Parsel Documentation

Release 1.11.0

Scrapy Project

Jan 29, 2026

CONTENTS

Parsel is a BSD-licensed Python library to extract data from HTML, JSON, and XML documents.

It supports:

- CSS and XPath expressions for HTML and XML documents
- JMESPath expressions for JSON documents
- Regular expressions

Find the Parsel online documentation at https://parsel.readthedocs.org.

Example (open online demo):

```
>>> from parsel import Selector
>>> text = """
... <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>
...         <script type="application/json">{"a": ["b", "c"]}</script>
...     </body>
... </html>"""
>>> selector = Selector(text=text)
>>> selector.css("h1::text").get()
'Hello, Parsel!'
>>> selector.xpath("//h1/text()").re(r"\w+")
['Hello', 'Parsel']
>>> for li in selector.css("ul > li"):
...     print(li.xpath(".//@href").get())
...
http://example.com
http://scrapy.org
>>> selector.css("script::text").jmespath("a").get()
'b'
>>> selector.css("script::text").jmespath("a").getall()
['b', 'c']
```

CHAPTER

ONE

PARSEL DOCUMENTATION CONTENTS

Contents:

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
```

1.2 Usage

Create a *Selector* object for your input text.

For HTML or XML, use CSS or XPath expressions to select data:

```
>>> from parsel import Selector
>>> html_text = "<html><body><h1>Hello, Parsel!</h1></body></html>"
>>> html_selector = Selector(text=html_text)
>>> html_selector.css('h1')
[<Selector query='descendant-or-self::h1' data='<h1>Hello, Parsel!</h1>'>]
>>> html_selector.xpath('//h1')  # the same, but now with XPath
[<Selector query='//h1' data='<h1>Hello, Parsel!</h1>'>]
```

For JSON, use JMESPath expressions to select data:

```
>>> json_text = '{"title":"Hello, Parsel!"}'
>>> json_selector = Selector(text=json_text)
>>> json_selector.jmespath('title')
[<Selector query='title' data='Hello, Parsel!'>]
```

And extract data from those elements:

```
>>> html_selector.xpath('//h1/text()').get()
'Hello, Parsel!'
>>> json_selector.jmespath('title').getall()
['Hello, Parsel!']
```

1.2.1 Learning expression languages

CSS is a language for applying styles to HTML documents. It defines selectors to associate those styles with specific HTML elements. Resources to learn CSS selectors include:

- CSS selectors in the MDN
- XPath/CSS Equivalents in Wikibooks

Parsel support for CSS selectors comes from cssselect, so read about CSS selectors supported by cssselect.

XPath is a language for selecting nodes in XML documents, which can also be used with HTML. Resources to learn XPath include:

- XPath Tutorial in W3Schools
- XPath cheatsheet

For HTML and XML input, you can use either CSS or XPath. CSS is usually more readable, but some things can only be done with XPath.

JMESPath allows you to declaratively specify how to extract elements from a JSON document. Resources to learn JMESPath include:

- JMESPath Tutorial
- JMESPath Specification

1.2.2 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:

> https://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 = 'https://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 query='//title/text()' data='Example website'>]
```

You can also ask the same thing using CSS instead:

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

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

```
>>> selector.xpath('//title/text()').getall()
['Example website']
>>> selector.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:

```
>>> selector.css('title::text').get()
'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').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 parsel versions):

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

It returns `None` if no element was found:

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

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

```
>>> [img.attrib['src'] for img in selector.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:

```
>>> selector.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:

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

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

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

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

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

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

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

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

>>> selector.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 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:

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

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

```
>>> selector.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  ']
```

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

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

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.

Note

See also: *Selecting element attributes*.

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.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):
...     args = (index, link.xpath('@href').get(), link.xpath('img/@src').get())
...     print('Link number %d points to url %r and image %r' % args)
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:

```
>>> selector.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.

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

```
>>> selector.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 selector.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):

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

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

```
>>> selector.css('foo').attrib
{}
```

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 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*(.*)')
['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:

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

Removing elements

If for any reason you need to remove elements based on a Selector or a SelectorList, you can do it with the `drop()` method, available for both classes.

> **Warning**
>
> this is a destructive action and cannot be undone. The original content of the selector is removed from the elements tree. This could be useful when trying to reduce the memory footprint of Responses.

Example removing an ad from a blog post:

```
>>> from parsel import Selector
>>> doc = """
... <article>
...     <div class="row">Content paragraph...</div>
...     <div class="row">
...         <div class="ad">
...             Ad content...
...             <a href="http://...">Link</a>
...         </div>
...     </div>
...     <div class="row">More content...</div>
... </article>
... """
>>> sel = Selector(text=doc)
>>> sel.xpath('//div/text()').getall()
['Content paragraph...', '\n        ', '\n            Ad content...\n            ', '\n␣
↪      ', '\n    ', 'More content...']
>>> sel.xpath('//div[@class="ad"]').drop()
>>> sel.xpath('//div//text()').getall()
['Content paragraph...', 'More content...']
```

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 = """
... <div>
```

(continues on next page)

(continued from previous page)

```
...     <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').getall()
['link1.html', 'link2.html', 'link3.html', 'link4.html', 'link5.html']
>>> sel.xpath(r'//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 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">
```

(continues on next page)

(continued from previous page)

```
...     <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("    properties: %s" % (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']
```

(continues on next page)

(continued from previous page)

```
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

Parsel also defines a sorely missed XPath extension function `has-class` that returns `True` for nodes that have all of the specified HTML classes:

```
>>> from parsel import Selector
>>> sel = Selector("""
...         <p class="foo bar-baz">First</p>
...         <p class="foo">Second</p>
...         <p class="bar">Third</p>
...         <p>Fourth</p>
... """)
...
>>> sel.xpath('//p[has-class("foo")]')
[<Selector query='//p[has-class("foo")]' data='<p class="foo bar-baz">First</p>'>,
 <Selector query='//p[has-class("foo")]' data='<p class="foo">Second</p>'>]
>>> sel.xpath('//p[has-class("foo", "bar-baz")]')
[<Selector query='//p[has-class("foo", "bar-baz")]' data='<p class="foo bar-baz">First</
↪p>'>]
>>> sel.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.

`parsel.xpathfuncs.set_xpathfunc`(*fname: str*, *func: Callable | None*) → None

Register a custom extension function to use in XPath expressions.

The function `func` registered under `fname` identifier will be called for every matching node, being passed a `context` parameter as well as any parameters passed from the corresponding XPath expression.

If `func` is `None`, the extension function will be removed.

See more in lxml documentation.

Some XPath tips

Here are some tips that you may find useful when using XPath with Parsel, based on this post from Zyte'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()').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>']
```

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>
```

(continues on next page)

(continued from previous page)

```
...         <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>']
```

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></div>')
>>> 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 how script and style tags differ from other tags

Following the standard, the contents of `script` and `style` elements are parsed as plain text.

This means that XML-like structures found within them, including comments, are all treated as part of the element text, and not as separate nodes.

For example:

```
>>> from parsel import Selector
>>> selector = Selector(text="""
...     <script>
...         text
...         <!-- comment -->
...         <br/>
...     </script>
...     <style>
...         text
...         <!-- comment -->
...         <br/>
...     </style>
...     <div>
...         text
...         <!-- comment -->
...         <br/>
...     </div>""")
>>> for tag in selector.xpath('//*[contains(text(), "text")]'):
...     print(tag.xpath('name()').get())
...     print('    Text: ' + (tag.xpath('text()').get() or ''))
...     print('    Comment: ' + (tag.xpath('comment()').get() or ''))
...     print('    Children: ' + ''.join(tag.xpath('*').getall()))
...
script
    Text:
        text
        <!-- comment -->
        <br/>

    Comment:
    Children:
style
    Text:
        text
        <!-- comment -->
        <br/>

    Comment:
    Children:
div
    Text:
        text

    Comment: <!-- comment -->
    Children: <br>
```

extract() and extract_first()

If you're a long-time parsel (or Scrapy) user, you're probably familiar with `.extract()` and `.extract_first()` selector methods. These methods are still supported by parsel, there are no plans to deprecate them.

However, `parsel` usage docs are now written using `.get()` and `.getall()` methods. We feel that these new methods result in 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()`:

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

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

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

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

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

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

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

With the `.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()` needed to be called, depending whether you had a `Selector` or `SelectorList`.

So, the main difference is that the outputs of `.get()` and `.getall()` are more predictable: `.get()` always returns a single result, `.getall()` always returns a list of all extracted results.

Using CSS selectors in multi-root documents

Some webpages may have multiple root elements. It can happen, for example, when a webpage has broken code, such as missing closing tags.

You can use XPath to determine if a page has multiple root elements:

```
>>> len(selector.xpath('/*')) > 1
True
```

CSS selectors only work on the first root element, because the first root element is always used as the starting current element, and CSS selectors do not allow selecting parent elements (XPath's `..`) or elements relative to the document root (XPath's `/`).

If you want to use a CSS selector that takes into account all root elements, you need to precede your CSS query by an XPath query that reaches all root elements:

```
selector.xpath('/*').css('<your CSS selector>')
```

1.2.3 Command-Line Interface Tools

There are third-party tools that allow using Parsel from the command line:

- Parsel CLI allows applying Parsel selectors to the standard input. For example, you can apply a Parsel selector to the output of cURL.
- parselcli provides an interactive shell that allows applying Parsel selectors to a remote URL or a local file.

1.2.4 Examples

Working on HTML

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 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'])
   ```

Working on XML (and namespaces)

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").getall()
   ```

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 the Python Insider 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://feeds.feedburner.com/PythonInsider').text
>>> sel = Selector(text=text, type='xml')
```

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">
  ...
</feed>
```

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".

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 query='//link' data='<link rel="alternate" type="text/html...'>,
 <Selector query='//link' data='<link rel="next" type="application/at...'>,
 ...]
```

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.

Ad-hoc namespaces references

Selector objects also allow passing namespaces references along with the query, through a `namespaces` argument, with the prefixes you declare being used in your XPath or CSS query.

Let's use the same Python Insider Atom feed:

```
>>> import requests
>>> from parsel import Selector
>>> text = requests.get('https://feeds.feedburner.com/PythonInsider').text
>>> sel = Selector(text=text, type='xml')
```

And try to select the links again, now using an "atom:" prefix for the "link" node test:

```
>>> sel.xpath("//atom:link", namespaces={"atom": "http://www.w3.org/2005/Atom"})
[<Selector query='//atom:link' data='<link xmlns="http://www.w3.org/2005/A...'>,
 <Selector query='//atom:link' data='<link xmlns="http://www.w3.org/2005/A...'>,
 ...]
```

You can pass several namespaces (here we're using shorter 1-letter prefixes):

```
>>> sel.xpath("//a:entry/a:author/g:image/@src",
...           namespaces={"a": "http://www.w3.org/2005/Atom",
...                       "g": "http://schemas.google.com/g/2005"}).getall()
['https://img1.blogblog.com/img/b16-rounded.gif',
 'https://img1.blogblog.com/img/b16-rounded.gif',
 ...]
```

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 normalized string-value:

```
>>> str_to_match = "Name: My image 3"
>>> selector.xpath('//a[normalize-space(.)=$match]',
...                match=str_to_match).get()
'<a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>'
```

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.

Here's another example using a position range passed as two integers:

```
>>> start, stop = 2, 4
>>> selector.xpath('//a[position()>=$_from and position()<=$_to]',
...                _from=start, _to=stop).getall()
['<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>']
```

Named variables can be useful when strings need to be escaped for single or double quotes characters. The example below would be a bit tricky to get right (or legible) without a variable reference:

```
>>> html = '''<html>
... <body>
...   <p>He said: "I don't know why, but I like mixing single and double quotes!"</p>
... </body>
... </html>'''
>>> selector = Selector(text=html)
>>>
>>> selector.xpath('//p[contains(., $mystring)]',
...                mystring='''He said: "I don't know''').get()
'<p>He said: "I don\'t know why, but I like mixing single and double quotes!"</p>'
```

Converting CSS to XPath

parsel.**css2xpath**(*query: str*) → str

Return translated XPath version of a given CSS query

When you're using an API that only accepts XPath expressions, it's sometimes useful to convert CSS to XPath. This allows you to take advantage of the conciseness of CSS to query elements by classes and the easeness of manipulating XPath expressions at the same time.

On those occasions, use the function *css2xpath()*:

```
>>> from parsel import css2xpath
>>> css2xpath('h1.title')
"descendant-or-self::h1[@class and contains(concat(' ', normalize-space(@class), ' '), '␣
↪title ')]"
>>> css2xpath('.profile-data') + '//h2'
"descendant-or-self::*[@class and contains(concat(' ', normalize-space(@class), ' '), '␣
↪profile-data ')]//h2"
```

As you can see from the examples above, it returns the translated CSS query into an XPath expression as a string, which you can use as-is or combine to build a more complex expression, before feeding to a function expecting XPath.

1.2.5 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.

1.3 API reference

1.3.1 parsel.csstranslator

class parsel.csstranslator.**GenericTranslator**

Bases: *TranslatorMixin*, GenericTranslator

css_to_xpath(*css: str*, *prefix: str = 'descendant-or-self::'*) → str

Translate a *group of selectors* to XPath.

Pseudo-elements are not supported here since XPath only knows about "real" elements.

Parameters

- **css** – A *group of selectors* as a string.
- **prefix** – This string is prepended to the XPath expression for each selector. The default makes selectors scoped to the context node's subtree.

Raises

SelectorSyntaxError on invalid selectors, ExpressionError on unknown/unsupported selectors, including pseudo-elements.

Returns
The equivalent XPath 1.0 expression as a string.

class parsel.csstranslator.**HTMLTranslator**(*xhtml: bool = False*)

Bases: *TranslatorMixin*, HTMLTranslator

css_to_xpath(*css: str*, *prefix: str = 'descendant-or-self::'*) → str

Translate a *group of selectors* to XPath.

Pseudo-elements are not supported here since XPath only knows about "real" elements.

Parameters

- **css** – A *group of selectors* as a string.
- **prefix** – This string is prepended to the XPath expression for each selector. The default makes selectors scoped to the context node's subtree.

Raises
SelectorSyntaxError on invalid selectors, ExpressionError on unknown/unsupported selectors, including pseudo-elements.

Returns
The equivalent XPath 1.0 expression as a string.

class parsel.csstranslator.**TranslatorMixin**

Bases: object

This mixin adds support to CSS pseudo elements via dynamic dispatch.

Currently supported pseudo-elements are ::text and ::attr(ATTR_NAME).

xpath_attr_functional_pseudo_element(*xpath: XPathExpr*, *function: FunctionalPseudoElement*) → *XPathExpr*

Support selecting attribute values using ::attr() pseudo-element

xpath_element(*selector: Element*) → *XPathExpr*

xpath_pseudo_element(*xpath: XPathExpr*, *pseudo_element: FunctionalPseudoElement | str*) → XPathExpr

Dispatch method that transforms XPath to support pseudo-element

xpath_text_simple_pseudo_element(*xpath: XPathExpr*) → *XPathExpr*

Support selecting text nodes using ::text pseudo-element

class parsel.csstranslator.**TranslatorProtocol**(**args*, ***kwargs*)

Bases: Protocol

css_to_xpath(*css: str*, *prefix: str = Ellipsis*) → str

xpath_element(*selector: Element*) → XPathExpr

class parsel.csstranslator.**XPathExpr**(*path: str = ''*, *element: str = '*'*, *condition: str = ''*, *star_prefix: bool = False*)

Bases: XPathExpr

attribute: str | None = None

classmethod from_xpath(*xpath: OriginalXPathExpr*, *textnode: bool = False*, *attribute: str | None = None*) → Self

join(*combiner: str*, *other: OriginalXPathExpr*, **args: Any*, ***kwargs: Any*) → Self

textnode: bool = False

parsel.csstranslator.**css2xpath**(*query: str*) → str

Return translated XPath version of a given CSS query

1.3.2 parsel.selector

XPath and JMESPath selectors based on the lxml and jmespath Python packages.

class parsel.selector.**CTGroupValue**

Bases: TypedDict

exception parsel.selector.**CannotDropElementWithoutParent**

Bases: *CannotRemoveElementWithoutParent*

exception parsel.selector.**CannotRemoveElementWithoutParent**

Bases: Exception

exception parsel.selector.**CannotRemoveElementWithoutRoot**

Bases: Exception

class parsel.selector.**SafeXMLParser**(**args: Any*, ***kwargs: Any*)

Bases: XMLParser

class parsel.selector.**Selector**(*text: str | None = None*, *type: str | None = None*, *body: bytes | bytearray = b''*, *encoding: str = 'utf-8'*, *namespaces: Mapping[str, str] | None=None*, *root: Any | None = <object object>*, *base_url: str | None = None*, *_expr: str | None = None*, *huge_tree: bool = True*)

Bases: object

Wrapper for input data in HTML, JSON, or XML format, that allows selecting parts of it using selection expressions.

You can write selection expressions in CSS or XPath for HTML and XML inputs, or in JMESPath for JSON inputs.

`text` is an `str` object.

`body` is a `bytes` object. It can be used together with the `encoding` argument instead of the `text` argument.

`type` defines the selector type. It can be `"html"` (default), `"json"`, or `"xml"`.

`base_url` allows setting a URL for the document. This is needed when looking up external entities with relative paths. See the documentation for `lxml.etree.fromstring()` for more information.

`huge_tree` controls the lxml/libxml2 feature that forbids parsing certain large documents to protect from possible memory exhaustion. The argument is `True` by default if the installed lxml version supports it, which disables the protection to allow parsing such documents. Set it to `False` if you want to enable the protection. See this lxml FAQ entry for more information.

property attrib: dict[str, str]

Return the attributes dictionary for underlying element. For JSON selectors, return an empty dict.

body

css(*query: str*) → *SelectorList*[Self]

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.

drop() → None

Drop matched nodes from the parent element.

extract() → Any

Serialize and return the matched nodes.

For HTML and XML, the result is always a string, and percent-encoded content is unquoted.

get() → Any

Serialize and return the matched nodes.

For HTML and XML, the result is always a string, and percent-encoded content is unquoted.

getall() → list[str]

Serialize and return the matched node in a 1-element list of strings.

jmespath(*query: str*, ***kwargs: Any*) → *SelectorList*[Self]

Find objects matching the JMESPath `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 JMESPath query to apply.

Any additional named arguments are passed to the underlying `jmespath.search` call, e.g.:

```
selector.jmespath('author.name', options=jmespath.Options(dict_cls=collections.
↪OrderedDict))
```

namespaces

re(*regex: str | Pattern[str]*, *replace_entities: bool = True*) → list[str]

Apply the given regex and return a list of 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)`.

By default, character entity references are replaced by their corresponding character (except for `&` and `<`). Passing `replace_entities` as `False` switches off these replacements.

re_first(*regex: str | Pattern[str]*, *default: None = None*, *replace_entities: bool = True*) → str | None

re_first(*regex: str | Pattern[str]*, *default: str*, *replace_entities: bool = True*) → str

Apply the given regex and return the first string which matches. If there is no match, return the default value (`None` if the argument is not provided).

By default, character entity references are replaced by their corresponding character (except for `&` and `<`). Passing `replace_entities` as `False` switches off these replacements.

register_namespace(*prefix: str*, *uri: str*) → None

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 *Working on XML (and namespaces)*.

remove_namespaces() → None

Remove all namespaces, allowing to traverse the document using namespace-less xpaths. See *Removing namespaces*. For JSON selectors, this method does nothing.

root: Any

selectorlist_cls

alias of *SelectorList*[Selector]

type

xpath(*query: str*, *namespaces: Mapping[str, str] | None = None*, ***kwargs: Any*) → *SelectorList*[Self]

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.

`namespaces` is an optional `prefix: namespace-uri` mapping (dict) for additional prefixes to those registered with `register_namespace(prefix, uri)`. Contrary to `register_namespace()`, these prefixes are not saved for future calls.

Any additional named arguments can be used to pass values for XPath variables in the XPath expression, e.g.:

```
selector.xpath('//a[href=$url]', url="http://www.example.com")
```

class parsel.selector.**SelectorList**(*iterable=()*, */ (Positional-only parameter separator (PEP 570))*)

Bases: list[_SelectorType]

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

property attrib: Mapping[str, str]

Return the attributes dictionary for the first element. If the list is empty, return an empty dict.

css(*query: str*) → *SelectorList*[_SelectorType]

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()*

drop() → None

Drop matched nodes from the parent for each element in this list.

extract() → list[str]

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

extract_first(*default: str | None = None*) → Any

Return the result of `.get()` for the first element in this list. If the list is empty, return the default value.

get(*default: None = None*) → str | None

get(*default: str*) → str

Return the result of `.get()` for the first element in this list. If the list is empty, return the default value.

getall() → list[str]

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

jmespath(*query: str*, ***kwargs: Any*) → *SelectorList*[_SelectorType]

Call the `.jmespath()` 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.jmespath()*.

Any additional named arguments are passed to the underlying `jmespath.search` call, e.g.:

```
selector.jmespath('author.name', options=jmespath.Options(dict_cls=collections.
↪OrderedDict))
```

re(*regex: str | Pattern[str]*, *replace_entities: bool = True*) → list[str]

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

By default, character entity references are replaced by their corresponding character (except for `&` and `<`. Passing `replace_entities` as `False` switches off these replacements.

re_first(*regex: str | Pattern[str]*, *default: None = None*, *replace_entities: bool = True*) → str | None

re_first(*regex: str | Pattern[str]*, *default: str*, *replace_entities: bool = True*) → str

Call the `.re()` method for the first element in this list and return the result in an string. If the list is empty or the regex doesn't match anything, return the default value (`None` if the argument is not provided).

By default, character entity references are replaced by their corresponding character (except for `&` and `<`. Passing `replace_entities` as `False` switches off these replacements.

xpath(*xpath: str*, *namespaces: Mapping[str, str] | None = None*, ***kwargs: Any*) → *SelectorList*[_SelectorType]

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

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

`namespaces` is an optional `prefix: namespace-uri` mapping (dict) for additional prefixes to those registered with `register_namespace(prefix, uri)`. Contrary to `register_namespace()`, these prefixes are not saved for future calls.

Any additional named arguments can be used to pass values for XPath variables in the XPath expression, e.g.:

```
selector.xpath('//a[href=$url]', url="http://www.example.com")
```

parsel.selector.**create_root_node**(*text: str*, *parser_cls: type[XMLParser | HTMLParser]*, *base_url: str | None = None*, *huge_tree: bool = True*, *body: bytes = b''*, *encoding: str = 'utf-8'*) → _Element

Create root node for text using given parser class.

1.3.3 parsel.utils

parsel.utils.**extract_regex**(*regex: str | Pattern[str]*, *text: str*, *replace_entities: bool = True*) → list[str]

Extract a list of strings from the given text/encoding using the following policies: * if the regex contains a named group called "extract" that will be returned * if the regex contains multiple numbered groups, all those will be returned (flattened) * if the regex doesn't contain any group the entire regex matching is returned

parsel.utils.**flatten**(*sequence*) → list

Returns a single, flat list which contains all elements retrieved from the sequence and all recursively contained sub-sequences (iterables). Examples: >>> [1, 2, [3,4], (5,6)] [1, 2, [3, 4], (5, 6)] >>> flatten([[[1,2,3], (42,None)], [4,5], [6], 7, (8,9,10)]) [1, 2, 3, 42, None, 4, 5, 6, 7, 8, 9, 10] >>> flatten(["foo", "bar"]) ['foo', 'bar'] >>> flatten(["foo", ["baz", 42], "bar"]) ['foo', 'baz', 42, 'bar']

parsel.utils.**iflatten**(*sequence*) → Iterator

Similar to `.flatten()`, but returns iterator instead Examples: >>> list(iflatten([[1, 2], (3, 4)])) [1, 2, 3, 4]

parsel.utils.**shorten**(*text: str*, *width: int*, *suffix: str = '...'*) → str

Truncate the given text to fit in the given width.

1.4 History

1.4.1 1.11.0 (2026-01-29)

- Removed support for Python 3.9 and PyPy 3.10.
- Added support for Python 3.14 and PyPy 3.11.
- The following dependencies now have a minimum supported version:
  - `lxml >= 5.1.0`
  - `packaging >= 23.0`
  - `jmespath >= 1.0.0`
- Removed `Selector.remove()` and `SelectorList.remove()`, deprecated in 1.7.0.
- The `Selector()` constructor now accepts `bytearray` values for the `body` argument in addition to `bytes`.
- `attrib` and `remove_namespaces()` no longer fail with unhandled exceptions on JSON selectors.
- Switched the build system to `hatchling`.
- CI fixes and improvements.

1.4.2 1.10.0 (2024-12-16)

- Removed support for Python 3.8.
- Added support for Python 3.13.
- Changed the default encoding name from `"utf8"` to `"utf-8"` everywhere. The former name is not supported in certain environments.
- CI fixes and improvements.

1.4.3 1.9.1 (2024-04-08)

- Removed the dependency on `pytest-runner`.
- Removed the obsolete `Makefile`.

1.4.4 1.9.0 (2024-03-14)

- Now requires `cssselect >= 1.2.0` (this minimum version was required since 1.8.0 but that wasn't properly recorded)
- Removed support for Python 3.7
- Added support for Python 3.12 and PyPy 3.10
- Fixed an exception when calling `__str__` or `__repr__` on some JSON selectors
- Code formatted with `black`
- CI fixes and improvements

1.4.5 1.8.1 (2023-04-18)

- Remove a Sphinx reference from NEWS to fix the PyPI description
- Add a `twine check` CI check to detect such problems

1.4.6 1.8.0 (2023-04-18)

- Add support for JMESPath: you can now create a selector for a JSON document and call `Selector.jmespath()`. See the documentation for more information and examples.
- Selectors can now be constructed from `bytes` (using the `body` and `encoding` arguments) instead of `str` (using the `text` argument), so that there is no internal conversion from `str` to `bytes` and the memory usage is lower.
- Typing improvements
- The `pkg_resources` module (which was absent from the requirements) is no longer used
- Documentation build fixes
- New requirements:
  - `jmespath`
  - `typing_extensions` (on Python 3.7)

1.4.7 1.7.0 (2022-11-01)

- Add PEP 561-style type information
- Support for Python 2.7, 3.5 and 3.6 is removed
- Support for Python 3.9-3.11 is added
- Very large documents (with deep nesting or long tag content) can now be parsed, and `Selector` now takes a new argument `huge_tree` to disable this
- Support for new features of cssselect 1.2.0 is added
- The `Selector.remove()` and `SelectorList.remove()` methods are deprecated and replaced with the new `Selector.drop()` and `SelectorList.drop()` methods which don't delete text after the dropped elements when used in the HTML mode.

1.4.8 1.6.0 (2020-05-07)

- Python 3.4 is no longer supported
- New `Selector.remove()` and `SelectorList.remove()` methods to remove selected elements from the parsed document tree
- Improvements to error reporting, test coverage and documentation, and code cleanup

1.4.9 1.5.2 (2019-08-09)

- `Selector.remove_namespaces` received a significant performance improvement
- The value of `data` within the printable representation of a selector (`repr(selector)`) now ends in `...` when truncated, to make the truncation obvious.
- Minor documentation improvements.

1.4.10 1.5.1 (2018-10-25)

- `has-class` XPath function handles newlines and other separators in class names properly;
- fixed parsing of HTML documents with null bytes;
- documentation improvements;
- Python 3.7 tests are run on CI; other test improvements.

1.4.11 1.5.0 (2018-07-04)

- New `Selector.attrib` and `SelectorList.attrib` properties which make it easier to get attributes of HTML elements.
- CSS selectors became faster: compilation results are cached (LRU cache is used for `css2xpath`), so there is less overhead when the same CSS expression is used several times.
- `.get()` and `.getall()` selector methods are documented and recommended over `.extract_first()` and `.extract()`.
- Various documentation tweaks and improvements.

One more change is that `.extract()` and `.extract_first()` methods are now implemented using `.get()` and `.getall()`, not the other way around, and instead of calling `Selector.extract` all other methods now call `Selector.get` internally. It can be **backwards incompatible** in case of custom Selector subclasses which override `Selector.extract` without doing the same for `Selector.get`. If you have such Selector subclass, make sure `get` method is also overridden. For example, this:

```
class MySelector(parsel.Selector):
    def extract(self):
        return super().extract() + " foo"
```

should be changed to this:

```
class MySelector(parsel.Selector):
    def get(self):
        return super().get() + " foo"
    extract = get
```

1.4.12 1.4.0 (2018-02-08)

- `Selector` and `SelectorList` can't be pickled because pickling/unpickling doesn't work for `lxml.html.HtmlElement`; parsel now raises TypeError explicitly instead of allowing pickle to silently produce wrong output. This is technically backwards-incompatible if you're using Python < 3.6.

1.4.13 1.3.1 (2017-12-28)

- Fix artifact uploads to pypi.

1.4.14 1.3.0 (2017-12-28)

- `has-class` XPath extension function;
- `parsel.xpathfuncs.set_xpathfunc` is a simplified way to register XPath extensions;
- `Selector.remove_namespaces` now removes namespace declarations;
- Python 3.3 support is dropped;

- `make htmlview` command for easier Parsel docs development.
- CI: PyPy installation is fixed; parsel now runs tests for PyPy3 as well.

1.4.15 1.2.0 (2017-05-17)

- Add `SelectorList.get` and `SelectorList.getall` methods as aliases for `SelectorList.extract_first` and `SelectorList.extract` respectively
- Add default value parameter to `SelectorList.re_first` method
- Add `Selector.re_first` method
- Add `replace_entities` argument on `.re()` and `.re_first()` to turn off replacing of character entity references
- Bug fix: detect `None` result from lxml parsing and fallback with an empty document
- Rearrange XML/HTML examples in the selectors usage docs
- Travis CI:
  - Test against Python 3.6
  - Test against PyPy using "Portable PyPy for Linux" distribution

1.4.16 1.1.0 (2016-11-22)

- Change default HTML parser to lxml.html.HTMLParser, which makes easier to use some HTML specific features
- Add css2xpath function to translate CSS to XPath
- Add support for ad-hoc namespaces declarations
- Add support for XPath variables
- Documentation improvements and updates

1.4.17 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`

1.4.18 1.0.2 (2016-04-26)

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

1.4.19 1.0.1 (2015-08-24)

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

1.4.20 1.0.0 (2015-08-22)

- Documentation fixes

1.4.21 0.9.6 (2015-08-14)

- Updated documentation
- Extended test coverage

1.4.22 0.9.5 (2015-08-11)

- Support for extending SelectorList

1.4.23 0.9.4 (2015-08-10)

- Try workaround for travis-ci/dpl#253

1.4.24 0.9.3 (2015-08-07)

- Add base_url argument

1.4.25 0.9.2 (2015-08-07)

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

1.4.26 0.9.1 (2015-08-04)

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

1.4.27 0.9.0 (2015-07-30)

- First release on PyPI.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

S

T

X