Intro to Web Scraping With Scrapy

When it comes to web scraping, Python Scrapy is the godfather of web scraping frameworks.

At nearly 15 years old, **Scrapy** is a extremely powerful web scraping framework that has collectively scraped trillions of web pages, and is trusted by thousands of companies and developers to power their data feeds.

If you had to learn only one web scraping framework, then there is a very strong case to be made that Python Scrapy should be the one you go for.

In this guide, we're going to explore:

What Exactly Is Python Scrapy

Scrapy is a Python framework designed specifically for web scraping. Built using Twisted, an event-driven networking engine, Scrapy uses an asynchronous architecture to crawl & scrape websites at scale fast.

With Scrapy you write **Spiders** to retrieve HTML pages from websites and scrape the data you want, clean and validate it, and store it in the data format you want.

Here is an example Spider:

```
import scrapy

class QuotesSpider(scrapy.Spider):
        name = 'quotes'

        def start_requests(self):
                url = 'https://quotes.toscrape.com/'
                yield scrapy.Request(url, callback=self.parse)

        def parse(self, response):
                for quote in response.css('div.quote'):
                        yield {
                'text': quote.css('span.text::text').get(),
                'author': quote.css('small.author::text').get(),
                'tags': quote.css('div.tags a.tag::text').getall()
            }

        # go to next page
        next_page = response.css("li.next a::attr(href)").extract_first()
        if next_page:
            yield response.follow(next_page, callback=self.parse)
```

Scrapy is a highly customisable web scraping framework, that also has a large community of developers building extensions & plugins for it.

So if the vanilla version of Scrapy doesn't have everything you want, then it is easily customised with open-source Scrapy extensions or your own middlewares/extensions.

How Does Scrapy Differ From Other Scraping Libraries & Frameworks

Taking inspiration from Python Django, Scrapy isn't just a HTTP request library like Python Requests, or a parsing library like BeautifulSoup or lxml, it a purpose built web scraping framework that covers:

- Making GET, POST, etc. requests
- Extracting data from the page with CSS & XPath selectors
- Detecting failed requests & automatically retrying
- Parallelising requests with inbuilt concurrency functionality
- Crawling entire websites with pagination, sitemap and link following
- Cleaning, validatiing and post-processing scraped data with pipelines
- Saving data to CSV/JSON files, databases and object storage
- And much much more.

With Scrapy everything is ready to use out of the box.

Just write your Scrapy spider, and configure the Scrapy settings to get it to perform as you want it to and away you go.

Whereas, if you were to use the common Python Requests and BeautifulSoup stack then you would have to build your own concurrency management layer, ban detection and retry layer, data processing layer, etc.

Essentially, recreate your own version of Scrapy.

That's not to say that Scrapy is the only option, every web scraping stack has its pros and cons, and are better suited to certain use cases and situations.

When Should You Use Scrapy?

It can come down to personal perference and your existing tech stack but Scrapy can be used for virtually every web scraping use case.

However, there are some situations where it really shines and others where using another web scraping stack is just as good or even better.

#1 Large Scale Web Scraping

Where Scrapy really shines is if you want to build scalable web scrapers that power your production data feeds. This is the use case Scrapy was built for and is the king at.

Out of the box, with Scrapy you have a highly scalable web scraping stack that is trusted by thousands of developers every day in production applications.

#2 Want To Learn A Powerful Web Scraping Framework

If you wanted to learn a web scraping technology to expand your skillset and enhance your CV then Scrapy is the best option. Companies all over the world use Scrapy in their production applications so if you can call yourself an experienced Scrapy developer you will never be short of work.

Learning Scrapy is also a great side skill for any data scientist, as you can quickly develop scrapers to power your data pipelines without the need of building your own web scraping stack from scratch.

#3 New To Web Scraping & Have Small Project

Generally speaking, the learning curve associated with Python Scrapy is a bit steeper than other approaches. So if you are new to web scraping and just have a small once off project like scraping a couple pages, then using the **Python Requests/BeautifulSoup stack** is a good option.

It is quick and easy to get setup, and you don't need to worry about learning about Scrapy Items, Pipelines, Feed Exporters, etc.

Just create the simplest possible scraper to get the job done.

However, if you want to get into web scraping, need to build a production system or want to learn a powerful web scraping framework then learning Scrapy is well worth the extra hour or two to learn it.

#4 Scraping Javascript Heavy Websites

Although, Scrapy can't render Javascript out of the box, it can be easily configured to do so by using one of the many headless browser extensions for Scrapy.

However, if you would like to focus solely on making a bot that heavily interacts with a page, logging in and navigating around then it might be better for you if you used the Puppeteer or Playwright in their native language of Javascript.

An Overview of Scrapy

Scrapy has a huge amount of functionality but it is a very structured web scraping framework and has a Scrapy way of doing things.

So to get the most out of Scrapy, you need to understand the main building blocks within Scrapy and how they work together.

#1 The Scrapy Project

You can use Scrapy like a normal python script, however, the preferred way of developing with Scrapy is using Scrapy projects.

When you run the `startproject` command, you generate a template Scrapy project ready for you to build your scrapers with.

```
scrapy startproject <project_name>
```

This project should look like this.

```
├── scrapy.cfg
└── myproject
    ├── __init__.py
```

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

This folder tree illustrates the 5 main building blocks of every Scrapy project: **Spiders**, **Items**, **Middlewares**, **Pipelines** and **Settings**.

Using these 5 building blocks you can create a scraper to do pretty much anything.

The most fundamental of which are **Spiders**.

#2 Scrapy Spiders

Scrapy spiders is where the magic happen. Spiders is the Scrapy name for scrapers that extract the data you need.

In your Scrapy project, you can have multiple Spiders all scraping the same or different websites and storing the data in different places.

Anything you could do with a **Python Requests/BeautifulSoup scraper** you can do with a **Scrapy Spider**.

```
import scrapy

class QuotesSpider(scrapy.Spider):
        name = 'quotes'

        def start_requests(self):
                url = 'https://quotes.toscrape.com/'
                yield scrapy.Request(url, callback=self.parse)

        def parse(self, response):
                for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').get(),
                'author': quote.css('small.author::text').get(),
                'tags': quote.css('div.tags a.tag::text').getall()
            }
```

To run this Spider, you simply need to run:

```
scrapy crawl quotes
```

When the above Spider is run, it will send a request to https://quotes.toscrape.com/ and once it has responded it will scrape all quotes from the page.

There are a couple things to point out here:

1. **Asynchronous** - As Scrapy is built using the Twisted framework, when you send a request to a website it isn't blocking. Scrapy will send the request to the website, and once it has retrieved a successful response it will tigger the `parse` method using the callback defined in the original Scrapy Request `yield scrapy.Request(url, callback=self.parse)`.
2. **Spider Name** - Every spider in your Scrapy project must have a unique name so that Scrapy can identify it. You set this using the `name = 'quotes'` attribute.
3. **Start Requests** - You define the starting points for your spider using the `start_requests()` method. Subsequent requests can be generated successively from these initial requests.
4. **Parse** - You use the `parse()` method to process the response from the website and extract the data you need. After extraction this data is sent to the Item Pipelines using the `yield` command.

Instead of just scraping one page, this basic spider can be easily reconfigured to scrape all the pages of the website by paginating through all the pages and scrape everything from those pages too:

```
import scrapy

class QuotesSpider(scrapy.Spider):
        name = 'quotes'
```

```
        def start_requests(self):
                url = 'https://quotes.toscrape.com/'
                yield scrapy.Request(url, callback=self.parse)

        def parse(self, response):
                for quote in response.css('div.quote'):
                        yield {
                'text': quote.css('span.text::text').get(),
                'author': quote.css('small.author::text').get(),
                'tags': quote.css('div.tags a.tag::text').getall()
            }

        # go to next page
        next_page = response.css("li.next a::attr(href)").extract_first()
        if next_page:
            yield response.follow(next_page, callback=self.parse)
```

Although this Scrapy spider is a bit more structured than your typical **Python Requests/BeautifulSoup scraper** it accomplishes the same things.

However, it is with Scrapy **Items**, **Middlewares**, **Pipelines** and **Settings** that Scrapy really stands out versus **Python Requests/BeautifulSoup**.

#3 Scrapy Items

Scrapy Items are how we store and process our scraped data. They provide a structured container for the data we scrape so that we can clean, validate and store it easily with Scrapy **ItemLoaders**, **Item Pipelines**, and **Feed Exporters**.

Using Scrapy Items have a number of advantages:

- Structures your data and gives it a clear schema.
- Enables you to easily clean and process your scraped data.
- Enables you to validate, deduplicate and monitor your data feeds.
- Enables you to easily store and export your data with Scrapy Feed Exports.
- Makes using Scrapy Item Pipelines & Item Loaders.

We typically define our **Items** in out `items.py` file.

```
# items.py

from scrapy.item import Item, Field

class QuoteItem(Item):
    text = Field()
    tags = Field()
    author = Field()
```

Then inside in your spider, instead of yielding a dictionary you would create a new Item with the scraped data before yielding it.

```
import scrapy
from demo.items import QuoteItem

class QuotesSpider(scrapy.Spider):
        name = 'quotes'

        def start_requests(self):
                url = 'https://quotes.toscrape.com/'
                yield scrapy.Request(url, callback=self.parse)

        def parse(self, response):
                quote_item = QuoteItem()
                for quote in response.css('div.quote'):
                        quote_item['text'] = quote.css('span.text::text').get()
                        quote_item['author'] = quote.css('small.author::text').get()
                        quote_item['tags'] = quote.css('div.tags a.tag::text').getall()
                        yield quote_item
```

#4 Scrapy Item Pipelines

Item Pipelines are the data processors of Scrapy, which all our scraped Items will pass through and from where we can clean, process, validate, and store our data.

Using Scrapy Pipelines we can:

- Clean our data (ex. remove currency signs from prices)
- Format our data (ex. convert strings to ints)
- Enrich our data (ex. convert relative links to absolute links)
- Valdiate our data (ex. make sure the price scraped is a viable price)
- Store our data in databases, queues, files or object storage buckets.

For example, here is a Item pipeline that stores our scraped into a Postgres Database:

```
# pipelines.py

import psycopg2

class PostgresDemoPipeline:

    def __init__(self):
        ## Connection Details
        hostname = 'localhost'
        username = 'postgres'
        password = '******' # your password
        database = 'quotes'

        ## Create/Connect to database
        self.connection = psycopg2.connect(host=hostname, user=username, password=password, dbname=database)

        ## Create cursor, used to execute commands
        self.cur = self.connection.cursor()

        ## Create quotes table if none exists
        self.cur.execute("""
        CREATE TABLE IF NOT EXISTS quotes(
            id serial PRIMARY KEY,
            content text,
            tags text,
            author VARCHAR(255)
        )
        """)

    def process_item(self, item, spider):

        ## Define insert statement
        self.cur.execute(""" insert into quotes (content, tags, author) values (%s,%s,%s)""", (
            item["text"],
            str(item["tags"]),
            item["author"]
        ))

        ## Execute insert of data into database
        self.connection.commit()
        return item

    def close_spider(self, spider):

        ## Close cursor & connection to database
        self.cur.close()
        self.connection.close()
```

#5 Scrapy Middlewares

As we've discussed, Scrapy is a complete web scraping framework that manages a lot of the complexity of scraping at scale for you behind the scenes without you having to configure anything.

Most of this functionality is contained within **Middlewares** in the form of Downloader Middlewares and Spider Middlewares.

Downloader Middlewares

Downloader middlewares are specific hooks that sit between the Scrapy Engine and the Downloader, which process requests as they pass from the Engine to the Downloader, and responses as they pass from Downloader to the Engine.

By default Scrapy has the following downloader middlewares enabled:

```
# settings.py

DOWNLOADER_MIDDLEWARES_BASE = {
    # Engine side
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware': 300,
    'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware': 350,
    'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware': 400,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': 500,
    'scrapy.downloadermiddlewares.retry.RetryMiddleware': 550,
    'scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware': 560,
    'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware': 580,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 590,
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': 600,
    'scrapy.downloadermiddlewares.cookies.CookiesMiddleware': 700,
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.downloadermiddlewares.stats.DownloaderStats': 850,
    'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware': 900,
    # Downloader side
}
```

These middlewares control everything from:

- Timing out requests
- What headers to send with your requests
- What user agents to use with your requests
- Retrying failed requests
- Managing cookies, caches and response compression

You can disable any of these default middlewares by setting it to `none` in your `settings.py` file. Here is an example of disabling the **RobotsTxtMiddleware**.

```
# settings.py

DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': None,
}
```

You can also override existing middlewares, or insert your own completely new middlewares if you want to:

- alter a request just before it is sent to the website (change the proxy, user-agent, etc.)
- change received response before passing it to a spider
- retry a request if the response doesn't contain the correct data instead of passing received response to a spider
- pass response to a spider without fetching a web page
- silently drop some requests

Here is an example of inserting our own middleware to use a proxy with all of your requests. We will create this in our `middlewares.py` file:

```
## middlewares.py

import base64

class MyProxyMiddleware(object):

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings)

    def __init__(self, settings):
        self.user = settings.get('PROXY_USER')
        self.password = settings.get('PROXY_PASSWORD')
        self.endpoint = settings.get('PROXY_ENDPOINT')
        self.port = settings.get('PROXY_PORT')
```

```
    def process_request(self, request, spider):
        user_credentials = '{user}:{passw}'.format(user=self.user, passw=self.password)
        basic_authentication = 'Basic ' + base64.b64encode(user_credentials.encode()).decode()
        host = 'http://{endpoint}:{port}'.format(endpoint=self.endpoint, port=self.port)
        request.meta['proxy'] = host
        request.headers['Proxy-Authorization'] = basic_authentication
```

We would enable it in your `settings.py` file, and fill in your proxy connection details:

```
## settings.py

PROXY_USER = 'username'
PROXY_PASSWORD = 'password'
PROXY_ENDPOINT = 'proxy.proxyprovider.com'
PROXY_PORT = '8000'

DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.MyProxyMiddleware': 350,
}
```

Spider Middlewares

Spider middlewares are specific hooks that sit between the Scrapy Engine and the Spiders, and which process spider input (responses) and output (items and requests).

By default Scrapy has the following downloader middlewares enabled:

```
# settings.py

SPIDER_MIDDLEWARES_BASE = {
    # Engine side
    'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware': 50,
    'scrapy.spidermiddlewares.offsite.OffsiteMiddleware': 500,
    'scrapy.spidermiddlewares.referer.RefererMiddleware': 700,
    'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware': 800,
    'scrapy.spidermiddlewares.depth.DepthMiddleware': 900,
    # Spider side
}
```

Spider middleware are used to:

- post-process output of spider callbacks - change/add/remove requests or items
- post-process start_requests
- handle spider exceptions
- call errback instead of callback for some of the requests based on response content

Like **Downloader middlewares** you can disable any of these default **Spider middlewares** by setting it to `none` in your `settings.py` file. Here is an example of disabling the **RobotsTxtMiddleware**.

```
# settings.py

SPIDER_MIDDLEWARES = {
    'scrapy.spidermiddlewares.referer.RefererMiddleware': None,
}
```

#6 Scrapy Settings

The `settings.py` file is the central control panel for your Scrapy project. You can enable/disable default functionality or integrate your own custom middlewares and extensions.

You can change the settings on a project basis by updating the `settings.py` file, or on a individual Spider basis by adding `custom_settings` to each spider.

In the following example, we add custom settings to our spider so that the scraped data will be saved to a `data.csv` file using the

`custom_settings` attribute.

```
import scrapy
from demo.items import QuoteItem

class QuotesSpider(scrapy.Spider):
        name = 'quotes'
    custom_settings = {
                'FEEDS': { 'data.csv': { 'format': 'csv',}}
                }

        def start_requests(self):
                url = 'https://quotes.toscrape.com/'
                yield scrapy.Request(url, callback=self.parse)

        def parse(self, response):
                quote_item = QuoteItem()
                for quote in response.css('div.quote'):
                        quote_item['text'] = quote.css('span.text::text').get()
                        quote_item['author'] = quote.css('small.author::text').get()
                        quote_item['tags'] = quote.css('div.tags a.tag::text').getall()
                        yield quote_item
```

There are a huge range of settings you can configure in Scrapy, so if you'd like to explore them all, here is a complete list of the default settings Scrapy provides.

Getting Started With Scrapy

So far we've given you an overview all the main building blocks in Scrapy. However, if you would like a guide on how to get started then you can check out the official quick start tutorial.

Alternatively, you can check out our **5 Part Scrapy For Beginners Series** which covers:

- **Part 1: Basic Scrapy Spider -** We will go over the basics of Scrapy, and build our first Scrapy spider. (Part 1)
- **Part 2: Cleaning Dirty Data & Dealing With Edge Cases -** Web data can be messy, unstructured, and have lots of edge cases. In this tutorial we will make our spider robust to these edge cases, using Items, Itemloaders and Item Pipelines. (Part 2)
- **Part 3: Storing Our Data -** There are many different ways we can store the data that we scrape from databases, CSV files to JSON format, and to S3 buckets. We will explore several different ways we can store the data and talk about their Pro's, Con's and in which situations you would use them. (Part 3)
- **Part 4: User Agents & Proxies -** Make our spider production ready by managing our user agents & IPs so we don't get blocked. (Part 4)
- **Part 5: Deployment, Scheduling & Running Jobs -** Deploying our spider on Heroku with a Scrapyd server, and monitoring and scheduling jobs via ScrapeOps. (Part 5)

Supercharging Scrapy With Scrapy Extensions

Vanilla Scrapy is great, it has a huge range of functionality available straight out of the box.

However, one of Scrapy's real superpowers is the fact there is a huge range of free Scrapy extensions that you can integrate into your Scrapy projects to enable more advanced functionality.

The Scrapy community has built Scrapy extensions for:

- Managing and rotating pools of proxies
- Generating & managing user agents
- Storing data in databases
- Monitoring & validating your spiders
- Creating distributed scraping architectures

More Scrapy Tutorials

This was a high-level introduction to Scrapy. Howeer, if you would like to get started with Scrapy then be sure to check out our:

- 5 Part Scrapy For Beginners Series

Or, if you would like to learn more about Scrapy in general, then be sure to check out The Scrapy Playbook.