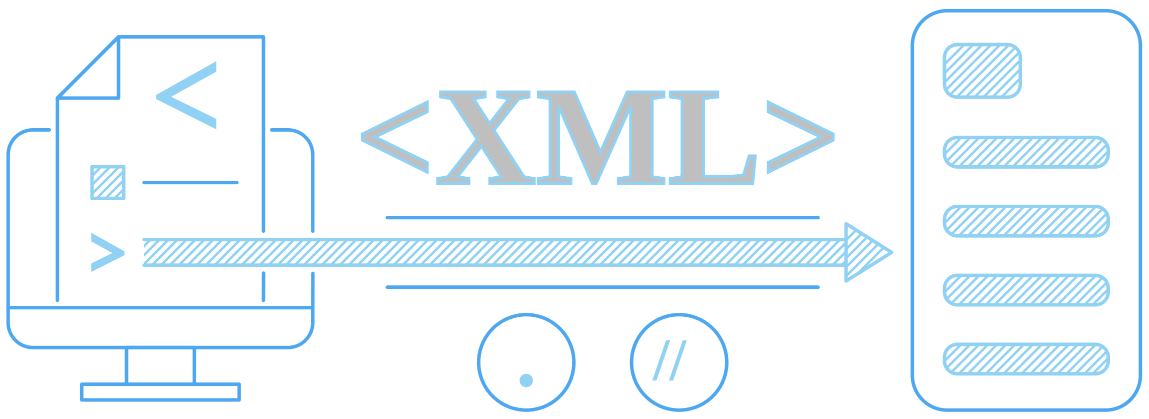
How to Parse XML

by Johann Saunier Apr 03, 2025

#Python #Css Selectors #XPath #Data Parsing



XML files are popular data types used to structure and store data in a hierarchal format. They are commonly used for exchanging data between servers and clients, making them valuable data sources. But what about parsing XML files for data extraction?

In this article, we'll explain how to parse XML files effectively for data extraction. We'll start by defining XML, its structure, and how to navigate it. Then, we'll dive into parsing methods using query languages and native tools. Finally, we'll walk through a practical Python example. Let's get started!

What are XML files?

XML (eXtensible Markup Language) is a markup language designed for creating files that are readable for both humans and machines. These files are used to exchange and transmit data between systems, such as servers, websites and databases.

Unlike HTML, XML files don't follow pre-defined tags. Instead, it's highly customizable and allows for defining any tag, which allows for better readability. Instead, they allow for creating custom tags, leading to better flexibility and readability. To better understand this, let's take a look at the XML structure.

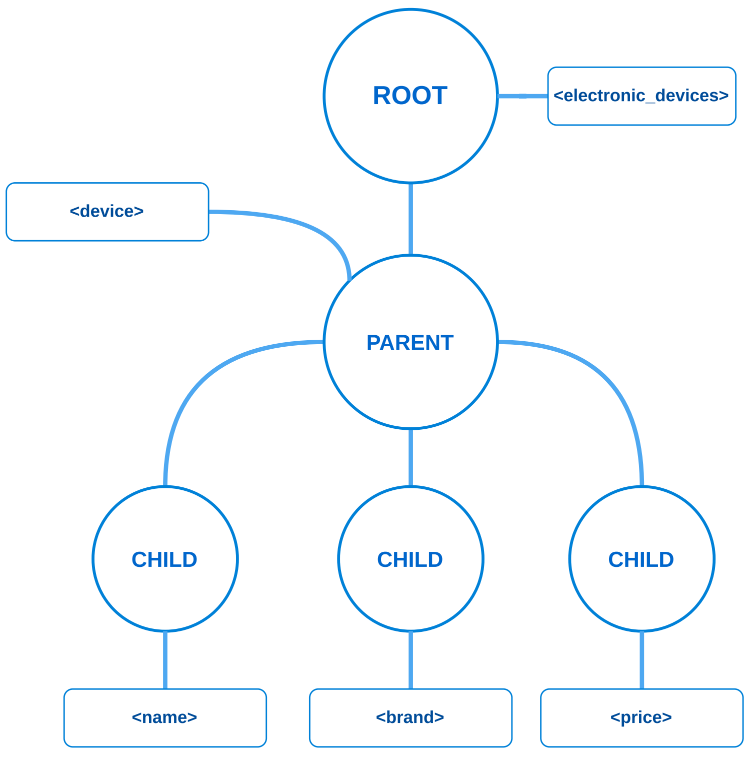
XML Structure

XML files follow a hierarchal structure, where elements are nested within each other. These elements are the building blocks of all XML files. For example, here is a simple XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<electronic_devices>
    <device>
        <name>Core I5 Laptop</name>
        <brand>HP</brand>
        <price>400$</price>
    </device>
</electronic_devices>
```

The `<electronic_devices>` represents the root element, and the `<device>` represents a parent with the `<name>`, `<brand>` and `<price>` as children elements.

When navigating the above XML file programmatically, it's represented as a tree-based structure:



XML file in a tree representation

XML elements can also include attributes that provide more context and details. These attributes also allow us to select and parse elements with more precision and efficiency:

```
<device category="laptops">
    <name>Core I5 Laptop</name>
    <!-- more elements-->
</device>
```

Now that we have an overview of XML files and their structure. Let's proceed to the XML parsing details.

Basic XML Navigation

When it comes to parsing an XML file, there are a few options:

- **Query languages**, such as XPath and CSS.
- **Native XML parses**, such as ElementTree in Python.

That being said, native XML parsers are limiting and ineffective. Therefore, it's better to use query languages due to their advanced capabilities. Moreover, XPath and CSS selectors work with HTML too.

We'll explain parsing XML with XPath and CSS selectors in detail. Then, we'll breifly go over the built-in XML parsing modules. But before proceeding into the details, let's revise the basics of XPath and CSS selectors. The below table represents the most popular XPath and CSS expressions:

XPath expression	CSS expression	description
`//node`	`node`	selects any descendant (child, grandchild, gran-grandchild etc.) that matches the node name
`/node`	`>node`	selects a direct child that matches the node name.
`[@attribute=]`	`[attribute=]`	selects a node by matching the attribute name or value.
`[@attribute=]`	`[attribute=]`	selects a node by matching the attribute name or value.
`/@attribute_name`	`::attr(attribute_name)`	selects an attribute value of a node e.g. `href` attribute of an `a` node
`/text()`	`:text`	selects a text value of a node.
`/`	`>`	sequence multiple selectors together.

The above expressions are commonly shared between XPath and CSS selectors. However, each one is slightly different from the other. For further details about these expressions, refer to our previous articles and XPath and CSS cheat sheets.

When parsing XML, it's important to know how to navigate through elements. For example, the following selectors can be used to select a specific element based on their parents:

<products> <product> <id>1</id> <name type="title">Box of Chocolate Candy</name> <price>24.99</price> <rate value="4.5 out of 5">Product rate</rate> </product> </products>

Here, we parse the product name by selecting the `products` element and then proceeding with its children to the desired element.

We reached the element by explicitly declaring its path. However, this approach can be inefficient when dealing with more complex XML files. Instead, it's better to shorten the path based on the elements' context. For example, we can see that the product name has a `type` of `title`. We can make use of this attribute to select the element directly:

<products> <product> <id>1</id> <name type="title">Box of Chocolate Candy</name> <price>24.99</price> <rate value="4.5 out of 5">Product rate</rate> </product> </products>

With this modification, we got rid of the long selector expression and made it more precise and reliable.

The previous expression can select an element by matching it against its attribute. Alternatively, we can reverse this logic and get the element's attribute value itself:

<products> <product> <id>1</id> <name type="title">Box of Chocolate Candy</name> <price>24.99</price> <rate value="4.5 out of 5">Product rate</rate> </product> </products>

Here, we parse the rate value by selecting a specific element value. However, we can even make it more sophisticated by searching for the attribute that contains part of the value:

<products> <product> <id>1</id> <name type="title">Box of Chocolate Candy</name> <price>24.99</price> <rate value="4.5 out of 5">Product rate</rate> </product> </products>

So far, we have explored XML parsing using basic XPath and CSS selectors' methods. Let's delve into more advanced approaches.

Advanced XML Navigation

With less descriptive XML files, it can be challenging to select specific elements. Therefore, we might need to parse XML elements' indexes or positions. For instance, the following XML code contains four duplicates of the same element. Here is how we can reach specific ones by matching against their position:

<steps> <step>intro step</step> <step>first step</step> <step>second second</step> <step>outro step</step> </steps>

Here, we use the `position` in XPath and `nth` in CSS to select the 2nd and 3rd elements from the XML file. We can also select these elements by matching or mismatching with the elements' text:

<steps> <step>intro step</step> <step>first step</step> <step>second second</step> <step>outro step</step> </steps>

Here, we select the elements that do not contain a specific text. Note that CSS selectors can't select elements using text values. The XPath selector outperforms it in this case.

When dealing with complex XML files, selecting a specific element can be challenging as elements can grow in breadth. However, we can overcome this challenge by selecting elements based on their children's attributes or values:

<product> <variant> <price>99$</price> <rate type="product A">4.5</rate> </variant> <variant> <price>120$</price> <rate type="product B">3.5</rate> </variant> </product>

Here, we select a specific variant based on its rate type and then select its price child.

We went through some XPath and CSS selector examples for parsing XML. However, XML files are designed to be readable for both machines and humans. So, it's not likely to encounter most of the previous challenges.

XML Parsing Clients

So far, we have explored parsing XML using query languages like XPath and CSS. However, there are built-in XML clients in most programming languages. Let's have a quick look at each language's XML client.

XML in Python

To parse an XML file in Python, we can use different packages. However, the most popular one is ElementTree. ElementTree allows for searching through XML files using their elements and supports most of the XPath syntax.

While using ElementTree, it's important to ensure your XML is well-formed. Otherwise, you might encounter an xml parse error, especially if there are missing tags or malformed elements.

Here is a simple overview of parsing XML using ElementTree:

```
import xml.etree.ElementTree as ET

xml_data = '''
<root>
  <products>
    <product>
      <name attribute="product_name">Dark Red Energy Potion</name>
      <price attribute="product_price">$4.99</price>
      <rate attribute="product_rate">4.7</rate>
      <description attribute="product_description">Bring out the best in your gaming
performance.</description>
    </product>
  </products>
</root>
'''

# Parse the XML data
root = ET.fromstring(xml_data)

# Iterate over elements
for parent in root:
    print(f"Parent tag: {parent.tag}")
    for child in parent:
        print(f"Child tag: {child.tag}")
        for grandchild in child:
            print(f"Grandchild tag: {grandchild.tag}, Attribute: {grandchild.attrib}")

# Access elements
product = root.find(".//product")
name = product.find(".//name").text
price = product.find(".//price").text
rate = product.find(".//rate").text
description = product.find(".//description").text

print(f"Name: {name}, Price: {price}, Rate: {rate}, Description: {description}")
```

Other Python XML parsing tools are:

- Minidom - A built-in library that converts XML files into a DOM representation.
- xml.sax - A built-in SAX (Simple API for XML) library for parsing XML in an event-driven approach.
- lxml - A third-party library for processing XML and HTML files in high performance.

XML in PHP

The most popular XML processing tool in PHP is SmipleXML:

```
<?php
$xml_data = '
<root>
  <products>
    <product>
      <name attribute="product_name">Dark Red Energy Potion</name>
      <price attribute="product_price">$4.99</price>
      <rate attribute="product_rate">4.7</rate>
      <description attribute="product_description">Bring out the best in your gaming
performance.</description>
    </product>
  </products>
</root>
';

// Parse the XML data
$xml = simplexml_load_string($xml_data);

// Iterate over elements
foreach ($xml->children() as $parent) {
    echo "Parent tag: {$parent->getName()}\n";
    foreach ($parent->children() as $child) {
        echo "Child tag: {$child->getName()}\n";
        foreach ($child->attributes() as $attr => $value) {
            echo "Attribute: $attr = $value\n";
        }
        foreach ($child->children() as $grandchild) {
            echo "Grandchild tag: {$grandchild->getName()}, Attribute: {$grandchild-
>attributes()['attribute']}\n";
        }
    }
}

// Access elements
$product = $xml->xpath('//product')[0];
$name = (string)$product->name;
$price = (string)$product->price;
$rate = (string)$product->rate;
$description = (string)$product->description;

echo "Name: $name, Price: $price, Rate: $rate, Description: $description\n";
?>
```

Other PHP XML parsing tools are:

- DOMDocument - Converts HTML and XPath files into a DOM representation.
- XMLReader - A memory-efficient tool for processing extensive XML files, it reads XML files in a streaming approach.

XML in JavaScript

JavaScript natively supports processing XML through the DOM in the browser. The most popular representation of this interface is the DOMParser:

```
const xmlData = `
<root>
  <products>
    <product>
      <name attribute="product_name">Dark Red Energy Potion</name>
      <price attribute="product_price">$4.99</price>
      <rate attribute="product_rate">4.7</rate>
      <description attribute="product_description">Bring out the best in your gaming
performance.</description>
    </product>
  </products>
</root>
`;

// Parse the XML data
const parser = new DOMParser();
const xmlDoc = parser.parseFromString(xmlData, 'text/xml');

// Access elements
const product = xmlDoc.getElementsByTagName('product')[0];
const name = product.getElementsByTagName('name')[0].textContent;
const price = product.getElementsByTagName('price')[0].textContent;
const rate = product.getElementsByTagName('rate')[0].textContent;
const description = product.getElementsByTagName('description')[0].textContent;

console.log(`Name: ${name}, Price: ${price}, Rate: ${rate}, Description:
${description}`);
```

Other tool recommendations:

- xml2js - A module that converts XML into JavaScript objects.
- xmldom - An implementation of the DOMParser for Node.js environments.

XML in Other Languages

XML files are commonly used for data exchange, so most programming languages come with built-in XML clients to process them. XPath clients can also be used for parsing XML and they can be found in almost all programming languages, either as built-in features or community packages such as:

- nokogiri and rexml in Ruby.
- encoding/xml in Golang
- libxml2 in C.
- jdom in Java.

XML Parsing Project

In this section, we'll create a Python XML parser to scrape XML data on a real target website. We'll be scraping real estate property links from this Realtor.com page. It contains the latest added properties on the website. The XML data on this page looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
  <channel>
    <title>Price Changed</title>
    <link>https://www.realtor.com</link>
    <description/>
    <atom:link href="https://pubsubhubbub.appspot.com/" rel="hub"/>
    <atom:link href="https://www.realtor.com/realestateandhomes-detail/sitemap-rss-
price/rss-price-ca.xml" rel="self"/>
    <item>
      <link>https://www.realtor.com/realestateandhomes-detail/275-Howell-Ave_Red-
Bluff_CA_96080_M27031-58587</link>
      <pubDate>Fri, 05 Jan 2024 13:43:02</pubDate>
    </item>
    <item>
      <link>https://www.realtor.com/realestateandhomes-detail/2575-Thunder-Mountain-
Rd_Upland_CA_91784_M19026-44599</link>
      <pubDate>Fri, 05 Jan 2024 13:47:32</pubDate>
    </item>
    <item>
      <link>https://www.realtor.com/realestateandhomes-detail/20631-W-Chestnut-
Cir_Porter-Ranch_CA_91326_M90130-33610</link>
      <pubDate>Fri, 05 Jan 2024 13:47:34</pubDate>
    </item>
  </channel>
</rss>
```

To scrape this data, we'll request the page URL and then parse the XML content using the XPath selector:

Python ScrapFly

```
import json
from httpx import Client
from parsel import Selector

client = Client(
    # enable http2
    http2=True,
    # add basic browser like headers to prevent being blocked
    headers={
        "accept-language": "en-US,en;q=0.9",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36",
        "accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
",
        "accept-encoding": "gzip, deflate, br",
    },
)

# send a request
response = client.get("https://www.realtor.com/realestateandhomes-detail/sitemap-rss-
price/rss-price-ca.xml")
# get the XML content from the response body
body = response.content
# convert the XML to a Pasrsel selector
selector = Selector(response.text, type="xml")
results = {}
# iterate over the items
for item in selector.xpath("//item"):
    url = item.xpath("./link/text()").get()
    pub_date = item.xpath("./pubDate/text()").get()
    results[url] = pub_date

# save the results into a JSON file
with open("data.json", "w", encoding="utf-8") as file:
    json.dump(results, file, indent=2, ensure_ascii=False)
```

Here, we request the page URL to get the XML content. Then, we convert the data we got into a Prasel XML selector. Finally, we parse the data to get the URL and date of each item and save the data into a JSON file. Here is a sample output of the result we got:

```
{
  "https://www.realtor.com/realestateandhomes-detail/231-S-Laxore-St_Anaheim_CA_92804_M14771-41700": "Fri, 05 Jan 2024 14:39:35",
  "https://www.realtor.com/realestateandhomes-detail/60815-Burnt-Valley-Rd_Anza_CA_92539_M21181-92033": "Fri, 05 Jan 2024 14:39:43",
  "https://www.realtor.com/realestateandhomes-detail/463-Buckeye-St_Vacaville_CA_95688_M24868-08760": "Fri, 05 Jan 2024 14:50:33",
  "https://www.realtor.com/realestateandhomes-detail/61-Arthur-Dr_Santa-Rosa_CA_95403_M29889-89374": "Fri, 05 Jan 2024 14:57:14",
  "https://www.realtor.com/realestateandhomes-detail/29413-Fenders-Ferry-Rd_Round-Mountain_CA_96084_M29662-29330": "Fri, 05 Jan 2024 15:00:47"
}
```

FAQ

To wrap up this guide, let's have a look at some frequently asked questions about XML parsing.

What is parsing in XML?

Parsing in XML means searching through the XML elements to get their data. This can be achieved by matching against elements' tags, attributes, or values.

How to parse XML in Python?

You can parse XML files in Python using tools such as ElementTree, lxml, or query languages like XPath. Be sure to handle potential issues like an xml parse error using proper error handling.

What's the difference between XML and HTML?

HTML is an XML-based markup language used for creating web pages so it's a more restrictive version of XML. Fortunately, both markups are very similar and can often be parsed using the same tools.

Summary

In this article, we explained XML files - markup language files used for exchanging data in a readable format.

We have explained how to use query languages such as XPath and CSS to parse XML, which allows us to precisely select elements in the least amount of expressions. We also reviewed the available methods for parsing XML in different programming languages.

Finally, we went through an example project of creating a Python XML parser.

Check out ScrapFly Python SDK

Try ScrapFly for FREE!

Related Questions

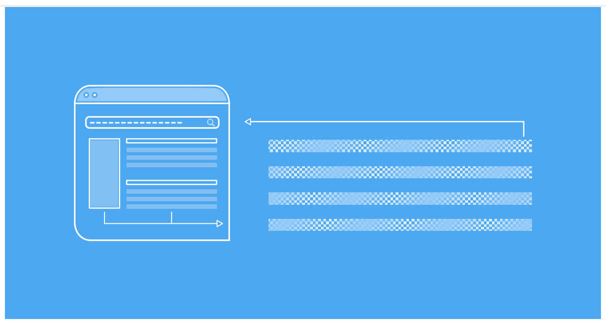
- What Python libraries support HTTP2?
- Python httpx vs requests vs aiohttp - key differences
- How to scrape HTML table to Excel Spreadsheet (.xlsx)?
- How to handle popup dialogs in Playwright?

- How to use proxies with Python httpx?
- How to select dictionary key recursively in Python?
- What are some ways to parse JSON datasets in Python?
- Selenium: chromedriver executable needs to be in PATH?
- How to fix python requests ConnectTimeout error?
- How to fix Python requests MissingSchema error?
- How to scrape images from a website?
- How to check if element exists in Playwright?
- How to use cURL in Python?
- Selenium: geckodriver executable needs to be in PATH?
- How to fix Python requests ReadTimeout error?

More >

PYTHON CSS SELECTORS XPATH DATA PARSING

Related Posts

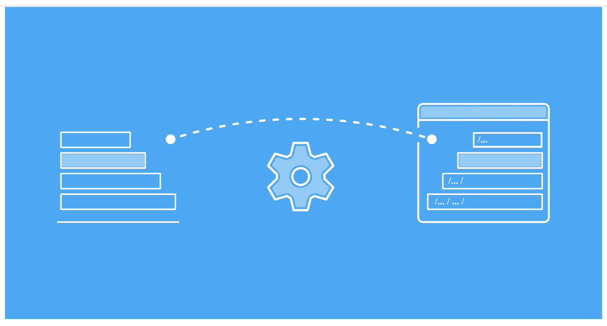


Mar 10, 2025

Guide to List Crawling: Everything You Need to Know

In-depth look at list crawling - how to extract valuable data from list-formatted content like tables, listicles and paginated pages.

WEB CRAWLING BEAUTIFULSOUP PYTHON

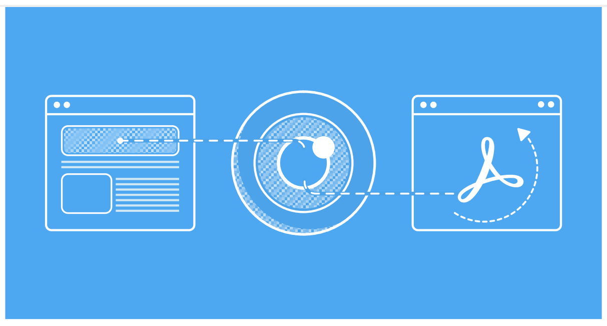


Jan 29, 2025

How to Find All URLs on a Domain

Learn how to efficiently find all URLs on a domain using Python and web crawling. Guide on how to crawl entire domain to collect all website data

WEB CRAWLING PYTHON



Jan 22, 2025

How to Capture and Convert a Screenshot to PDF

Quick guide on how to effectively capture web screenshots as PDF documents

SCREENSHOTS PYTHON NODEJS

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