Introduction to Scrapy: Web Scraping in Python

Dec 04, 2019 • 20 minutes • 4010 views

Introduction

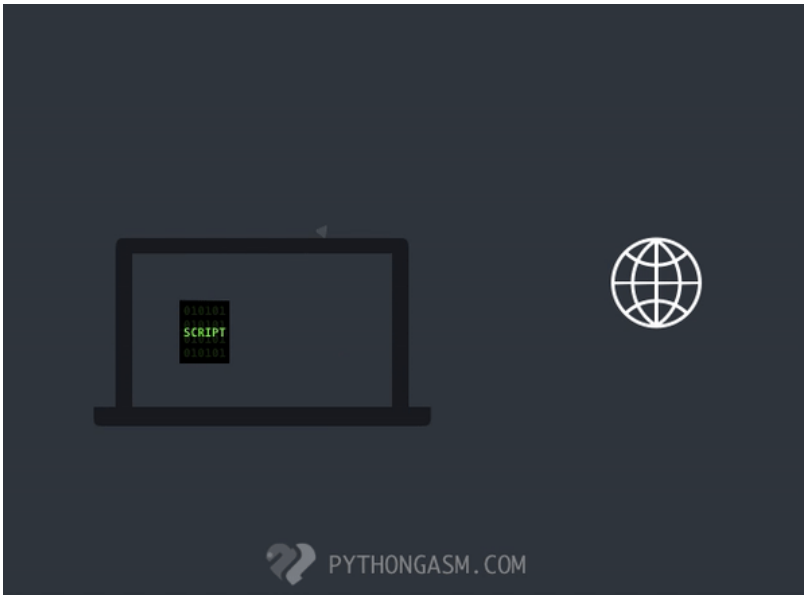
Scrapy is an open-source web scraping framework, and it does a lot more than just a library. It manages requests, parses HTML webpages, collects data, and saves it to the desired format. Hence, you don't need separate libraries for every other step. You can also use middlewares in scrapy. Middlewares are sort of 'plugins' that add additional functionalities to scrapy. There are plenty of open-source middlewares that we can attach to scrapy for extra features. This article will teach you how to collect data from webpages using scrapy. More specifically, we will be scraping Craigslist, and collect some real state data from their webpage. It would be good to have some prior HTML/CSS experience but you can proceed even if you are not familiar with HTML as a tiny portion of this article has been dedicated to HTML.

Installation

```
pip install scrapy
```

If any permission error shows up, try using sudo.

Scrapy shell



You won't want to send new requests every time you have to make small changes in your code. Instead, it's more logical to 'save' the webpage locally with one request, and then see how to call functions, and extract data. This is why we use scrapy shell for debugging. It's quick, easy and efficient.

Run this command to start the scrapy shell:

```
scrapy shell
```

We are scraping real estate section of Craigslist (New York) at:

choose real estate from the list of options)

We use the `fetch()` function to, well, fetch the response from an URL.

```
fetch("https://newyork.craigslist.org/d/real-estate/search/rea")
```

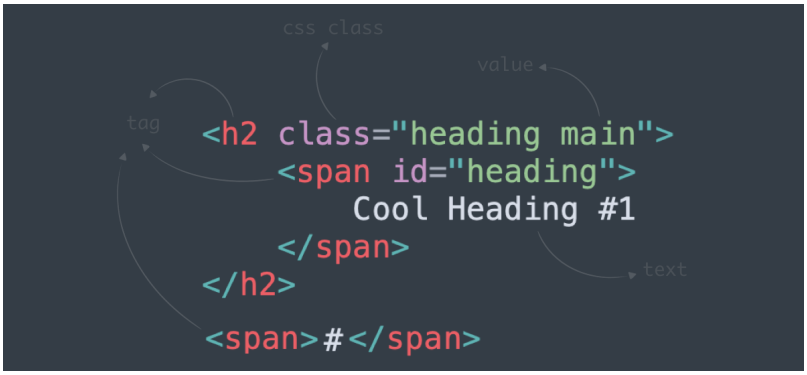
This will return a `HtmlResponse` object stored in the `response` variable.

You can view this response (that is saved locally on the device) in your browser:

```
view(response)
```

Parsing HTML

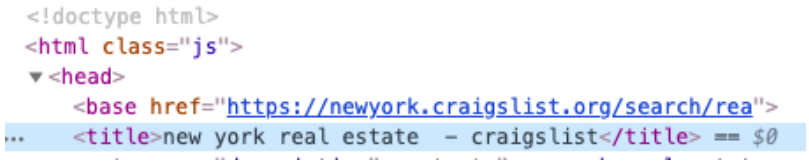
Let us have a look at this example:



To parse a webpage, we locate elements using the tag name and/or CSS class of that tag. We can instruct our spider to find the `h2` tag which has "heading main" as its CSS class, and it will return us this snippet. In this case, there's only

we can either get it by specifying the `id` OR by telling the scraper that we are looking for one that's inside an `h2` tag. The goal is to make your scraper uniquely identify the element you are looking for.

Now let us get back to the browser, and find the title of the webpage we just downloaded. Right-click on the webpage, and select Inspect. Titles are generally present inside the `head` tag.



We use this `css()` function (also called CSS Selector) to select webpage elements. For example, we can select the `title` tag like this:

```
response.css("title")
```

```
[<Selector xpath='descendant-or-self::title'
data='<title>new york real estate  - craigs…'
>]
```

`css()` function returns a list (called `SelectorList`), and each element of this list is a `Selector` object. There's only one `title` tag on the webpage, hence, it's not surprising that the length of this list is 1.

However, this isn't the desired output. We are looking for the text inside the title and not the

```
response.css("title::text")
```

```
[<Selector xpath='descendant-or-self::title/text()' data='new york real estate  - craigslist'<]
```

You can see that the data in the output has now changed to `data='new york real estate'`, which is the text inside the `title` tag.

We can get string values from `Selector` objects by calling `get()` function on them.

```
response.css("title::text")[0].get()
```

```
'new york real estate  - craigslist'
```

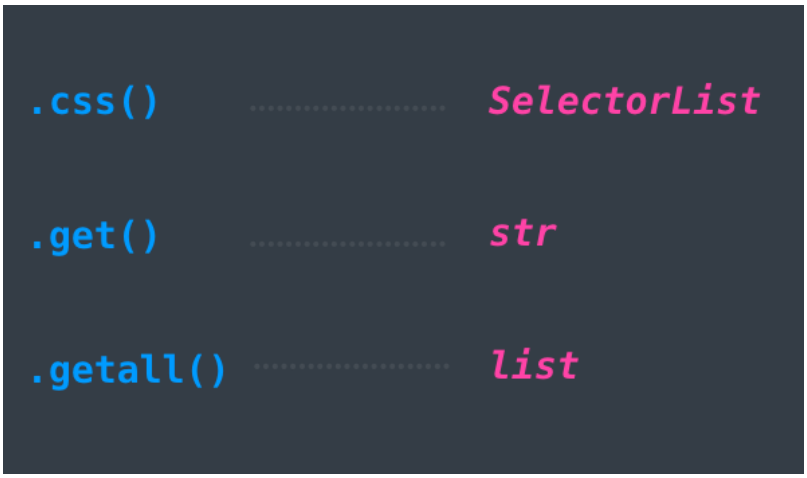
Interestingly, when you call the `get()` function directly on the list (`SelectorList`), it still gives you the same output.

```
response.css("title::text").get()
```

```
'new york real estate  - craigslist'
```

This is because when you call `get()` function without specifying the index, it returns the string value of the very first element in the `SelectorList`. In other words, you can call `get()` on a `SelectorList`, and rather than

This function simply calls the `get()` function on every element inside a `SelectorList`. Hence, `getall()` function can be called on a `SelectorList` to obtain a list of strings.



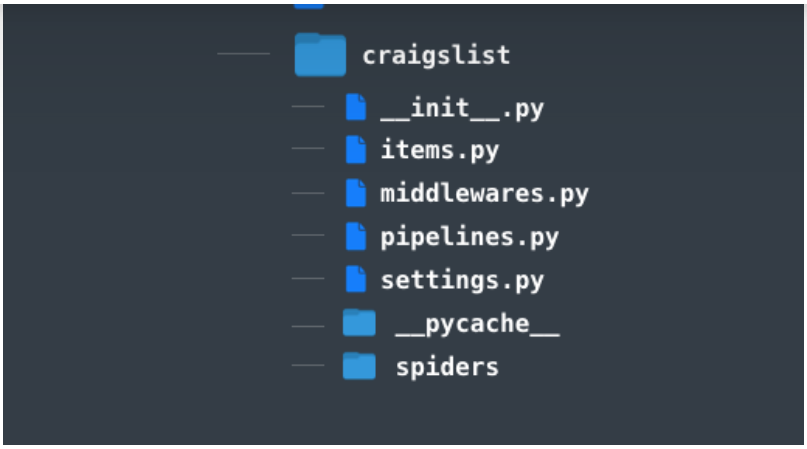
Just like we scraped the title text from the webpage, we can extract other elements from the page using these functions.

Scrapy project

Run the following command to start a new scrapy project:

```
scrapy startproject craigslist
```

This will create a folder "craigslist" in your current working directory with the following structure:



Go to the spiders directory. This is where we will be saving our spiders (crawlers). You can create a new python file in there, and start writing code. However, we can also run the following command to generate a spider with some initial code.

```
scrapy genspider realestate newyork.craigslis
t.org/d/real-estate/search/rea
```

Now open the realestate.py file inside the spiders directory. It should look like this:

```
# -*- coding: utf-8 -*-
import scrapy


class RealestateSpider(scrapy.Spider):
    name = 'realestate'
    allowed_domains = ['https://newyork.craigs
list.org/d/real-estate/search/rea']
    start_urls = ['http://https://newyork.crai
gslist.org/d/real-estate/search/rea/']
```

When we run the spider, a request is sent to all the URLs inside `start_urls`. This is a special variable that automatically calls the `parse()` function, and passes response as the argument.

`parse()` function is where we will be writing our code to parse the response. Let us define it like this:

```
def parse(self, response):
    print("\n")
    print("HTTP STATUS: "+str(response.status)
)
    print(response.css("title::text").get())
    print("\n")
```

Save the file, and run the spider with this command:

```
scrapy crawl realestate
```

```
HTTP STATUS 200
new york real estate  - craigslist
```

Now, let us extract other elements on the webpage. Inspect the HTML of the first advert. (Right-click > Inspect on the element you want to inspect):

You can see all the information about this advert (tilte, date, price, etc.) is inside this p tag with class name "result-info"


```
<p class="result-info"> == $0
  <span class="icon icon-star" role="button">…
  </span>
  <time class="result-date" datetime="2019-12-03 01:
  30" title="Tue 03 Dec 01:30:04 AM">Dec  3</time>
  <a href="https://newyork.craigslist.org/brk/reo/d/
  brooklyn-luxury-2br-apartment-for-rent/
  7028444373.html" data-id="7028444373" class=
  "result-title hdrlnk">🔥🔥🔥LUXURY 2BR APARTMENT
  FOR RENT IN PARK SLOPE BROOKLYN🔥🔥🔥</a>
  <span class="result-meta">
    <span class="result-price">$3900</span>
    <span class="housing">
                        2br -
                    </span>
    <span class="result-hood"> (PARK SLOPE,
    BROOKLYN)</span>
    <span class="result-tags">…</span>
    <span class="banish icon icon-trash" role=
  "button">…</span>
    <span class="unbanish icon icon-trash red" role=
    "button" aria-hidden="true"></span>
    <a href="#" class="restore-link">…</a>
  </span>
  ::after
</p>
```

There should be 120 (or equal to the number of listings) such snippets on the webpage.

```
allAds = response.css("p.result-info")
```

This will select all such snippets, and make a `SelectorList` out of them.

Let's try to extract date, price, ad title, and hyperlink from the first advert

```
firstAd = allAds[0]
```

```
date = firstAd.css("time").get()
```

Now extract the title and the hyperlink from the ad, which are present inside the a tag. We need to remove whitespaces in a CSS class name with . :

> Whitespaces inside the CSS function represent nesting of tags. Anything seperated with a whitespace is considered as a child tag of what's before the whitespace.

```
title = firstAd.css("a.result-title.hdrlnk::te
xt").get()
```

```
link = firstAd.css("a.result-title.hdrlnk::att
r(href)").get()
```

Also, as you can see we add `::attr()` in the argument to extract a attributes like `src`, `href` from tags.

Finally, we need to find the price which is located inside a `span` tag with class name "result-price" :

```
price = firstAd.css("span.result-price::text")
.get()
```

we did with the first Selector object

```
def parse(self, response):
    allAds = response.css("p.result-info")

    for ad in allAds:
        date = ad.css("time::text").get()
        title =  ad.css("a.result-title.hdrln
k::text").get()
        price = ad.css("span.result-price::tex
t").get()
        link = ad.css("a::attr(href)").get()

        print("====NEW PROPERTY===")
        print(date)
        print(title)
        print(price)
        print(link)

        print("\n")
```

Run this spider again, and you would see an output like this:

```
https://newyork.craigslist.org/brk/reb/d/brooklyn-bank-owned-2-family-7-bed-2/7031786990.html


=====NEW PROPERTY=====
Dec  3
Bank Owned 2 Family Home For Sale
$459900
https://newyork.craigslist.org/brk/reb/d/brooklyn-bank-owned-2-family-home-for/7031786904.html


=====NEW PROPERTY=====
Dec  3
pacific street area reo cash offer close 30 days or less one family
$0
https://newyork.craigslist.org/lgi/reb/d/west-babylon-pacific-street-area-reo/7031780313.html


=====NEW PROPERTY=====
Dec  3
beautiful condo with 1 bedroom and 1 bathroom
$489900
https://newyork.craigslist.org/mnh/reb/d/new-york-beautiful-condo-with-1-bedroom/7031780127.html


=====NEW PROPERTY=====
Dec  3
```

We have successfully scraped the data we wanted from Craigslist. You can always include additional information you might want to extract. With little tweaks, we can also scrape the next few pages of Craigslist (2,3,4, and so on).

Export data

Now it's time to structure it once we have the scraper ready, and make a useful dataset. It's pretty easy to do it.

Go to Craigslist > Craigslist > items.py

You just need to define new attributes of the `CraigslistItem` class (It's very much like a Python dictionary class):

```
import scrapy

class CraigslistItem(scrapy.Item):
    date = scrapy.Field()
    title = scrapy.Field()
```

Open the realestate.py file, and import the `CraigslistItem` class from items.py:

```
from ..items import CraigslistItem
```

Create an instance of this class, called `items`, and assign values just like you would do to a dictionary.

```
def parse(self, response):
    allAds = response.css("p.result-info")

        for ad in allAds:
            date = ad.css("time::text").get()
            title = ad.css("a.result-title.hdr
lnk::text").get()
            price = ad.css("span.result-pric
e::text").get()
            link = ad.css("a.result-title.hdrl
nk::attr(href)").get()

            items = CraigslistItem()

            items['date'] = date
            items['title'] = title
            items['price'] = price
            items['link'] = link

            yield items
```

Run the spider with an argument `-o output.csv` like this:

```
scrapy crawl realestate -o output.csv
```

You will find that the output has been saved in the parent directory (Craigslist). Apart from CSV, you can also export the data to other formats like JSON, XML, etc.

Scrapy settings

Go to craigslist > craigslist > settings.py. This file basically allows you to customise you a lot of things.

For example, before crawling a webpage, scrapy spiders visit the robots.txt file to check permissions/limitations set by the site owner for web crawlers. If a particular page that you want to scrape, is 'restricted' by the website, scrapy won't go to that page. However, you can disable this functionality by simply changing the value of `ROBOTSTXT_OBEY` to `False` in the settings.py file, and your crawler will stop following the guidelines inside robots.txt.

Similarly, you can customize user-agent of your crawler (user-agent strings are identifiers through which website recognizes what kind of request it is receiving). You can read more about user-agent here. Let's say you want to spoof the user agent of Google Chrome

```
USER_AGENT  = "Mozilla/5.0 (Macintosh; Intel M
ac OS X 10_15_1) AppleWebKit/537.36 (KHTML, li
ke Gecko) Chrome/78.0.3904.108 Safari/537.36".
```

Conclusion

This was an introduction to scrapy. As we have already discussed, there are a lot of other things that we can do with scrapy (or web scraping in general). There are many public websites that you might want to scrape, and convert there content into huge datasets for later use (visualisation, prediction, etc.). You can scrape all the tweets made by a user, for example. Hope you enjoyed reading this article.

Further readings

- Spider middleware
- Item pipeline

Permanent link to this article

pygs.me/001

Read Next

Major Differences Between Python 2 and Python 3

Comments

styling code quotes links

name comment

Arribo

Hi

Thanks for this useful tuto. these remarks : * in the first picture of code : ... allAds = response.css(“p.result-info”) --> allAds = response.css("p.result-info") ... date = ad.css("time").get() --> date = ad.css("time::text").get() ... link = ad.css(a::attr(href)).get() --> link = ad.css("a::attr(href)").get() and * from ..items import scrapy --> from ..items import CraigslistItem ... link = ad.css(a::attr(href)).get() -->

Orlando

There is 1 mistake at:

title = firstAd.css(a.result-title.hdrlnk::text).get()

It needs to be inside quotes:

title = firstAd.css("a.result-title.hdrlnk::text").get()

[Author]

Thanks a lot for the corrections @Orlando, and @Arribo, I have made the changes.

PYTHONGASM
Python articles, tutorials, and news