A Classy Spider

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

Your Spider

```
import scrapy
from scrapy.crawler import CrawlerProcess

class SpiderClassName(scrapy.Spider):
    name = "spider_name"
    # the code for your spider
    ...

process = CrawlerProcess()

process.crawl(SpiderClassName)

process.start()
```

Your Spider

- Required imports

```
import scrapy
from scrapy.crawler import CrawlerProcess
```

- The part we will focus on: the actual spider

```
class SpiderClassName(scrapy.Spider):
    name = "spider_name"
    # the code for your spider
    ...
```

- Running the spider

```
# initiate a CrawlerProcess
process = CrawlerProcess()

# tell the process which spider to use
process.crawl(YourSpider)

# start the crawling process
process.start()
```

3. Your Spider
The first part is simply the necessary importing of scrapy and the CrawlerProcess object. The second part, which happens to be the most important part for us, is the code for the actual spider. This code will tell scrapy what websites to scrape and how to scrape them with all the techniques we've built so far in this course. The code for the spider comes in the form of a class, a python object to house together methods and variables that relate to each other. We can name this class whatever we like. Though, the class must take scrapy.Spider as an input, which is why we needed to import scrapy above. The third part runs the spider, using the CrawlerProcess function. For our purposes in this course, we will only need to make sure that the spider name we pass to crawl method (in this case process.crawl) to the actual name of our spider. From here on, we will focus on the code for the spider.

Weaving the Web

```
class DCspider( scrapy.Spider ):

    name = 'dc_spider'

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )

    def parse( self, response ):
        # simple example: write out the html
        html_file = 'DC_courses.html'
        with open( html_file, 'wb' ) as fout:
            fout.write( response.body )
```

- Need to have a function called `start_requests`
- Need to have at least one parser function to handle the HTML code

4. Weaving the Web
We now narrow in on creating the actual spider. This is the class object which basically tells us what sites we want to scrape and how we want to scrape them. Again, the code may look a little complicated, but we will walk through it and see that we've built up the technique to cover most of the work. Here we have a code to scrape the DataCamp course directory. It includes all the basic pieces we need. We can name the class anything we want, here we called the class "DCspider" and defined the name variable within that class with a similar name (although we can assign any string to the name variable we want). This name variable is important for some of the action that happens under the hood of scrapy. We must have a start_requests method to define which site or sites we want to scrape, and which tells us where to send the information from these sites to be parsed. Finally, we need to have at least one method to parse to the website we scrape; we can call the parsing method (or methods) anything we want as long as we correctly identify the method within the start_requests function. All we are doing in this parser (which we named parse) is taking the HTML code and writing it to a file.

We'll Weave the Web Together

WEB SCRAPING IN PYTHON

A Request for Service

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

Spider Recall

```
import scrapy
from scrapy.crawler import CrawlerProcess

class SpiderClassName(scrapy.Spider):
    name = "spider_name"
    # the code for your spider
    ...

process = CrawlerProcess()

process.crawl(SpiderClassName)

process.start()
```

2. Spider Recall
Remember that the code to setup and run a spider roughly looked like the code we see here, where most of the work will go into coding the class we define for our spider.

Spider Recall

3. Spider Recall
We've already seen an example of a spider which effectively would save the HTML code from the DataCamp course directory into a file. So, let's now narrow in on the start_requests method.

```
class DCspider( scrapy.Spider ):
    name = "dc_spider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )

    def parse( self, response ):
        # simple example: write out the html
        html_file = 'DC_courses.html'
        with open( html_file, 'wb' ) as fout:
            fout.write( response.body )
```

The Skinny on start_requests

```
def start_requests( self ):
    urls = ['https://www.datacamp.com/courses/all']
    for url in urls:
        yield scrapy.Request( url = url, callback = self.parse )
```

```
def start_requests( self ):
    url = 'https://www.datacamp.com/courses/all'
    yield scrapy.Request( url = url, callback = self.parse )
```

- `scrapy.Request` here will fill in a response variable for us
- The `url` argument tells us which site to scrape
- The `callback` argument tells us where to send the response variable for processing

4. The Skinny on start_requests

First, within the spider class, we must define a method called start_requests which should take self as an input. The reason we don't have flexibility in this method name is because scrapy looks for the start_requests method by name within the class we define for our spider. We then have a list of the url or urls we would like to start scraping (but in this case, only one). It doesn't matter that we named this list urls, but it seems like a convenient name considering. Finally, we will take each url within the urls list and send it off to be dealt with. Now, this "send-off" is the most complicated part! But even before worrying about the "send-off" let's notice that we really did not need to loop over the urls list with only one url. We could have easily defined it without the for loop, but we wrote it originally with the loop to give you an idea of how you might construct this if you had several urls you want to initiate the spider with. Back to the "send-off": ***the yield command, which you might be familiar with already, acts kind of like a return command in that it returns values when the start_requests is run; yield is a python call, and is not specific to scrapy.*** We won't go into more detail about yield vs return here, but just note we are using yield in this method. The object we are yielding is a scrapy.Request object. We haven't seen these before, but again, that's OK. What yielding the scrapy.Request object does is send a response variable (the same response variable we are familiar with) pre-loaded with the HTML code from the url argument of the scrapy.Request call, to the parsing function defined in the callback argument.

Zoom Out

5. Zoom Out
So, if we look at the entire spider class again, what we see is that the start_request call will pre-load a response variable with the HTML code from the DataCamp course directory, and send it to the method we have named parse. As a glimpse into the future, notice that the parse method has response as its second input variable, this is the variable passed from the scrapy.Request call. Also, notice that while there are many wheels turning in the start_requests method, most of them are happening under the hood, and the only real adjustments we need to make are to define which url or urls we are going to scrape, and what callback method we want to use to parse those scraped sites.

```
class DCspider( scrapy.Spider ):
    name = "dc_spider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )

    def parse( self, response ):
        # simple example: write out the html
        html_file = 'DC_courses.html'
        with open( html_file, 'wb' ) as fout:
            fout.write( response.body )
```

End Request

WEB SCRAPING IN PYTHON

Move Your Bloomin' Parse

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

datacamp

Once Again

```
class DCspider( scrapy.Spider ):
    name = "dcspider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )

    def parse( self, response ):
        # simple example: write out the html
        html_file = 'DC_courses.html'
        with open( html_file, 'wb' ) as fout:
            fout.write( response.body )
```

You Already Know!

```
def parse( self, response ):
    # input parsing code with response that you already know!
    # output to a file, or...
    # crawl the web!
```

DataCamp Course Links: Save to File

```
class DCspider( scrapy.Spider ):
    name = "dcspider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )
    def parse( self, response ):
        links = response.css('div.course-block > a::attr(href)').extract()
        filepath = 'DC_links.csv'
        with open( filepath, 'w' ) as f:
            f.writelines( [link + '/n' for link in links] )
```

DataCamp Course Links: Parse Again

```
class DCspider( scrapy.Spider ):
    name = "dcspider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )
    def parse( self, response ):
        links = response.css('div.course-block > a::attr(href)').extract()
        for link in links:
            yield response.follow( url = link, callback = self.parse2 )
    def parse2( self, response ):
        # parse the course sites here!
```

5. DataCamp Course Links: Parse Again

Another, much more interesting and powerful example is to create a spider which crawls between different sites. We give you an example of that here. We start by first extracting the course links from the DataCamp course directory (as we did in the last chapter). Then, instead of printing anything to a file here, we will have the spider follow those links and parse those sites in a second parser. You see, finishing the first parse method, we loop over the links we extracted from the course directory, then we send the spider to follow each of those links and scrape those sites with the method parse2. Notice here that when we send our spider from the first parser to the second, we again use a yield command. But, instead of creating a scrapy.Request call (like we did in start_requests), we use the follow method in the response variable itself. The follow method works similarly to the scrapy.Request call, where we need to input the url we want to use to load a response variable and use the callback argument to point the spider to which parsing method we are going to use next.

Johnny Parsin'

WEB SCRAPING IN PYTHON

Capstone

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

Inspecting Elements

```
import scrapy
from scrapy.crawler import CrawlerProcess

class DC_Chapter_Spider(scrapy.Spider):

    name = "dc_chapter_spider"

    def start_requests( self ):
        url = 'https://www.datacamp.com/courses/all'
        yield scrapy.Request( url = url,
                              callback = self.parse_front )

    def parse_front( self, response ):
        ## Code to parse the front courses page

    def parse_pages( self, response ):
        ## Code to parse course pages
        ## Fill in dc_dict here

dc_dict = dict()

process = CrawlerProcess()
process.crawl(DC_Chapter_Spider)
process.start()
```

2. Inspecting Elements
The structure of our spider will be as you see here. You will notice all the usual suspects set up for us, including the naming variable within the spider class, and a start_requests method which directs us to the DataCamp course directory site. You will also notice at the bottom, we have an empty dictionary, called dc_dict, which is what we want to fill in with the course titles and course chapter titles during the scrape. Our first objective of scraping the course directory (to extract the course page links) will be coded in the parsing method we call parse_front within our spider class. From there, the spider will crawl to each of those course pages and fill in the dc_dict using the course titles as the keys, and a list of the course chapter titles as the items; this second order scraping method we will call parse_pages.

Parsing the Front Page

```
    def parse_front( self, response ):
        # Narrow in on the course blocks
        course_blocks = response.css( 'div.course-block' )
        # Direct to the course links
        course_links = course_blocks.xpath( './a/@href' )
        # Extract the links (as a list of strings)
        links_to_follow = course_links.extract()
        # Follow the links to the next parser
        for url in links_to_follow:
            yield response.follow( url = url,
                                   callback = self.parse_pages )
```

3. Parsing the Front Page
It remains for us to fill in the parsing code. Starting with parse_front, we will first direct to all the course block div elements (as we have done before). These course blocks divvy up the course information for each course in the directory. From each course block, we then direct to the course page link, again as we have done before. Using the extract method, we create a list of the links (as strings) that we want to follow. And finally, we will iterate through the links and yield a call to response.follow, directing the spider to crawl to each of these course pages. Note that the follow callback method is directed to parse_pages, which is the name of the parsing method we want to spider to use at the next step.

Parsing the Course Pages

```
def parse_pages( self, response ):
    # Direct to the course title text
    crs_title = response.xpath('//h1[contains(@class,"title")]/text()')
    # Extract and clean the course title text
    crs_title_ext = crs_title.extract_first().strip()
    # Direct to the chapter titles text
    ch_titles = response.css( 'h4.chapter__title::text' )
    # Extract and clean the chapter titles text
    ch_titles_ext = [t.strip() for t in ch_titles.extract()]
    # Store this in our dictionary
    dc_dict[ crs_title_ext ] = ch_titles_ext
```

4. Parsing the Course Pages

Now, to fill in the parse_pages method, we remember that we want to extract the course title and the titles of the course chapters. After inspecting the HTML on one of the course pages, we discover that the course title is defined by the text within an h1 element whose class contains the word title. So, we start by directing to this text. To extract the course title, we will call extract_first rather than extract since we want to be left with the title as string, rather than a list containing the title, as extract would leave us with. Although it is unnecessary, when we do this, we can clean the text a little, removing strange character returns that often crop up in HTML. Fortunately, strings in python already include a strip method to do this cleaning for us! Next, we want to get to the chapter titles. On inspection, we discover that these are defined as the text within h4 elements whose class is chapter__title. So, we direct the spider to these pieces of text, extract the text and clean it as before. This time, we use extract to get a list of the many chapter titles per course. We finally end by filling in our dictionary whose keys are the course titles and corresponding elements are the chapter titles. And now that we have finished this parsing method, we have finished our spider.

It's time to Weave

WEB SCRAPING IN PYTHON

Stop Scratching and Start Scraping!

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

datacamp

Feeding the Machine

Scraping Skills

- **Objective:** Scrape a website computationally
- **How**? We decide to use `scrapy`
- **How**? We need to work with:
  - `Selector` and `Response` objects
  - Maybe even create a Spider
- **How**? We need to learn XPath or CSS Locator notation
- **How**? Understand the structure of HTML

What'd'ya Know?

- Structure of HTML
- XPath and CSS Locator notation
- How to use `Selector` and `Response` objects in `scrapy`
- How to set up a spider
- How to scrape the web

EOT

WEB SCRAPING IN PYTHON