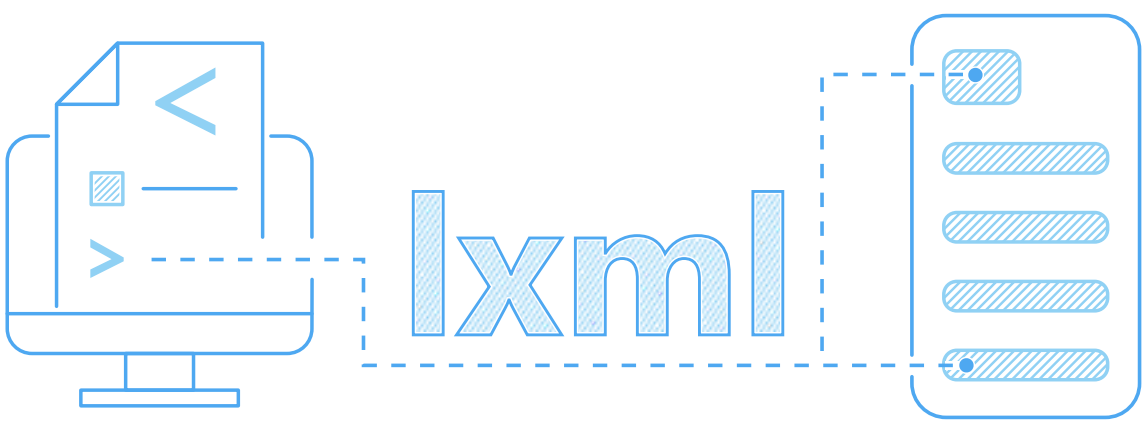
Intro to Parsing HTML and XML with Python and lxml

by Johann Saunier Mar 05, 2025 #Python #Tools #Data Parsing



Data parsing is an essential step in almost every web scraping process. It's used in defining scraping and crawling logic as well as the final data output structure.

In this tutorial, we'll take a deep dive into lxml - a powerful Python library that allows for parsing HTML and XML documents effectively. We'll start by explaining what lxml is, how to install it and using lxml processing XML and HTML documents.

Finally, we'll go over a practical web scraping with lxml example by using it to scrape e-commerce product data. Let's get started!

What is Lxml?

The Python LXML module is a Python interface for the libxml2 and libxslt C parsers. It combines the speed of XML features with the high efficiency of the underlying C parses to create an API that closely follows the Python ElementTree API.

The Lxml project follows the ElementTree concept. To put it shortly: **it treats the HTML and XML documents as a tree of node objects**. In this tree, each element is represented as a node, where each node contains attributes, text values and child elements.

Since Lxml provides fast Python XML processing, it's used by popular parsing libraries, such as BeautifulSoup and Parsel.

How to Install Lxml?

Lxml can be installed using the following `pip` command:

```
pip install lxml
```

The Lxml XML toolkit is not a pure Python language library and relies on the C libraries `libxml2` and `libxslt` which are often provided by your operating system. For more see the official installation instructions.

Since we'll use lxml for web scraping, for our example scrapers we'll need an HTTP client to request the web pages and return the web page data as am HTML document. In this article, we'll use httpx, but it can be replaced with other clients, such as the requests module. Install `httpx` using the following command:

```
pip install httpx
```

ElementTree With Lxml

As discussed earlier, lxml follows the ElementTree API. This API was designed to parse XML files. However, it can parse HMTL since the latter is a superset of XML. For clarity, we'll still split our examples into two code tabs for each data type.

Let's explore the `etree` API capabilities by navigating through some example documents:

HTML XML

```
<div id="root">
  <div id="products">
    <div class="product">
      <div id="product_name">Dark Red Energy Potion</div>
      <div id="product_price">$4.99</div>
      <div id="product_rate">4.7</div>
      <div id="product_description">Bring out the best in your gaming performance.
</div>
    </div>
  </div>
</div>
```

The above data represent simple product data, where each product contains a few data fields: name, price, rate and description.

We'll parse each document using `etree` and navigate through its XML and HTML elements:

HTML XML

```
from lxml import etree

html_data = '''
<div id="root">
  <div id="products">
    <div class="product">
      <div id="product_name">Dark Red Energy Potion</div>
      <div id="product_price">$4.99</div>
      <div id="product_rate">4.7</div>
      <div id="product_description">Bring out the best in your gaming performance.
</div>
    </div>
  </div>
</div>
'''

# Parse the HTML data using lxml
root = etree.fromstring(html_data)

# Navigate by parsing HTML tags
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},
Text: {grandchild.text}")
```

We start by importing the `etree` module and parsing the document into a `root` variable (as in, the start of the tree). Then, we create multiple nested loops to get the subsequent layers of the element tree - each element's tag, attributes and text:

```
Parent tag: products
Child tag: product
Grandchild tag: name, Attribute: {'attribute': 'product_name'}, Text: Dark Red Energy
Potion
Grandchild tag: price, Attribute: {'attribute': 'product_price'}, Text: $4.99
Grandchild tag: rate, Attribute: {'attribute': 'product_rate'}, Text: 4.7
Grandchild tag: description, Attribute: {'attribute': 'product_description'}, Text:
Bring out the best in your gaming performance.
```

Each tree layer is represented as a list of elements and each element as a dictionary of attributes so once we have the element we can easily retrieve any attribute:

HTML XML

```
from lxml import etree

html_data = '<div type="product_rate" review_count="774">4.7</div>'

# Parse the HTML data using lxml
element = etree.fromstring(html_data)

# Get a specific attribute value
print(element.get("review_count"))
"774"
```

Here, we get the `review_count` attribute value from the element. This method is especially helpful when scraping data from web pages through specific attributes, such as links found in `href` attributes.

XPath With Lxml

Lxml natively supports XPath expressions, meaning we can use XPath selectors to filter and find elements through the entire tree structure:

HTML XML

```
from lxml import etree

html_data = '''
<div id="products">
  <div class="product">
    <div id="product_name">Dark Red Energy Potion</div>
    <div class="pricing">
      <div>Price with discount: $4.99</div>
      <div>Price without discount: $11.99</div>
    </div>
    <div id="product_rate" review_count="774">4.7 out of 5</div>
    <div id="product_description">Bring out the best in your gaming performance.</div>
  </div>
</div>
'''

# Parse the HTML data using lxml
root = etree.fromstring(html_data)

# Iterate over the product div elements
for element in root.xpath("//div/div//div"):
    print(element.text)
```

Here, we use XPath to select all the product's descending elements and save them to a list. Then, we iterate over all the elements we got and return their text:

```
Dark Red Energy Potion
Price with discount: $4.99
Price without discount: $11.99
4.7
Bring out the best in your gaming performance.
```

We can also make our lxml parsing logic more reliable by using more precise XPath expressions by matching attributes or text values:

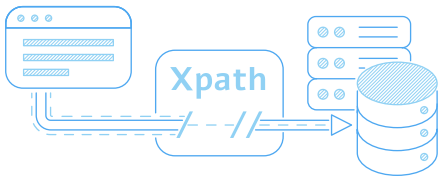
HTML XPath

```
from lxml import etree

html_data = '''
<div id="products">
  <div class="product">
    <div id="product_name">Dark Red Energy Potion</div>
    <div class="pricing">
      <div>Price with discount: $4.99</div>
      <div>Price without discount: $11.99</div>
    </div>
    <div id="product_rate" review_count="774">4.7 out of 5</div>
    <div id="product_description">Bring out the best in your gaming performance.</div>
  </div>
</div>
'''

# Parse the HTML data using lxml
root = etree.fromstring(html_data)

# parse elements
name = root.xpath("//div[@id='product_name']/text()")[0]
discount_price = root.xpath("//div[contains(text(), 'with discount')]/text()")[0]
review_count = root.xpath("//div[@id='product_rate']/@review_count")[0]
descirption = root.xpath("//div/div/div[4]/text()")[0]
```

In the above code, we parse the elements based on their attributes, text and order index. However, XPath allows for even more advanced and clever expressions so for more details, refer to our previous guide on XPath 👇

Parsing HTML with Xpath

Learn about XPath selectors and how to write effective XPath expressions. You will also learn about different XPath clients for different programming languages.



CSS With Lxml

While lxml doesn't directly support CSS selector expressions it can be enabled through the cssselect module. This package can convert CSS selectors to XPath which can be used by lxml engine.

Note that `cssselect` is built in the lxml package, and you don't have to install it separately though it's also available on its own as `csselect` on pypi.

Let's start parsing HTML and XML with lxml by navigating through elements using CSS selectors:

HTML XML

```
from lxml import etree

html_data = '''
<div id="products">
  <div class="product">
    <div id="product_name">Dark Red Energy Potion</div>
    <div class="pricing">
      <div>Price with discount: $4.99</div>
      <div>Price without discount: $11.99</div>
    </div>
    <div id="product_rate" review_count="774">4.7 out of 5</div>
    <div id="product_description">Bring out the best in your gaming performance.</div>
  </div>
</div>
'''

# Parse the HTML data using lxml
root = etree.fromstring(html_data)

# Iterate over the product div elements
for element in root.cssselect("div div div"):
    print(element.text)
```

We select all the descending `div` elements of the product class. Then, we iterate over each element and return its text:

```
Dark Red Energy Potion
Price with discount: $4.99
Price without discount: $11.99
4.7 out of 5
Bring out the best in your gaming performance.
```

Although the `cssselect` module allows for executing CSS expressions. It does not support pseudo-elements, such as matching with attributes or text. However, since we are able to get the attributes and text values of each element, we can implement the filtering logic ourselves:

Python ScrapFly

```
from lxml import etree

html_data = '''
<div id="products">
  <div class="product">
    <div id="product_name">Dark Red Energy Potion</div>
    <div class="pricing">
      <div>Price with discount: $4.99</div>
      <div>Price without discount: $11.99</div>
    </div>
    <div id="product_rate" review_count="774">4.7 out of 5</div>
    <div id="product_description">Bring out the best in your gaming performance.</div>
  </div>
</div>
'''

# Parse the HTML data using lxml
root = etree.fromstring(html_data)

# get all the div elements
div_elements = root.cssselect("div div div")

# match by a specific text value
discount_price = [element for element in div_elements if "without discount" in
element.text][0].text
print(discount_price)
"Price without discount: $11.99"

# match by an attribute value
rate = [element for element in div_elements if element.get("id") == "product_rate"]
[0].text
print(rate)
"4.7 out of 5"
```

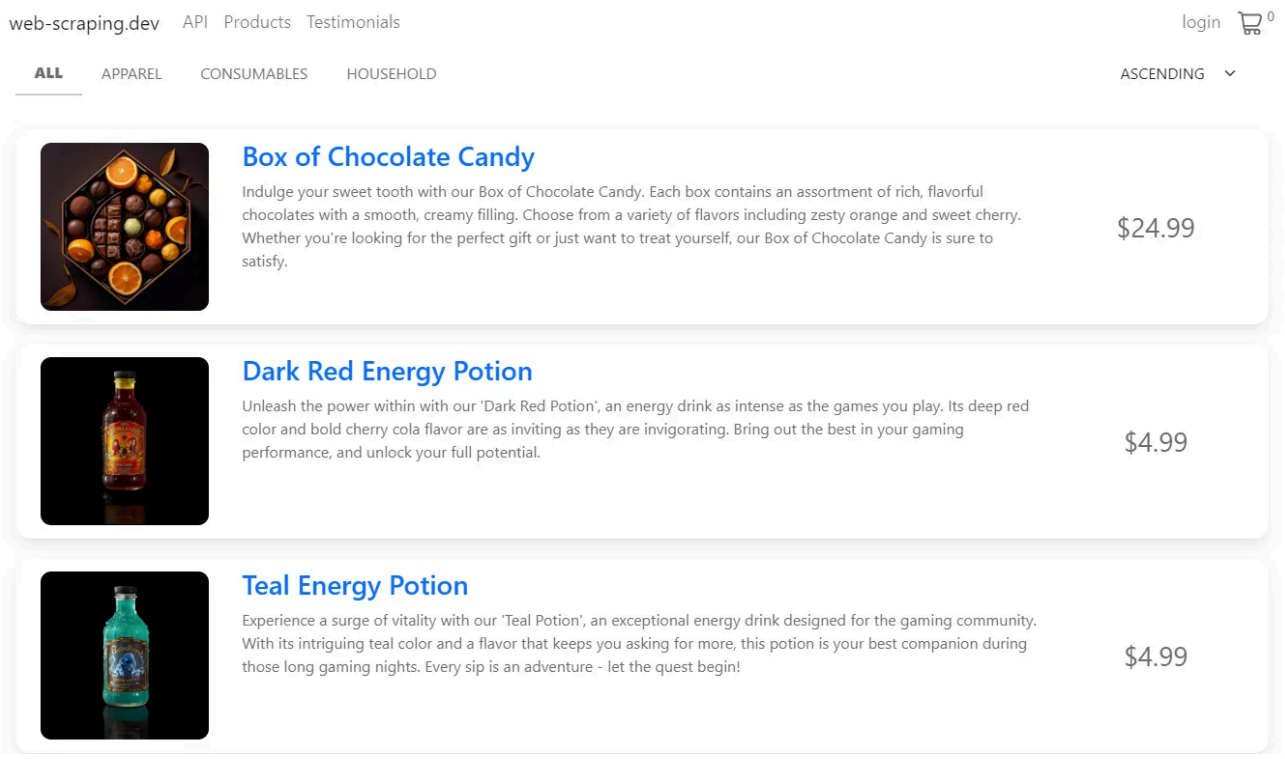
We have seen that using CSS selectors with lxml can be quite limiting due to the lack of support for pseudo-elements. However, it's available for other parsing libraries, such as Parsel. For more details, refer to our previous guide on CSS selectors 👇

Parsing HTML with CSS Selectors

Learn about CSS selectors, their limitations and how to write effective CSS expressions. You will also learn about different CSS clients for different programming languages.

Web Scraping With Lxml

In this section, we'll go over a practical lxml web scraping example. We'll scrape product data from web-scraping.dev and parse the HTML using lxml:



Product page on web-scraping.dev

Let's begin with the parsing logic. We'll iterate over product cards and extract each product's details:

```
def parse_products(response: Response) -> List[Dict]:
    """parse products from HTML"""
    # create a lxml selector
    selector = etree.fromstring(response.text)
    data = []
    for product in selector.xpath("//div[@class='row product']"):
        name = product.xpath(".//div[contains(@class,
description)]/h3/a/text()").get()
        link = product.xpath(".//div[contains(@class, description)]/h3/a/@href").get()
        product_id = link.split("/product/")[-1]
        price = float(product.xpath(".//div[@class='price']/text()").get())
        data.append({
            "product_id": int(product_id),
            "name": name,
            "link": link,
            "price": price
        })
    return data
```

Here, we use lxml to create a selector by parsing the HMTL using the ElementTree API. Next, we'll utilize the `parse_products` function while requesting the product pages to extract the product data:

```
import asyncio
import json
from lxml import etree
from httpx import AsyncClient, Response
from typing import List, Dict

# initializing a async httpx client
client = AsyncClient(
    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",
    }
)


def parse_products(response: Response) -> List[Dict]:
    """parse products from HTML"""
    # create a lxml selector
    parser = etree.HTMLParser()
    selector = etree.fromstring(response.text, parser)
    data = []
    for product in selector.xpath("//div[@class='row product']"):
        name = product.xpath(".//div[contains(@class, description)]/h3/a/text()")[0]
        link = product.xpath(".//div[contains(@class, description)]/h3/a/@href")[0]
        price = float(product.xpath(".//div[@class='price']/text()")[0])
        data.append({
            "name": name,
            "link": link,
            "price": price
        })
    return data


async def scrape_products(url: str) -> List[Dict]:
    """scrape product pages"""
    # scrape the first product page first
    first_page = await client.get(url)
    products_data = parse_products(first_page)
    # add the remaining product pages to a scraping list
    other_pages = [
        client.get(url + f"?page={page_number}")
        # the maximum available pages are 5
        for page_number in range(2, 5 + 1)
    ]
    for response in asyncio.as_completed(other_pages):
        response = await response
        data = parse_products(response)
        # extend the first page data with new ones
        products_data.extend(data)
    print(f"scraped {len(products_data)} products")
    return products_data
```

▶ Run the code

In the above code, we request the first product page and parse its HTML to extract the product data and save it into the `products_data` list. Then, we add the remaining pages to a scraping list, scrape them concurrently and extend the `products_data` list with the new data we get.

Here is a sample output of the result we got:

▶ Sample output

Our HTML parsing with lxml example was a success! Although this parsing example is implemented for HTML documents. We have covered parsing XML in an example project before. It's worh noting that our earlier project uses XPath with Parsel instead of lxml.

However, since lxml supports XPath, the code from the previous example can be easily adapted to parse XML with lxml.

Lxml vs BeautifulSoup vs Parsel

There are various HTML parses available in Python. Which one to choose for web scraping? Let's compare them to find out. We'll list the pros and cons of each one to find the optimal parsing library for your case.

BeautifulSoup

BeautifulSoup is a very popular library for parsing in Python due to its ease of use.

Pros

- Beginner-friendly and easy to learn.
- Supports CSS pseudo-elements.

Cons

- Doesn't support XPath selectors.
- Slow parsing performance, even if lxml is used as the backend.

Lxml

Lxml is one of the oldest Python parsing libraries. Its high performance makes it outstanding.

Pros

- Supports both XPath and CSS selectors.
- Very efficient in terms of performance.

Cons

- Doesn't support CSS pseudo-elements.
- Doesn't provide additional utilities for parsing while web scraping.
- Its low-level API can be confusing for beginners.

Parsel

Parsel is Python parsing library that's tailored for web scraping. Yet, it's not very popular.

Pros

- Supports both XPath and CSS selectors.
- Supports CSS pseudo elements.
- Supports JMESPath and regular expressions.
- Provides various parsing utilities in a Pythonic way.

Cons

- It may have a steeper learning curve due to its many features and advanced functionality.

FAQ

To wrap up this guide on parsing with lxml for web scraping, let's have a look at some frequently asked questions.

Is lxml built in Python?

No, lxml doesn't come pre-installed in Python. It can be installed using pip: `pip install lxml`.

What is the best parsing library in Python?

Each parsing library is unique with its own features and the best parsing library can differ based on the use case. If the document structure is straightforward and performance is crucial, **lxml** is a strong choice. For beginners and quick web scraping tasks, **BeautifulSoup** can be more suitable. For complex HTML structures and heavy web scraping tasks, **Parsel** can be a better choice.

Is lxml faster than BeautifulSoup?

Yes, `lxml` is the fastest way to parse large amount of HTML and XML data. Though note that in web scraping parsing speed is almost never the bottle neck when it comes to scraping speed.

Lxml Parsing Summary Summary

In this article, we explored the lxml library - a highly efficient Pythonic binding on top of C parsers.

We started by explaining what lxml is and how to install it. Then, we explained using lxml for parsing HTML and XML documents using the `etree` API, XPath and CSS selectors. Finally, we went through a quick comparison between different parsing libraries in Python.

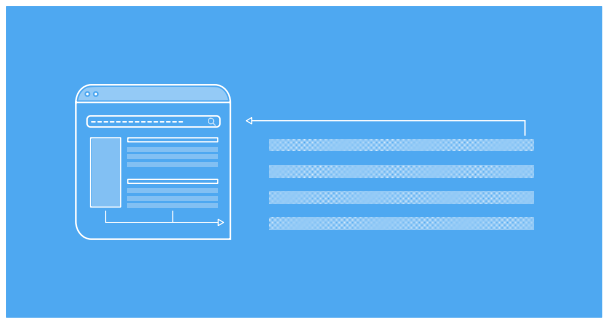
Related Questions

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

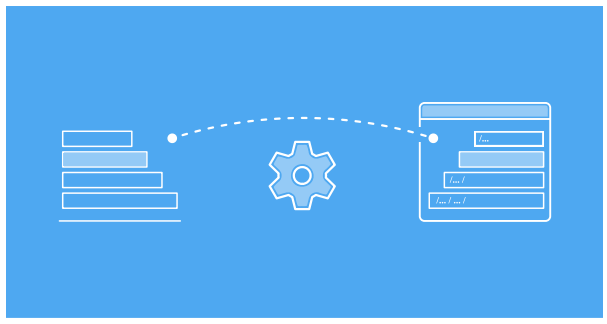
Related Posts



Mar 10, 2025

Guide to List Crawling: Everything You Need to Know

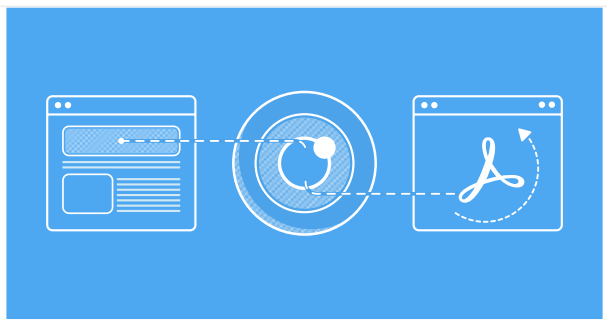
In-depth look at list crawling - how to extract valuable data from list-formatted content like tables, listicles and paginated



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



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

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