



SCRAPY FUNDAMENTALS

Scrapy Fundamentals

by Attila Toth

Copyright © 2016 Attila Toth

<http://ScrapingAuthority.com>

Table of Contents

| | |
|-----------------------------------|----|
| Introduction..... | 4 |
| Item..... | 10 |
| First Scrapy Spider..... | 12 |
| Exporting Data..... | 18 |
| Itemloader..... | 22 |
| Debugging..... | 28 |
| Login to Websites..... | 35 |
| Scrapy Settings..... | 37 |
| Downloading Images..... | 41 |
| Javascript Generated Content..... | 46 |
| AJAX Requests..... | 52 |
| Infinite Scrolling..... | 54 |
| Scrapy Cloud..... | 56 |
| Scrapy Code Breakdown..... | 60 |
| Where to Go Next..... | 66 |

Introduction

About me

Hi, my name is Attila Toth I'm from Hungary. I am the owner of [ScrapingAuthority.com](https://scrapingauthority.com). I also wrote an ebook „Learn Web Scraping From Scratch” [which you can download for free here.](#) Currently I'm a computer science student and I have been obsessed with Scrapy for 4 years. I've created thousands of Scrapy spiders and scraped millions of web pages.

I will teach you how to scrape the web with Scrapy.

About the book

This book is created to help you get started with Scrapy and web scraping. You need to have at least a bit of programming knowledge. I put as much real code in the book as possible to teach you better and you can find all the piece of code I'm mentioning here: <https://github.com/zseta/scrapypfundamentals>

How to reach out to me

I'm always glad to help whenever I can. Feel free to reach out to me if you have a question or something to say. Also, **tell me what you want to learn more about in my next web scraping learning material!**

- Email: attila@scrapingauthority.com
- Twitter: <http://twitter.com/scrapingA>
- Facebook: <https://www.facebook.com/scrapingauthority/>
- Website: <http://www.scrapingauthority.com/>

You can find the content of this book wrapped in blog posts here:
<http://scrapingauthority.com/tutorials/scrapy>

I really appreciate you picked up my book, I hope you will like it!

ATTILA TOTH

Scrapy Fundamentals

I. Spider, Item

We use web scraping to turn unstructured data into highly structured data. Essentially, it's the goal of web scraping. Structured data means collected information in database such as mongoDB or SQL database. Also, in most cases we only need some simple data structure such as JSON, CSV or XML. This way you can process data to fit your desires. Web scraping collects gross information from websites into an organized system. Data has to be stored in built-in data types, at least for a while, so you can process it into a database. In Scrapy you can use the Item class to store data before extracting into any kind of structured data system mentioned before.

Scrapy Item

To extract data with Scrapy we use particular Item classes in our project to store information we need to fetch. You can have as many items as you wish. Strive to organize your scrapy items in the most readable and logical way. Each item class has to derive from scrapy.Item class.

```
class BookExampleItem(scrapy.Item):  
    title = scrapy.Field()  
    author = scrapy.Field()  
    length = scrapy.Field()  
    paperback = scrapy.Field()  
    publisher = scrapy.Field()
```

It's as simple as that. You create Scrapy Fields in your item to store data.

Associate Fields with data

Now you know how to create Item classes. Setting values to them works the same way as you set values in a Dictionary.

```
book_item = BookExampleItem()
book_item["title"] = 'title'
book_item["author"] = 'author'
book_item["length"] = 'length'
book_item["paperback"] = 'paperback'
book_item["publisher"] = 'publisher'
```

This way you can easily set values to your Item Fields in your Spider script just like a dict.

Now you have a basic understanding how Scrapy Items work so you can start web scraping right away. Move on to the next part!

Web scraping is something that can be really useful, inevitable and a good framework makes it really easy. When working with Python, I like using [Scrapy](#) framework because it's very powerful and easy to use even for a novice and capable of scraping large sites like amazon.com.

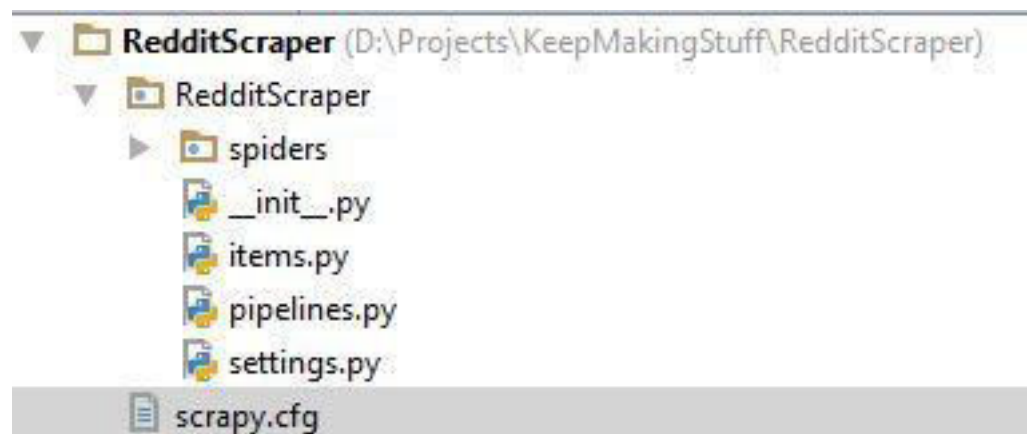
Create a Web Scraper with Scrapy

We're gonna create a scraper that crawl some data from reddit's programming subreddit. More specifically, we are going to crawl the hottest programming topics from the first page, their links and their posting time.

Very first, we create a scrapy project using this command:

```
scrapy startproject RedditScraper
```

Our project structure looks like this:



In Scrapy, we have to store scraped data in Item classes. In our case, an Item will have fields like title, link and posting_time.

```
class RedditscraperItem(scrapy.Item):
    title = scrapy.Field()
    link = scrapy.Field()
    posting_time = scrapy.Field()
```

Next, we create a new python file in Spiders folder in which we're going to implement parse method. But first, we should check out the [page](#)'s HTML to figure out a way to achieve the information we need.

A screenshot of a web browser's developer tools, specifically the 'Elements' panel. It shows a tree view of HTML elements. The selected element is a <div class="entry unrated">. It contains a <p class="title"> with an FLIF - Free Lossless Image Format, a , a <p class="tagline">, a <ul class="flat-list buttons">, a <div class="reportform report-t3_49bs2c">, a <div class="expando expando-uninitialized" style="display: none;">, a <div class="child">, and a <div class="clearleft">.

We can see that each data we need is inside a class named “entry unrated”.

Now in our custom spider we need to define its name as a string and its urls as a list. Then in parse method we create a for loop which go through each “*entry unrated*” div and find every title, link and posting time. We are using xpath because now I find it more efficient and readable than CSS selectors.

```
import scrapy
from RedditScraper.items import RedditiItem

class RedditSpider(scrapy.Spider):
    name = "reddit"
    start_urls = [
        "https://www.reddit.com/r/programming"
    ]

    def parse(self, response):
        for sel in response.xpath("//div[@class='entry unvoted']"):
            item = RedditiItem()
            item["title"] = sel.xpath("p[@class='title']/a/text()").extract()
            item["link"] = sel.xpath("p[@class='title']/a/@href").extract()
            item["posting_time"] = sel.xpath("p[@class='tagline']/time/text()").extract()
            yield item
```

Running the *scrapy crawl reddit* command we get what we wanted wrapped in Items:

```
2016-03-07 18:49:52 [scrapy] DEBUG: Scraped from 200
https://www.reddit.com/r/programming
{'link': [u'http://thecodelesscode.com/case/225'],
 'posting_time': [u'3 hours ago'],
 'title': [u'[Codeless Code] Case 225: The Three Most Terrifying Words']}
2016-03-07 18:49:52 [scrapy] DEBUG: Scraped from 200
https://www.reddit.com/r/programming
{'link': [u'https://textplain.wordpress.com/2016/03/06/using-https-properly/'],
 'posting_time': [u'2 hours ago'],
 'title': [u'Using HTTPS Properly']}
...
```

In the previous Scrapy tutorial you learnt how to scrape information from a single page. Going further with web scraping, you will need to visit a bunch of URLs within a website and execute the same scraping script again and again. In my Jsoup tutorial and BeautifulSoup tutorial I showed you how you can paginate on a website now you will learn how to do this with Scrapy. The CrawlSpider module makes it super easy.

Pagination with Scrapy

As relevant example, we are going to scrape some data from Amazon. As usual, scrapy will do most of the work and now we're using its CrawlSpider Module. It provides an attribute called rule. This is a tuple in which we define rules about links we want our crawler to follow.

First and foremost, we should setup a User Agent because we want our crawler to see the site like we see it in browser and it's fine because we don't intend to do anything harmful. You can setup a User Agent in settings.py:

```
USER_AGENT='Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.36 Safari/535.7'
```

Scrapy CrawlSpider Rule Attribute

So for example we're looking for the most reviewed books in each category.

In our spider file we create a *Rule* which contains where book category links are on the page then callback the method we want to execute inside each category page(our starting url is [amazon books page](#)):

```
url_xpath = "(//div[@class='categoryRefinementsSection']/ul/li)"
rules = (
    Rule(LinkExtractor(restrict_xpaths = url_xpath), callback="sort_books"),
)
```

Now our crawler knows where to go. We know that amazon, like most of modern sites, uses javascript to display content. In some cases it makes scraping much more complicated but it's a good thing that amazon works perfectly without any javascript so we don't have to use any kind of headless browser or such.

Scrapy FormRequest

As I said, we need the most reviewed books in each category, let's say the first 12 we can find on the first page. But first we should sort the books by most reviews.

```
<form id="searchSortForm" class="s-inline-form" method="get" action="/gp/search/ref=sr_st">
  <input name="rh" value="n:283155,n:11000,n:1" type="hidden"></input>
  <input name="qid" value="1458242511" type="hidden"></input>
  <span class="a-size-base">Sort by </span>
  <select id="sort" class="a-spacing-top-mini" style="vertical-align: baseline;" name="sort" onchange="">
    <option value="featured-rank" selected="selected">Featured</option>
    <option value="price-asc-rank">Price: Low to High</option>
    <option value="price-desc-rank">Price: High to Low</option>
    <option value="review-rank">Avg. Customer Review</option>
    <option value="date-desc-rank">Publication Date</option>
    <option value="review-count-rank">Most reviews</option>
  </select>
  <noscript><input type="image" src="http://g-ecx.images-amazo...</noscript>
</form>
```

If you have a look at the source you'll see that we need to parse a form in order to sort. The "Go" button which will refresh the page according to the form is visible only if visiting the page without javascript enabled like our crawler does. In Scrapy we can use a FormRequest object to pass a form:

```
def sort_books(self, response):
    sorted_form = FormRequest.from_response(
        response,
        formxpath="//form[@id='searchSortForm']", #xpath of form
        formdata={"sort": "review-count-rank"},
        clickdata={"value": "Go"}, #the button we "click"
        callback=self.parse_category
    )
    yield sorted_form
```

Data Extraction

The next and final thing we have to do is to parse each link that redirect the crawler to a book's page where you invoke the *parse_book_page* method which will take care of scraping the data we're looking for.

```
def parse_category(self, response):
    links_xpath = "//a[@class='a-link-normal s-access-detail-page a-text-normal']/@href"
    links = response.xpath(links_xpath)
    for link in links:
        url = response.urljoin(link.extract())
        yield Request(url, callback=self.parse_book_page)
```

Finally, we extract the desired details we need in *parse_book_page* like we did in the previous tutorial.

The real beauty in web scraping is actually to be able to use the scraped data. In most cases, the easiest and smartest way to store scraped data is a simple Json or CSV file. They are readable by humans and other softwares as well so it should be applicable almost everytime though when you work with huge amount of data it might be better to choose a database structure which is more scalable.

Exporting Json and CSV in Scrapy

There are some ways to produce Json or CSV files including your data in Scrapy.

The first way is to use Feed Exports inside command line. You can run your scraper and store your data from the command line by setting the filename and desired format.

You may want to customize your output and produce structured Json or CSV while your scraper runs. You can use Item Pipeline to set your output properties in a pipeline and not from command line.

Exporting from Command Line

As you learnt in the first post of this series you can run your scraper from command line with the *scrapy crawl myspider* command. If you want to create output files you have to set the filename and extension you want to use.

```
scrapy crawl myspider -o data.json
```

```
scrapy crawl myspider -o data.csv
```

```
scrapy crawl myspider -o data.xml
```

Scrapy has its built-in tool to generate json, csv, xml and other serialization formats.

If you want to specify either relative or absolute path of the produced file or set other properties from command line you can do it as well.

```
scrapy crawl reddit -s FEED_URI='/home/user/folder/mydata.csv' -s  
FEED_FORMAT=csv
```

```
scrapy crawl reddit -s FEED_URI='mydata.json' -s FEED_FORMAT=json
```

Exporting with Item Pipeline

Scrapy Item Pipeline is a universal tool to process your data. Typical usages are cleaning html, validating scraped data, dropping duplicates and storing scraped data in database. You can use pipelines if you want a convenient and customizable process to store your data.

You need to use `JsonItemExporter`:

```
class JsonPipeline(object):  
    def __init__(self):  
        self.file = open("books.json", 'wb')  
        self.exporter = JsonItemExporter(self.file, encoding='utf-8',  
ensure_ascii=False)  
        self.exporter.start_exporting()  
  
    def close_spider(self, spider):  
        self.exporter.finish_exporting()  
        self.file.close()  
  
    def process_item(self, item, spider):  
        self.exporter.export_item(item)  
        return item
```

It works the same way with CSV but you have to invoke `CsvItemExporter`:

```
class CsvPipeline(object):
    def __init__(self):
        self.file = open("booksdata.csv", 'wb')
        self.exporter = CsvItemExporter(self.file, unicode)
        self.exporter.start_exporting()

    def close_spider(self, spider):
        self.exporter.finish_exporting()
        self.file.close()

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item
```

Be aware that in a csv file the fields are separated with “,” (comma) by default. If your fields contain text with commas which screw up the whole structure you may want to create a function which fixes this:

```
def create_valid_csv(self, item):
    for key, value in item.items():
        is_string = (isinstance(value, basestring))
        if (is_string and ("," in value.encode('utf-8'))):
            item[key] = "\"" + value + "\""
```

You have to invoke this function before exporting the item so the `ItemExporter` will recognize commas in the data and structure accordingly.

Configure settings.py

It's very important to tell scrapy you dare to use Item Pipelines otherwise your pipelines won't be invoked.

You have to add these lines to your settings.py in your Scrapy project:

```
ITEM_PIPELINES = {  
    'RedditScraper.pipelines.JsonPipeline': 300,  
    'RedditScraper.pipelines.CsvPipeline': 500,  
}
```

If you are wondering what those numbers mean, those are meant to indicate the priority of the pipelines. In this example the JsonPipeline will be executed sooner. The values must be in range of 0-1000.

Item Loaders are used to populate your items. Earlier, you learnt how to create Scrapy Items and store your scraped data in them. Essentially, Item Loaders provide a way to populate these Items and run any input or output process you want alongside. Maybe you need to parse the scraped items even more, apply a filter on them or simply cleansing html or validate data. You can do all these with Item Loader.

Scrapy Item Loader

When instantiating an Item Loader to populate items you don't use dict-like syntax instead you define it by giving an item which you want to populate and a Response or Selector object in the constructor. The item is a Scrapy Item instance that you want to populate. The selector or response object will define where is the data you need to extract.

```
def parse(self, response):  
  
    loader = ItemLoader(item=MyItem(), selector=response)  
  
    loader.add_css("text", ".text")  
    loader.add_css("author", ".author")  
    loader.add_xpath("tags", "///div[@class='tag']")  
    loader.add_value("name", "Attila")  
    yield loader.load_item()
```

This is how you create an item loader object and then how you populate the item. With `add_css(field_name, css_selector)` method you populate the text field which is defined in `MyItem`. Then you define a css selector which will select the data from response. As you see in the snippet above xpath is supported as well. You also can assign value which is not scraped from the website but directly defined in the code.

It's important that when you invoke `add_css` or `add_xpath` or `add_value` the data will be extracted but the item will not be populated yet. The item will be populated when you invoke `load_item()` method. It returns the scraped item.

Input and Output Processors

As I mentioned you can do further parsing and processing on your scraped data with Item Loader. When you declare an `ItemLoader` you should assign an item to it. Now, each field of this item has an input processor and an output processor.

Here's a small snippet and explanation how it works:

```
loader = ItemLoader(item=MyItem(), selector=response)

loader.add_css("name", selector1) #1.
loader.add_css("name", selector2) #2.
loader.add_xpath("name", xpath1) #3.
loader.add_value("name", "my_title") #4.
yield loader.load_item() #5.
```

1. The data is extracted according to selector1, input processor is invoked and scraped data is stored in ItemLoader(Not in the item!) as name.
2. The data is extracted according to selector2, the same input processor is invoked and scraped data is appended to name(collected in 1.), data is stored in ItemLoader.
3. The data is extracted according to xpath1, the same input processor is invoked and scraped data is appended to name(collected in 1. and 2.), data is stored in ItemLoader.
4. my_title is assigned to name and the input processor is invoked.
5. As we collected all data we need now output processor is invoked and the data is populated in items by ItemLoader.

Now that you know how and when input/output processors are invoked you're going to learn how to declare them. You can do it many ways now we will declare them in the Item class.

```
class QuoteItem(scrapy.Item):
    text = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=TakeFirst()
    )

    author = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=TakeFirst()
    )

    tags = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=Join(', ')
    )
```

You can see that we use the same input processor for each field. `MapCompose(remove_tags)` removes html tags around the text which contain the data. For the first and second field we apply the same output processor which is `TakeFirst()`. It does the same thing as `Selector's extract_first()` in a spider. It returns a single value not a list. For the last field we apply `Join()` as output processor which simply appends the elements one after another with comma separation.

You can define a default input/output processor to your `ItemLoader` like this:

```
from scrapy.loader.processors import TakeFirst
from scrapy.loader.processors import Identity

loader.default_output_processor = TakeFirst()
loader.default_input_processor = Identity()
```


Scrapy ItemLoader Built-in Processors

Scrapy has some built-in input/output processors which you can use. Also, you can call any function as a processor.

```
from scrapy.loader.processors import Identity
input_processor=Identity()
```

It returns the original values it doesn't modify anything. You might use it when you've defined a different a default processor for your ItemLoader.

```
from scrapy.loader.processors import TakeFirst
output_processor=TakeFirst()
```

It returns the first value which is not null or empty out of the selected elements. It is usually used as an output processor if we want only one single value to be stored.

```
from scrapy.loader.processors import Join
output_processor=Join(separator=u' ')
```

It returns the values joined together with the separator specified in the constructor.(u' ' is the default).

```
from scrapy.loader.processors import Compose
output_processor=Compose(*functions, **default_loader_context)
```

This processor takes multiple functions in its constructor then it invokes them in the given order. So the first function will return the modified value then the next function will modify it and return it as well and so on.

```
from scrapy.loader.processors import MapCompose
input_processor=MapCompose(*functions, **default_loader_context)
```

It works the same way as Compose processor the difference is how returned values are passed to the next function. The input value(probably a list) of this processor is iterated and the first function is applied to each element of the list. The returned values of the first function are appended to create a new iterable which will be given to the next(the second) function and so on. The output iterable of the last function will be the output of the processor.

When you write a software it's obvious that sooner or later there will be a function or method which doesn't work as you expected or doesn't work at all. It's the same when you code a web scraper and it doesn't scrape a piece of data or the response you get from the server is somewhat different to what you would expect. There are bunch of possible mistakes your scraper can make while running. To find the root of the problem as soon as possible you have to debug your code. In Scrapy there are several ways to do this. So let's dive into it.

Debugging Your Web Scraper

The first thing that you and probably many developers do when something doesn't work perfectly is to put some print statements here and there and let's hope it's gonna show you what's wrong quickly. Well, it might be the first step but what if it doesn't help you? You should give scrapy's parse command a shot. It will show you what happens under the hood. Or maybe you want to go hardcore and debug with Scrapy Shell. You will definitely want to use Scrapy Shell at some point because it's a really effective way to track each of your objects while scraping. Perhaps you would rather have a look at the html response in a real browser. You will learn exactly how to debug your spiders in all the different ways now.

Logging

A really similar way to print statements is to log short messages in your methods. There are 5 logging levels in Scrapy (and Python): critical, error, warning, info and debug. As you're debugging you tend to use the debug level. But if you would like to log general messages to track your scraper you may use the other ones.

There are two ways to log a message:

```
import logging

logging.log(logging.DEBUG, "Debug message")
logging.debug("Debug message")
```

In your spider you can easily use logger because each Scrapy Spider instance has a logger object. It's a good practice to log about something significant which was done or should have done by your scraper. So when you run your scraper you can be sure if it's working or not. You might want to check if your scraper scraped data properly.

```
def parse(self, response):
    item = DataItem()
    data = response.css(some_selector)
    if (len(data) > 0):
        item['data_field'] = data
        return item
    else:
        self.logger.warning('data field list is empty here: '. response.url)
```

Scrapy Commands

Ok so after you logged the hell out of your spider and yet haven't found any solution to your problem it's time to move on and try a different debugging method. You should look what is happening under scrapy's hood. The scrapy parse command gives you a good insight on method level. You can check the scraped items by defining the name of your spider, the name of the parsing(callback) function, depth level and the website url. And if you want you can add -v (--verbose) to get information about each depth level.

```
scrapy parse --spider=toscrape -c parse_data -d 2 http://quotes.toscrape.com/ -v
```

```
>>> DEPTH LEVEL: 1 <<<
# Scraped Items -----
[]

# Requests -----
[<GET item_details_url>]

>>> DEPTH LEVEL: 2 <<<
# Scraped Items -----
[{'url': <item_url>}]

# Requests -----
[]
```

Beside scrapy parse, there are two more commands which could help you debugging: scrapy fetch and scrapy view.

scrapy fetch downloads the HTML file from the server and prints to stdout.

scrapy view opens the response in a real browser so you can see what Scrapy “sees” while scraping.

```
scrapy fetch http:scrapingauthority.com  
scrapy view http://tosrape.com
```

You can see that both commands take a url which is then downloaded.

Scrapy Shell

Here comes the monster. If you want to track and inspect your spider thoroughly Scrapy Shell is the best tool to do it. A good thing about Scrapy Shell is that you can use it in the command line and you can invoke it from your spider code as well. You can check and test each object you have: crawler, spider, request, response, settings.

You can launch the shell with a website url you want to scrape.

```
scrapy shell "http://tosrape.com"
```

Then you will see that the shell gives you a list of objects you can use and other stuff.

Now it's up to you what you want to test or check or just randomly play around.

A really useful thing Scrapy Shell can do and I like it a lot is that it's capable of testing your selectors and xpath right away. So you can check your extraction code alone don't have to run the whole scraper.

```
response.css('.text-right > h1::text').extract_first()
u'Web Scraping Sandbox'

fetch("http://toscrape.com")

response.css('.col-md-10 > h2').extract()
[u'Books']

view(response)
```

This is how to invoke the shell from your spider:

```
def parse(self, response):

    from scrapy.shell import inspect_response
    inspect_response(response, self)
```

When you invoke Scrapy Shell from your spider code the crawler stops at that point and lets you check if everything works as expected. Then when you are done you press Ctrl+Z(or Ctrl+D) to exit the shell and resume your scraper.

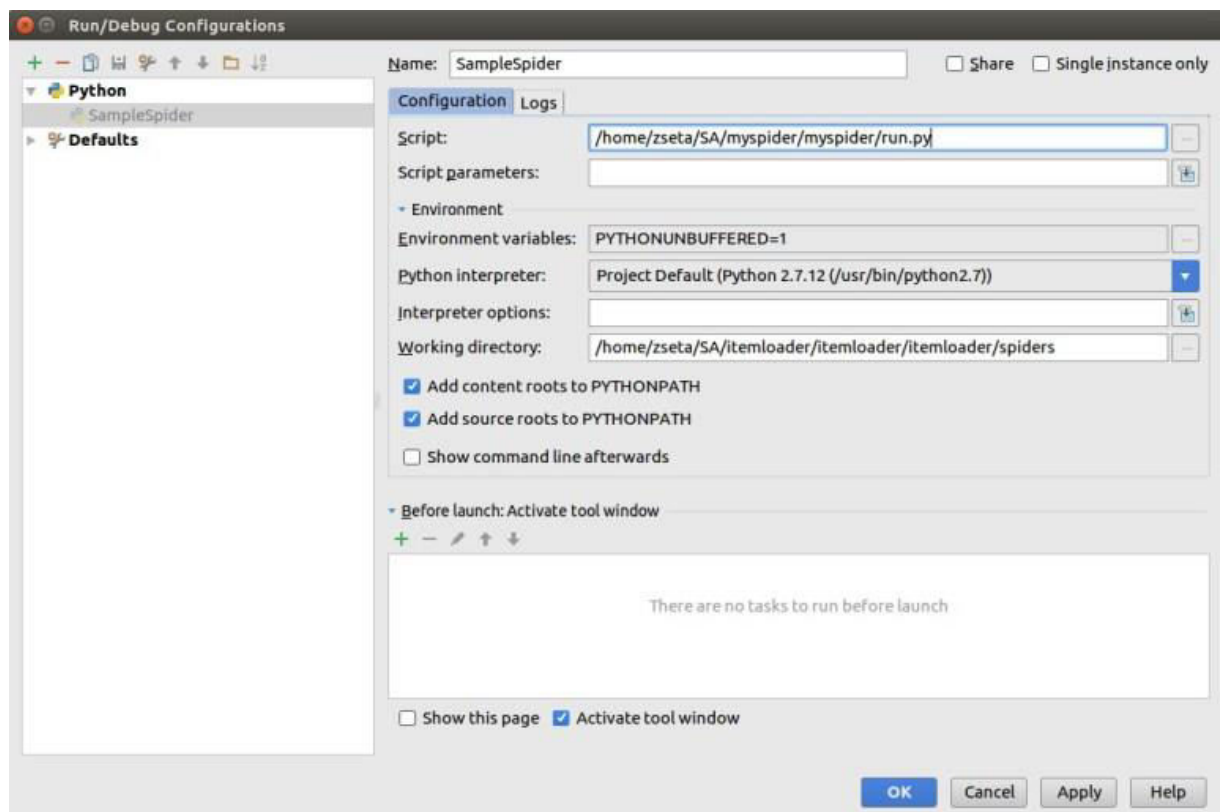
Attaching PyCharm Debugger

I usually use IntelliJ every time as my IDE so I use PyCharm(which is essentially IntelliJ) as my Python IDE. Sometimes when I don't feel like using the shell or other debugging methods I just create some breakpoints and run debugger. I show you how you can run and debug your scrapy project in PyCharm.

First you need to create a new python file(run.py) in your project directory. Then in that file you import scrapy's cmdline module and add the command you would write in the command line as a string parameter like this, this is your run.py:

```
from scrapy import cmdline  
cmdline.execute("scrapy crawl toscrape".split())
```

Finally, you should create a python run configuration and add run.py as script:



Now you can define breakpoints and debug your scraper in PyCharm.

So, you have learnt some ways to debug your web scraper. I definitely suggest using Scrapy Shell most of the time because I found that it's very useful in every case and you can test everything very quickly.

Login to Websites

When you visit the site you want to login you have your username(or maybe email) and password. That's all you need when you login with scrapy too. You don't have to deal with sending POST request, storing cookies or anything else. Scrapy does all the hard work for you.

Scrapy FormRequest

You need to use scrapy's FormRequest object. It will handle the login form and try to login with the given credentials in the constructor. This is the general use of a FormRequest:

```
def parse(self, response):
    return scrapy.FormRequest.from_response(
        response,
        formdata={'username': 'randomuser', 'password': 'topsecret'},
        callback=self.after_login
    )
```

In the example above, the response object is the HTTP response of the page where you need to fill in the login form. As you see, the FormRequest has to contain information about where the form is(response), the credentials which will be sent to the server(formdata) and a callback function that will probably scrape the page after logged in.

Before scraping the page, you should make sure you are correctly logged in. A simple solution would be searching for an error message in the

response body. If there is no login-failed error message you are good to go with scraping the page.

```
def after_login(self, response):
    if "Error while logging in" in response.body:
        self.logger.error("Login failed!")
    else:
        self.logger.error("Login succeeded!")
        item = SampleItem()
        item["quote"] = response.css(".text").extract()
        item["author"] = response.css(".author").extract()
        return item
```

If the page you are redirected to after login is not the one where the data is you should Request the right page this way:

```
scrapy.Request(url="http://scrape.this.com", callback=self.parse_something)
```

Pagination After Login

You may not find all the data you need on one page so you need some sort of pagination. As I mentioned scrapy does everything you need under the hood so takes care of staying logged in while paginate through pages. Cookies are stored automatically.

Here's a simple example of pagination on a website which has a "Next Page" button:

```
next_page_url = response.css("li.next > a::attr(href)").extract_first()
if next_page_url is not None:
    yield scrapy.Request(response.urljoin(next_page_url))
```

You include this or something similar to this at the end of your parsing method and you will be able to parse each page which need authentication.

Scrapy provides a convenient way to customize the crawling settings of your scraper. Including the core mechanism, pipelines and spiders. When you create a new scrapy project with scrapy startproject command you will find a settings.py file. Here you can customize your scraper's settings.

Scrapy Settings

Let's examine the key settings which you may have to modify for each project.

Settings.py

```
USER_AGENT = 'myscraper (http://www.scrapingauthority.com)'
```

The user agent should identify who you are. Most websites you cannot visit without a user agent.

```
USER_AGENT = 'myscraper (http://www.scrapingauthority.com)'
```

By default this is set to True so your scraper will follow the guidelines defined in the site's robots.txt. Every time you scrape a website your scraper should operate ethically.

```
ITEM_PIPELINES = {  
    'myscraper.pipelines.ExportPipeline': 100,  
    #'myscraper.pipelines.SamplePipeline': 200  
}
```

Pipelines are meant to process items right after scraping. It's important that you have to make it clear which pipelines you want to apply while scraping. In this example the second pipeline is commented out so not activated it won't be invoked.

```
SPIDER_MODULES = ['myscraper.spiders']
```

Here you have to declare where the spiders are inside your project.

```
CONCURRENT_REQUESTS = 16
```

The requests scrapy can make at the same time. This is 16 by default. Be careful when you set it to avoid damaging the website!

```
DOWNLOAD_DELAY = 0
```

The delay between requests given in seconds. Default value is 0! You might want modify this to be nicer to the website.

```
DEFAULT_REQUEST_HEADERS = {  
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',  
    'Accept-Language': 'en',  
}
```

Some website recognizes scrapy's default request headers so it might be a good idea to customize it.

```
AUTOTHROTTLER_ENABLED = True
AUTOTHROTTLER_START_DELAY = 5
AUTOTHROTTLER_MAX_DELAY = 60
AUTOTHROTTLER_TARGET_CONCURRENCY = 1.0
```

Scrapy's AutoThrottle extension is designed to adjust the speed of crawling according to the scrapy server and the crawled website server. In high-volume projects it's useful to enable.

This is a very brief guide to scrapy settings. These are the most frequent settings I adjust in almost every project. I suggest you [checking out the official doc here](#) if you want to know more.

Modify Settings in Command Line

You can override any settings in the command line with -s (or --set):

```
scrapy crawl spidername -s DOWNLOAD_DELAY=3
```

Settings for Specific Spiders

You can define settings specifically for certain spiders:

```
class SampleSpider(scrapy.Spider):
    name = 'samplespider'

    custom_settings = {
        'DOWNLOAD_DELAY': '3',
    }
```

It will override the DOWNLOAD_DELAY attribute in settings.py.

Accessing Settings Objects

In your spiders you have access to the settings through `self.settings`:

```
class SampleSpider(scrapy.Spider):
    name = 'samplespider'

    def parse(self, response):
        print("Settings: %s" % self.settings.attributes.keys())
```

If you want to access settings in your pipeline you have to override `from_crawler` method. Crawler has settings attribute:

```
@classmethod
def from_crawler(cls, crawler):
    settings = crawler.settings
    return cls(settings.getint('DOWNLOAD_DELAY'))
```

One of the most useful features of Scrapy is that it can download and process images. For example in the ecommerce world, retail companies use web scraping technology to make use of online data of products. Scraping images is necessary in order to match competitors' products with their own products. With scrapy, you can easily download images from websites with the ImagesPipeline.

Downloading Images with Scrapy

The process of downloading images:

1. Very first, you should install Pillow, an imaging library because scrapy uses it.
2. You have to enable ImagesPipeline. Go to settings.py and include ImagesPipeline as an item pipeline.
3. Again in settings.py, define IMAGES_STORE which is the path where the images should be downloaded.
4. In your item class you create these fields: image_urls and images
5. Inside your spider you scrape the URLs of the images you want to download and put it into image_urls field(It has to be a list). Now your job is done in your spider. scrapy's ImagesPipeline downloads the image(s) and the scraper waits until downloading finishes(or fails).
6. After the images are downloaded the images field will be populated with the results. It will contain a list of dictionaries of information about the image such as download path, URL, and the checksum of the file.

So let's do it step by step:

1. Install Pillow with pip:

```
pip install Pillow
```

2. Enable ImagesPipeline:

```
ITEM_PIPELINES = {'scrapy.pipelines.images.ImagesPipeline': 1}
```

3. Define a path for the images:

```
IMAGES_STORE = '/home/user/SA/ImageScraper/ImageScraper/product_
```

4. Create field in item class:

```
class BookItem(Item):  
    #other fields...  
    images = Field()  
    image_urls = Field()  
    #other fields...
```

5. Scrape URLs in your spider:

```
def parse_book(self, response):  
    book = BookItem()  
    relative_img_urls = response.css("div.item.active > img::attr(src)").extract()  
    book["image_urls"] = self.url_join(relative_img_urls, response)  
  
    return book  
  
def url_join(self, urls, response):  
    joined_urls = []  
    for url in urls:  
        joined_urls.append(response.urljoin(url))  
  
    return joined_urls
```

7. If everything works correctly you will see an output something like this:

```
{'image_name': u'1,000 Places to See Before You Die',
 'image_urls':
 [u'http://books.toscrape.com/media/cache/9e/10/9e106f81f65b293e488718a4f54a6a3f.jpg'],
 'images': [{'checksum': '6a732297bf33405b3b161f33edb2bcd5',
               'path': 'full/e81aa83c5f23d396bc2953c061af6a5c454cd3e4.jpg',
               'url':
'http://books.toscrape.com/media/cache/9e/10/9e106f81f65b293e488718a4f54a6a3f.jpg'}]}}
```

Custom Names for Image Downloading fields

You can define your own field names instead of `image_urls` and `images`. In the settings file set this:

```
FILES_URLS_FIELD = 'instead_of_image_urls_field_name'
FILES_RESULT_FIELD = 'instead_of_images_field_name'
```

Create Thumbnails of the Images

The `ImagesPipeline` can do it for you. You just have to include in `settings.py` the dimensions of the desired thumbnails and it creates them automatically. Like this:

```
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (260, 260),
}
```

,

It generates two kinds of thumbnails(a smaller and a bigger) for each images saving them into two different folder. The aspect ratio will be kept.

File Expiration

Scrapy is capable of checking if the image has been already downloaded recently so it won't download it again if not necessary. You can define how long scrapy should not download the same image again in the settings:

```
IMAGES_EXPIRES = 15 #15 days of delay for image expiration
(default:90 days)
```

Custom Filenames for Images

The default filenames of the downloaded images are based on a SHA1 hash of their URLs. But in real world it doesn't help you to know what's on the image without opening it. You can use whatever filenames you want for the images. You have to extend the ImagesPipeline. You override two functions: `get_media_requests` and `file_path`. In the first function you will return a Request object with meta information. This meta information will carry the name of the desired filename. In the second function you will simply use the meta information you passed to override the default file path. Sample code:

```
class CustomImageNamePipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        return [Request(x, meta={'image_name': item["image_name"]})
                for x in item.get('image_urls', [])]

    def file_path(self, request, response=None, info=None):
        return '%s.jpg' % request.meta['image_name']
```

Scrapy Fundamentals

II. Javascript, AJAX, Cloud

It's really hard to find a modern website which doesn't use javascript technology. It just makes it easier to create dynamic and fancy websites. When you want to scrape javascript generated content from a website you will realize that Scrapy or other web scraping libraries cannot run javascript code while scraping. First, you should try to find a way to make the data visible without executing any javascript code. If you can't you have to use a headless or lightweight browser.

Scraping Javascript Generated Content

So you come across a website which uses javascript to load data. What do you do? First, you should check the website in your real browser with JS disabled. There's a good chance that the website is fully loaded and functioning even without JS (like Amazon.com). If you need to enable JS to reach the data you want there's not much you can do but use a headless or lightweight browser to load data for scraping.

Headless and Lightweight Browsers

Headless browsers are real full-fledged web browsers without a GUI. So that you can drive the browser via an API or command line interface. Popular browsers like mozilla and chrome have their own official web driver. These browsers can load JS so you can use them in your web scraper. One such headless browser is Selenium.

On the other hand, lightweight browsers are not fully functioning browsers. They have only the main features so they can behave like real browsers. They can load JS as well. Splash is a lightweight browser.

When you choose between these two options for your web scraping project you should consider one major factor: hardware resource requirements. As I mentioned headless browsers are real full-featured browser instances working in the background. That's why they consume system resources like hell which can be a nightmare considering that a simple scraper makes thousands of requests while running. I highly discourage you from using Selenium for web scraping projects.

Instead you should try Splash. It is created to render JS content only. This is exactly what you need for web scraping. This tutorial will be a quick introduction to using Splash and Scrapy together. This tutorial will help you to get started.

Install Splash

In order to install Splash you should have Docker already installed. If you haven't, install it now with pip:

```
sudo apt install docker.io
```

Using docker you can install Splash:

```
sudo docker pull scrapinghub/splash
```

Now you can test if Splash is installed properly you have to start Splash server every time you want to use it:

```
sudo docker run -p 8050:8050 scrapinghub/splash
```

This command will start Splash service on <http://localhost:8050>

You will see this on the screen:

Splash v2.2.2

Splash is a javascript rendering service. It's a lightweight browser with an HTTP API, implemented in Python using Twisted and QT.

- Process multiple webpages in parallel
- Get HTML results and/or take screenshots
- Turn OFF images ([Run live example](#)) or use **Adblock Plus** rules to make rendering faster
- Execute custom JavaScript in page context ([Run live example](#))
- Write Lua browsing scripts:
- Get detailed rendering info in **HAR** format ([Run live example](#))

Splash is free & open source. Commercial support is also available by [Scrapinghub](#).

[Documentation](#)[Examples ▾](#)[Source code](#)

On the right side of the page you can render a website with Splash and then run a Lua script on it. When using Splash, in order to interact with JS elements(buttons, forms, etc..) you need to write Lua scripts. This tutorial will not delve in Splash scripting but you can learn about it [here](#).

Now it's time to set up our Scrapy project to work with Splash properly. The easiest way to do it is using scrapy-splash. You can download it with pip:

```
sudo pip install scrapy-splash
```

Then go to your scrapy project's settings.py and set these middlewares:

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware':
810,
}
```

The url of the Splash server(if you're using Win or OSX this should be the URL of the docker machine):

```
sudo pip install scrapy-splash
```

And finally you need to set these values too:

```
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

Now you've integrated Scrapy and Splash properly. Move on how you can use it in your spider.

SplashRequest

In a normal spider you have Request objects which you can use to open URLs. If the page you want to open contains JS generated data you have to use SplashRequest(or SplashFormRequest) to render the page. Here's a simple example:

```
class MySpider(scrapy.Spider):
    name = "jsscrafer"

    start_urls = ["http://quotes.toscrape.com/js/"]

    def start_requests(self):
        for url in self.start_urls:
            yield SplashRequest(url=url, callback=self.parse, endpoint='render.html')

    def parse(self, response):
        for q in response.css("div.quote"):
            quote = QuotelItem()
            quote["author"] = q.css(".author::text").extract_first()
            quote["quote"] = q.css(".text::text").extract_first()
            yield quote
```


Have look at what arguments you can give to SplashRequest:

- url: The URL of the page you want to scrape.
- callback: a method that will get the (html) response of the request
- execute: This is a special one it executes your custom Lua script or Javascript given as parameter to modify the response page. As follows:

```
yield SplashRequest(url=url,  
                    callback=self.parse,  
                    endpoint='execute',  
                    args={'lua_source':your_lua_script,  
                          'js_source':your_js_code})
```

- endpoint: it will define what kind of response get.

Possible values:

- render.html: It is used by default. Returns the html of the rendered page.
- render.png: Returns a PNG screenshot of the rendered page.
- render.jpeg: Returns a JPEG screenshot of the rendered page.
- render.json: Return information about the rendered page in JSON
- render.har: Returns information about requests, responses made by SplashRequest

Splash Responses

The returned response of a `SplashRequest` or `SplashFormRequest` can be:

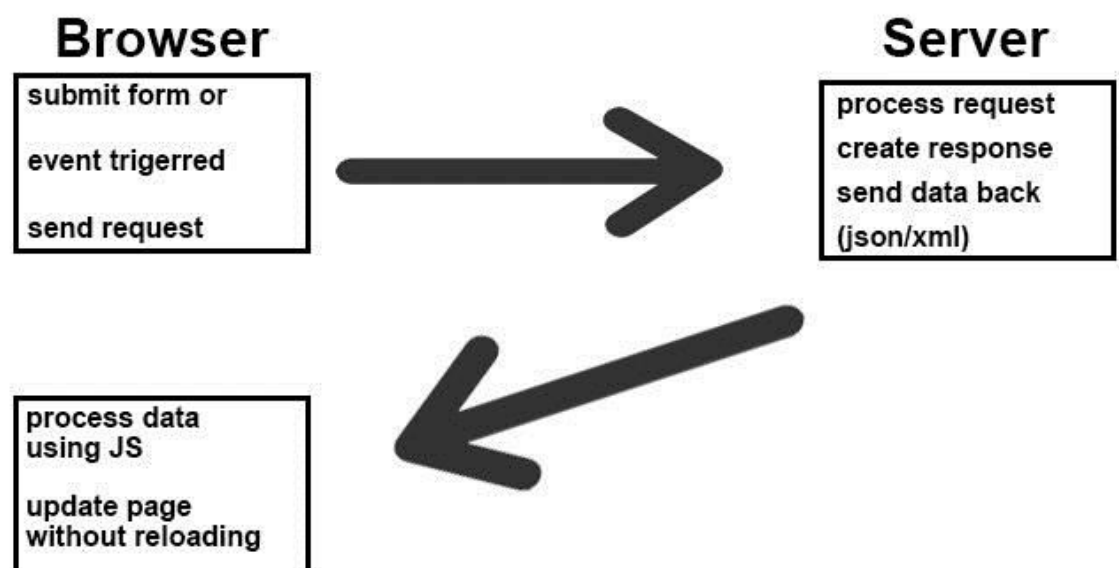
- `SplashResponse`: binary response such as `render.jpeg` responses
- `SplashTextResponse`: `render.html` responses
- `SplashJsonResponse`: `render.json` responses

These objects extend scrapy Responses so if you want to scrape from the response page you have to use `render.html` as endpoint and you'll be able to use `response.css` and `response.xpath` methods as usual.

AJAX stands for Asynchronous JavaScript And XML (nowadays JSON instead). With AJAX websites can send and receive data from the server in the background, without reloading the whole page. This technique became really popular because it makes it easier to load data from the server in a convenient way. In this tutorial I will cover two major usages of AJAX: infinite scrolling and filtering forms(ViewState). Though you may think that you need to run Javascript to be able to scrape data which was retrieved using AJAX, I'm gonna show you how to do it without launching Splash or other Javascript rendering service.

Using Scrapy to Simulate AJAX Requests

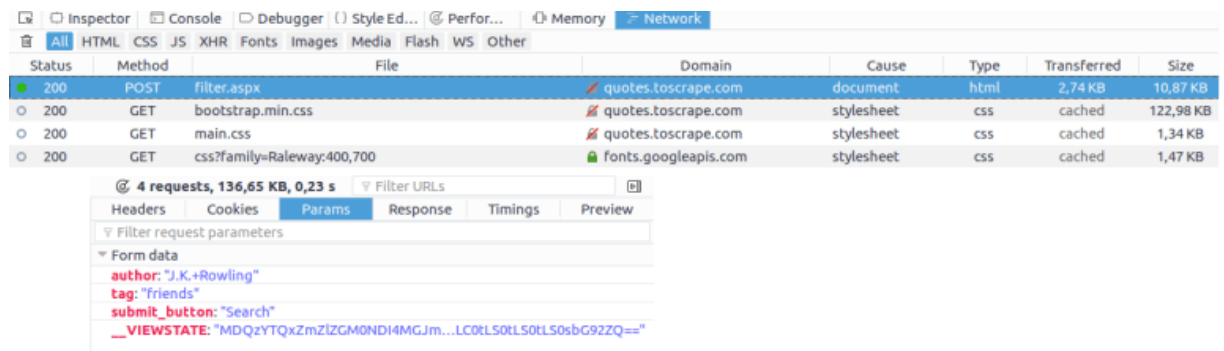
One way to get through AJAX is simply launching a headless browser which renders javascript. But now I will teach you a more effective and faster solution: inspect your browser and see what requests are made during submitting a form or triggering a certain event. Try to simulate the same requests as your browser sends. If you can replicate the request(s) correctly you will get the data you need.



Submit AJAX Form

[Take this website for example.](#)

You have to choose an *author* and then a *tag* before submitting the form. Now let's open the inspector of your browser(F12) and find *network* tab. After submitting you'll see that the user needs to define two of the sent parameters. So it means that you have to pass these two data in your request.



In Scrapy, simply use `FormRequest` which will take care of sending the parameters:

```
class AjaxScraper(scrapy.Spider):
    name = "ajaxscraper"

    start_urls = ["http://quotes.toscrape.com/search.aspx"]

    def parse(self, response):
        yield scrapy.FormRequest.from_response(response=response,
                                              formdata={'author': 'Steve Martin', 'tag': 'humor'},
                                              callback=self.parse_item)

    def parse_item(self, response):
        quote = QuotelItem()
        quote["author"] = response.css(".author::text").extract_first()
        quote["quote"] = response.css(".content::text").extract_first()
        yield quote
```

That's all you have to do to pass an AJAX form. Figure it out what parameters you should send and use `FormRequest` in your spider.

Be aware that in the case of the example website above, you have to send one request with both data. Though in the browser you do this in two phases.

Dealing with Infinite Scrolling

Infinite scrolling is an alternative to usual pagination. Instead of using *previous* and *next* buttons, it is a good way to load a huge amount of content without reloading the page. Fortunately, infinite scrolling is implemented in a way that you don't need to actually scrape the html of the page. The content is stored on the client side in a structured json or xml file most times. As you scroll, the next portion of content is being loaded.

For example [this website](#) uses AJAX to implement infinite scrolling. Inspect the page while scrolling:

| Status | Method | File | Domain | Cause | Type |
|--------|--------|----------------------------|----------------------|------------|------|
| 200 | GET | scroll | quotes.toscrape.com | document | html |
| 200 | GET | bootstrap.min.css | quotes.toscrape.com | stylesheet | css |
| 200 | GET | main.css | quotes.toscrape.com | stylesheet | css |
| 200 | GET | jquery.js | quotes.toscrape.com | script | js |
| 200 | GET | css?family=Raleway:400,700 | fonts.googleapis.com | stylesheet | css |
| 200 | GET | quotes?page=1 | quotes.toscrape.com | xhr | json |
| 200 | GET | bootstrap.min.css | quotes.toscrape.com | stylesheet | css |
| 200 | GET | main.css | quotes.toscrape.com | stylesheet | css |
| 200 | GET | quotes?page=2 | quotes.toscrape.com | xhr | json |
| 200 | GET | quotes?page=3 | quotes.toscrape.com | xhr | json |
| 200 | GET | quotes?page=4 | quotes.toscrape.com | xhr | json |

You can see that each URL is based on this template:

```
http://quotes.toscrape.com/api/quotes?page=1
```

On each endpoint you find a json file containing the data you're looking for.

Now you just have to create a spider which iterates over these URLs and extracts items from the json.

```
class ScrollScraper(Spider):
    name = "scrollingscraper"

    quote_url = "http://quotes.toscrape.com/api/quotes?page="
    start_urls = [quote_url + "1"]

    def parse(self, response):
        quote_item = QuoteItem()
        print response.body
        data = json.loads(response.body)
        for item in data.get('quotes', []):
            quote_item['author'] = item.get('author', {}).get('name')
            quote_item['quote'] = item.get('text')
            quote_item['tags'] = item.get('tags')
            yield quote_item

        if data['has_next']:
            next_page = data['page'] + 1
            yield Request(self.quote_url + str(next_page))
```

Making raw web data useful is very important nowadays. If you've followed my Scrapy tutorial series you already know how to scrape hundreds of thousands of pages with Scrapy.

Another great thing about web scraping is that you can make it work fully automatically on a server and it will give you data periodically. When using Scrapy you should use Scrapinghub Scrapy Cloud. It provides a convenient way to easily run, schedule and track your scrapers on a remote server.

Scrapy Cloud

What is Scrapy Cloud? It is a platform which is capable of deploying your scrapy spiders and scale them if you need to. Also, you can watch your spiders running then review the collected data. Furthermore, main features include monitoring spiders and tracking them while they're running, you can read the log real-time, reviewing the scraped items in your browser, and scheduling periodical jobs. The data is stored on Scrapinghub's servers. You can interact with your spiders and fetch structured data through a simple HTTP API.

Deploy Your Spider

After you coded your web scraper and it works as expected on your machine it's time to deploy to the Scrapy Cloud. [If you don't have a scrapy project that you can use to try scrapy cloud here's a sample project you can deploy.](#)

Create Project on Scrapy Cloud

First, you have to [register to scrapinghub](#). It's free. Then, you can create your first project with defining its name and that it's a scrapy project(not Portia).

Create a new project


Organization

Scraping Authority


Name

bookscraper

Build spider with



Portia



Scrapy

Cancel

Create

If you click create you land on the project's Job Dashboard. Here you can check your completed, running and scheduled jobs.

Deploy

On the side bar click Code & Deploys. You'll find here your API key and the ID of this project. You'll need these in order to deploy your spider. You have to install shub this command will help you to deploy and interact with your spider.


```
pip install shub
```

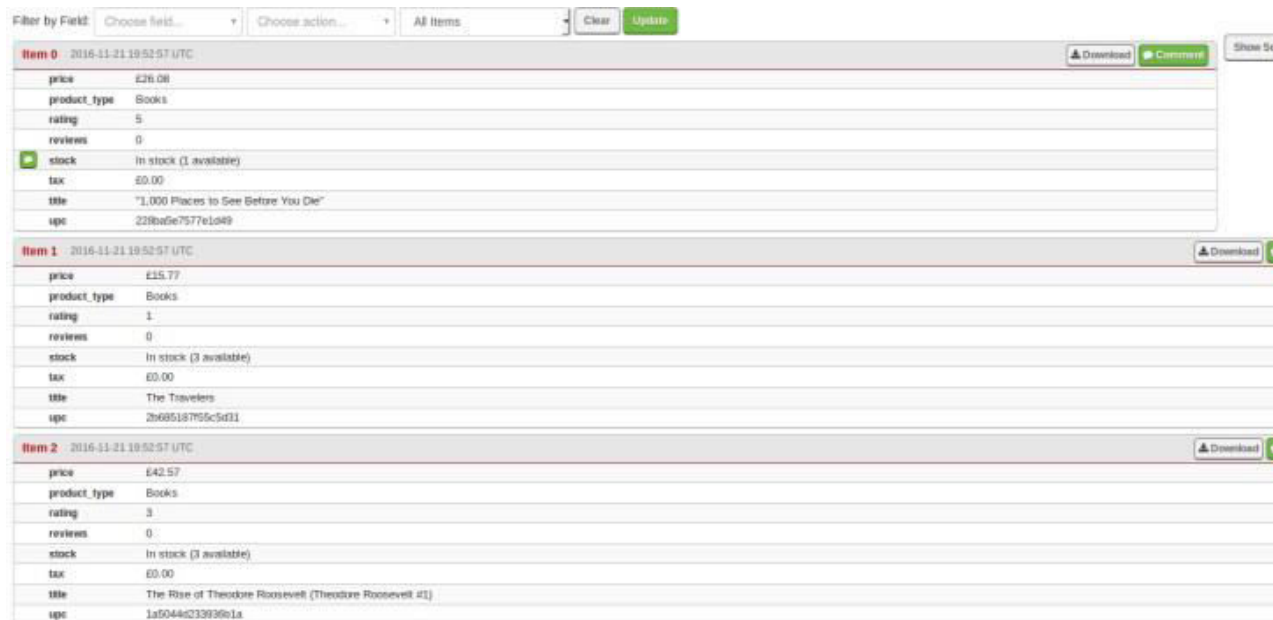
After you successfully installed the scrapinghub command line tool, cd into your scrapy project folder and run this:

```
shub deploy
```

The first time you deploy it will ask you to provide your API key and project ID.

Whoala, if you did everything well you've just deployed your first scrapy project to the cloud. Now It's up to you what your scraper needs to do and when.

On Job Dashboard you can test your scraper right away just click Run and watch the results of your scraper.



The screenshot shows the Scrapy Job Dashboard interface. At the top, there are filters for 'Filter by Field', 'Choose field...', 'Choose action...', and 'All Items'. Below the filters, there are three items listed, each with a 'Download' button and a 'Comment' button. The items are:

| Item | Timestamp | price | product_type | rating | reviews | stock | tax | title | upc |
|--------|-------------------------|--------|--------------|--------|---------|------------------------|-------|--|------------------|
| Item 0 | 2016-11-21 19:52:57 UTC | £26.08 | Books | 5 | 0 | In stock (1 available) | £0.00 | "1,000 Places to See Before You Die" | 228ba6e7577e1d89 |
| Item 1 | 2016-11-21 19:52:57 UTC | £15.77 | Books | 1 | 0 | In stock (3 available) | £0.00 | The Travelers | 2b685187f5c9d31 |
| Item 2 | 2016-11-21 19:52:57 UTC | £42.57 | Books | 3 | 0 | In stock (3 available) | £0.00 | The Rise of Theodore Roosevelt (Theodore Roosevelt #1) | 1a5044c23939e1a |

Maybe you want to add a periodic job so your scraper will run regularly:

Add Periodic Job

| | |
|--|--|
| Spiders <input type="text" value="bookscraper"/> | Choose Month <input type="text" value="Every month"/> |
| Priority <input type="text" value="Normal"/> | Choose Day of Week <input type="text" value="Every day"/> |
| Description <input type="text"/> | Choose Day of Month <input type="text" value="Every day"/> |
| Tags + | Choose Hour <input type="text" value="Every hour"/> |
| Arguments + | Choose Minutes <input type="text" value="00"/> |

Scrapinghub API

Ok now you have all the data you need on a remote web server but how do you access it?

You have to use a single API to read your data as json file, like this:

```
https://storage.scrapinghub.com/items/*project_id*?apikey=*your_api_key*&format=json
```

I have created a repository on Github which contains full working Scrapy web scraping projects. At the end, I'm introducing one of them and explain you how it works.

Scrapy Code Breakdown

BookScraper

Find it here:

<https://github.com/zseta/scrapyfundamentals/tree/master/booksrape/bookscraper>

This web scraper fetch all the information available on

<http://books.tosrape.com/> a fictional bookstore.

Specifically it scrapes these data: title, price, rating, stock, upc, product type, tax, amount of reviews and product images.

Items.py:

```
import scrapy

class BookItem(scrapy.Item):
    title = scrapy.Field()
    price = scrapy.Field()
    rating = scrapy.Field()
    stock = scrapy.Field()
    upc = scrapy.Field()
    product_type = scrapy.Field()
    tax = scrapy.Field()
    reviews = scrapy.Field()
    images = scrapy.Field()
    image_urls = scrapy.Field()
```

I defined each field I want to scrape in *BookItem*. This should be the very first step when you start to write your scraper. Define what fields you want to scrape

I have two spiders *BookScraperCss.py* and *BookScraperXPath.py* they do the same except that the first one is implemented using CSS selectors the second one uses XPATH to navigate in the HTML.

This is the full *BookScraperCss.py* and I'm gonna breakdown each element:

```
class BookScraperCss(CrawlSpider):
    name = 'bookscraper'
    start_urls = ['http://books.toscrape.com/']

    rules = (
        Rule(LinkExtractor(restrict_css=".nav-list > li > ul > li > a"), follow=True),
        Rule(LinkExtractor(restrict_css=".product_pod > h3 > a"), callback="parse_book")
    )

    def parse_book(self, response):
        book_loader = ItemLoader(item=BookItem(), response=response)
        book_loader.default_input_processor = MapCompose(remove_tags)

        book_loader.add_value("image_urls", response.urljoin(response.css(".item.active >
img::attr(src)").extract_first()))

        book_loader.add_css("title", ".col-sm-6.product_main > h1", TakeFirst())
        book_loader.add_css("price", ".price_color", TakeFirst())
        book_loader.add_css("upc", ".table.table-striped > tr:nth-child(1) > td", TakeFirst())
        book_loader.add_css("product_type", ".table.table-striped > tr:nth-child(2) > td",
TakeFirst())
        book_loader.add_css("tax", ".table.table-striped > tr:nth-child(5) > td", TakeFirst())
        book_loader.add_css("stock", ".table.table-striped > tr:nth-child(6) > td", TakeFirst())
        book_loader.add_css("reviews", ".table.table-striped > tr:nth-child(7) > td", TakeFirst())
        book_loader.add_css("rating", ".star-rating::attr(class)", TakeFirst())
        return book_loader.load_item()
```

```
Rule(LinkExtractor(restrict_css=".nav-list > li > ul > li > a"), follow=True),
```

The LinkExtractor will find every href attributes inside the given CSS selection and follow them.

```
Rule(LinkExtractor(restrict_css=".product_pod > h3 > a"),
callback="parse_book")callback="parse_book")
```

This rule will be applied on the followed URLs by the previous rule. It extracts hrefs inside the given selection and call `parse_book` function which scrapes the page on that URL.

```
book_loader = ItemLoader(item=BookItem(), response=response)
```

Inside `parse_book` you see the declaration of a itemloader. Given the actual item and the response in the constructor.

```
book_loader.default_input_processor = MapCompose(remove_tags)
```

This line sets `MapCompose(remove_tags)` as default input processor of the itemloader. It removes all the HTML tags of the field while scraping.

```
book_loader.add_value("image_urls",
response.urljoin(response.css(".item.active > img::attr(src)").extract_first()))
```

This piece of code fetches the product image URL so later it can be downloaded. The reason I don't use `add_css` is that the scraped href attr is a relative URL and in order to download the image we need its absolute URL so I called `urljoin` which creates the absolute URL.

```
book_loader.add_css("title", ".col-sm-6.product_main > h1", TakeFirst())
book_loader.add_css("price", ".price_color", TakeFirst())
book_loader.add_css("upc", ".table.table-striped > tr:nth-child(1) > td", TakeFirst())
book_loader.add_css("product_type", ".table.table-striped > tr:nth-child(2) > td", TakeFirst())
book_loader.add_css("tax", ".table.table-striped > tr:nth-child(5) > td", TakeFirst())
book_loader.add_css("stock", ".table.table-striped > tr:nth-child(6) > td", TakeFirst())
book_loader.add_css("reviews", ".table.table-striped > tr:nth-child(7) > td", TakeFirst())
book_loader.add_css("rating", ".star-rating::attr(class)", TakeFirst())
```

These lines are easy to understand. They add the selected field to the item loader. Because the selections return a list I also call TakeFirst method which takes only the first (and only) element of that list.

```
return book_loader.load_item()
```

Finally this returns the item filled with scraped data.

Pipelines

The project contains four individual pipelines: *BookRatingPipeline*, *StockPipeline*, *JsonPipeline* and *CsvPipeline*. *JsonPipeline* and *CsvPipeline* are meant only to export data right after scraping.

The *BookRatingPipeline* process „rating” field. It is needed because we have to transform text into plain integer. So this pipeline recognizes which number can be found in the text and set the value of rating accordingly. Eventually returns the item with the modify rating value.

```
def process_item(self, item, spider):
    rating = str(item["rating"])

    if ("One" in rating):
        item["rating"] = 1

    elif ("Two" in rating):
        item["rating"] = 2

    elif ("Three" in rating):
        item["rating"] = 3

    elif ("Four" in rating):
        item["rating"] = 4

    elif ("Five" in rating):
        item["rating"] = 5

    return item
```

StockPipeline is just a simple filtering pipeline. If there are more than five available of the product it drops the item.

```
class StockPipeline(object):  
  
    max_stock = 5  
  
    def process_item(self, item, spider):  
        stock = int(filter(str.isdigit, str(item["stock"])))  
        if stock > self.max_stock:  
            raise DropItem("More than 5 available here:" % item)  
        else:  
            return item
```

More on:

<http://github.com/zseta/scrapypipeline>

Where to Go Next

Contact

I write blog posts every week about web scraping, subscribe to my email list to keep up and get exclusive content:

<http://www.scrapingauthority.com/signup>

Email me here: attila@scrapingauthority.com

My Twitter: <http://twitter.com/scrapingA>

My Facebook: <http://facebook.com/scrapingauthority/>

Share

Did you like this book?

Share it with your friends!

Thank you for reading my book I hope you liked it.

You can share this book with this link:

<http://scrapingauthority.com/scrapypfundamentals>

At the end, I'd like to tell you that I'm in the making of a [package for fellows who want to delve deep into Web Scraping with Scrapy Framework](#). The package will contain [video tutorials](#), [screencasts](#), [PDFs](#) and other exclusive contents to teach you web scraping with Scrapy.

Make sure to [signup](#) to my email list to get noticed first when I publish it:

<http://www.scrapingauthority.com/signup>