oxylabs®

Best Python Libraries for Web Scraping

Top python library tutorials to get you started in web scraping

oxylabs®

Introduction

Although web scraping in its totality is a complex and nuanced field of knowledge, building your own basic web scraper is not all that difficult. And that's mostly due to coding languages such as Python. Python is one of the easiest ways to get started as it is an object-oriented language. Python's classes and objects are significantly easier to use than in any other language. Additionally, many libraries exist that make building a tool for web scraping in Python an absolute breeze. This document will go through four most popular libraries and the basics on how to get started in web scraping.

Why Use Python?

If you need to start writing code for web scraping, it is definitely worth it to learn Python. The best part is that Python, compared to other programming languages, is easy to learn, clear to read, and simple to write in.

Python advantages for web scraping

Diverse libraries. Python has a fantastic collection of libraries such as *BeautifulSoup, Selenium, lxml,* and much more. These libraries are a perfect fit for web scraping and, also, for further work with extracted data. You will find more information about these libraries below.

Easy to use. To put it simply, Python is easy to code. Of course, it is wrong to believe that you would easily write a code for web scraping without any programming knowledge. But, compared to other languages, it is much easier to use as you do not have to add semicolons like ";" or curly-brackets "{}" everywhere. Many developers agree that this is the reason why Python is less messy. Furthermore, Python syntax is clear and easy to read. Developers can simply navigate between different blocks in the code.

Saves time. As you probably know, web scraping was created to simplify time-consuming tasks like collecting vast amounts of data manually. Using Python for web scraping is similar because you are able to write a little bit of code that completes a large task. Python saves a bunch of developers' time.

Community. As Python is one of the most popular programming languages, it also has a very active community. Developers are sharing their knowledge on various questions, so if you are struggling while writing the code, you can always search for help.

Comparing Python to other languages

Python is the most popular programming language for web scraping because it can handle almost all processes related to data extraction smoothly. However, there are other languages that can be used by developers for web scraping such as *Ruby, C ++, PHP*.

All of these languages have their pros and cons compared to Python, so let's compare them in terms of web scraping.

Programming language	Pros for web scraping	Comparison with Python	Conclusion
Ruby	+ Specific libraries (gems) for web scraping, such as **NokoGiri** and **HTTParty**. + Easy to follow and convenient syntax.	– Has lower performance. – Hard to locate good documentation. – Complicated to find help when struggling with coding.	Even if Python is more widely used for web scraping, Ruby is a suitable choice for web scraping too. Choosing Ruby depends on developers skills and tasks.
C++	+ High performance and speed. + C++ is one of the most popular programming languages with a large and active community.	– C++ is harder to learn. – In C++, the web scraping concept is expressed in more lines of code. – C++ is a static programming language. Coding for web scraping is more comfortable with a dynamic programming language like Python.	Even if C++ is not the best choice to set up a crawler, **libcurl** can solve this problem as developers use this library to fetch URLs. If needed, C++ can be used for web scraping.
PHP	+ High-quality web scraping libraries such as **Goutte**, **cURL**, **HTTPful**, and much more. + Widely used programming language with an active community.	– Writing a web crawler program with PHP language requires additional time for problems like task scheduling.	Python is less complicated and more comfortable to learn than PHP. However, if there is a need, PHP can be suitable for web scraping because of various great libraries.

Python is a perfect fit for building web scrapers and extracting data as it has a large selection of libraries, and an active community to search for help if you have issues with coding. One of the most important parts why use Python for web scraping is that Python is easy to learn, clear to read, and simple to write in.

Web Scraping Libraries: Where to Start?

Your dev team, of course, will be working with various libraries, integration tools, etc. There are many libraries to choose from. However, we list out the most popular, tried and tested libraries and tools you might need when building your infrastructure. Here are the top four:

- **Puppeteer** for JavaScript-heavy websites. If you are scraping hotel listings, e-commerce product pages, or similar – this will become your main headache. Many modern sites use JavaScript to load content asynchronously (i.e., hides part of the content to not be visible during the initial page load). The easiest way to manage JavaScript-heavy sites is to use a headless browser – a browser, but without a graphical user interface. This is where Puppeteer comes into the picture.

- **Selenium**. Similarly to Puppeteer, it is a solution that helps control headless browsers. It is one of the more popular browser automation tools out there, so experimenting with both is suggested.

- **lxml**. lxml is one of the fastest and feature-rich libraries for processing XML and HTML in Python. By using the lxml library, XML and HTML documents can be created, parsed, and queried.

- **Beautiful soup** for parsing. We will cover parsing a little bit later in this article, but to put it simply, there is no real point to web scraping without being able to parse your data to make it more readable. Beautiful soup is a Python package used for parsing HTML and XML documents.

oxylabs®

Puppeteer: Scraping With a Headless Browser

Automating web scraping

Generally, there are two methods of accessing and parsing web pages. The first method uses packages e.g., Axios. It directly sends a get request to the web page and receives HTML content. This can then be parsed using packages like Cheerio.

Though this is a fast method, it has its limitations. The biggest is that it cannot handle dynamic sites – sites that are rendered using JavaScript. The easiest way to manage these sites is to open a browser and load the site.

Unfortunately, loading a browser would take a lot of resources because it has to load a lot of other things like the toolbar and buttons. These UI elements are not needed when everything is being controlled with code. Fortunately, there are better solutions – headless browsers.

What is a headless browser?

A headless browser is simply a browser but without a graphical user interface. Think of it as a hidden browser. Headless browsers have complete functionality offered by a browser while being faster and taking up a lot less memory because there is no user interface. Everything is controlled programmatically.

The most commonly used browsers, Chrome and Firefox, support headless mode. There are few more browsers with headless mode supported, for example, Splash, Chromium, etc. Splash is aimed at Python programmers. In this Puppeteer tutorial, we will be focusing on Chromium.

Chromium is an open-source web browser made by Google. Note that Chromium and Chrome are two different browsers. Chromium is an open-source project. Chrome and is built over Chromium by adding many features. In addition to Chrome, many other browsers are based on Chromium, for example, Microsoft Edge, Opera, Brave, etc.

Now that we know what a headless browser is, it's time to understand the available options to control the browsers programmatically.

Controlling the browsers programmatically

There are several solutions to control headless browsers. Perhaps the most widely known solution is Selenium. We will cover selenium later in this article, but to quickly answer is Puppeteer better than selenium – if you need a lightweight and fast headless browser for web scraping, Google Puppeteer would be the better choice.

This Puppeteer tutorial will cover Puppeteer in much detail. Puppeteer, however, is a Node.js package, making it exclusive for JavaScript developers. Python programmers, therefore, have a similar option – Pyppeteer.

Pyppeteer

Pyppeteer is an unofficial port of Puppeteer for Python. This also bundles Chromium and works smoothly with it. Pyppeteer can work with Chrome as well, similar to Puppeteer.

The syntax is very similar as it uses the asyncio library for Python, except the syntactical differences between Python and JavaScript. Here are two scripts in JavaScript and Python that load a page and then take a screenshot of it.

Puppeteer example:

```
const puppeteer = require('puppeteer');
async function main() {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://oxylabs.io/');
  await page.screenshot({'path': 'oxylabs_js.png'})
  await browser.close();
}
main();
```

Pyppeteer Example:

```
import asyncio
import pyppeteer
async def main():
    browser = await pyppeteer.launch()
    page = await browser.newPage()
    await page.goto('https://oxylabs.io/')
    await page.screenshot({'path': 'oxylabs_python.png'})
   await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

The code is very similar. For web scraping dynamic websites, Pyppeteer can be an excellent alternative to Selenium for Python developers. But for the sake of making a Puppeteer tutorial, the following sections, we will cover Puppeteer, starting with the installation.

Installation

Before moving on with this Puppeteer tutorial, let's get the basic tools installed.

Prerequisite

There are only two pieces of software that will be needed:

- Node.js (which is bundled with npm—the package manager for Node.js)
- Any code editor

The only thing that you need to know about Node.js is that it is a runtime framework. This means that JavaScript code, which typically runs in a browser, can run without a browser.

Node.js is available for Windows, Mac OS, and Linux. It can be downloaded at their official download page.

Create node.js project

Before writing any code to web scrape using node js, create a folder where JavaScript files will be stored. All the code for Puppeteer is written in .js files and is run by Node.

Once the folder is created, navigate to this folder and run the initialization command:

```
npm init -y
```

This will create a package.json file in the directory. This file will contain information about the packages that are installed in this folder. The next step is to install the Node.js Packages in this folder.

How do you run Puppeteer

Installing Puppeteer is very easy. Just run the npm install command from the terminal. Note that the working directory should be the one which contains package.json:

```
npm install puppeteer
```

Note that Puppeteer is bundled with a full instance of Chromium. When it is installed, it downloads a recent version of Chromium that is guaranteed to work with the version of Puppeteer being installed.

Getting started with Puppeteer

Puppeteer is a promise-based library, which means it performs asynchronous calls. This Puppeteer tutorial will have all of the examples in async-await syntax.

Simple example of using Puppeteer

Create a new file in your node project directory (the directory that contains package.json and node_modules). Save this file as example1.js and add this code:

```
const puppeteer = require('puppeteer');
async function main() {
    // Add code here
}
main();
```

The code above can be simplified by making the function anonymous and calling it on the same line:

```
const puppeteer = require('puppeteer');
(async () => {
    // Add code here
})();
```

The required keyword will ensure that the Puppeteer library is available in the file. The rest of the lines are the placeholder where an anonymous, asynchronous function is being created and executed. For the next step, launch the browser.

```
const browser = await puppeteer.launch();
```

Note that by default, the browser is launched in the headless mode. If there is an explicit need for a user interface, the above line can be modified to include an object as a parameter.

```
const browser = await puppeteer.launch({headless:false}); //
default is true
```

The next step would be to open a page:

```
const page = await browser.newPage();
```

Now that a page or in other words, a tab, is available, any website can be loaded by simply calling the goto() function:

```
await page.goto('https://oxylabs.io/');
```

Once the page is loaded, the DOM elements, as well the rendered page is available. This can be verified by taking a quick screenshot:

```
await page.screenshot({path: 'oxylabs_1080.png'})
```

This, however, will create only an 800×600 pixel image. The reason is that Puppeteer sets an initial page size to 800×600px. This can be changed by setting the viewport, before taking the screenshot.

```
 await page.setViewport({
    width: 1920,
    height: 1080,
  });
```

Finally, remember to close the browser:

```
await browser.close();
```

Putting it altogether, here is the complete script.

```
const puppeteer = require('puppeteer');
(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.setViewport({
    width: 1920,
    height: 1080,
  });
  await page.goto('https://oxylabs.io/');
  await page.screenshot({path: 'oxylabs_1080.png'})
  await browser.close();
})();
```

Run this file from the terminal using this command:

```
node example1.js
```

This should create a new file oxylabs_1080.png in the same directory.

Bonus tip: If you need a PDF, you can use the pdf() function:

```
await page.pdf({path: 'oxylabs.pdf', format: 'A4'});
```

Scraping an element from a page

Puppeteer loads the complete page in DOM. This means that we can extract any data from the page. The easiest way to do this is to use the function evaluate(). This allows JavaScript functions like document.querySelector(). Consequently, it lets us extract any Element from the DOM.

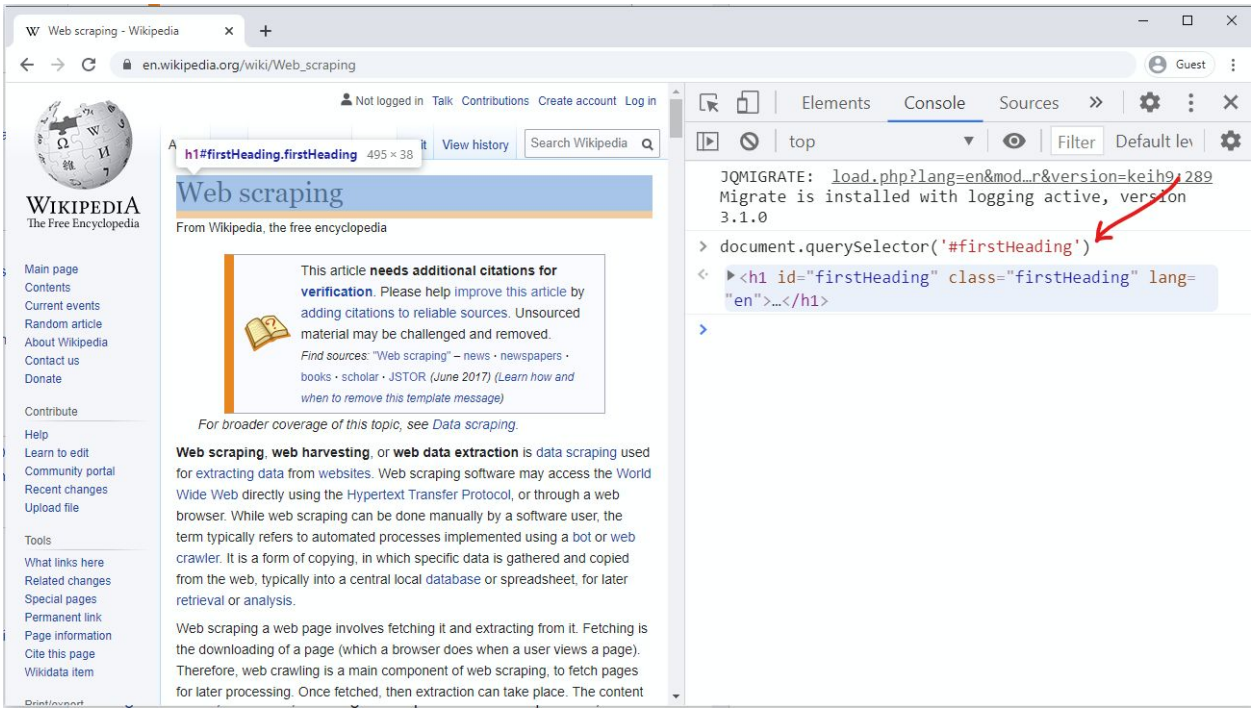
To understand this, open this link in your preferred browser: https://en.wikipedia.org/wiki/Web_scraping

Once the page is loaded, right-click the heading of the page, and select Inspect. This should open developer tools with the Elements tab activated. Here it is visible that the page's heading is in h1 element, with id and class both set to firstHeading.

Now, go to the Console tab in the developer toolbox and write in this line:

```
document.querySelector('#firstHeading')
```

You will immediately see that our desired tag is extracted.



This returns one element from the page. For this particular element, all we need is text. Text can be easily extracted with this line of code:

```
document.querySelector('#firstHeading').textContent
```

The text can now be returned using the return keyword. The next step is to surround this in the evaluate method. This will ensure that this querySelector can be run.

```
await page.evaluate(() => {
    return document.querySelector("#firstHeading").textContent;
});
```

The result of the evaluate() function can be stored in a variable to complete the functionality. Finally, do not forget to close the browser. Here is the complete script:

```
const puppeteer = require("puppeteer");
(async () => {
  const browser = await puppeteer.launch();
```

```
  const page = await browser.newPage();
  await page.goto("https://en.wikipedia.org/wiki/Web_scraping");
  title = await page.evaluate(() => {
    return
document.querySelector("#firstHeading").textContent.trim();
  });
  console.log(title);
  await browser.close();
})();
```

Scraping multiple elements

Extracting multiple elements would involve three steps:

1. Use of querySelectorAll to get all elements matching the selector:

```
headings_elements = document.querySelectorAll("h2
.mw-headline");
```

2. create an array, as heading_elements is of type NodeList.

```
headings_array = Array.from(headings_elements);
```

3. Call the map() function can be called to process each element in the array and return it.

```
return headings_array.map(heading => heading.textContent);
```

This of course needs to be surrounded by page.evaluate() function. Putting everything together, this is the complete script. You can save this as wiki_toc.js:

```
const puppeteer = require("puppeteer");
```

```
(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto("https://en.wikipedia.org/wiki/Web_scraping");

  headings = await page.evaluate(() => {
    headings_elements = document.querySelectorAll("h2
.mw-headline");
    headings_array = Array.from(headings_elements);
    return headings_array.map(heading => heading.textContent);
  });
  console.log(headings);
  await browser.close();
})();
```

This file can now be run from your terminal:

```
node wiki_toc.js
```

Bonus tip: Array.from() function can be supplied with a map function directly, without a separate call to map. Depending on the comfort level, the same code can thus be written as:

```
headings = await page.evaluate(() => {
    return Array.from(document.querySelectorAll("h2
.mw-headline"),
      heading => heading.innerText.trim());
  });
```

Scraping a hotel listing page

This section will explain how a typical listing page can be scraped to get a JSON object with all the required information. The concepts presented in this section will be applicable for any listing, whether it is an online store, a directory, or a hotel listing.

The example that we will take is an Airbnb. Apply some filters so that you reach a page similar to the one in the screenprint. In this particular example, we will be scraping this Airbnb page that lists 20 hotels. To scrape all 20 hotels, the first step is to identify the selector for each hotel section.

```
root = Array.from(document.querySelectorAll("#FMP-target
[itemprop='itemListElement']"));
```

This returns a NodeList of length 20 and stores in the variable root. Note that so far, text or any attribute has not been extracted. All we have is an array of elements. This will be done in the map() function.

```
hotels = root.map(hotel => ({
// code here
}));
```

The URL of the photo of the hotel can be extracted with a code like this:

```
hotel.querySelector("img").getAttribute("src")
```

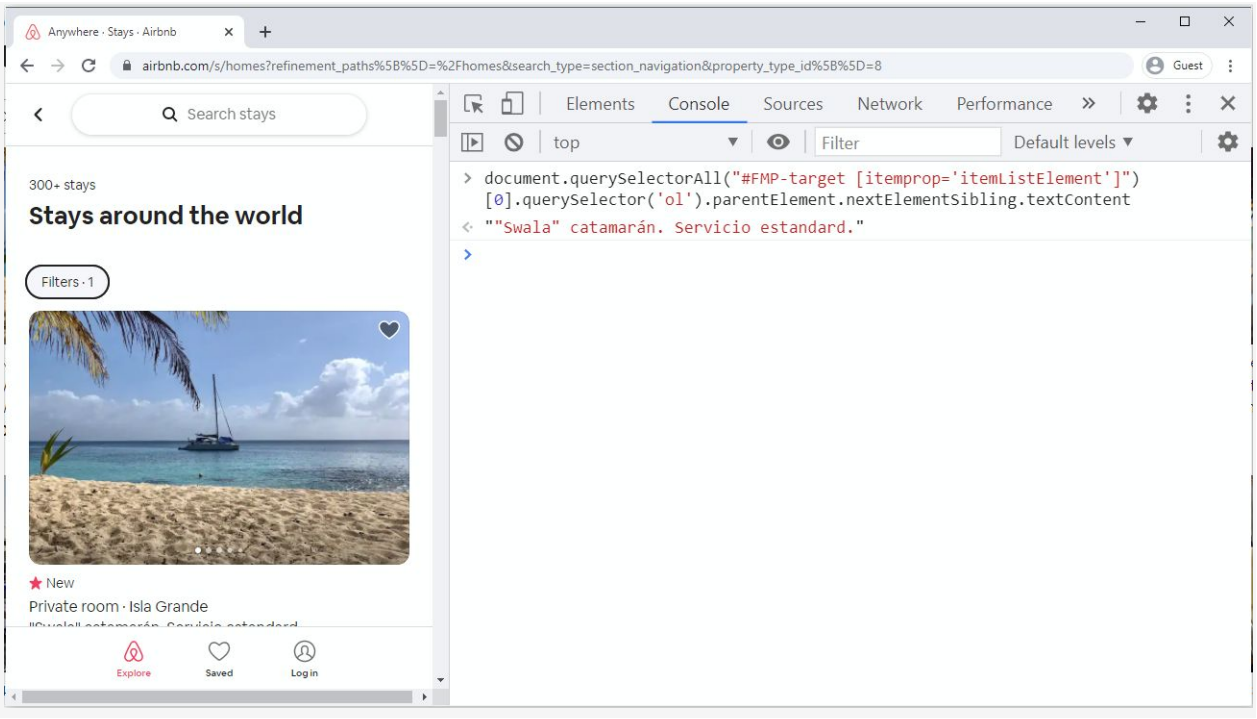
Getting the name of the hotel is a little trickier. The classes used on this page are some random words like _krjbj and _mvzr1f2. These class names appear to be generated dynamically and may change later on. It is better to have selectors which do not rely on these class names.

The hotel name can be extracted by combining parentElement and nextElementSibling selectors:

```
hotel.querySelector('ol').parentElement.nextElementSibling.textC
ontent
```

The most important concept to understand here is that we are concatenating querySelectors. Effectively, the first hotel name is being extracted with this line of code:

```
document.querySelectorAll("#FMP-target
[itemprop='itemListElement']")[0].querySelector('ol').parentElem
ent.nextElementSibling.textContent
```



Finally, we can create an object containing both of these values. The syntax to create an object is like this:

```
Hotel = {
        Name: 'x',
        Photo: 'y'
      }
```

Putting everything together, here is the final script. Save it as bnb.js.

```
const puppeteer = require("puppeteer");
(async () => {
```

```
  let url =
"https://www.airbnb.com/s/homes?refinement_paths%5B%5D=%2Fhomes&
search_type=section_navigation&property_type_id%5B%5D=8";
  const browser = await puppeteer.launch(url);
  const page = await browser.newPage();
  await page.goto(url);
  data = await page.evaluate(() => {
    root = Array.from(document.querySelectorAll("#FMP-target
[itemprop='itemListElement']"));
    hotels = root.map(hotel => ({
      Name:
hotel.querySelector('ol').parentElement.nextElementSibling.textC
ontent,
      Photo: hotel.querySelector("img").getAttribute("src")
    }));
    return hotels;
  });
  console.log(data);
  await browser.close();
})();
```

Run this file from the terminal using:

```
node bnb.js
```

You should be able to see a JSON object printed on the console.

We recommend that you look at the official Puppeteer documentation for more detailed information.

Web Scraping With Selenium

How does Selenium work? It automates your written script processes, as the script needs to interact with a browser to perform repetitive tasks like clicking, scrolling, etc. As described on Selenium's official webpage, it is "primarily for automating web applications for testing purposes, but is certainly not limited to just that."

In this guide, on how to web scrape with Selenium, we will be using Python 3.x. as our main input language (as it is not only the most common scraping language but the one we closely work with as well).

Setting up Selenium

Firstly, to download the Selenium package, execute the pip command in your terminal:

```
pip install selenium
```

You will also need to install Selenium drivers, as it enables python to control the browser on OS-level interactions. This should be accessible via the PATH variable if doing a manual installation.

You can download the drivers for Firefox, Chrome, and Edge from here.

Quick starting Selenium

Let's begin the automatization by starting up your browser:

- Open up a new browser window (in this instance, Firefox)

- Load the page of your choice (our provided URL)

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://oxylabs.io/')
```

This will launch it in the headful mode. In order to run your browser in headless mode and run it on a server, it should look something like this:

```
from selenium import webdriver
from selenium.webdriver.firefox.options import Options

options = Options()
options.headless = True
options.add_argument("--window-size=1920,1200")

driver = webdriver.firefox(options=options,
executable_path=DRIVER_PATH)
driver.get("https://www.oxylabs.io/")
print(driver.page_source)
driver.quit()
```

Data extraction with Selenium by locating elements

find_element

Selenium offers a variety of functions to help locate elements on a page:

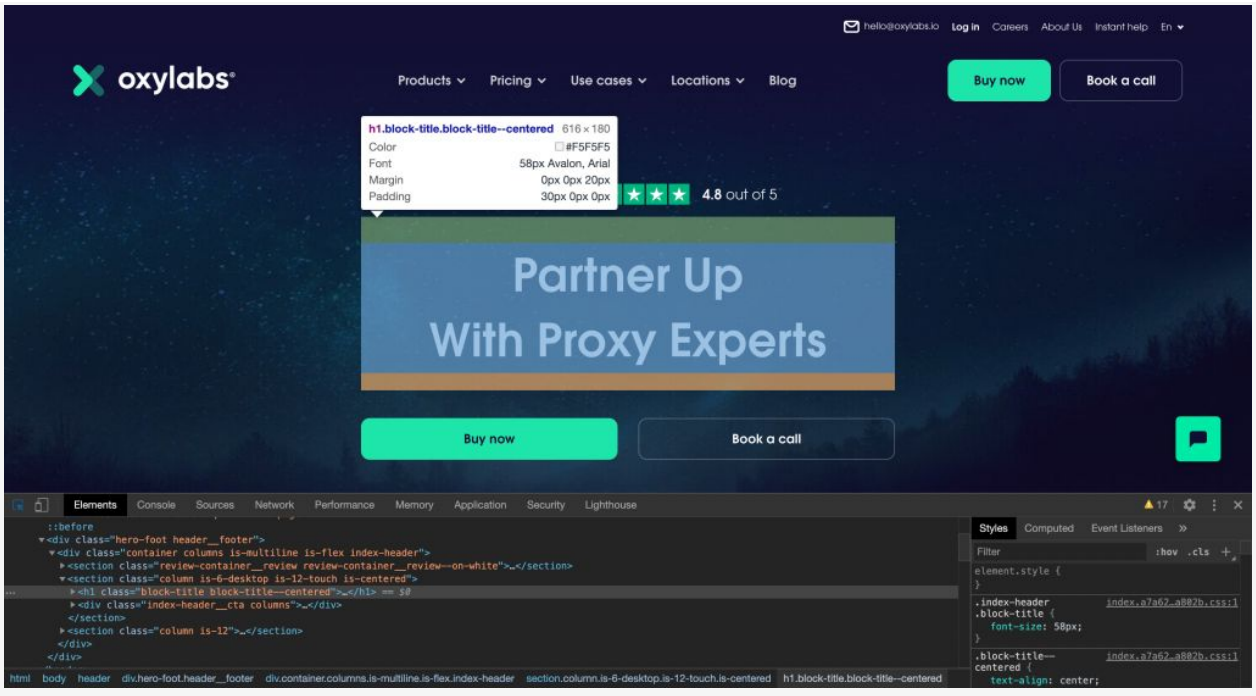
- find_element_by_id
- find_element_by_name
- find_element_by_xpath

- find_element_by_link_text (find element by using text value)
- find_element_by_partial_link_text (find element by matching some part of a hyperlink text(anchor tag))
- find_element_by_tag_name
- find_element_by_class_name
- find_element_by_css_selector (find element by using a CSS selector for id class)

As an example, let's try and locate the H1 tag on oxylabs.io homepage with Selenium:

```
<html>
    <head>
        ... something
    </head>
    <body>
        <h1 class="someclass" id="greatID"> Partner Up With
Proxy Experts</h1>
    </body>
</html>

h1 = driver.find_element_by_name('h1')
h1 = driver.find_element_by_class_name('someclass')
h1 = driver.find_element_by_xpath('//h1')
h1 = driver.find_element_by_id('greatID')
```



You can also use the find_elements (plural form) to return a list of elements. E.g.:

```
all_links = driver.find_elements_by_tag_name('a')
```

This way, you'll get all anchors in the page.

However, some elements are not easily accessible with an ID or a simple class. This is why you will need XPath.

XPath

XPath is a syntax language that helps find a specific object in DOM. XPath syntax finds the node from the root element either through an absolute path or by using a relative path. e.g.:

- **/** : Select node from the root. /html/body/div[1] will find the first div

- **//**: Select node from the current node no matter where they are. //form[1] will find the first form element

- **[@attributename='value']**: a predicate. It looks for a specific node or a node with a specific value.

Example:

```
//input[@name='email'] will find the first input element with
the name "email".

<html>
 <body>
   <div class = "content-login">
     <form id="loginForm">
         <div>
            <input type="text" name="email" value="Email
Address:">
            <input type="password"
name="password"value="Password:">
         </div>
        <button type="submit">Submit</button>
     </form>
   </div>
 </body>
</html>
```

WebElement

WebElement in Selenium represents an HTML element. Here are the most commonly used actions:

- element.text (accessing text element)

- element.click() (clicking on the element)

- element.get_attribute('class') (accessing attribute)
- element.send_keys('mypassword') (sending text to an input)

Slow website render solutions

Some websites use a lot of JavaScript to render content, and they can be tricky to deal with as they use a lot of AJAX calls. There are a few ways to solve this:

- time.sleep(ARBITRARY_TIME)
- WebDriverWait()

Example:

```
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "mySuperId"))
    )
finally:
   driver.quit()
```

This will allow the located element to be loaded after 10 seconds. To dig deeper into this topic, go ahead and check out the official Selenium documentation.

Selenium vs Puppeteer

The biggest reason for Selenium's popularity and complexity is that it supports writing tests in multiple programming languages. This includes C#, Groovy, Java, Perl, PHP, Python, Ruby, Scala, and even

JavaScript. It supports multiple browsers, including Chrome, Firefox, Edge, Internet Explorer, Opera, and Safari.

However, for web scraping tasks, Selenium is perhaps more complex than it needs to be. Remember that Selenium's real purpose is functional testing. For effective functional testing, it mimics what a human would do in a browser. Selenium thus needs three different components:

- A driver for each browser
- Installation of each browser
- The package/library depending on the programming language used

In the case of Puppeteer, though, the node package puppeteer includes Chromium. It means no browser or driver is needed. It makes it simpler. It also supports Chrome if that is what you need.

On the other hand, multiple browser support is missing. Firefox support is limited. Google announced Puppeteer for Firefox, but it was soon deprecated. As wehn writing this, Firefox support is experimental. So, to sum up, if you need a lightweight and fast headless browser for web scraping, Puppeteer would be the better choice.

Web Scraping With lxml

Prerequisite

This tutorial is aimed at developers who have at least a basic understanding of Python. A basic understanding of XML and HTML is also required. Simply put, if you know what an attribute is in XML, that is enough to understand this article.

This tutorial uses Python 3 code snippets but everything works on Python 2 with minimal changes as well.

What is lxml in Python?

lxml is one of the fastest and feature-rich libraries for processing XML and HTML in Python. This library is essentially a wrapper over C libraries *libxml2* and *libxslt*. This combines the speed of the native C library and the simplicity of Python.

Using Python lxml library, XML and HTML documents can be created, parsed, and queried. It is a dependency on many of the other complex packages like Scrapy.

Installation

The best way to download and install the lxml library is from Python Package Index (PyPI). If you are on Linux (debian-based), simply run:

```
sudo apt-get install python3-lxml
```

Another way is to use the pip package manager. This works on Windows, Mac, and Linux:

```
pip3 install lxml
```

On windows, just use *pip install lxml*, assuming you are running Python 3.

Creating a simple XML document

Any XML or any XML compliant HTML can be visualized as a tree. A tree has a root and branches. Each branch optionally may have further branches. All these branches and the root are represented as an *Element*.

A very simple XML document would look like this:

```
<root>
    <branch>
        <branch_one>
        </branch_one>
        <branch_one>
        </branch_one >
    </branch>
</root>
```

If an HTML is XML compliant, it will follow the same concept.

Note that HTML may or may not be XML compliant. For example, if an HTML has
 without a corresponding closing tag, it is still valid HTML, but it will not be a valid XML. In the later part of this tutorial, we will see how these cases can be handled. For now, let's focus on XML compliant HTML.

The Element class

To create an XML document using python lxml, the first step is to import the *etree* module of lxml:

```
>>> from lxml import etree
```

Every XML document begins with the root element. This can be created using the *Element* type. The Element type is a flexible container object which can store hierarchical data. This can be described as a cross between a dictionary and a list.

In this python lxml example, the objective is to create an HTML, which is XML compliant. It means that the root element will have its name as html:

```
>>> root = etree.Element("html")
```

Similarly, every html will have a head and a body:

```
>>> head = etree.Element("head")
>>> body = etree.Element("body")
```

To create parent-child relationships, we can simply use the append() method.

```
>>> root.append(head)
>>> root.append(body)
```

This document can be serialized and printed to the terminal with the help of *tostring()* function. This function expects one mandatory argument, which is the root of the document. We can optionally set *pretty_print* to True to make the output more readable. Note that

tostring() serializer actually returns bytes. This can be converted to string by calling *decode()*:

```
>>> print(etree.tostring(root, pretty_print=True).decode())
```

The SubElement class

Creating an *Element* object and calling the *append()* function can make the code messy and unreadable. The easiest way is to use the *SubElement* type. Its constructor takes two arguments – the parent node and the element name. Using *SubElemen*t, the following two lines of code can be replaced by just one.

```
body = etree.Element("body")
root.append(body)
# is same as
body = etree.SubElement(root,"body")
```

Setting text and attributes

Setting text is very easy with the lxml library. Every instance of the *Element* and *SubElement* exposes two methods – text and set, the former is used to specify the text and later is used to set the attributes. Here are the examples:

```
para = etree.SubElement(body, "p")
para.text="Hello World!"
```

Similarly, attributes can be set using key-value convention:

```
para.set("style", "font-size:20pt")
```

One thing to note here is that the attribute can be passed in the constructor of SubElement:

```
para = etree.SubElement(body, "p", style="font-size:20pt",
id="firstPara")
para.text = "Hello World!"
```

The benefit of this approach is saving lines of code and clarity. Here is the complete code. Save it in a python file and run it. It will print an HTML which is also a well-formed XML.

```
from lxml import etree

root = etree.Element("html")
head = etree.SubElement(root, "head")
title = etree.SubElement(head, "title")
title.text = "This is Page Title"
body = etree.SubElement(root, "body")
heading = etree.SubElement(body, "h1", style="font-size:20pt",
id="head")
heading.text = "Hello World!"
para = etree.SubElement(body, "p",  id="firstPara")
para.text = "This HTML is XML Compliant!"
para = etree.SubElement(body, "p",  id="secondPara")
para.text = "This is the second paragraph."

etree.dump(root)  # prints everything to console. Use for debug
only
```

Note that here we used *etree.dump()* instead of calling *etree.tostring()*. The difference is that *dump()* simply writes everything to the console and doesn't return anything, *tostring()* is used for serialization and returns a string which you can store in a variable or write to a file. *dump()* is good for debug only and should not be used for any other purpose.

Add the following lines at the bottom of the snippet and run it again:

```
with open('input.html', 'wb') as f:
    f.write(etree.tostring(root, pretty_print=True)
```

This will save the contents to input.html in the same folder you were running the script. Again, this is a well-formed XML, which can be interpreted as XML or HTML.

How do you parse an XML file using LXML in Python?

The previous section was a Python lxml tutorial on creating XML files. In this section, we will look at traversing and manipulating an existing XML document using the lxml library.

Before we move on, save the following snippet as input.html.

```
<html>
  <head>
    <title>This is Page Title</title>
  </head>
  <body>
    <h1 style="font-size:20pt" id="head">Hello World!</h1>
    <p id="firstPara">This HTML is XML Compliant!</p>
    <p id="secondPara">This is the second paragraph.</p>
  </body>
</html>
```

When an XML document is parsed, the result is an in-memory ElementTree object.

The raw XML contents can be in a file system or a string. If it is in a file system, it can be loaded using the *parse* method. Note that the *parse* method will return an object of type ElementTree. To get the root element, simply call the *getroot()* method.

```
from lxml import etree

tree = etree.parse('input.html')
elem = tree.getroot()
etree.dump(elem) #prints file contents to console
```

The lxml.etree module exposes another method that can be used to parse contents from a valid xml string — *fromstring()*

```
xml = '<html><body>Hello</body></html>'
root = etree.fromstring(xml)
etree.dump(root)
```

One important difference to note here is that *fromstring()* method returns an object of element. There is no need to call *getroot()*.

If you want to dig deeper into parsing, we have already written a tutorial on BeautifulSoup, a Python package used for parsing HTML and XML documents. But to quickly answer what is lxml in BeautifulSoup, lxml can use BeautifulSoup as a parser backend. Similarly, BeautifulSoup can employ lxml as a parser.

Finding elements in XML

Broadly, there are two ways of finding elements using the Python lxml library. The first is by using the Python lxml querying languages: XPath and ElementPath. For example, the following code will return the first paragraph element.

Note that the selector is very similar to XPath. Also note that the root element name was not used because elem contains the root of the XML tree.

```
tree = etree.parse('input.html')
elem = tree.getroot()
para = elem.find('body/p')
etree.dump(para)

# Output
# <p id="firstPara">This HTML is XML Compliant!</p>
```

Similarly, *findall()* will return a list of all the elements matching the selector.

```
elem = tree.getroot()
para = elem.findall('body/p')
for e in para:
    etree.dump(e)

# Outputs
# <p id="firstPara">This HTML is XML Compliant!</p>
# <p id="secondPara">This is the second paragraph.</p>
```

The second way of selecting the elements is by using XPath directly. This approach is easier to follow by developers who are familiar with XPath. Furthermore, XPath can be used to return the instance of the element, the text, or the value of any attribute using standard XPath syntax.

```
para = elem.xpath('//p/text()')
for e in para:
    print(e)

# Output
# This HTML is XML Compliant!
# This is the second paragraph.
```

Handling HTML with lxml.html

Throughout this article, we have been working with a well-formed HTML which is XML compliant. This will not be the case a lot of the time. For these scenarios, you can simply use lxml.html instead of *lxml.etree*.

Note that reading directly from a file is not supported. The file contents should be read in a string first. Here is the code to print all paragraphs from the same HTML file.

```
from lxml import html
```

```
with open('input.html') as f:
    html_string = f.read()
tree = html.fromstring(html_string)
para = tree.xpath('//p/text()')
for e in para:
    print(e)

# Output
# This HTML is XML Compliant!
# This is the second paragraph
```

lxml web scraping tutorial

Now that we know how to parse and find elements in XML and HTML, the only missing piece is getting the HTML of a web page.

For this, the 'requests' library is a great choice. It can be installed using the pip package manager:

```
pip install requests
```

Once the requests library is installed, HTML of any web page can be retrieved using a simple get() method. Here is an example.

```
import requests

response = requests.get('http://books.toscrape.com/')
print(response.text)
# prints source HTML
```

This can be combined with lxml to retrieve any data that is required.

Here is a quick example that prints a list of countries from Wikipedia:

```
import requests
from lxml import html
```

```
response =
requests.get('https://en.wikipedia.org/wiki/List_of_countries_by
_population_in_2010')

tree = html.fromstring(response.text)
countries = tree.xpath('//span[@class="flagicon"]')
for country in countries:
    print(country.xpath('./following-sibling::a/text()')[0])
```

In this code, the HTML returned by response.text is parsed into the variable tree. This can be queried using standard XPath syntax. The XPaths can be concatenated. Note that the *xpath()* method returns a list and thus only the first item is taken in this code snippet.

This can easily be extended to read any attribute from the HTML. For example, the following modified code prints the country name and image URL of the flag.

```
for country in countries:
    flag = country.xpath('./img/@src')[0]
    country = country.xpath('./following-sibling::a/text()')[0]
    print(country, flag)
```

Python lxml library is a light-weight, fast, and feature-rich library. This can be used to create XML documents, read existing documents, and find specific elements. This makes this library equally powerful for both XML and HTML documents. Combined with requests library, it can also be easily used for web scraping.

Using Beautiful Soup to Parse Data

This tutorial is useful for those seeking to quickly grasp the value that Python and Beautiful Soup v4 offers. After following the provided examples you should be able to understand the basic principles of how to parse HTML data. The examples will demonstrate traversing a document for HTML tags, printing the full content of the tags, finding elements by ID, extracting text from specified tags and exporting it to a .csv file.

What is Beautiful Soup?

Beautiful Soup is a Python package for parsing HTML and XML documents. It creates a parse tree for parsed pages based on specific criteria that can be used to extract, navigate, search and modify data from HTML, which is mostly used for web scraping. It is available for Python 2.7 and Python 3. A useful library, it can save programmers loads of time.

Installing Beautiful Soup

Before working on this tutorial, you should have a Python programming environment set up on your machine. For this tutorial we will assume that PyCharm is used since it's a convenient choice even for the less experienced with Python and is a great starting point. Otherwise, simply use your go-to IDE.

On Windows, when installing Python make sure to tick the "PATH installation" checkbox. PATH installation adds executables to the

default Windows Command Prompt executable search. Windows will then recognize commands like “pip” or “python” without having to point to the directory of the executable which makes things more convenient.

You should also have Beautiful Soup installed on your system. No matter the OS, you can easily do it by using this command on the terminal to install the current latest version of Beautiful Soup:

```
pip install BeautifulSoup4
```

If you are using Windows, it is recommended to run terminal as administrator to ensure that everything works out smoothly.

Finally, since we will be working with a sample file written in HTML, you should be at least somewhat familiar with HTML structure.

Getting started

A sample HTML file will help demonstrate the main methods of how Beautiful Soup parses data. This file is much more simple than your average modern website, however, it will be sufficient for the scope of this tutorial.

```
<!DOCTYPE html>
<html>
    <head>
        <title>What is a Proxy?</title>
        <meta charset="utf-8">
    </head>

    <body>
        <h2>Proxy types</h2>
```

```
        <p>
There are many different ways to categorize proxies. However,
two of the most popular types are residential and data center
proxies. Here is a list of the most common types.
        </p>

        <ul id="proxytypes">
            <li>Residential proxies</li>
            <li>Datacenter proxies</li>
            <li>Shared proxies</li>
            <li>Semi-dedicated proxies</li>
            <li>Private proxies</li>
      </ul>

    </body>
</html>
```

For PyCharm to use this file, simply copy it to any text editor and save it with the .html extension to the directory of your PyCharm project.

Going further, open PyCharm and after a right click on the project area navigate to New -> Python File. Congratulations and welcome to your new playground!

Traversing for HTML tags

First, we can use Beautiful Soup to extract a list of all the tags used in our sample HTML file. For this, we will use the *soup.descendants* generator.

```
from bs4 import BeautifulSoup

with open('index.html', 'r') as f:
    contents = f.read()

    soup = BeautifulSoup(contents, features="html.parser")
```

```
    for child in soup.descendants:

        if child.name:
            print(child.name)
```

After running this code (right click on code and click “Run”) you should get the below output:

```
html
head
title
meta
body
h2
p
ul
li
li
li
li
li
```

What just happened? Beautiful Soup traversed our HTML file and printed all the HTML tags that it has found sequentially. Let’s take a quick look at what each line did.

```
from bs4 import BeautifulSoup
```

This tells Python to use the Beautiful Soup library.

```
with open('index.html', 'r') as f:
    contents = f.read()
```

And this code, as you could probably guess, gives an instruction to open our sample HTML file and read its contents.

```
   soup = BeautifulSoup(contents, features="html.parser")
```

This line creates a BeautifulSoup object and passes it to Python's built in HTML parser. Other parsers, such as lxml, might also be used, but it is a separate external library and for the purpose of this tutorial the built-in parser will do just fine.

```
for child in soup.descendants:
    if child.name:
        print(child.name)
```

The final pieces of code, namely the *soup.descendants* generator, instruct Beautiful Soup to look for HTML tags and print them in the PyCharm console. The results can also easily be exported to a .csv file but we will get to this later.

Getting the full content of tags

To get the content of tags, this is what we can do:

```
from bs4 import BeautifulSoup

with open('index.html', 'r') as f:
    contents = f.read()

    soup = BeautifulSoup(contents, features="html.parser")

    print(soup.h2)
    print(soup.p)
    print(soup.li)
```

This is a simple instruction that outputs the HTML tag with its full content in the specified order. Here's what the output should look like:

```
<h2>Proxy types</h2>
<p>
          There are many different ways to categorize proxies.
However, two of the most popular types are residential and data
center proxies. Here is a list of the most common types.
```

```
        </p>
<li>Residential proxies</li>
```

You could also remove the HTML tags and print text only, by using, for example:

```
print(soup.li.text)
```

Which in our case will give the following output:

```
Residential proxies
```

Note that this only prints the first instance of the specified tag. Let's continue to see how to find elements by ID or using the *find_all* method to filter elements by specific criteria.

Using Beautiful Soup to find elements by ID

We can use two similar ways to find elements by ID:

```
print(soup.find('ul', attrs={'id': 'proxytypes'}))
```

or

```
print(soup.find('ul', id='proxytypes'))
```

Both of these will output the same result in the Python Console:

```
<ul id="proxytypes">
<li>Residential proxies</li>
<li>Datacenter proxies</li>
<li>Shared proxies</li>
<li>Semi-dedicated proxies</li>
<li>Private proxies</li>
```

```
</ul>
```

Finding all specified tags and extracting text

The *find_all* method is a great way to extract specific data from an HTML file. It accepts many criteria that make it a flexible tool allowing us to filter data in convenient ways. Yet for this tutorial we do not need anything more complex. Let's find all items of our list and print them as text only:

```
  for tag in soup.find_all('li'):
        print(tag.text)
```

This is how the full code should look like:

```
from bs4 import BeautifulSoup
with open('index.html', 'r') as f:
    contents = f.read()
    soup = BeautifulSoup(contents, features="html.parser")
    for tag in soup.find_all('li'):
        print(tag.text)
```

And here's the output:

```
Residential proxies
Datacenter proxies
Shared proxies
Semi-dedicated proxies
Private proxies
```

Congratulations, you should now have a basic understanding of how Beautiful Soup might be used to parse data. It should be noted that the information presented in this article is useful as introductory material yet real-world web scraping with BeautifulSoup and the consequent parsing of data is usually much more complicated than this. For a more in-depth look at Beautiful Soup you will hardly find a better source than its documentation, so be sure to check it out too.

As you can see, Beautiful Soup is a greatly useful HTML parser. With a relatively low learning curve, you can quickly grasp how to navigate, search, and modify the parse tree. With the addition of libraries such as pandas you can further manipulate and analyze the data which offers a powerful package for a near infinite amount of data collection and analysis use cases.

oxylabs®

Want to know more?

If you would like to know more about any of the topics mentioned in this compendium or learn about our products, please get in touch! Our team is ready to answer any of your questions and offer you the best solution for your business needs.

Get in touch with Oxylabs

Our Mission

Our mission is to share all the know-how that we collected over the years in the industry in order to create the future where big data is accessible to all businesses. We seek to create a healthy environment for everyone to grow and thrive in.

Our Values

As a leading company in the proxy and web scraping industry, we ensure that the highest standards of business ethics lead all our operations. Our core values guide us toward achieving our mission. Learn more