"
      xmlns:georss="http://www.georss.org/georss"
      xmlns:gd="http://schemas.google.com/g/2005"
      xmlns:thr="http://purl.org/syndication/thread/1.0"
      xmlns:feedburner="http://rssnamespace.org/feedburner/ext/1.0">
  ...
```

You can see several namespace declarations including a default "http://www.w3.org/2005/Atom" and another one using the "gd:" prefix for "http://schemas.google.com/g/2005".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 686)

Unknown directive type "highlight".

```
.. highlight:: python
```

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 :meth:`Selector.remove_namespaces` method, all nodes can be accessed directly by their names:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 694); *backlink*

Unknown interpreted text role "meth".

```
>>> response.selector.remove_namespaces()
>>> response.xpath("//link")
[<Selector xpath='//link' data='<link rel="alternate" type="text/html" h'>,
    <Selector xpath='//link' data='<link rel="next" type="application/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 by default 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.

Using EXSLT extensions

Being built atop lxml, Scrapy 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 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').getall()
['link1.html', 'link2.html', 'link3.html', 'link4.html', 'link5.html']
>>> sel.xpath('//li[re:test(@class, "item-\d$")]//@href').getall()
['link1.html', 'link2.html', 'link4.html', '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 https://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').getall())
...     props = scope.xpath('''
...                 set:difference(./descendant::*/@itemprop,
...                                .//*[@itemscope]/*/@itemprop)''')
...     print(f"    properties: {props.getall()}")
...     print("")

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

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

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

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

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

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

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

Other XPath extensions

Scrapy selectors also provide a sorely missed XPath extension function `has-class` that returns `True` for nodes that have all of the specified HTML classes.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 866)

Unknown directive type "highlight".

```
.. highlight:: html
```

For the following HTML:

```
<p class="foo bar-baz">First</p>
<p class="foo">Second</p>
<p class="bar">Third</p>
<p>Fourth</p>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 875)

Unknown directive type "highlight".

```
.. highlight:: python
```

You can use it like this:

```
>>> response.xpath('//p[has-class("foo")]')
[<Selector xpath='//p[has-class("foo")]' data='<p class="foo bar-baz">First</p>'>,
 <Selector xpath='//p[has-class("foo")]' data='<p class="foo">Second</p>'>]
>>> response.xpath('//p[has-class("foo", "bar-baz")]')
[<Selector xpath='//p[has-class("foo", "bar-baz")]' data='<p class="foo bar-baz">First</p>'>]
>>> response.xpath('//p[has-class("foo", "bar")]')
[]
```

So XPath `//p[has-class("foo", "bar-baz")]` is roughly equivalent to CSS `p.foo.bar-baz`. Please note, that it is slower in most of the cases, because it's a pure-Python function that's invoked for every node in question whereas the CSS lookup is translated into XPath and thus runs more efficiently, so performance-wise its uses are limited to situations that are not easily described with CSS selectors.

Parsel also simplifies adding your own XPath extensions.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 896)

Unknown directive type "autofunction".

```
.. autofunction:: parsel.xpathfuncs.set_xpathfunc
```

Built-in Selectors reference

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 904)

Unknown directive type "module".

```
.. module:: scrapy.selector
   :synopsis: Selector class
```

Selector objects

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 910)

Unknown directive type "autoclass".

```
.. autoclass:: Selector

  .. automethod:: xpath

      .. note::

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

  .. automethod:: css

      .. note::

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

  .. automethod:: get

     See also: :ref:`old-extraction-api`

  .. autoattribute:: attrib

     See also: :ref:`selecting-attributes`.

  .. automethod:: re

  .. automethod:: re_first

  .. automethod:: register_namespace

  .. automethod:: remove_namespaces

  .. automethod:: __bool__

  .. automethod:: getall

     This method is added to Selector for consistency; it is more useful
     with SelectorList. See also: :ref:`old-extraction-api`
```

SelectorList objects

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 950)

Unknown directive type "autoclass".

```
.. autoclass:: SelectorList

   .. automethod:: xpath

   .. automethod:: css

   .. automethod:: getall

      See also: :ref:`old-extraction-api`

   .. automethod:: get

      See also: :ref:`old-extraction-api`

   .. automethod:: re

   .. automethod:: re_first

   .. autoattribute:: attrib

      See also: :ref:`selecting-attributes`.
```

Examples

Selector examples on HTML response

Here are some :class:`Selector` examples to illustrate several concepts. In all cases, we assume there is already a :class:`Selector` instantiated with a :class:`~scrapy.http.HtmlResponse` object like this:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 982**); ***backlink***

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 982**); ***backlink***

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 982**); ***backlink***

Unknown interpreted text role "class".

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

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

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 988**); ***backlink***

   Unknown interpreted text role "class".

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 988**); ***backlink***

   Unknown interpreted text role "class".

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

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

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

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

   ```
   for node in sel.xpath("//p"):
       print(node.attrib['class'])
   ```

Selector examples on XML response

Here are some examples to illustrate concepts for :class:`Selector` objects instantiated with an :class:`~scrapy.http.XmlResponse` object:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 1010**); ***backlink***

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 1010**); ***backlink***

Unknown interpreted text role "class".

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

1. Select all `<product>` elements from an XML response body, returning a list of :class:`Selector` objects (i.e. a :class:`SelectorList` object):

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, **line 1015**); ***backlink***

   Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scrapy-master\docs\topics\[scrapy-master][docs][topics]selectors.rst`, line 1015); *backlink*

Unknown interpreted text role "class".

```
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").getall()
```