CSS Locators

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

Rosetta CSStone

- `/` replace by `>` (except first character)
  - **XPath:** `/html/body/div`
  - **CSS Locator:** `html > body > div`
- `//` replaced by a blank space (except first character)
  - **XPath:** `//div/span//p`
  - **CSS Locator:** `div > span p`
- `[N]` replaced by `:nth-of-type(N)`
  - **XPath:** `//div/p[2]`
  - **CSS Locator:** `div > p:nth-of-type(2)`

2. Rosetta CSStone
Since we are so familiar with XPath at this point, let's quickly translate between what we know in XPath to what we use in CSS Locator notation. A single forward-slash in XPath is replaced by a greater-than symbol in CSS Locator notation; so, a greater-than symbol moves us forward one generation. There is an exception where if the first character of an XPath is a single forward-slash, we ignore it. The double forward-slash in XPath is replaced by a blank space; so a blank space looks forward all generations. And again, we ignore a double forward-slash if its the first part of the xpath string. Given an XPath with square brackets filled with a number we replace it with the colon-nth-of-type call with that number as its argument.

Rosetta CSStone

XPATH

```
xpath = '/html/body//div/p[2]'
```

CSS

```
css = 'html > body div > p:nth-of-type(2)'
```

3. Rosetta CSStone
Notice that the single forward-slash between html and body as well as between div and p are replaced with greater-than symbols, but that the very first forward slash is ignored. We see that the double forward-slash between body and div is replaced by a blank space; and that the square-bracket 2 is replaced by :nth-of-type(2).

Attributes in CSS

- To find an element by class, use a period `.`
  - Example: `p.class-1` selects all paragraph elements belonging to `class-1`
- To find an element by id, use a pound sign `#`
  - Example: `div#uid` selects the `div` element with `id` equal to `uid`

4. Attributes in CSS
One good reason to learn CSS Locator notation is that selecting elements by class or id attributes uses a very simple notation. For a CSS Locator, to select elements by which class they belong to, we simply follow the tag-name by a period followed by the class name. To select an element by id, we follow the tag-name by a pound sign followed by the id.

Attributes in CSS

Select paragraph elements within class `class1` :

```
css_locator = 'div#uid > p.class1'
```

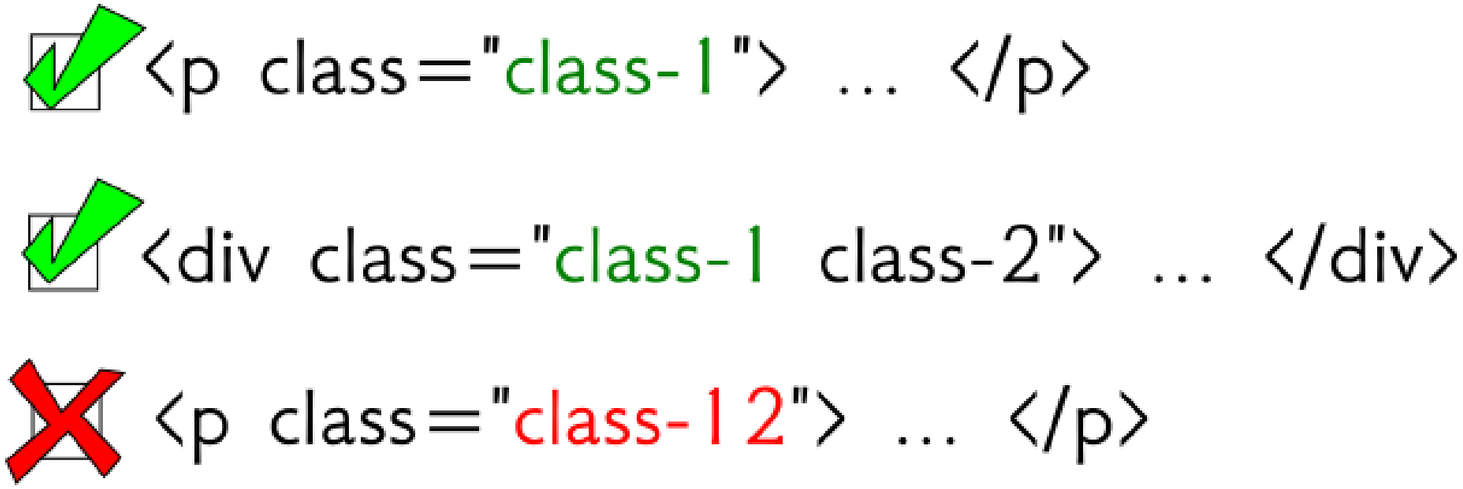
Select all elements whose class attribute belongs to `class1` :

```
css_locator = '.class1'
```

5. Attributes in CSS
To illustrate how this works, we could create a CSS Locator string which first navigates to the div element whose id is "uid" (using pound sign), then down one generation (using the greater-than sign), and finally to whichever paragraph element within that generation has a class attribute which belongs to class1 (using a period followed by the class name of interest). Alternatively, we could write a CSS Locator string which directs to all elements in the HTML document whose class attribute belongs to class1 by simply writing .class1.

Class Status

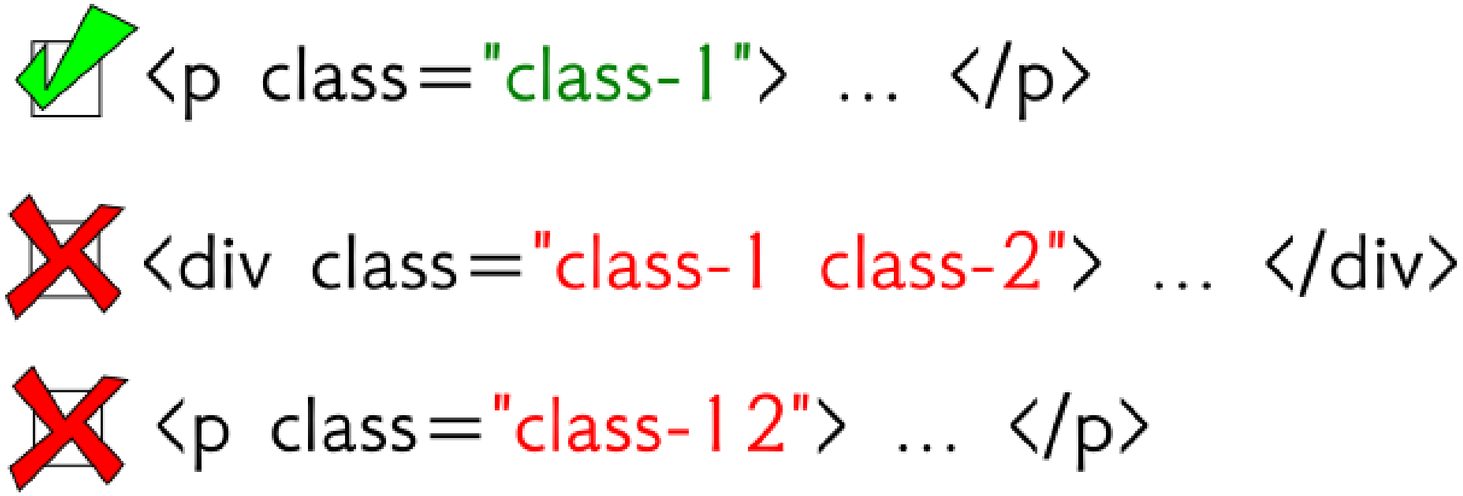
```
css = '.class1'
```



6. Class Status
Note that selecting a class like this in a CSS Locator directs us to all elements belonging to that class, even if they belong to other classes.

Class Status

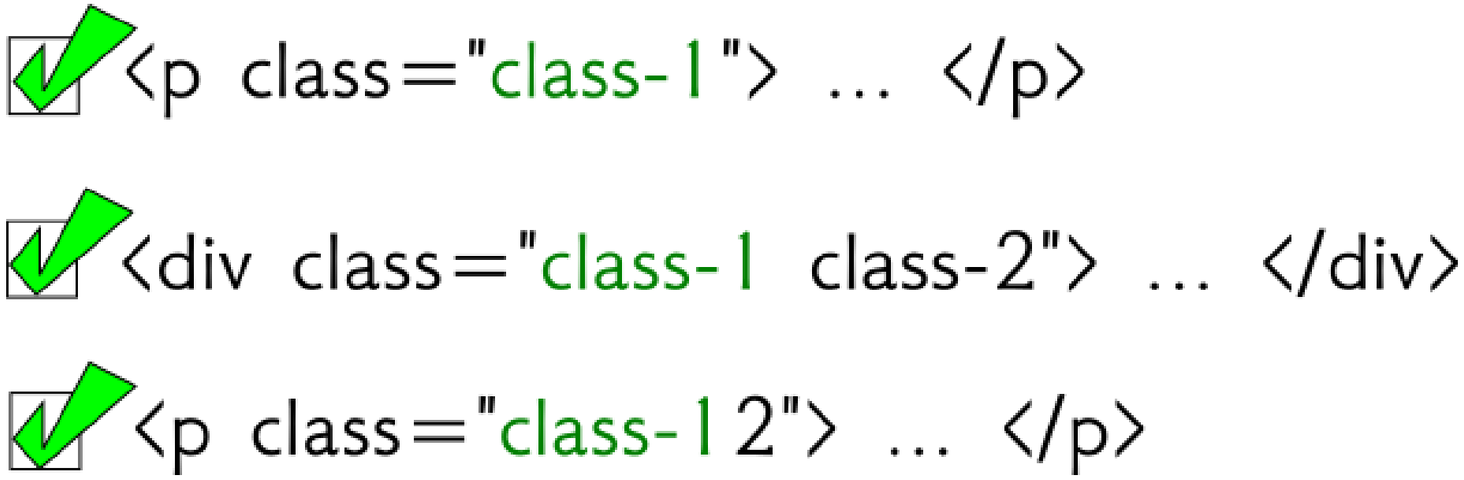
```
xpath = '//*[@class="class1"]'
```



7. Class Status
Recall that this is different than the XPath that we've learned where either using equality in the brackets forces an exact match of the class attribute ...or,

Class Status

```
xpath = '//*[contains(@class,"class1")]'
```



8. Class Status
using the contains function, which searches for all matching substrings.

Selectors with CSS

```
from scrapy import Selector

html = '''
<html>
  <body>
    <div class="hello datacamp">
      <p>Hello World!</p>
    </div>
    <p>Enjoy DataCamp!</p>
  </body>
</html>
'''
sel = Selector( text = html )
```

9. Selectors with CSS
In the last chapter we learned how to use XPath within scrapy Selectors to select specific HTML elements and from there extract the data. Well, we can do the same with CSS Locators by simply using the css method rather than the xpath method within the Selector.

```
>>> sel.css("div > p")
out: [<Selector xpath='...' data='<p>Hello World!</p>'>]

>>> sel.css("div > p").extract()
out: [ '<p>Hello World!</p>' ]
```

C(SS) You Soon!

WEB SCRAPING IN PYTHON

10. C(SS) You Soon!
Now, we've seen how to translate many of the things we've learned from our lessons using XPath to that of CSS Locators.

Attribute and Text Selection

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

datacamp

You Must have Guts to use your Colon

- Using XPath: `<xpath-to-element>/@attr-name`

```
xpath = '//div[@id="uid"]/a/@href'
```

- Using CSS Locator: `<css-to-element>::attr(attr-name)`

```
css_locator = 'div#uid > a::attr(href)'
```

2. You Must have Guts to use your Colon
Recall that to select an attribute in XPath, we use the @ symbol, so that the general form of the XPath would be to first select the element whose attribute we want to get to, followed by a forward-slash, followed by the @ symbol connected to the attribute type name we want to select. For example, if we first select the div element with id equal to uid, and from there select the href attributes from all the hyperlink children, this XPath string would do the trick. For a CSS Locator, we again direct to the element whose attribute we want to get to, and follow this by a double-colon connected to the attr attribute function. Then, the argument within the attr function is the attribute name. So, to select the same href attributes as we did in the XPath string above, we would write the following. Remember that the pound-sign tells us to select the div element by its id attribute, the greater than symbol tells us to move down one generation, and here we see our newly introduced double-colon attr piece, this is to select the desired attribute (in this case href).

Text Extraction

```
<p id="p-example">
  Hello world!
  Try <a href="http://www.datacamp.com">DataCamp</a> today!
</p>
```

- In XPath use `text()`

```
sel.xpath('//p[@id="p-example"]/text()').extract()
# result: ['\n Hello world!\n Try ', ' today!\n']
```

```
sel.xpath('//p[@id="p-example"]//text()').extract()
# result: ['\n Hello world!\n Try ', 'DataCamp', ' today!\n']
```

3. Text Extraction
We are going to switch gears a bit to hit on an extraction point we've neglected so far. Suppose that we have navigated to a paragraph element with id "p-example" and we want to direct to the text within that paragraph element. To do this, we can use the text() method within the XPath. Here we've gone ahead and put the XPath into a scrapy Selector to look at the output. By using the single forward-slash before the text method, we will direct to all chunks of text that are within that element, but not within future generations. On the other hand, if we use a double forward-slash, then we will point to all chunks of text that are within that element and within its descendants; in this case we pick up the "DataCamp" text, since it belongs to the next generation hyperlink element.

Text Extraction

```
<p id="p-example">
  Hello world!
  Try <a href="http://www.datacamp.com">DataCamp</a> today!
</p>
```

- For CSS Locator, use `::text`

```
sel.css('p#p-example::text').extract()
# result: ['\n Hello world!\n Try ', ' today!\n']
```

```
sel.css('p#p-example ::text').extract()
# result: ['\n Hello world!\n Try ', 'DataCamp', ' today!\n']
```

4. Text Extraction
Similar to attribute selection, to navigate to this text in CSS Locator notation, we again follow the element selection by the double-colon. But this time, we follow the double colon only by the word text. As we did with XPath, we can indicate whether we want only the text in the current element (but not from future generations), or if we want to also include the text within future generations. To grab only the text within the element, but not future generations, we use the double-colon without preceding it by a space. On the other hand, if we also want to include the text within future generations, we simply add a space before the double-colon. As a note: In both XPath and CSS Locator notation, the extracted text is broken up by elements. So in this example, since there is a hyperlink child, the text is broken into the chunk before the hyperlink child, the text of the hyperlink child, and the text following the hyperlink child.

Scoping the Colon

WEB SCRAPING IN PYTHON

Getting Ready to Crawl

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

datacamp

Let's Respond

Selector vs Response:

- The Response **has all the tools we learned with Selectors:**
  - `xpath` and `css` methods followed by `extract` and `extract_first` methods.
- The Response also **keeps track of the url** where the HTML code was loaded from.
- The Response **helps us move from one site to another**, so that we can "crawl" the web while scraping.

2. Let's Respond
In the next chapter, we will learn exactly how to scrape a site and load its HTML code into a scrapy variable without having to do all the work we have previously done: loading the HTML code into a string and then passing that string as a variable to a Selector object. But for now, we will focus on Response objects which already have HTML pre-loaded. You ask: "Tom, why are you making us learn a new Response object when you just taught us about Selectors?!". My "Response" to you is that you can use everything we've learned for Selectors with Responses. The Selector object was our introduction to a Response object! What makes us want to use a Response object rather than a Selector is that, on top of all the Selector functionality, the Response object keeps track of which URL the HTML code is from, and hence gives us a tool to not only scrape one single site, but crawl between links on sites and scrape multiple sites automatically!

What We Know!

- `xpath` method works like a Selector

```
response.xpath( '//div/span[@class="bio"]' )
```

- `css` method works like a Selector

```
response.css( 'div > span.bio' )
```

- Chaining works like a Selector

```
response.xpath('//div').css('span.bio')
```

- Data extraction works like a Selector

```
response.xpath('//div').css('span.bio').extract()
response.xpath('//div').css('span.bio').extract_first()
```

3. What We Know!
As an illustration of what we already know for Response variables from our Selector expertise, suppose we have pre-loaded a Response variable with the HTML from some website, and we are interested in the span elements which are children of some div element and whose class attribute is "bio". We can still use the xpath method as we have before. We can still use the css method as we've learned about in this chapter. We can chain together these methods. And we can extract the selected data using the extract or extract_first methods we already know about.

What We Don't Know

- The `response` keeps track of the URL within the response url variable.

```
response.url
>>> 'http://www.DataCamp.com/courses/all'
```

- The `response` lets us "follow" a new link with the `follow()` method

```
# next_url is the string path of the next url we want to scrape
response.follow( next_url )
```

- We'll learn more about `follow` later.

4. What We Don't Know
What we gain by using a Response object is the functionality to keep track of the URL where the HTML was scraped from, which it stores as a string in its url variable; and the ability to follow a new link using the follow method, which allows us to crawl through multiple pages for scraping. We will learn more about the follow method in the next chapter, for now just realize that this ability to "follow" links automatically makes using Response the correct choice when we want to crawl between websites for scraping.

In Response

WEB SCRAPING IN PYTHON

5. In Response
In this lesson we introduced the scrapy Response object, showing how it can be used like a Selector, but adds crawling capability. Although we still have some gaps to fill in with regards to our understanding of creating a Response and utilizing the follow method for crawling, those gaps will be the focus of our next chapter, and the culmination of our work so far. You will be able to create a web spider to crawl and scrape multiple sites automatically.

Scraping For Reals

WEB SCRAPING IN PYTHON

Thomas Laetsch
Data Scientist, NYU

datacamp

DataCamp Site

https://www.datacamp.com/courses/all

What's the Div, Yo?

```
# response loaded with HTML from https://www.datacamp.com/courses/all
```

```
course_divs = response.css('div.course-block')
```

```
print( len(course_divs) )
>>> 185
```

3. What's the Div, Yo?
We have taken the HTML for the DataCamp course directory and loaded it into a scrapy Response variable. After I manually inspected the HTML code, I noticed that each of the courses displayed on the DataCamp site belong to a div element within the class "course-block". So, let's go ahead and move to those elements using a CSS Locator. We'll store this output in the variable course_divs. By the way, at the time we scraped this site, DataCamp had 185 courses listed in this directory, and it seems we've got them all in these selected div elements.

Inspecting course-block

```
first_div = course_divs[0]
children = first_div.xpath('./*')
print( len(children) )
>>> 3
```

4. Inspecting course-block
Examining the first of the div elements in the course-block class, we notice that there are three children.

The first child

```
first_div = course_divs[0]
children = first_div.xpath('./*')
```

```
first_child = children[0]
print( first_child.extract() )
>>> <a class=... />
```

5. The first child
The first child is a hyperlink element to the course website. It also contains several other elements as children which comprise the upper portion of the course block, highlighted here. Let's note that we took the first element from the SelectorList as the variable first_child, meaning that first_child is a Selector object. So, to get to the data in the first_child element, we can easily apply the extract function.

The second child

```
first_div = course_divs[0]
children = first_div.xpath('./*')
```

```
second_child = children[1]
print( second_child.extract() )
>>> <div class=... />
```

6. The second child
The second child is another div element, which also contains several other elements which comprise the footer of the course block; the section of the course block highlighted here.

The forgotten child

```
first_div = course_divs[0]
children = first_div.xpath('./*')
```

```
third_child = children[2]
print( third_child.extract() )
>>> <span class=... />
```

7. The forgotten child
The third child is a span element which acts as an invisible container for some specific information, but really isn't visible in the course block itself.

Listful

- In one CSS Locator

```
links = response.css('div.course-block > a::attr(href)').extract()
```

- Stepwise

```
# step 1: course blocks
course_divs = response.css('div.course-block')
# step 2: hyperlink elements
hrefs = course_divs.xpath('./a/@href')
# step 3: extract the links
links = hrefs.extract()
```

8. Listful
After this inspection, we are now in the position to easily create the list of course links, our original goal. Here I present two options, though others are certainly possible. The first and possibly simplest is to use just a single CSS Locator to direct to the course-block div elements, then direct to the href attributes of the hyperlink child (the hyperlink child we found when exploring the children of the course block). From there, a quick call to extract. The second is to do this stepwise, mixing CSS Locator and XPath methods. First we collect the course divs with a CSS Locator; then direct to the href attributes of the hyperlink child using XPath; and finally extract these.

Get Schooled

```
for l in links:
    print( l )

>>> /courses/free-introduction-to-r
>>> /courses/data-table-data-manipulation-r-tutorial
>>> /courses/dplyr-data-manipulation-r-tutorial
>>> /courses/ggvis-data-visualization-r-tutorial
>>> /courses/reporting-with-r-markdown
>>> /courses/intermediate-r
...
```

9. Get Schooled
And guess what? At this point, we're done! Let's look at the list we created. We have collected all links to the courses!

10. Links Achieved
We've made it through this example and were able to really scrape the DataCamp course directory. You saw some of the exploratory methods and implementations that I would code myself for the task. In the next chapter, we will move on to building spiders!

Links Achieved

WEB SCRAPING IN PYTHON