Ari Bajo

Ari is an expert Data Engineer and a talented technical writer. He wrote the entire Scrapy integration for ScrapingBee and this awesome article.

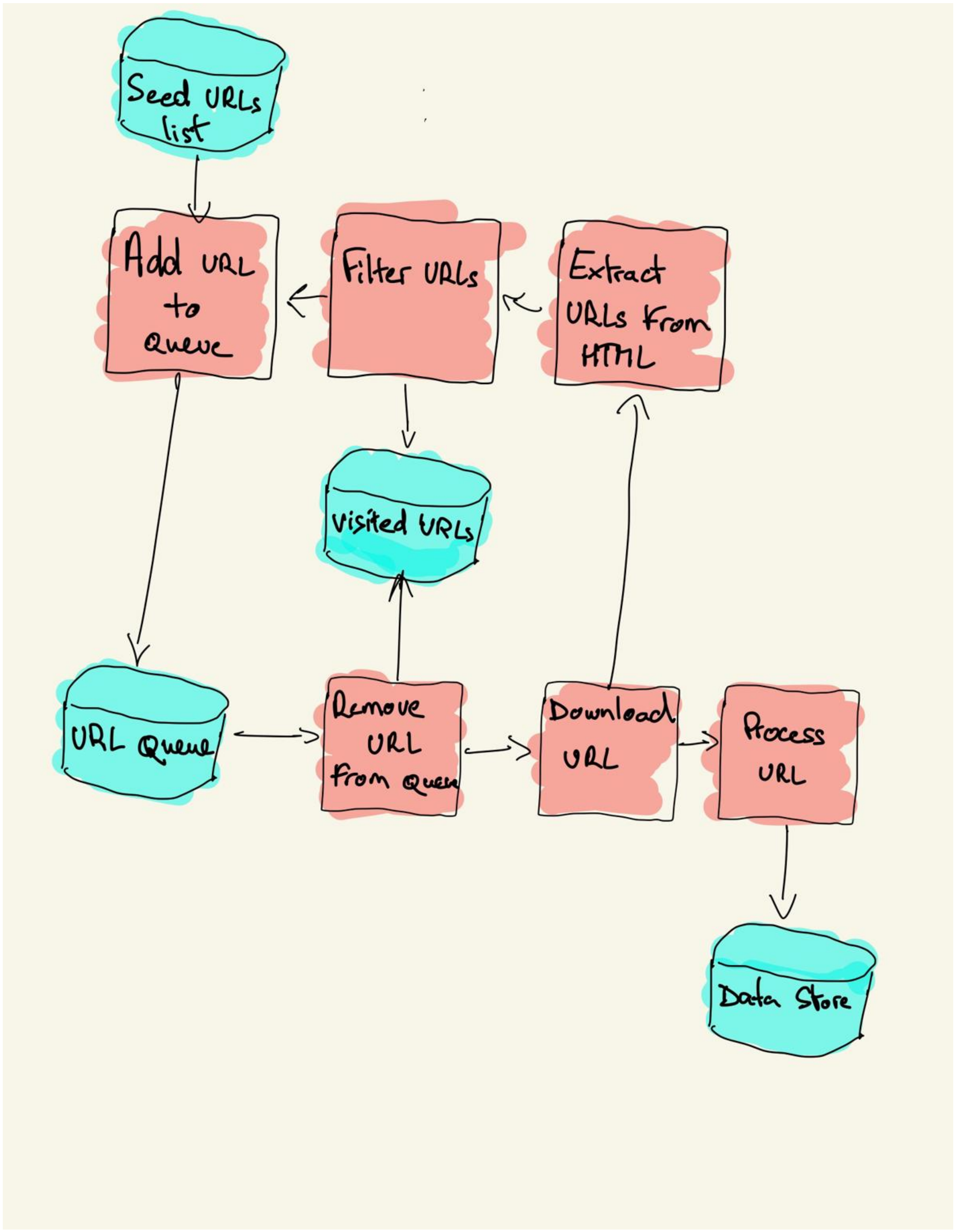
Web crawling is a powerful technique to collect data from the web by finding all the URLs for one or multiple domains. Python has several popular web crawling libraries and frameworks.

In this article, we will first introduce different crawling strategies and use cases. Then we will build a simple web crawler from scratch in Python using two libraries: requests and Beautiful Soup. Next, we will see why it's better to use a web crawling framework like Scrapy. Finally, we will build an example crawler with Scrapy to collect film metadata from IMDb and see how Scrapy scales to websites with several million pages.

What is a web crawler?

Web crawling and web scraping are two different but related concepts. Web crawling is a component of web scraping, the crawler logic finds URLs to be processed by the scraper code.

A web crawler starts with a list of URLs to visit, called the seed. For each URL, the crawler finds links in the HTML, filters those links based on some criteria and adds the new links to a queue. All the HTML or some specific information is extracted to be processed by a different pipeline.


Seed URLs list
Add URL to Queue
Filter URLs
Extract URLs From HTML
Visited URLs
URL Queue
Remove URL from Queue
Download URL
Process URL
Data Store

Web crawling strategies

In practice, web crawlers only visit a subset of pages depending on the crawler budget, which can be a maximum number of pages per domain, depth or execution time.

Most popular websites provide a robots.txt file to indicate which areas of the website are disallowed to crawl by each user agent. The opposite of the robots file is the sitemap.xml file, that lists the pages that can be crawled.

Popular web crawler use cases include:

- Search engines (Googlebot, Bingbot, Yandex Bot...) collect all the HTML for a significant part of the Web. This data is indexed to make it searchable.
- SEO analytics tools on top of collecting the HTML also collect metadata like the response time, response status to detect broken pages and the links between different domains to collect backlinks.
- Price monitoring tools crawl e-commerce websites to find product pages and extract metadata, notably the price. Product pages are then periodically revisited.
- Common Crawl maintains an open repository of web crawl data. For example, the archive from October 2020 contains 2.71 billion web pages.

Next, we will compare three different strategies for building a web crawler in Python. First, using only standard libraries, then third party libraries for making HTTP requests and parsing HTML and finally, a web crawling framework.

Building a simple web crawler in Python from scratch

To build a simple web crawler in Python we need at least one library to download the

HTML from a URL and an HTML parsing library to extract links. Python provides standard libraries urllib for making HTTP requests and html.parser for parsing HTML. An example Python crawler built only with standard libraries can be found on Github.

The standard Python libraries for requests and HTML parsing are not very developer-friendly. Other popular libraries like requests, branded as HTTP for humans, and Beautiful Soup provide a better developer experience.

If you wan to learn more, you can check this guide about the best Python HTTP client.

You can install the two libraries locally.

```
pip install requests bs4
```

A basic crawler can be built following the previous architecture diagram.

```
import logging
from urllib.parse import urljoin
import requests
from bs4 import BeautifulSoup

logging.basicConfig(
    format='%(asctime)s %(levelname)s:%(message)s',
    level=logging.INFO)

class Crawler:

    def __init__(self, urls=[]):
        self.visited_urls = []
        self.urls_to_visit = urls

    def download_url(self, url):
        return requests.get(url).text

    def get_linked_urls(self, url, html):
        soup = BeautifulSoup(html, 'html.parser')
        for link in soup.find_all('a'):
            path = link.get('href')
            if path and path.startswith('/'):
                path = urljoin(url, path)
            yield path

    def add_url_to_visit(self, url):
        if url not in self.visited_urls and url not in self.urls_to_visit:
            self.urls_to_visit.append(url)

    def crawl(self, url):
        html = self.download_url(url)
        for url in self.get_linked_urls(url, html):
            self.add_url_to_visit(url)

    def run(self):
        while self.urls_to_visit:
            url = self.urls_to_visit.pop(0)
            logging.info(f'Crawling: {url}')
            try:
                self.crawl(url)
            except Exception:
                logging.exception(f'Failed to crawl: {url}')
            finally:
                self.visited_urls.append(url)

if __name__ == '__main__':
    Crawler(urls=['https://www.imdb.com/']).run()
```

The code above defines a Crawler class with helper methods to download_url using the requests library, get_linked_urls using the Beautiful Soup library and add_url_to_visit to filter URLs. The URLs to visit and the visited URLs are stored in two separate lists. You can run the crawler on your terminal.

```
python crawler.py
```

The crawler logs one line for each visited URL.

```
2020-12-04 18:10:10,737 INFO:Crawling: https://www.imdb.com/
2020-12-04 18:10:11,599 INFO:Crawling: https://www.imdb.com/?ref_=nv_home
2020-12-04 18:10:12,868 INFO:Crawling: https://www.imdb.com/calendar/?ref_=nv_mv_ca
2020-12-04 18:10:13,526 INFO:Crawling: https://www.imdb.com/list/ls016522954/?ref_=
2020-12-04 18:10:19,174 INFO:Crawling: https://www.imdb.com/chart/top/?ref_=nv_mv_2
2020-12-04 18:10:20,624 INFO:Crawling: https://www.imdb.com/chart/moviemeter/?ref_=
2020-12-04 18:10:21,556 INFO:Crawling: https://www.imdb.com/feature/genre/?ref_=nv_
```

The code is very simple but there are many performance and usability issues to solve before successfully crawling a complete website.

- The crawler is slow and supports no parallelism. As can be seen from the timestamps, it takes about one second to crawl each URL. Each time the crawler makes a request it waits for the request to be resolved and no work is done in between.
- The download URL logic has no retry mechanism, the URL queue is not a real queue and not very efficient with a high number of URLs.
- The link extraction logic doesn't support standardizing URLs by removing URL query string parameters, doesn't handle URLs starting with #, doesn't support filtering URLs by domain or filtering out requests to static files.
- The crawler doesn't identify itself and ignores the robots.txt file.

Next, we will see how Scrapy provides all these functionalities and makes it easy to extend for your custom crawls.

Web crawling with Scrapy

Scrapy is the most popular web scraping and crawling Python framework with 40k stars on Github. One of the advantages of Scrapy is that requests are scheduled and handled asynchronously. This means that Scrapy can send another request before the previous one is completed or do some other work in between. Scrapy can handle many concurrent requests but can also be configured to respect the websites with custom settings, as we'll see later.

Scrapy has a multi-component architecture. Normally, you will implement at least two different classes: Spider and Pipeline. Web scraping can be thought of as an ETL where you extract data from the web and load it to your own storage. Spiders extract the data and pipelines load it into the storage. Transformation can happen both in spiders and pipelines, but I recommend that you set a custom Scrapy pipeline to transform each item independently of each other. This way, failing to process an item has no effect on other items.

On top of all that, you can add spider and downloader middlewares in between components as it can be seen in the diagram below.

Scrapy architecture diagram

Scrapy Architecture Overview [source]

If you have used Scrapy before, you know that a web scraper is defined as a class that inherits from the base Spider class and implements a parse method to handle each response. If you are new to Scrapy, you can read this article for easy scraping with Scrapy.

```
from scrapy.spiders import Spider

class ImdbSpider(Spider):
    name = 'imdb'
    allowed_domains = ['www.imdb.com']
    start_urls = ['https://www.imdb.com/']

    def parse(self, response):
        pass
```

Scrapy also provides several generic spider classes: CrawlSpider, XMLFeedSpider, CSVFeedSpider and SitemapSpider. The CrawlSpider class inherits from the base Spider class and provides an extra rules attribute to define how to crawl a website. Each rule uses a LinkExtractor to specify which links are extracted from each page. Next, we will see how to use each one of them by building a crawler for IMDb, the Internet Movie Database.

Building an example Scrapy crawler for IMDb

Before trying to crawl IMDb, I checked IMDb robots.txt file to see which URL paths are allowed. The robots file only disallows 26 paths for all user-agents. Scrapy reads the robots.txt file beforehand and respects it when the ROBOTSTXT_OBEY setting is set to true. This is the case for all projects generated with the Scrapy command startproject.

```
scrapy startproject scrapy_crawler
```

This command creates a new project with the default Scrapy project folder structure.

```
scrapy_crawler/

├── scrapy.cfg
└── scrapy_crawler
    ├── __init__.py
    ├── items.py
    ├── middlewares.py
    ├── pipelines.py
    ├── settings.py
    └── spiders
        ├── __init__.py
```

Then you can create a spider in scrapy_crawler/spiders/imdb.py with a rule to extract all links.

```python
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor

class ImdbCrawler(CrawlSpider):
    name = 'imdb'
    allowed_domains = ['www.imdb.com']
    start_urls = ['https://www.imdb.com/']
    rules = (Rule(LinkExtractor()),)
```

You can launch the crawler in the terminal.

```
scrapy crawl imdb --logfile imdb.log
```

You will get lots of logs, including one log for each request. Exploring the logs I noticed that even if we set allowed_domains to only crawl web pages under https://www.imdb.com, there were requests to external domains, such as amazon.com.

```
2020-12-06 12:25:18 [scrapy.downloadermiddlewares.redirect] DEBUG: Redirecting (302
```

IMDb redirects from URLs paths under whitelist-offsite and whitelist to external domains. There is an open Scrapy Github issue that shows that external URLs don't get filtered out when the OffsiteMiddleware is applied before the RedirectMiddleware. To fix this issue, we can configure the link extractor to deny URLs starting with two regular expressions.

```
class ImdbCrawler(CrawlSpider):
    name = 'imdb'
    allowed_domains = ['www.imdb.com']
    start_urls = ['https://www.imdb.com/']
    rules = (
        Rule(LinkExtractor(
            deny=[
                re.escape('https://www.imdb.com/offsite'),
                re.escape('https://www.imdb.com/whitelist-offsite'),
            ],
        )),
    )
```

Rule and LinkExtractor classes support several arguments to filter out URLs. For example, you can ignore specific URL extensions and reduce the number of duplicate URLs by sorting query strings. If you don't find a specific argument for your use case you can pass a custom function to process_links in LinkExtractor or process_values in Rule.

For example, IMDb has two different URLs with the same content.

https://www.imdb.com/name/nm1156914/

https://www.imdb.com/name/nm1156914/?mode=desktop&ref_=m_ft_dsk

To limit the number of crawled URLs, we can remove all query strings from URLs with the url_query_cleaner function from the w3lib library and use it in process_links.

```
from w3lib.url import url_query_cleaner

def process_links(links):
    for link in links:
        link.url = url_query_cleaner(link.url)
        yield link

class ImdbCrawler(CrawlSpider):

    name = 'imdb'
    allowed_domains = ['www.imdb.com']
    start_urls = ['https://www.imdb.com/']
    rules = (
        Rule(LinkExtractor(
            deny=[
                re.escape('https://www.imdb.com/offsite'),
                re.escape('https://www.imdb.com/whitelist-offsite'),
            ],
        ), process_links=process_links),
    )
```

Now that we have limited the number of requests to process, we can add a parse_item method to extract data from each page and pass it to a pipeline to store it. For example, we can either extract the whole response.text to process it in a different pipeline or select the HTML metadata. To select the HTML metadata in the header tag we can code our own XPATHs but I find it better to use a library, extruct, that extracts all metadata from an HTML page. You can install it with pip install extract.

```
import re
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from w3lib.url import url_query_cleaner
import extruct
```

```python
def process_links(links):
    for link in links:
        link.url = url_query_cleaner(link.url)
        yield link

class ImdbCrawler(CrawlSpider):
    name = 'imdb'
    allowed_domains = ['www.imdb.com']
    start_urls = ['https://www.imdb.com/']
    rules = (
        Rule(
            LinkExtractor(
                deny=[
                    re.escape('https://www.imdb.com/offsite'),
                    re.escape('https://www.imdb.com/whitelist-offsite'),
                ],
            ),
            process_links=process_links,
            callback='parse_item',
            follow=True
        ),
    )

    def parse_item(self, response):
        return {
            'url': response.url,
            'metadata': extruct.extract(
                response.text,
                response.url,
                syntaxes=['opengraph', 'json-ld']
            ),
        }
```

I set the follow attribute to True so that Scrapy still follows all links from each response even if we provided a custom parse method. I also configured extruct to extract only Open Graph metadata and JSON-LD, a popular method for encoding linked data using JSON in the Web, used by IMDb. You can run the crawler and store items in JSON lines format to a file.

```
scrapy crawl imdb --logfile imdb.log -o imdb.jl -t jsonlines
```

The output file imdb.jl contains one line for each crawled item. For example, the extracted Open Graph metadata for a movie taken from the <meta> tags in the HTML looks like this.

```
{
    "url": "http://www.imdb.com/title/tt2442560/",
    "metadata": {"opengraph": [{
         "namespace": {"og": "http://ogp.me/ns#"},
         "properties": [
             ["og:url", "http://www.imdb.com/title/tt2442560/"],
             ["og:image", "https://m.media-amazon.com/images/M/MV5BMTkzNjEzMDEzMF5B
             ["og:type", "video.tv_show"],
             ["og:title", "Peaky Blinders (TV Series 2013\u2013 ) - IMDb"],
             ["og:site_name", "IMDb"],
             ["og:description", "Created by Steven Knight.  With Cillian Murphy, Pa
        ]
   }]}
}
```

The JSON-LD for a single item is too long to be included in the article, here is a sample of what Scrapy extracts from the <script type="application/ld+json"> tag.

```
"json-ld": [
    {
        "@context": "http://schema.org",
        "@type": "TVSeries",
        "url": "/title/tt2442560/",
        "name": "Peaky Blinders",
        "image": "https://m.media-amazon.com/images/M/MV5BMTkzNjEzMDEzMF5BMl5BanBn
        "genre": ["Crime","Drama"],
        "contentRating": "TV-MA",
        "actor": [
            {
                "@type": "Person",
                "url": "/name/nm0614165/",
                "name": "Cillian Murphy"
            },
            ...
        ]
        ...
    }
]
```

Exploring the logs, I noticed another common issue with crawlers. By sequentially clicking on filters, the crawler generates URLs with the same content, only that the filters were applied in a different order.

https://www.imdb.com/name/nm2900465/videogallery/content_type-trailer/related_titles-tt0479468

https://www.imdb.com/name/nm2900465/videogallery/related_titles-tt0479468/content_type-trailer

Long filter and search URLs is a difficult problem that can be partially solved by limiting the length of URLs with a Scrapy setting, URLLENGTH_LIMIT.

I used IMDb as an example to show the basics of building a web crawler in Python. I didn't let the crawler run for long as I didn't have a specific use case for the data. In case you need specific data from IMDb, you can check the IMDb Datasets project that provides a daily export of IMDb data and IMDbPY, a Python package for retrieving and managing the data.

Web crawling at scale

If you attempt to crawl a big website like IMDb, with over 45M pages based on Google, it's important to crawl responsibly by configuring the following settings. You can identify your crawler and provide contact details in the BOT_NAME setting. To limit the pressure you put on the website servers you can increase the DOWNLOAD_DELAY, limit the CONCURRENT_REQUESTS_PER_DOMAIN or set AUTOTHROTTLE_ENABLED that will adapt those settings dynamically based on the response times from the server.

Notice that Scrapy crawls are optimized for a single domain by default. If you are crawling multiple domains check these settings to optimize for broad crawls, including changing the default crawl order from depth-first to breath-first. To limit your crawl budget, you can limit the number of requests with the CLOSESPIDER_PAGECOUNT setting of the close spider extension.

With the default settings, Scrapy crawls about 600 pages per minute for a website like IMDb. To crawl 45M pages it will take more than 50 days for a single robot. If you need to crawl multiple websites it can be better to launch separate crawlers for each big website or group of websites. If you are interested in distributed web crawls, you can read how a developer crawled 250M pages with Python in 40 hours using 20 Amazon EC2 machine instances.

In some cases, you may run into websites that require you to execute JavaScript code to render all the HTML. Fail to do so, and you may not collect all links on the website. Because nowadays it's very common for websites to render content dynamically in the browser I wrote a Scrapy middleware for rendering JavaScript pages using ScrapingBee's API.

You might also like:

Conclusion

We compared the code of a Python crawler using third-party libraries for downloading URLs and parsing HTML with a crawler built using a popular web crawling framework. Scrapy is a very performant web crawling framework and it's easy to extend with your custom code. But you need to know all the places where you can hook your own code and the settings for each component.

Configur ing websites with millions t that you pick a popular s, which makes the topic fas

Sou

- Scrap
- Confi
- How
- How

How to execute JavaScript with Scrapy?

Ari Bajo 7 min read

Learn how to use Scrapy with website using JavaScript rendering.

Easy web scraping with Scrapy

Kevin Sahin 12 min read

Scrapy is the most popular Python web scraping framework. In this tutorial we will see how to scrape an E-commerce website with Scrapy from scratch.

Practical XPath for Web Scraping

Kevin Sahin 7 min read

In this tutorial, we are going to see how to use XPath expressions in your Python code to extract data from the web

Ready to get started?

Get access to 1,000 free API credits

Try ScrapingBee for Free

ScrapingBee API handles headless browsers and rotates proxies for you.

Company

Team

Company's journey

Blog

Rebranding

Affiliate Program

Legal

Terms of Service

Privacy Policy

GDPR Compliance

Data Processing Agreement

Product

Features

Pricing

Status

How we compare

Alternative to Crawlera

Alternative to Luminati

Alternative to Smartproxy

Alternative to Oxylabs

Alternative to NetNut

Alternatives to ScrapingBee

No code web scraping

No code web scraping

No code competitor monitoring

How to put scraped website data into Google Sheets

Send stock prices update to Slack

Scrape Amazon products' price with no code

Scrape Amazon products' price with no code

Extract job listings, details and salaries

Learning Web Scraping

A guide to Web Scraping without getting blocked

Web Scraping Tools

Best Free Proxies

Best Mobile proxies

Web Scraping vs Web Crawling

Rotating and residential proxies

Web Scraping with Python

Web Scraping with PHP

Web Scraping with Java

Web Scraping with Ruby

Web Scraping with NodeJS

Web Scraping with R

Web Scraping with C#

Web Scraping with Go