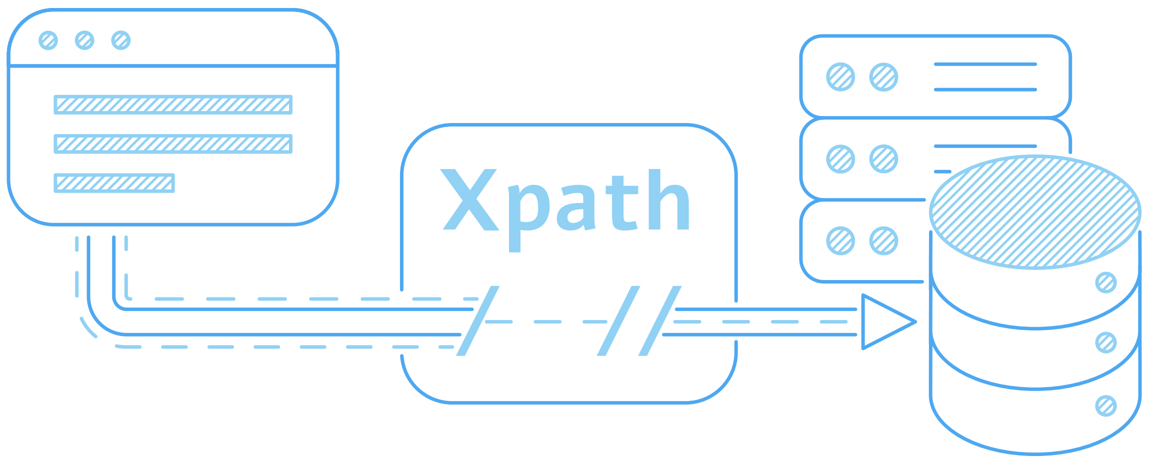
Parsing HTML with Xpath

by Bernardas Ališauskas Aug 22, 2024

#Data Parsing #Parsel #XPath #Python



When it comes to parsing web-scraped HTML content, there are multiple techniques to select the data we want.
For simple text parsing, regular expression can be used, but HTML is designed to be a machine-readable text structure. We can take advantage of this fact and use special path languages like CSS selectors and XPath to extract data in a much more efficient and reliable way!

You are probably familiar with CSS selectors from the style sheet documents (`.css`) however, XPath goes beyond that and implements full document navigation in its own unique syntax.

Parsing HTML with CSS Selectors

For parsing using CSS selectors see the CSS version of this article

In this article, we'll be taking a deep look at this unique path language and how can it be used to extract needed details from modern, complex HTML documents. We'll start with a quick introduction and expression cheatsheet and explore concepts using an interactive XPath tester.

Finally, we'll wrap up by covering XPath implementations in various programming languages and some common idioms and tips when it comes to XPath in web scraping. Let's dive in!

What is Xpath?

XPath stands for "XML Path Language" which essentially means it's a query language that described a path from point A to point B for XML/HTML type of documents.
Other path languages you might know of are CSS selectors which usually describe paths for applying styles, or tool-specific languages like jq which describe paths for JSON-type documents.

For HTML parsing, Xpath has some advantages over CSS selectors:

- Xpath can traverse HTML trees in every direction and is location-aware.
- Xpath can transform results before returning them.
- Xpath is easily extendable with additional functionality.

Before we dig into Xpath let's have a quick overview of HTML itself and how it enables xpath language to find anything with the right instructions.

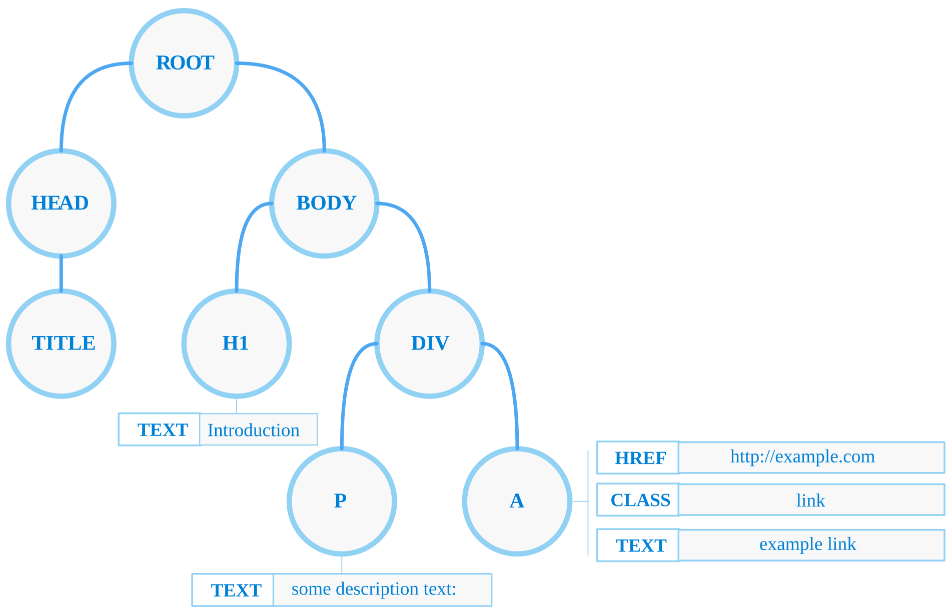
HTML Overview

HTML (HyperText Markup Language) is designed to be easily machine-readable and parsable. In other words, HTML follows a tree-like structure of nodes and their attributes, which we can easily navigate programmatically.

Let's start off with a small example page and illustrate its structure:

```
<head>
  <title>
  </title>
</head>
<body>
  <h1>Introduction</h1>
  <div>
    <p>some description text: /p>
    <a class="link" href="https://example.com">example link</a>
  </div>
</body>
```

In this basic example of a simple web page, we can see that the document already resembles a data tree. Let's go a bit further and illustrate this:



HTML tree is made of nodes which can contain attributes such as classes, ids and text itself.

Here we can wrap our heads around it a bit more easily: it's a tree of nodes and each node can also have properties attached to them like keyword attributes (like `class` and `href`) and natural attributes such as text.

Now that we're familiar with HTML let's familiarize ourselves with Xpath itself!

Xpath Syntax Overview

Xpath selectors are usually referred to as "xpaths" and a single xpath indicates a destination from the root to the desired endpoint. It's a rather unique path language, so let's start off with a quick glance over basic syntax.

Average xpath selector in web scraping often looks something like this:

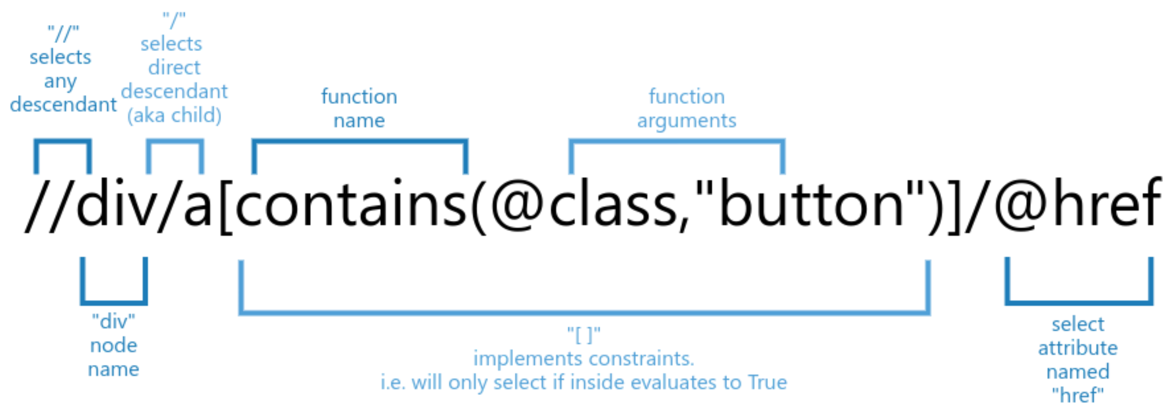

"//" selects any descendant
"/" selects direct descendant (aka child)
function name
function arguments
//div/a[contains(@class,"button")]/@href
"div" node name
"[ ]" implements constraints. i.e. will only select if inside evaluates to True
select attribute named "href"

illustration of a usual xpath selector's structure

In this example, XPath would select `href` attribute of an `<a>` node that has a class "button" which is also directly under `<div>` node:

<div> <a class="button" href="http://scrapfly.io">ScrapFly</a> </div>

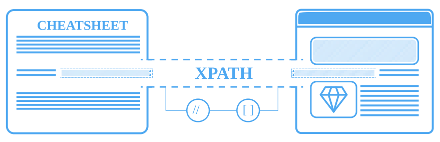
Xpath selectors are made up of multiple expressions joined together into a single string. Let's see the most commonly used expressions in this XPath cheat sheet:

expression	description
`/node`	selects a direct child that matches node name.
`//node`	selects any descendant - child, grandchild, gran-grandchild etc.
`*`	wildcard can be used instead of node name

expression	description
`@`	selects an attribute of a node e.g. `a/@href` will select value of `href` attribute of an `a` node
`text()`	selects text value of a node
`[]`	selector constraint - can be used to filter out nodes that do no match some condition
`parent` or `..`	select current nodes parent e.g. `/div/a/..` will select `div` element
`self` or `.`	select current node (this is useful as argument in xpath function, we'll cover more later)
`following-sibling::node`	selects all following siblings of type, e.g. `following-sibling::div` will select all `div` siblings below the current node
`preceding-sibling::node`	selects all preceding siblings of type, e.g. `preceding-sibling::div` will select all `div` siblings below the current node
`function()`	calls registered xpath function e.g. `/a/text()` will return text value of `a` node

Complete XPath cheatsheet

For more on XPath syntax see our full, interactive XPath cheatsheet which demonstrates all Xpath features used in web scraping



Let's solidify this knowledge with some real-life examples.

Basic Navigation

When writing xpaths the first thing we should be aware of is that all xpaths have to have a root (aka point A) and final target (aka point B). Knowing this and xpath axis syntax, we can start describing our selector path:

<div> <p class="socials"> Follow us on <a href="https://twitter.com/@scrapfly_dev">Twitter!</a> </p> </div>

Here, our simple xpath simply describes a path from the root to the `a` node.
All we used is `/` direct child syntax, however with big documents direct xpaths are often unreliable as any changes to the tree structure or order will break our path.

It's better to design our xpaths around some smart context. For example, here we can see that this `<a>` node is under `<p class="socials">` node - we can infer strong sense that these two will most likely go together:

<div> <p class="socials"> Follow us on <a href="https://twitter.com/@scrapfly_dev">Twitter!</a> </p> </div>

With this xpath, we get rid a lot of structure dependency in favor of context. Generally, modern websites have much more stable contexts than structures, and finding the right balance between context and structure is what creates reliable xpaths!

Further, we can combine constrains option (`[]`) with value testing functions such as `contains()` to make our xpath even more reliable:

<div> <p class="socials"> Follow us on <a href="https://twitter.com/@scrapfly_dev">Twitter!</a> </p> </div>

Using XPath `contains()` text function, we can filter out any results that don't contain a piece of text.

Xpath functions are very powerful and not only they can check for truthfulness but also modify content during runtime:

<div> <p class="socials"> Follow us on <a href="https://twitter.com/@scrapfly_dev">Twitter!</a> or connect with us on <a href="https://www.linkedin.com/company/scrapfly/">Linkedin</a> </p> </div>

Here, we've added `concat()` function, which joins all provided arguments into a single value and only then perform our match check.

Navigating Complex Structures

Sometimes tree complexity outgrows context based selectors and we have to implement some complex structure checks. For that, xpath has powerful tree navigation features that allow to select ancestors and siblings of any level:

<div> <span>For price contact </span> <a>Sales department </a> <div> <span>total: </span> </div> <span>166.00$</span> <span>*taxes apply</span> </div>

In this example, we find a text containing the phrase "total", navigate up to its parent and get the first following sibling using XPath.

Other times, we need to use position-based predicates and even combine multiple XPaths to reliably parse HTML data:

<div> <span>items: </span> <span>(taxes not included)</span> <span>166.00$</span> <span>25.00$</span> <span>*taxes apply</span> <div> <span>addons:</span> <span>0.5$</span> </div> </div>

In this example, we used `position()` function to select only siblings that are in specific range. We also combined to xpaths using the `|` operator (for or operation `or` operator can being used) to fully retrieve all pricing info.

As you can see, xpaths can be very powerful and parse almost any html structure if we get creative with path logic!

Extending Functions

Xpath in most clients can be extended with additional functions, and some clients even come with pre-registered non-standard functions.

For example, in Python's lxml (and it's based packages like parsel) we can easily register new functions like this:

```
from lxml import etree

def myfunc(context, *args):
    return True

xpath_namespace = etree.FunctionNamespace(None)
xpath_namespace['myfunc'] = myfunc
```

Other language clients follow a similar process.

Xpath Clients

Almost every programming language contains some sort of xpath client for XML file parsing. Since HTML is just a subset of XML we can safely use xpath in almost every modern language!

Xpath in Python

In Python there are multiple packages that implement xpath functionality, however most of them are based on lxml package which is a pythonic binding of **libxml2** and **libxslt** C language libraries. This means Xpath selectors in Python are blazing fast, as it's using powerful C components under the hood.

While lxml is a great wrapper, it lacks a lot of modern API usability features used in web scraping. For this, lxml based packages parsel (used by scrapy) and pyquery provide a richer feature set.

Example usage with parsel:

```
from parsel import Selector

html = """
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1 class="title">Page header by <a href="#">company</a></h1>
    <p>This is the first paragraph</p>
    <p>This is the second paragraph</p>
  </body>
</html>
"""
sel = Selector(html)
sel.xpath("//p").getall()
# [
# "<p>This is the first paragraph</p>",
# "<p>This is the second paragraph</p>",
# ]
```

Other tool recommendations:

- cssselect - converts css selectors to xpath selectors
- parsel-cli - real time REPL for testing css/xpath selectors

Xpath in PHP

In PHP most popular xpath processor is Symphony's DomCrawler:

```
use Symfony\Component\DomCrawler\Crawler;

$html = <<<'HTML'
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1 class="title">Page header by <a href="#">company</a></h1>
    <p>This is the first paragraph</p>
    <p>This is the second paragraph</p>
  </body>
</html>
HTML;

$crawler = new Crawler($html);
$crawler->filterXPath('//p');
```

Other tool recommendations:

- css-selector - converts css selectors to xpath selectors

Xpath in Javascript

Javascript supports xpath natively, you can read more about it on MDN's Introduction to Using Xpath in Javascript

Other tool recommendations:

- jQuery - extra syntax sugar and helpers for xpath querying.
- cash - lightweight, modern jQuery alternative.

Xpath in Other Languages

Most other languages have some sort of XPath client as XML parsing is an important data exchange feature. Meaning, we can parse web-scraped content in the language of our choice!

For example, C# supports XPath natively as well, you can read more about it over at the official documentation so does Objective C and other low-level languages.

While some languages might not have first-party XPath clients it's very likely there's a community package. For example, Go language has community packages for xpath in xml, html and even json.

Xpath in Browser Automation

Browser automation tools support XPath without any additional Javascript packages.

To access XPath in **Selenium** we can use `by.XPath` selector. For example, for Selenium in Python:

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://httpbin.org/html")

element = driver.find_element(By.XPATH, '//p')
```

To access XPath in **Playwright** we can use the locator functionality which take either CSS selectors or XPath as the argument. For example in Playwright and Python:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as pw:
    browser = pw.chromium.launch(headless=False)
    context = browser.new_context(viewport={"width": 1920, "height": 1080})
    page = context.new_page()

    page.goto("http://httpbin.org/html")
    paragraphs = page.locator("//p")  # this can also take CSS selectors
```

To access XPath in **Puppeteer** we can use the `$` and `$$` methods. For example in Puppeteer in Javascript:

```
const puppeteer = require('puppeteer')
const browser = await puppeteer.launch();
let page = await browser.newPage();
await page.goto('http://httpbin.org/html');
await page.$("//p");
```

FAQ

To wrap this introduction up let's take a look at some frequently asked questions regarding HTML parsing using XPath selectors:

Is XPATH faster than CSS selectors?

Many CSS selector libraries convert CSS selectors to XPATH because it's faster and more powerful. That being said it depends on each individual library and complexity of the selector itself. Some XPATH selectors which use broad selection paths like `//` can be very expensive computationally.

My xpath selects more data than it should

XPATH broad selector paths like `//` are global rather than relative. To make them relative we must add the relativity marker `.` -> `.//`

How to match nodes by multiple names?

To match nodes by multiple names we can use wildcard selector together with a name check condition: `//*[contains("p h1 head", name())]` - will select h1, p and head nodes.

How to select select elements between two nodes?

If we know two nodes like text headers we can select text between with clever use of `preceding-sibling` notation: `//h2[@id="faq"]//following-sibling::p[(preceding-sibling::h2[1])[@id="faq"]]` - this xpath selects all paragraph nodes under h2 tag with id `faq` and not elements under other `h2` nodes.

<div> <span>items: </span> <span>(taxes not included)</span> <span>166.00$</span> <span>25.00$</span> <span>*taxes apply</span> <div> <span>addons:</span> <span>0.5$</span> </div> </div>

Summary

In this article, we've introduced ourselves with xpath query language. We've discovered that HTML documents are data trees with nodes and attributes which can be machine parsed efficiently.

We glanced over the most commonly used XPath syntax and functions and explored common HTML parsing scenarios and best practices using our interactive XPath tester.

Xpath is a very powerful and flexible path language that is supported in many low-level and high-level languages: Python, PHP, Javascript etc. - so, whatever stack you're using for web-scraping, XPath can be your to-go tool for HTML parsing!

For more XPath help we recommend visiting Stackoverflow's #xpath tag and see our data parsing tag for more articles on data parsing.

Check out ScrapFly Python SDK

Try ScrapFly for FREE!

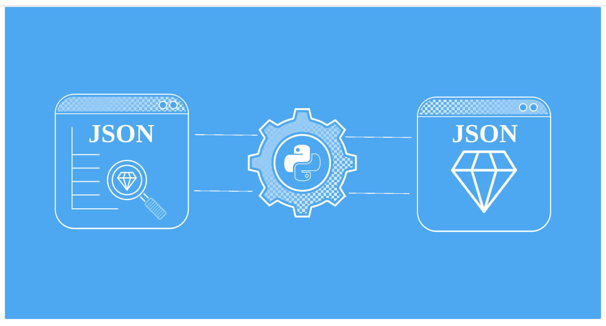
Related Questions

- How to scrape HTML table to Excel Spreadsheet (.xlsx)?
- How to select dictionary key recursively in Python?
- How to select all elements between two elements in XPath?
- How to parse dynamic CSS classes when web scraping?
- How to turn HTML to text in Python?
- How to use CSS selectors in NodeJS when web scraping?
- How to use XPath selectors in Python?
- How to select elements by text in XPath?
- How to select last element in XPath?
- What are some ways to parse JSON datasets in Python?
- What are devtools and how they're used in web scraping?
- How to select HTML elements by text using CSS Selectors?
- Scraper doesn't see the data I see in the browser - why?
- How to use XPath selectors in NodeJS when web scraping?
- How to select elements by class in XPath?

More >

DATA PARSING PARSEL XPATH PYTHON

Related Posts

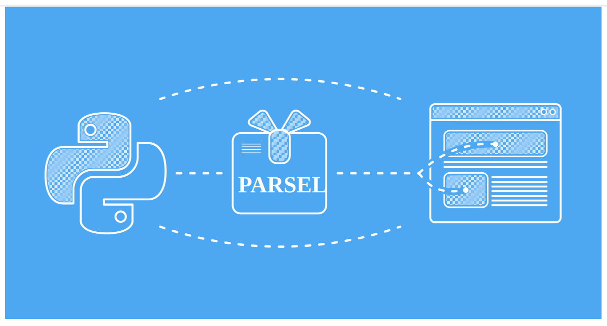


Jan 03, 2025

Ultimate Guide to JSON Parsing in Python

Learn JSON parsing in Python with this ultimate guide. Explore basic and advanced techniques using json, and tools like ijson and nested-lookup

DATA PARSING PYTHON

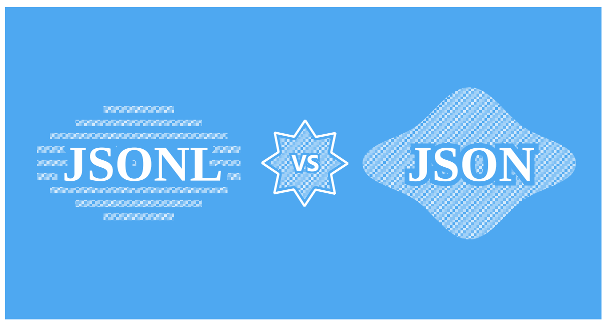


Dec 25, 2024

Guide to Parsel - the Best HTML Parsing in Python

Learn to extract data from websites with Parsel, a Python library for HTML parsing using CSS selectors and XPath.

DATA PARSING PARSEL



Dec 17, 2024

JSONL vs JSON

Learn the differences between JSON and JSONLines, their use cases, and efficiency. Why JSONLines excels in web scraping and real-time processing

Company

Careers
Terms of service
Privacy Policy
Data Processing Agreement
KYC Compliance
Status

Integrations

Zapier
Make
N8n
LlamaIndex
LangChain

Social

Tools

Convert cURL commands to Python code
JA3/TLS Fingerprint
HTTP2 Fingerprint
Xpath/CSS Selector Tester

Resources

API Documentation
Web Scraping Academy
Is Web Scraping Legal?
Web Scraping Tools
FAQ

Learn Web Scraping

Web Scraping with Python
Web Scraping with PHP
Web Scraping with Ruby
Web Scraping with R
Web Scraping with NodeJS
Web Scraping with Python Scrapy
How to Scrape without getting blocked tutorial
Web Scraping with Python and BeautifulSoup
Web Scraping with Nodejs and Puppeteer
How To Scrape Graphql
Best Proxies for Web Scraping
Top 5 Best Residential Proxies

Usage

What is Web Scraping used for?
Web Scraping for AI Training
Web Scraping for Compliance
Web Scraping for eCommerce
Web Scraping for Finance
Web Scraping for Fraud Detection
Web Scraping for Jobs
Web Scraping for Lead Generation
Web Scraping for News & Media
Web Scraping for Real Estate
Web Scraping for SERP & SEO
Web Scraping for Social Media
Web Scraping for Travel