PIC 16, Winter 2018 – Preparation 9W

Assigned 3/2/2018. To be completed by class 3/7/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- identify where in the hierarchy of an HTML (or XML) document the data in which they are interested lies, that is, which *node* it is contained in,
- write a CSS selector to identify that node (for very simple documents),
- write a subclass of Scrapy's `Item` class to create a customized dictionary-like data structure to hold information from an HTML document, and
- write a subclass of Scrapy's Spider class to define which webpages to search through, what pieces of information to extract, and what to do with it.

Tasks

- ☐ If you want to scrape data off of web pages, you have to know a tiny bit about HTML, the "markup language" with which web pages are described. Read W3 Schools HTML Intro for a very brief introduction to HTML. The main takeaway is that content is organized in a hierarchical structure of "tags". Under the "HTML Page Structure" heading, this hierarchical structure is visualized as boxes within boxes. For example, the `<head> … </head>` tags, and everything contained within them, is part of the `<html> … </html>` portion of the document; the `<title> … </title>` portion is part of the `<head> … </head>` portion, and thus part of the `<html> … </html>` portion, too. In order to tell a program what data to get from a website, we need to specify where the data is in this hierarchy.
- ☐ In order to understand the syntax for describing where the data is, you also have to know a bit about XML. Read the W3 Schools XML Intro for a brief introduction. Focus on the similarities rather than the differences. XML, like HTML, defines a hierarchy with tags. In the example, the `date` and `hour` live within the `note`.
- ☐ Let's go just a little further into this. Read through the end of "What is the DOM" W3 Schools XML DOM Intro (no need to go further), and then read about XML DOM Nodes. A node is the place in the hierarchy. We need to be able to describe which node we want the data from.
  - One notable point from the previous page (XML DOM Nodes) is that text like "`2005`" between tags like `<year>2005</year>` is not considered to be part of the element node `<year>`. It is considered to be part of a separate "text" node, beneath the element node `<year>` in the hierarchy. This text node is a *child* node of the `<year>` node, its *parent* node. This text node is also a *leaf* node because it does not have any children.
  - Note also that there are a few fundamentally different types of nodes. *Element* nodes are defined by the tags themselves. *Text* nodes are the text between tags that is not part of a tag itself. *Attribute* nodes are defined by additional text within tags.
- ☐ Now we're going to start learning the syntax for describing which node of an HTML document we are interested in. The syntax we will use is that of CSS (Cascading Syle Sheet) Selectors. CSS selectors are normally used for applying styles – colors, fonts, etc… - to the content of an HTML document. One would use a CSS selector/expression to select the parts of the HTML document to be styled, and then specify what styles to apply within curly braces `{}`. Since we're not interested in actually applying styles and we just want to select things, How CSS Selectors Work is a good introduction.
- ☐ Put your understanding to work – and learn much more – by playing the CSS Diner game.

- Now that we know how to describe what node(s) of an HTML document we want, we are ready to start the Scrapy Tutorial. Read the introduction.
    - As usual, you should try using `conda install scrapy` rather than `pip install scrapy` to ensure that Scrapy is installed for your Anaconda distribution of Python rather than some other installation.
    - You might want to follow the link to learn more about Spiders. You will write Spider *subclasses* to describe what webpage a Spider object should visit, what nodes it should extract information from (using CSS Selectors), and what links it should follow (to gather further information). Scrapy will instantiate these Spiders and send them off to do your bidding.
- Read "Creating a Project" in the Scrapy Tutorial. Note that the command `scrapy startproject tutorial` is *not* entered at a Python prompt but at an operating system command line (terminal/console). The command instructs the program `scrapy` to create a new project called `tutorial`. This creates a folder containing template Scrapy project files in whatever directory the command prompt was in before calling the command.
- BTW, "URL", or Universal Resource Locator, is the formal term for a "web address"; see here.
- Follow "Our First Spider". The Spider will begin at one of the `urls` `yield`ed by the generator function `start_requests`. (Recall that `yield` is like `return`, but the next time the generator function is called it picks up where it left off - right after `yield`. In this case, that's at the beginning of the loop for the next `url` in `urls`). The entire website located at that URL – the entire (mostly HTML) text that would load if you were to browse to that address – is put into an object called a `Response` (so named because when the Spider requests the data from a URL, the server at that URL *respond*s with the website). That `Response` object is then passed into the function `parse`, so named because it is going to *parse* the `Response` for the information we're interested in. In the example `parse` function, the line `page = response.url.split("/")[-2]`
    - `split`s the `url` of the `response` (really just the URL we provided) `http://quotes.toscrape.com/page/1/` at each "/" to generate a list of strings, like `['http:', '', 'quotes.toscrape.com', 'page', '1', '']`, and
    - stores the second to last string (`1`) from the list in `page`.

  Then `filename = 'quotes-%s.html' % page`
    - replaces the `%s` in the string with the value of page (`1`), and
    - stores the resulting string in the variable `filename`.

  Finally, it opens the file with that name (or rather, creates a file with that name) and writes the `body` of the response (the entire webpage – not just what's within the `<body>…</body>` tags of the webpage) to the file. You're right, we haven't even used all the CSS stuff yet. We're staring by saving the *entire* file; not just information from a particular node. And we're not making use of our `Item` yet, either! We're just saving all the contents of the webpage to a file!
  It also seems to log what it just did somewhere, but we don't need to bother with that.
- Read "How to run our spider", run the example (again the command is at the OS command line, not a Python prompt), and take a look at the resulting `quotes-1.html` in a text editor (not in a web browser). Note that this is the same as the HTML from the original website. (You can right click and select "View Page Source" or similar, depending on your browser, to see the website HTML.)
- Read "What just happened under the hood". When we try to visit a website, that's called a web "request", so Scrapy creates a "`Request`" object for each URL. When the `Request` is complete, it returns a "`Response`" object (the serve *responds* with data) and that object (the contents of the webpage) is fed into the "callback function", that is, the `parse` method.

- ☐ Read “A shortcut to the `start_requests`” method. It turns out we didn’t need to worry about writing a generator function after all; the base `Spider` class takes care of it for us.
- ☐ Now we’ll actually start thinking about extracting specific information from the website. Read “Extracting data”. You can ignore the part about XPath if you want, although I can provide additional information about XPath later for those who are interested.
  It is often difficult to write CSS expressions correctly, just like it is difficult to get regular expressions right, so it’s helpful to test CSS expressions at a command prompt just like it is helpful to test regular expressions at http://pythex.org/ before try to use them in code. That’s what we’re using the Scrapy shell for.
- ☐ Finish the tutorial, at least up to “Following Links”. Note that you can load the `.json` file you created using what you learned way back in the day about the `json` module in Preparation 8.
- ☐ You’re not expected to be an expert in writing CSS Selectors or Spiders yet; just following the tutorial is a lot to absorb! Try to understand the broad concepts (nodes, CSS selectors, the role of the `parse` method, etc…); we’ll learn more about how to apply it all soon.