Scrapy Tutorial

In this tutorial, we'll assume that Scrapy is already installed on your system. If that's not the case, see Installation guide.

We are going to scrape quotes.toscrape.com, a website that lists quotes from famous authors.

This tutorial will walk you through these tasks:

1. Creating a new Scrapy project
2. Writing a spider to crawl a site and extract data
3. Exporting the scraped data using the command line
4. Changing spider to recursively follow links
5. Using spider arguments

Scrapy is written in Python. If you're new to the language you might want to start by getting an idea of what the language is like, to get the most out of Scrapy.

If you're already familiar with other languages, and want to learn Python quickly, the Python Tutorial is a good resource.

If you're new to programming and want to start with Python, the following books may be useful to you:

- Automate the Boring Stuff With Python
- How To Think Like a Computer Scientist
- Learn Python 3 The Hard Way

You can also take a look at this list of Python resources for non-programmers, as well as the suggested resources in the learnpython-subreddit.

Creating a project

Before you start scraping, you will have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject tutorial
```

This will create a `tutorial` directory with the following contents:

```
tutorial/
    scrapy.cfg            # deploy configuration file

    tutorial/             # project's Python module, you'll import your code from here
        __init__.py

        items.py          # project items definition file

        middlewares.py    # project middlewares file

        pipelines.py      # project pipelines file

        settings.py       # project settings file

        spiders/          # a directory where you'll later put your spiders
            __init__.py
```

Our first Spider

Spiders are classes that you define and that Scrapy uses to scrape information from a website (or a group of websites). They must subclass `Spider` and define the initial requests to make, optionally how to follow links in the pages, and how to parse the downloaded page content to extract data.

This is the code for our first Spider. Save it in a file named `quotes_spider.py` under the `tutorial/spiders` directory in your project:

```
from pathlib import Path

import scrapy


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

    def start_requests(self):
        urls = [
            "https://quotes.toscrape.com/page/1/",
            "https://quotes.toscrape.com/page/2/",
        ]
        for url in urls:
            yield scrapy.Request(url=url, callback=self.parse)

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = f"quotes-{page}.html"
        Path(filename).write_bytes(response.body)
        self.log(f"Saved file {filename}")
```

As you can see, our Spider subclasses `scrapy.Spider` and defines some attributes and methods:

- `name`: identifies the Spider. It must be unique within a project, that is, you can't set the same name for different Spiders.
- `start_requests()`: must return an iterable of Requests (you can return a list of requests or write a generator function) which the Spider will begin to crawl from. Subsequent requests will be generated successively from these initial requests.
- `parse()`: a method that will be called to handle the response downloaded for each of the requests made. The response parameter is an instance of `TextResponse` that holds the page content and has further helpful methods to handle it.

  The `parse()` method usually parses the response, extracting the scraped data as dicts and also finding new URLs to follow and creating new requests (`Request`) from them.

How to run our spider

To put our spider to work, go to the project's top level directory and run:

```
scrapy crawl quotes
```

This command runs the spider with name `quotes` that we've just added, that will send some requests for the `quotes.toscrape.com` domain. You will get an output similar to this:

```
... (omitted for brevity)
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Spider opened
2016-12-16 21:24:05 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0
pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:24:05 [scrapy.extensions.telnet] DEBUG: Telnet console listening on
127.0.0.1:6023
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (404) <GET
https://quotes.toscrape.com/robots.txt> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET
https://quotes.toscrape.com/page/1/> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET
https://quotes.toscrape.com/page/2/> (referer: None)
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-1.html
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-2.html
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Closing spider (finished)
...
```

Now, check the files in the current directory. You should notice that two new files have been created: *quotes-1.html* and *quotes-2.html*, with the content for the respective URLs, as our `parse` method instructs.

Note

If you are wondering why we haven't parsed the HTML yet, hold on, we will cover that soon.

What just happened under the hood?

Scrapy schedules the `scrapy.Request` objects returned by the `start_requests` method of the Spider. Upon receiving a response for each one, it instantiates `Response` objects and calls the callback method associated with the request (in this case, the `parse` method) passing the response as argument.

A shortcut to the start_requests method

Instead of implementing a `start_requests()` method that generates `scrapy.Request` objects from URLs, you can just define a `start_urls` class attribute with a list of URLs. This list will then be used by the default implementation of `start_requests()` to create the initial requests for your spider.

```
from pathlib import Path

import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        "https://quotes.toscrape.com/page/1/",
        "https://quotes.toscrape.com/page/2/",
    ]

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = f"quotes-{page}.html"
        Path(filename).write_bytes(response.body)
```

The `parse()` method will be called to handle each of the requests for those URLs, even though we haven't explicitly told Scrapy to do so. This happens because `parse()` is Scrapy's default callback method, which is called for requests without an explicitly assigned callback.

Extracting data

The best way to learn how to extract data with Scrapy is trying selectors using the Scrapy shell. Run:

```
scrapy shell 'https://quotes.toscrape.com/page/1/'
```

Note

Remember to always enclose urls in quotes when running Scrapy shell from command-line, otherwise urls containing arguments (i.e. `&` character) will not work.

On Windows, use double quotes instead:

```
scrapy shell "https://quotes.toscrape.com/page/1/"
```

You will see something like:

```
[ ... Scrapy log here ... ]
2016-09-19 12:09:27 [scrapy.core.engine] DEBUG: Crawled (200) <GET
https://quotes.toscrape.com/page/1/> (referer: None)
[s] Available Scrapy objects:
[s]   scrapy     scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler    <scrapy.crawler.Crawler object at 0x7fa91d888c90>
[s]   item       {}
[s]   request    <GET https://quotes.toscrape.com/page/1/>
[s]   response   <200 https://quotes.toscrape.com/page/1/>
[s]   settings   <scrapy.settings.Settings object at 0x7fa91d888c10>
[s]   spider     <DefaultSpider 'default' at 0x7fa91c8af990>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser
```

Using the shell, you can try selecting elements using CSS with the response object:

```
>>> response.css("title")
[<Selector query='descendant-or-self::title' data='<title>Quotes to Scrape</title>'>]
```

The result of running `response.css('title')` is a list-like object called `SelectorList`, which represents a list of `Selector` objects that wrap around XML/HTML elements and allow you to run further queries to fine-grain the selection or extract the data.

To extract the text from the title above, you can do:

```
>>> response.css("title::text").getall()
['Quotes to Scrape']
```

There are two things to note here: one is that we've added `::text` to the CSS query, to mean we want to select only the text elements directly inside `<title>` element. If we don't specify `::text`, we'd get the full title element, including its tags:

```
>>> response.css("title").getall()
['<title>Quotes to Scrape</title>']
```

The other thing is that the result of calling `.getall()` is a list: it is possible that a selector returns more than one result, so we extract them all. When you know you just want the first result, as in this case, you can do:

```
>>> response.css("title::text").get()
'Quotes to Scrape'
```

As an alternative, you could've written:

```
>>> response.css("title::text")[0].get()
'Quotes to Scrape'
```

Accessing an index on a `SelectorList` instance will raise an `IndexError` exception if there are no results:

```
>>> response.css("noelement")[0].get()
Traceback (most recent call last):
...
IndexError: list index out of range
```

You might want to use `.get()` directly on the `SelectorList` instance instead, which returns `None` if there are no results:

```
>>> response.css("noelement").get()
```

There's a lesson here: for most scraping code, you want it to be resilient to errors due to things not being found on a page, so that even if some parts fail to be scraped, you can at least get **some** data.

Besides the `getall()` and `get()` methods, you can also use the `re()` method to extract using regular expressions:

```
>>> response.css("title::text").re(r"Quotes.*")
['Quotes to Scrape']
>>> response.css("title::text").re(r"Q\w+")
['Quotes']
>>> response.css("title::text").re(r"(\w+) to (\w+)")
['Quotes', 'Scrape']
```

In order to find the proper CSS selectors to use, you might find it useful to open the response page from the shell in your web browser using `view(response)`. You can use your browser's developer tools to inspect the HTML and come up with a selector (see Using your browser's Developer Tools for scraping).

Selector Gadget is also a nice tool to quickly find CSS selector for visually selected elements, which works in many browsers.

XPath: a brief intro

Besides CSS, Scrapy selectors also support using XPath expressions:

```
>>> response.xpath("//title")
[<Selector query='//title' data='<title>Quotes to Scrape</title>'>]
>>> response.xpath("//title/text()").get()
'Quotes to Scrape'
```

XPath expressions are very powerful, and are the foundation of Scrapy Selectors. In fact, CSS selectors are converted to XPath under-the-hood. You can see that if you read closely the text representation of the selector objects in the shell.

While perhaps not as popular as CSS selectors, XPath expressions offer more power because besides navigating the structure, it can also look at the content. Using XPath, you're able to select things like: *select the link that contains the text "Next Page"*. This makes XPath very fitting to the task of scraping, and we encourage you to learn XPath even if you already know how to construct CSS selectors, it will make scraping much easier.

We won't cover much of XPath here, but you can read more about using XPath with Scrapy Selectors here. To learn more about XPath, we recommend this tutorial to learn XPath through examples, and this tutorial to learn "how to think in XPath".

Extracting quotes and authors

Now that you know a bit about selection and extraction, let's complete our spider by writing the code to extract the quotes from the web page.

Each quote in https://quotes.toscrape.com is represented by HTML elements that look like this:

```
<div class="quote">
    <span class="text">“The world as we have created it is a process of our
    thinking. It cannot be changed without changing our thinking.”</span>
    <span>
        by <small class="author">Albert Einstein</small>
        <a href="/author/Albert-Einstein">(about)</a>
    </span>
    <div class="tags">
        Tags:
        <a class="tag" href="/tag/change/page/1/">change</a>
        <a class="tag" href="/tag/deep-thoughts/page/1/">deep-thoughts</a>
        <a class="tag" href="/tag/thinking/page/1/">thinking</a>
        <a class="tag" href="/tag/world/page/1/">world</a>
    </div>
</div>
```

Let’s open up scrapy shell and play a bit to find out how to extract the data we want:

```
scrapy shell 'https://quotes.toscrape.com'
```

We get a list of selectors for the quote HTML elements with:

```
>>> response.css("div.quote")
[<Selector query="descendant-or-self::div[@class and contains(concat(' ', normalize-
space(@class), ' '), ' quote ')]" data='<div class="quote" itemscope itemtype...'>,
<Selector query="descendant-or-self::div[@class and contains(concat(' ', normalize-
space(@class), ' '), ' quote ')]" data='<div class="quote" itemscope itemtype...'>,
...]
```

Each of the selectors returned by the query above allows us to run further queries over their sub-elements. Let’s assign the first selector to a variable, so that we can run our CSS selectors directly on a particular quote:

```
>>> quote = response.css("div.quote")[0]
```

Now, let’s extract `text`, `author` and the `tags` from that quote using the `quote` object we just created:

```
>>> text = quote.css("span.text::text").get()
>>> text
'“The world as we have created it is a process of our thinking. It cannot be changed
without changing our thinking.”'
>>> author = quote.css("small.author::text").get()
>>> author
'Albert Einstein'
```

Given that the tags are a list of strings, we can use the `.getall()` method to get all of them:

```
>>> tags = quote.css("div.tags a.tag::text").getall()
>>> tags
['change', 'deep-thoughts', 'thinking', 'world']
```

Having figured out how to extract each bit, we can now iterate over all the quotes elements and put them together into a Python dictionary:

```
>>> for quote in response.css("div.quote"):
...     text = quote.css("span.text::text").get()
...     author = quote.css("small.author::text").get()
...     tags = quote.css("div.tags a.tag::text").getall()
...     print(dict(text=text, author=author, tags=tags))
...
{'text': '“The world as we have created it is a process of our thinking. It cannot be
changed without changing our thinking.”', 'author': 'Albert Einstein', 'tags':
['change', 'deep-thoughts', 'thinking', 'world']}
{'text': '“It is our choices, Harry, that show what we truly are, far more than our
abilities.”', 'author': 'J.K. Rowling', 'tags': ['abilities', 'choices']}
...
```

Extracting data in our spider

Let’s get back to our spider. Until now, it doesn’t extract any data in particular, just saves the whole HTML page to a local file. Let’s integrate the extraction logic above into our spider.

A Scrapy spider typically generates many dictionaries containing the data extracted from the page. To do that, we use the `yield` Python keyword in the callback, as you can see below:

```
import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        "https://quotes.toscrape.com/page/1/",
        "https://quotes.toscrape.com/page/2/",
    ]

    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(),
            }
```

If you run this spider, it will output the extracted data with the log:

```
2016-09-19 18:57:19 [scrapy.core.scraper] DEBUG: Scraped from <200
https://quotes.toscrape.com/page/1/>
{'tags': ['life', 'love'], 'author': 'André Gide', 'text': '“It is better to be hated
for what you are than to be loved for what you are not.”'}
2016-09-19 18:57:19 [scrapy.core.scraper] DEBUG: Scraped from <200
https://quotes.toscrape.com/page/1/>
{'tags': ['edison', 'failure', 'inspirational', 'paraphrased'], 'author': 'Thomas A.
Edison', 'text': "“I have not failed. I've just found 10,000 ways that won't work.”"}
```

Storing the scraped data

The simplest way to store the scraped data is by using Feed exports, with the following command:

```
scrapy crawl quotes -O quotes.json
```

That will generate a `quotes.json` file containing all scraped items, serialized in JSON.

The `-O` command-line switch overwrites any existing file; use `-o` instead to append new content to any existing file. However, appending to a JSON file makes the file contents invalid JSON. When appending to a file, consider using a different serialization format, such as JSON Lines:

```
scrapy crawl quotes -o quotes.jsonl
```

The JSON Lines format is useful because it’s stream-like, you can easily append new records to it. It doesn’t have the same problem of JSON when you run twice. Also, as each record is a separate line, you can process big files without having to fit everything in memory, there are tools like JQ to help do that at the command-line.

In small projects (like the one in this tutorial), that should be enough. However, if you want to perform more complex things with the scraped items, you can write an Item Pipeline. A placeholder file for Item Pipelines has been set up for you when the project is created, in `tutorial/pipelines.py`. Though you don’t need to implement any item pipelines if you just want to store the scraped items.

Following links

Let’s say, instead of just scraping the stuff from the first two pages from https://quotes.toscrape.com, you want quotes from all the pages in the website.

Now that you know how to extract data from pages, let’s see how to follow links from them.

First thing is to extract the link to the page we want to follow. Examining our page, we can see there is a link to the next page with the following markup:

```
<ul class="pager">
    <li class="next">
        <a href="/page/2/">Next <span aria-hidden="true">&rarr;</span></a>
    </li>
</ul>
```

We can try extracting it in the shell:

```
>>> response.css('li.next a').get()
'<a href="/page/2/">Next <span aria-hidden="true">→</span></a>'
```

This gets the anchor element, but we want the attribute `href`. For that, Scrapy supports a CSS extension that lets you select the attribute contents, like this:

```
>>> response.css("li.next a::attr(href)").get()
'/page/2/'
```

There is also an `attrib` property available (see Selecting element attributes for more):

```
>>> response.css("li.next a").attrib["href"]
'/page/2/'
```

Let's see now our spider modified to recursively follow the link to the next page, extracting data from it:

```
import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        "https://quotes.toscrape.com/page/1/",
    ]

    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(),
            }

        next_page = response.css("li.next a::attr(href)").get()
        if next_page is not None:
            next_page = response.urljoin(next_page)
            yield scrapy.Request(next_page, callback=self.parse)
```

Now, after extracting the data, the `parse()` method looks for the link to the next page, builds a full absolute URL using the `urljoin()` method (since the links can be relative) and yields a new request to the next page, registering itself as callback to handle the data extraction for the next page and to keep the crawling going through all the pages.

What you see here is Scrapy's mechanism of following links: when you yield a Request in a callback method, Scrapy will schedule that request to be sent and register a callback method to be executed when that request finishes.

Using this, you can build complex crawlers that follow links according to rules you define, and extract different kinds of data depending on the page it's visiting.

In our example, it creates a sort of loop, following all the links to the next page until it doesn't find one – handy for crawling blogs, forums and other sites with pagination.

A shortcut for creating Requests

As a shortcut for creating Request objects you can use `response.follow`:

```
import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        "https://quotes.toscrape.com/page/1/",
    ]

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

        next_page = response.css("li.next a::attr(href)").get()
        if next_page is not None:
            yield response.follow(next_page, callback=self.parse)
```

Unlike scrapy.Request, `response.follow` supports relative URLs directly - no need to call urljoin. Note that `response.follow` just returns a Request instance; you still have to yield this Request.

You can also pass a selector to `response.follow` instead of a string; this selector should extract necessary attributes:

```
for href in response.css("ul.pager a::attr(href)"):
    yield response.follow(href, callback=self.parse)
```

For `<a>` elements there is a shortcut: `response.follow` uses their href attribute automatically. So the code can be shortened further:

```
for a in response.css("ul.pager a"):
    yield response.follow(a, callback=self.parse)
```

To create multiple requests from an iterable, you can use **`response.follow_all`** instead:

```
anchors = response.css("ul.pager a")
yield from response.follow_all(anchors, callback=self.parse)
```

or, shortening it further:

```
yield from response.follow_all(css="ul.pager a", callback=self.parse)
```

More examples and patterns

Here is another spider that illustrates callbacks and following links, this time for scraping author information:

```
import scrapy


class AuthorSpider(scrapy.Spider):
    name = "author"

    start_urls = ["https://quotes.toscrape.com/"]

    def parse(self, response):
        author_page_links = response.css(".author + a")
        yield from response.follow_all(author_page_links, self.parse_author)

        pagination_links = response.css("li.next a")
        yield from response.follow_all(pagination_links, self.parse)

    def parse_author(self, response):
        def extract_with_css(query):
            return response.css(query).get(default="").strip()

        yield {
            "name": extract_with_css("h3.author-title::text"),
            "birthdate": extract_with_css(".author-born-date::text"),
            "bio": extract_with_css(".author-description::text"),
        }
```

This spider will start from the main page, it will follow all the links to the authors pages calling the `parse_author` callback for each of them, and also the pagination links with the `parse` callback as we saw before.

Here we're passing callbacks to **`response.follow_all`** as positional arguments to make the code shorter; it also works for **`Request`**.

The `parse_author` callback defines a helper function to extract and cleanup the data from a CSS query and yields the Python dict with the author data.

Another interesting thing this spider demonstrates is that, even if there are many quotes from the same author, we don't need to worry about visiting the same author page multiple times. By default, Scrapy filters out duplicated requests to URLs already visited, avoiding the problem of hitting servers too much because of a programming mistake. This can be configured by the setting **`DUPEFILTER_CLASS`**.

Hopefully by now you have a good understanding of how to use the mechanism of following links and callbacks with Scrapy.

As yet another example spider that leverages the mechanism of following links, check out the `CrawlSpider` class for a generic spider that implements a small rules engine that you can use to write your crawlers on top of it.

Also, a common pattern is to build an item with data from more than one page, using a trick to pass additional data to the callbacks.

Using spider arguments

You can provide command line arguments to your spiders by using the `-a` option when running them:

```
scrapy crawl quotes -O quotes-humor.json -a tag=humor
```

These arguments are passed to the Spider's `__init__` method and become spider attributes by default.

In this example, the value provided for the `tag` argument will be available via `self.tag`. You can use this to make your spider fetch only quotes with a specific tag, building the URL based on the argument:

```
import scrapy


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

    def start_requests(self):
        url = "https://quotes.toscrape.com/"
        tag = getattr(self, "tag", None)
        if tag is not None:
            url = url + "tag/" + tag
        yield scrapy.Request(url, 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(),
            }

        next_page = response.css("li.next a::attr(href)").get()
        if next_page is not None:
            yield response.follow(next_page, self.parse)
```

If you pass the `tag=humor` argument to this spider, you'll notice that it will only visit URLs from the `humor` tag, such as `https://quotes.toscrape.com/tag/humor`.

You can learn more about handling spider arguments here.

Next steps

This tutorial covered only the basics of Scrapy, but there's a lot of other features not mentioned here. Check the What else? section in Scrapy at a glance chapter for a quick overview of the most important ones.

You can continue from the section Basic concepts to know more about the command-line tool, spiders, selectors and other things the tutorial hasn't covered like modeling the scraped data. If you prefer to play with an example project, check the Examples section.