10.3.2 Practice with Splinter and BeautifulSoup

Robin is feeling more comfortable with the different HTML components used to build webpages, and she knows that the data she wants to scrape will be nested within different HTML tags. An HTML page can get very confusing very quickly, so Robin would like to practice on a less sophisticated site first. There are several sites available specifically for newly minted web scrapers to practice and hone their skills with Splinter and BeautifulSoup. These practice sites contain several different components that we'll encounter out in the wild: buttons to navigate, search bars, and nested HTML tags. It's a great introduction to how the tools we'll use work together to gather the data we want.

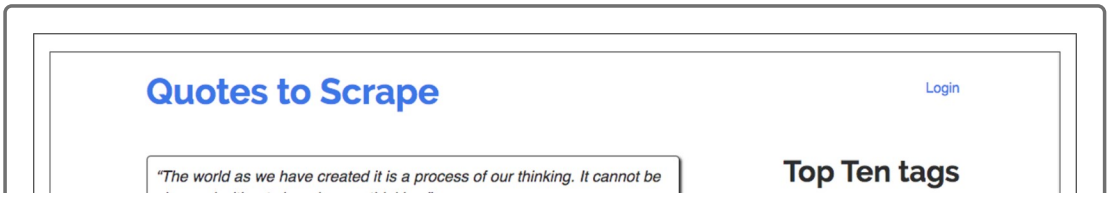
Before we start scraping things directly from a Mars website, let's practice on another, less-involved site first. Below is how our notebooks should currently look:

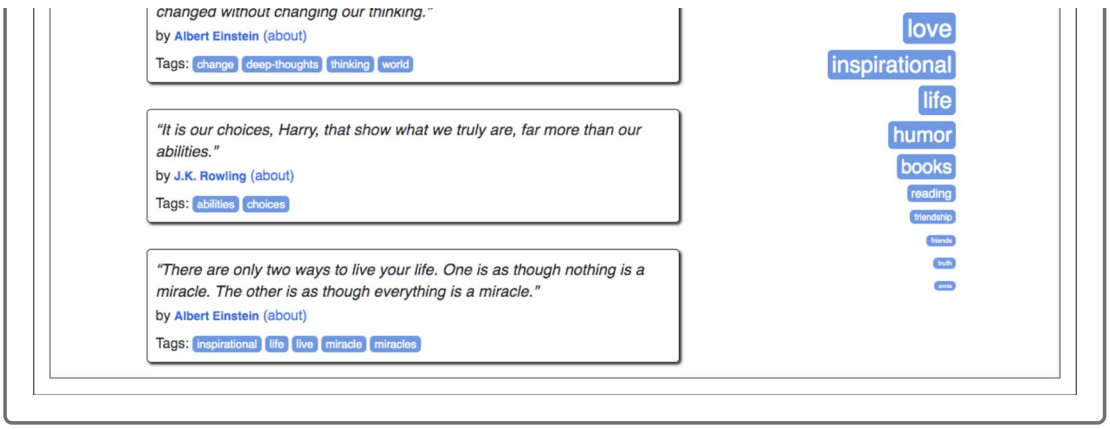
```
# Import Splinter and BeautifulSoup
from splinter import Browser
from bs4 import BeautifulSoup as soup
from webdriver_manager.chrome import ChromeDriverManager

# Set up Splinter
```

```
executable_path = {'executable_path': ChromeDriverManager().install()}
browser = Browser('chrome', **executable_path, headless=False)
```

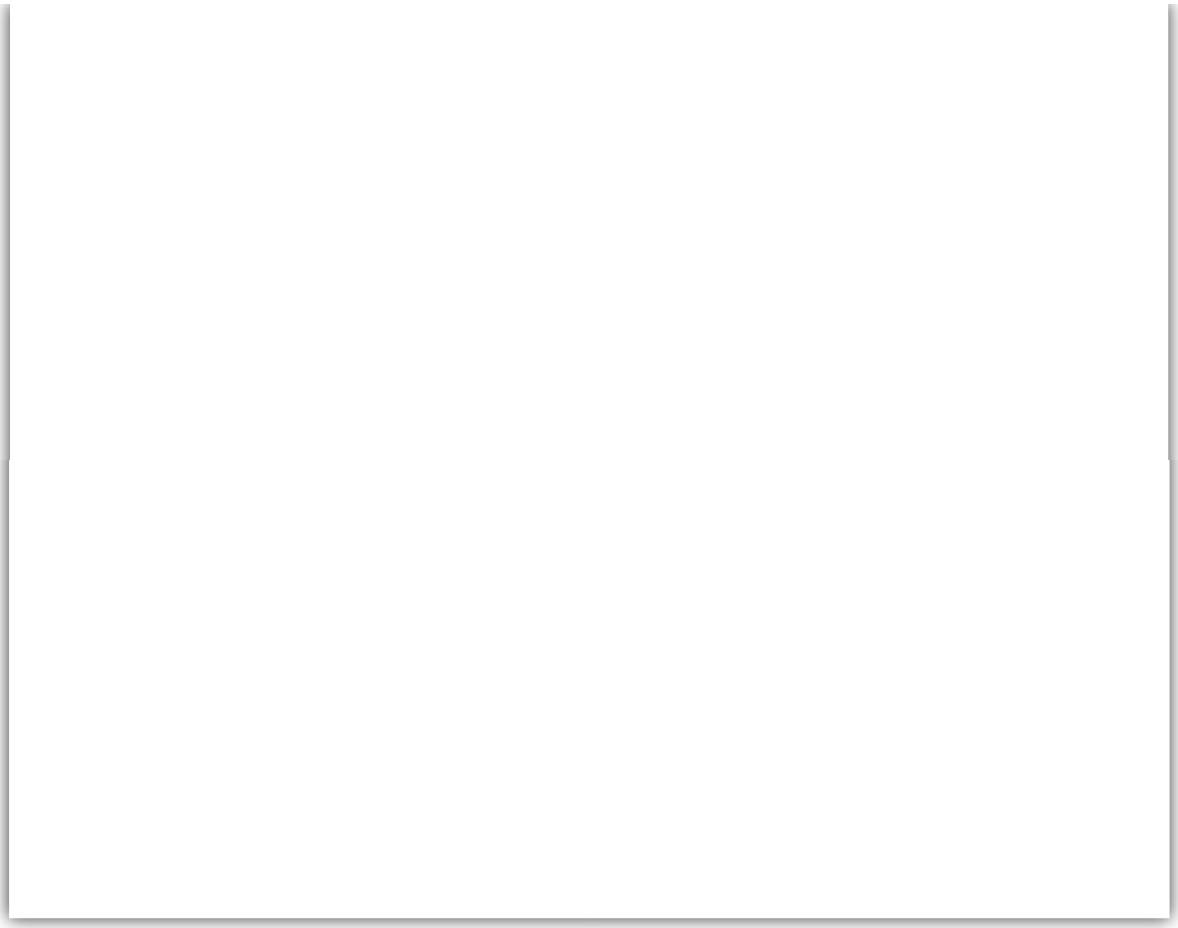
In the fourth cell, we'll scrape data from a website specifically created for practicing our skills: **Quotes to Scrape** **(http://quotes.toscrape.com/)**. Open the website in a browser and familiarize yourself with the page layout.





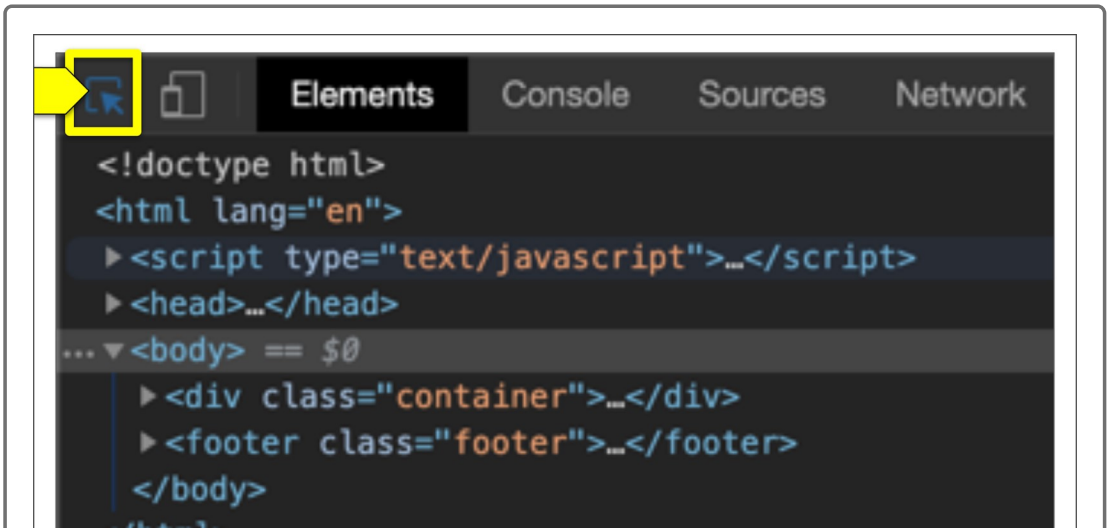
Scrape the Top 10 Tags

Interacting with webpages is Splinter's specialty, and there are lots of things to interact with on this one, such as the login button and tags. Our goal for this practice is to scrape the "Top Ten tags" text.



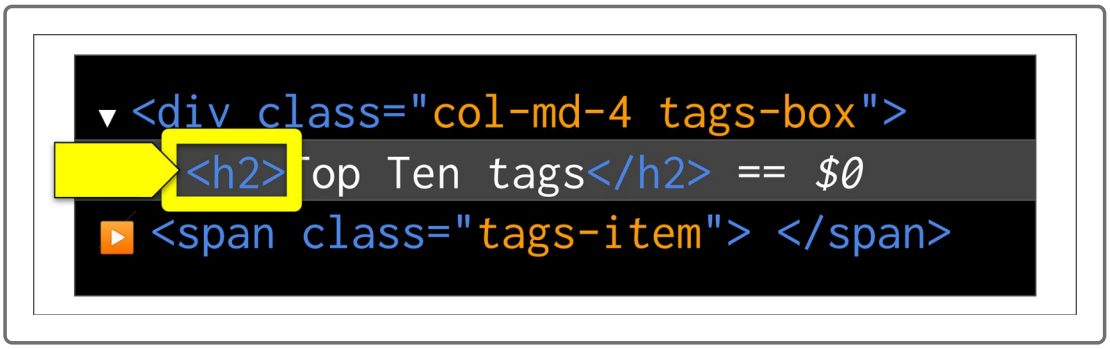
Before we start with the code, we'll want to use the DevTools to look at the details of this line. Right-click the webpage and select "Inspect." From the DevTools window, we can actually select an element on the page instead of searching through the tags.

First, select the inspect icon (the one to the far left).



Then, click the element you want to select on the page, such as the humor tag. This will direct your DevTools to the line of code the humor tag is nested in.

That was really quick. Sometimes we'll still have to dig through the tags to find the ones we want, but being able to select items directly from the webpage helps scale down time immensely. So with this shortcut, we've been able to select the `<h2 />` tag holding the text we want.



With this, we know that our data is in an `<h2 />` tag, and that's great! We've narrowed down where our data is hanging out. But what if there is more than one `<h2 />` tag on the page? When scraping one particular item, we will often need to be more specific in choosing the tag we're scraping from. We can narrow this down even further by using the search function in our DevTools.

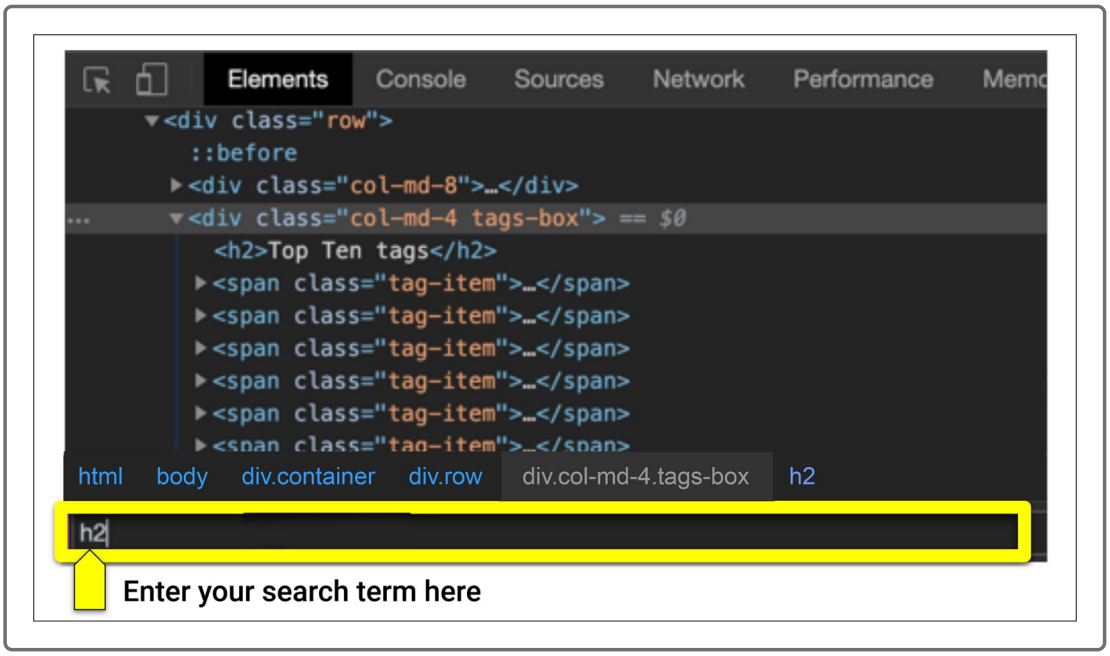
Search for Elements

Searching within the HTML code is another useful way to quickly find items we're looking for. Earlier, we were able to select a particular component from the page with the select tool. But there are times where we need to know how many of a certain type of tag are in the page. For example, the title we want to scrape is in an `<h2 />` tag, but there are several others on the page as well. Knowing this, we can expect to tailor

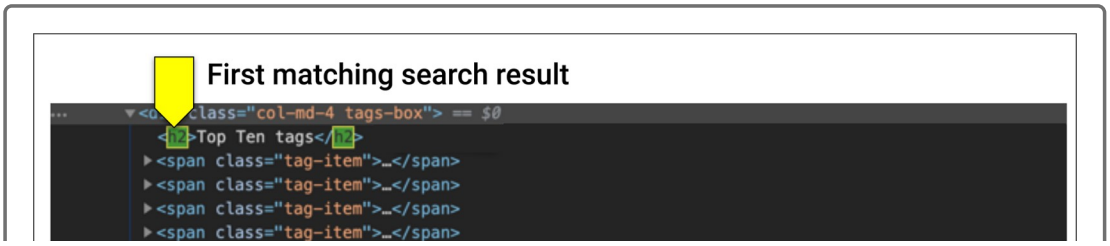
our code to pull only the `<h2 />` tag we want. First, let's practice searching in the HTML.

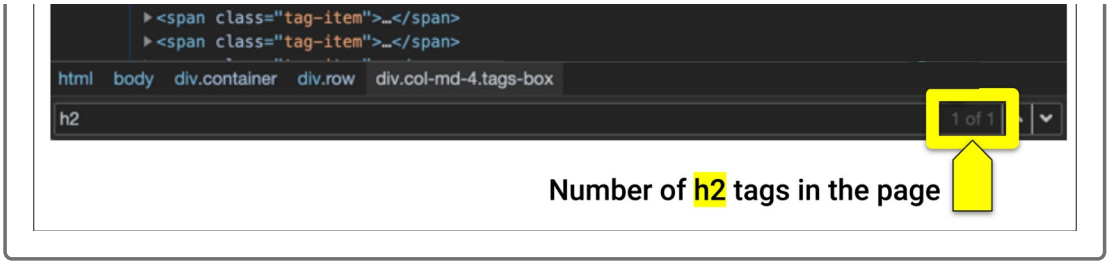
With the DevTools still active, press Command + F if you use a Mac, or CTRL + F if you use a Windows computer. This activates the search functionality, only instead of searching the webpage, we're searching the HTML of the webpage. So if we search for all of the `<h2 />` tags in the document, we'll know if we need to make our search more specific by adding attributes such as a class name or an id.

In the search bar that we just activated, type **h2** and then press Enter on your keyboard.



The result of our search immediately shows us two things: that the first tag we've searched for is highlighted, and also the number of those tags in the document.





Because there is only "1 of 1" h2 tags in the document, we know that we can scrape for an `<h2 />` without being more specific. In most other cases, we'll need to include a class or id, so we'll practice that in a little bit.

Scrape the Title

Now let's scrape that title. In the next cell in Jupyter Notebook, type the following:

```
# Visit the Quotes to Scrape site
url = 'http://quotes.toscrape.com/'
browser.visit(url)
```

This code tells Splinter which site we want to visit by assigning the link to a URL. After executing the cell above, we will use BeautifulSoup to parse the HTML. In the next cell, we'll add two more lines of code:

```
# Parse the HTML
html = browser.html
html_soup = soup(html, 'html.parser')
```

Now we've parsed all of the HTML on the page. That means that BeautifulSoup has taken a look at the different components and can now access them. Specifically, BeautifulSoup parses the HTML text and then stores it as an object.

In our code, we're using `'html.parser'` to parse the information, but there are other options available as well.

In our next cell, we will find the title and extract it.

```
# Scrape the Title
title = html_soup.find('h2').text
title
```

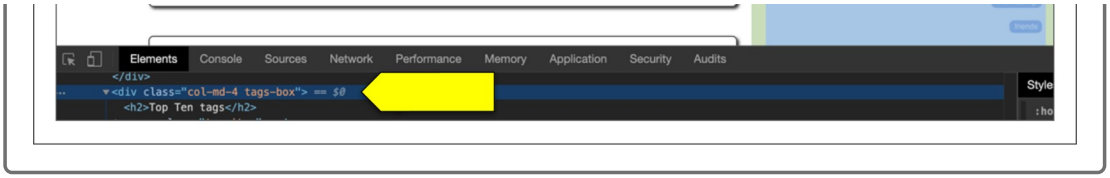
What we've just done in the last two lines of code is:

1. We used our html_soup object we created earlier and chained `find()` to it to search for the `<h2 />` tag.
2. We've also extracted only the text within the HTML tags by adding `.text` to the end of the code.

We've completed our first actual scrape. Let's practice again, this time using Splinter to scrape the actual tags to go with the title we just pulled.

Scrape All of the Tags

Using our DevTools again, look at the code for the tags. We want all of the tags instead of just one, so we want to first use our select tool to highlight the `<div />` container that holds all of the tags.



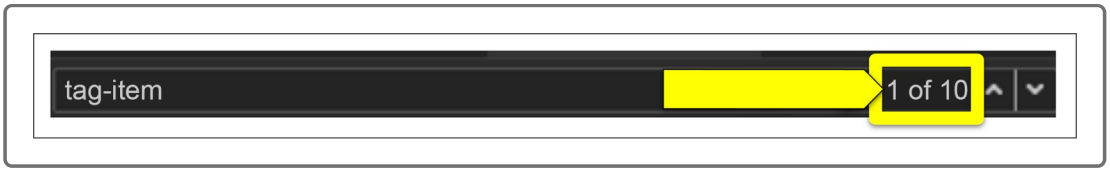
Notice that the `<div />` container holding all of the tags has two classes. The `col-md-4` class is a Bootstrap feature. Bootstrap is an HTML and CSS framework that simplifies adding functional components that look nice by default. In this case, `col-md-4` means that this webpage is using a grid layout, and it's a common class that many webpages use. We'll dive into that more later.

The other class, `tags-box`, looks custom, though. Let's make sure first by searching for it using our search box.

After searching for `tags-box`, we can see that only one result is returned. This means that it's unique in the HTML and can be used to locate specific data. Next, expand the `tags-box` div to take a look at the contents.

From here, we can see a list of `<span />` elements, each with a class of `tag-item`. Open some of the `<span />` elements to see what they contain; if you see `<a />` elements with the names in the list that we're targeting, then we're in the right place.

Since there are 10 items in the list displayed in the browser, let's use the dev tools' search function to verify the list item count. Search for `tag-item` and note the number of returned results. If there are 10, then we're ready to go.



Looks great! Let's scrape each of them. In the next cell of your Jupyter Notebook, type the following:

```
# Scrape the top ten tags
tag_box = html_soup.find('div', class_='tags-box')
# tag_box
tags = tag_box.find_all('a', class_='tag')

for tag in tags:
    word = tag.text
    print(word)
```

This code looks really similar to our last, but we've increased the difficulty a bit by incorporating a `for` loop, but let's start at the beginning.

The first line, `tag_box = html_soup.find('div', class_='tags-box')`, creates a new variable `tag_box`, which will be used to store the results of a search. In this case, we're looking for `<div />` elements with a class of `tags-box`, and we're searching for it in the HTML we parsed earlier and stored in the `html_soup` variable.

The second line, `tags = tag_box.find_all('a', class_='tag')`, is similar to the first but with a few tweaks to make the search more specific. The new "tags" variable will hold the results of a `find_all`, but this time we're searching through the parsed results stored in our `tag_box` variable to find `<a />` elements with a `tag` class.

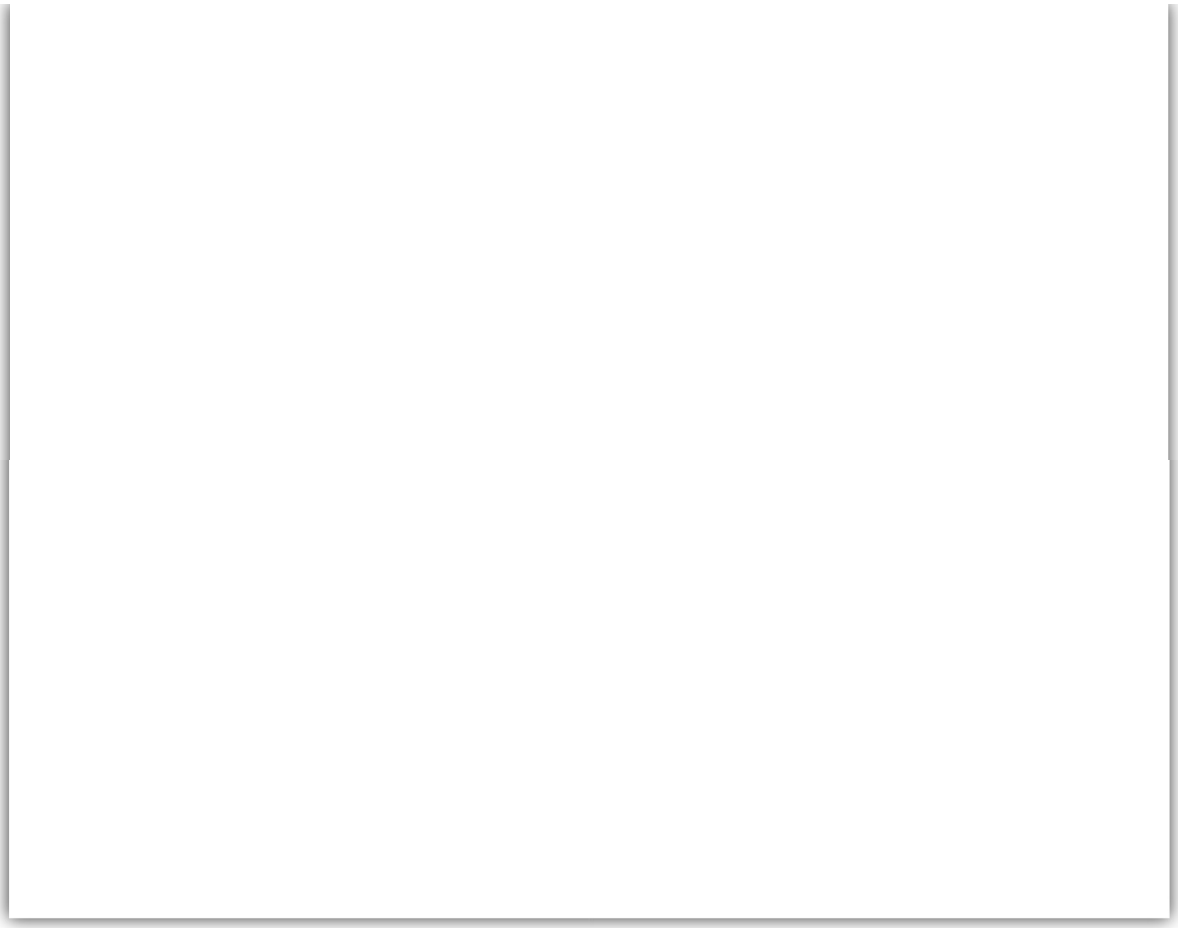
We used `find_all` this time because we want to capture all results, instead of a single or specific one.

Next, we've added a `for` loop. This `for` loop cycles through each tag in

the `tags` variable, strips the HTML code out of it, and then prints only the text of each tag.

Scrape Across Pages

Now that we've practiced scraping items from a single page, we're going to up the ante by scraping items that span multiple pages. Our next section of code will scrape the quotes on the first page, click the "Next" button, then scrape more quotes and so on until we have scraped the quotes on five pages.



We have already created the Browser instance and navigated to the `http://quotes.toscrape.com/` page with the `visit()` method. But, if you'd like to create the Browser instance again, run the following code in a new cell.

```
url = 'http://quotes.toscrape.com/'
browser.visit(url)
```

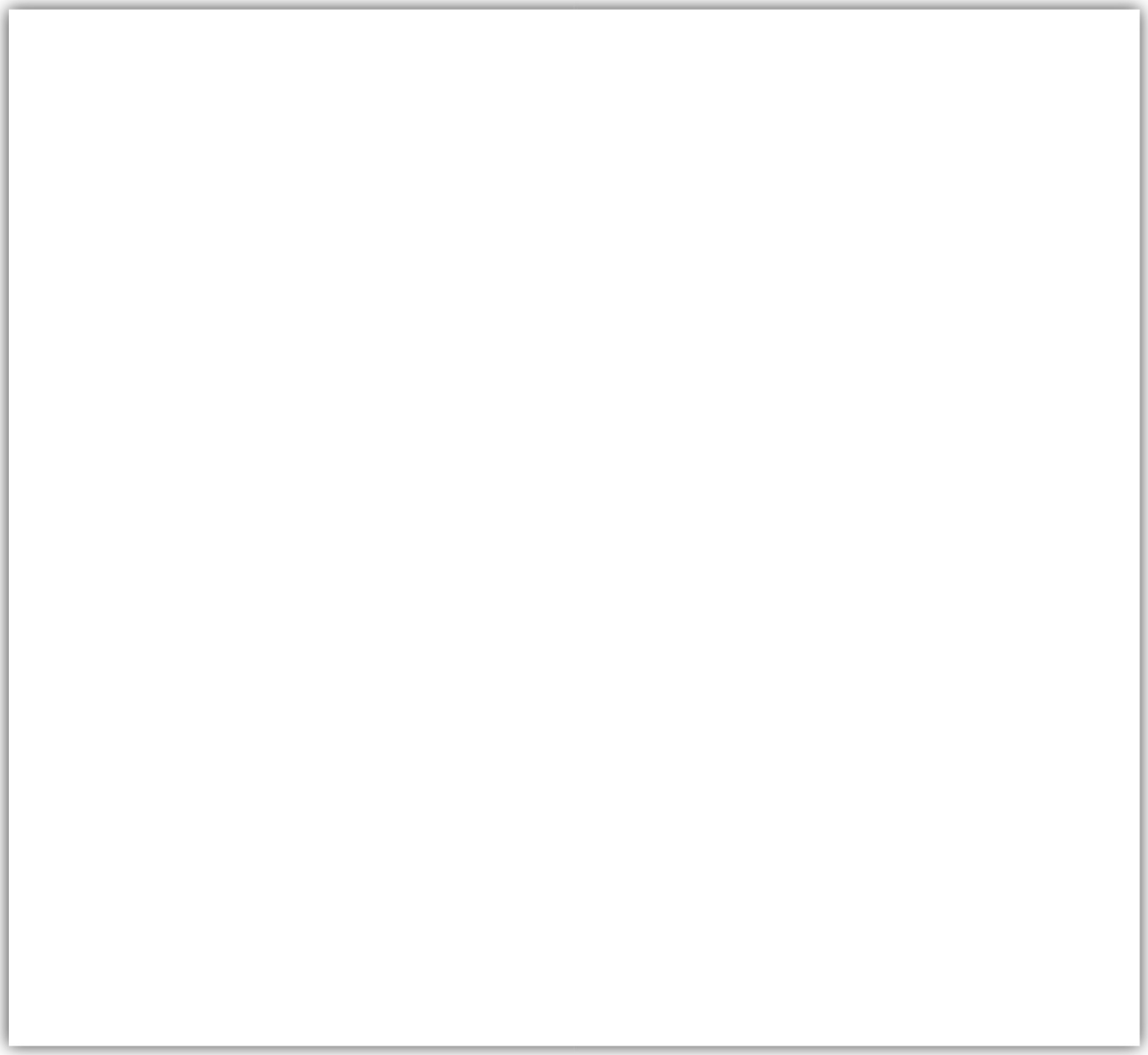
In the next cell, we'll create a `for` loop that will do the following:

1. Create a BeautifulSoup object
2. Find all the quotes on the page
3. Print each quote from the page
4. Click the "Next" button at the bottom of the page

We'll use `range(1, 6)` in our `for` loop to visit the first five pages of the website.

```
for x in range(1, 6):
   html = browser.html
   quote_soup = soup(html, 'html.parser')
   quotes = quote_soup.find_all('span', class_='text')
   for quote in quotes:
      print('page:', x, '----------')
      print(quote.text)
   browser.links.find_by_partial_text('Next').click()
```

Now let's take a look at what the code is doing.



It's important to note that there are many ways that BeautifulSoup can search for text, but the syntax is typically the same: we look for a tag first, then an attribute. We can search for items using only a tag, such as a `<span />` or `<h1 />`, but a **class** or **id** attribute makes the search that much more specific.

By including an attribute, we have a far better chance of scraping the data we want.

Go ahead and run the code in this cell. Thanks to our print statements, five pages worth of quotes should be right at our fingertips.

Now test your skills in the following Skill Drill.

SKILL DRILL

Stretch your scraping skills by visiting **Books to Scrape** **(http://books.toscrape.com/)** and scraping the book URL list on the first page.