11.5.1

# Performing an Automated Web Scrape

**Robin** now feels more familiar with HTML tags and how they fit together to create a webpage. She also has the necessary tools installed to get started with her web scraping. So, she feels eager to dive in.

The next step is to use Splinter to automate a browser. This will be fun, because we'll be able to watch a browser work without clicking anywhere or entering anything.

After we help Robin get Splinter set up, we'll scrape data by using Beautiful Soup. This is where our practice with HTML tags will come into play. To scrape the data that we want, we'll tell Beautiful Soup which HTML tag is being used and if it has a specific `class` or `id` attribute.

So far, you've scraped webpages by pasting the HTML code into a Jupyter notebook. In this lesson, you'll do an automated scrape on a real website. To do so, you'll bring together all the skills that you've learned so far. First, you'll create a Python script to capture the HTML content of a live website. Then, you'll use DevTools to identify HTML elements of interest, and you'll use their CSS selectors to extract their data. Finally, you'll use a `for` loop to cycle through the elements, extracting the data that you need.

**IMPORTANT**

Consider the ethics around web scraping. In particular, the "Terms of Service" and "Terms of Use" sections on many websites bring up an ethical issue about scraping data. Specifically, many websites don't allow automated browsing and scraping. That's because some scraping scripts are designed to quickly extract data. And, the constant traffic can overload a web server, disabling the website. Furthermore, an administrator might then ban the IP address of the person doing the scraping. This will make it difficult to even manually visit the site to review its data.

## Set Up Your Code to Use Your Tools

In this section, you'll set up your web scraping code to be able to use the tools that you'll need.

To begin, complete the following steps:

1. Create a new folder, and then use the terminal to navigate to it.
2. Activate Jupyter Notebook.
3. Create a new `Practice.ipynb` file.

**NOTE**

Later, you can either delete this file or save it as a reference.

Next, in the first cell of `Practice.ipynb`, enter the following code:

```
from splinter import Browser
from bs4 import BeautifulSoup as soup
from webdriver_manager.chrome import ChromeDriverManager
```

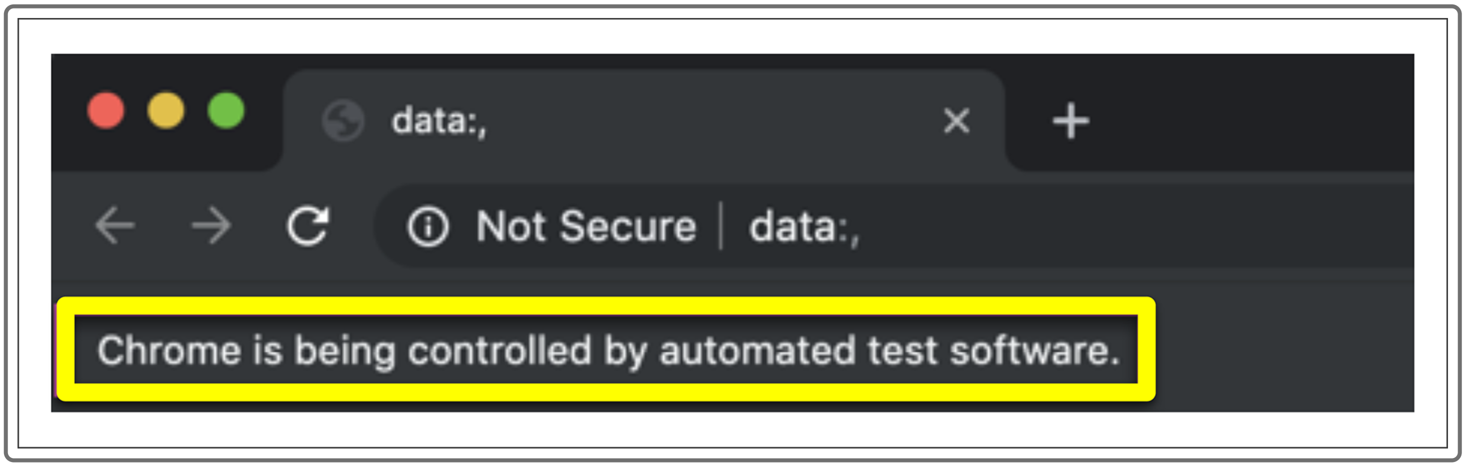
The preceding code imports your scraping tools: the Browser instance from Splinter, the Beautiful Soup object, and the driver object for Chrome (ChromeDriverManager). Notice that the code uses the `soup` alias so that later referencing the Beautiful Soup object will be simpler.

In the next cell, set the executable path and initialize a browser by entering the following code:

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

The preceding code creates an instance of Splinter to prepare your automated browser. The `ChromeDriverManager().install()` code indicates that you'll use Chrome as your browser. And, the `executable_path` statement as a whole specifies that you're installing ChromeDriver instead of manually downloading it. The `headless=False` argument means that all the browser's actions will display in a Chrome window so that you can observe them.

Go ahead and run the cells. The cell that instantiates a Splinter browser might take a few seconds to finish. When it's ready, an empty webpage opens with a message that states "Chrome is being controlled by automated test software."



Chrome is being controlled by automated test software.

Splinter is now controlling the browser. It's fun to observe Splinter navigate through webpages— without you needing to interact with anything. It's also a fantastic way to make sure that your code works as you expect it to. Although you can close the browser window at any time, it's generally best to leave it open until the session ends. Otherwise, your code might fail or generate an error.

NOTE

Splinter offers many ways to interact with webpages. For example, It can enter terms into a Google search bar and then click the Search button. It can even log in to your email account by entering a username-and-password combination that you provide.

Now that you're set up to perform an automated web scrape, you'll next practice performing one on a practice website.

## Practice Performing an Automated Web Scrape

Robin feels more confident about navigating the various HTML tags that build webpages. And, she knows that the data she wants to scrape will be nested within HTML tags. But, An HTML webpage can quickly become confusing. So, Robin wants to practice on a less-sophisticated site before working on the Mars project.

Several sites have been created specifically for new web scrapers to practice and hone their skills with Splinter and Beautiful Soup. These practice sites contain several components that we'll encounter when we work on our own projects. The components include buttons, search bars, and nested HTML tags. They offer a terrific introduction to how our tools work together to gather the data that we want. We'll begin by scraping a practice website that contains quotes from famous people. By doing so, we'll consolidate several skills that we've covered so far. These include using DevTools to inspect a website and using Beautiful Soup to parse HTML code.

**IMPORTANT**

Real websites are fantastic resources for new data. But, a site layout might get updated at any time. For example, an image might suddenly become embedded in an inaccessible block of code, because the developers decided to use a new library to make the website. When this happens, your scraping code has a good chance of breaking. In this case, you need to review and update your code to use it again. It's common to revise code with workarounds or even to search for a different site that's scraping friendly.

## Set Up Jupyter Notebook

Your `Practice.ipynb` notebook should already contain the following code:

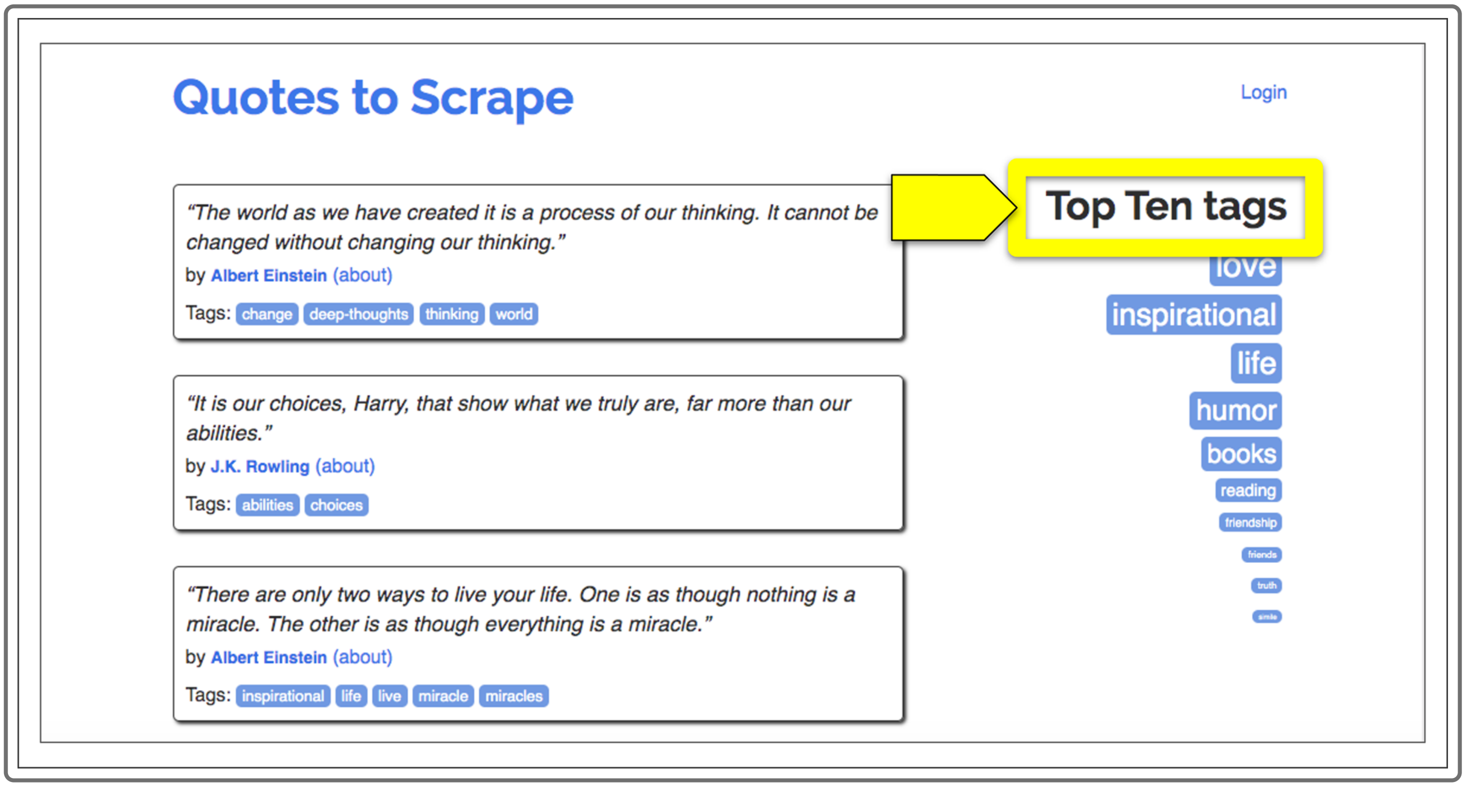
```
from splinter import Browser
from bs4 import BeautifulSoup as soup
from webdriver_manager.chrome import ChromeDriverManager

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

!assessment: zj5uP1Hk1nAy45Jy4HaHQ9N1

## Find Elements via a Shortcut

Next, we'll scrape data from a website that was created specifically for practicing our skills: Quotes to Scrape (http://quotes.toscrape.com/) . In Chrome, open the website, and then familiarize yourself with the page layout.

Interacting with webpages is Splinter's specialty. And, this page has lots to interact with, like the Login button and the Top Ten tags. In this section, we'll learn how to find the text in the Top Ten tags.


Quotes to Scrape

Login

"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."
by Albert Einstein (about)
Tags: change deep-thoughts thinking world

"It is our choices, Harry, that show what we truly are, far more than our abilities."
by J.K. Rowling (about)
Tags: abilities choices

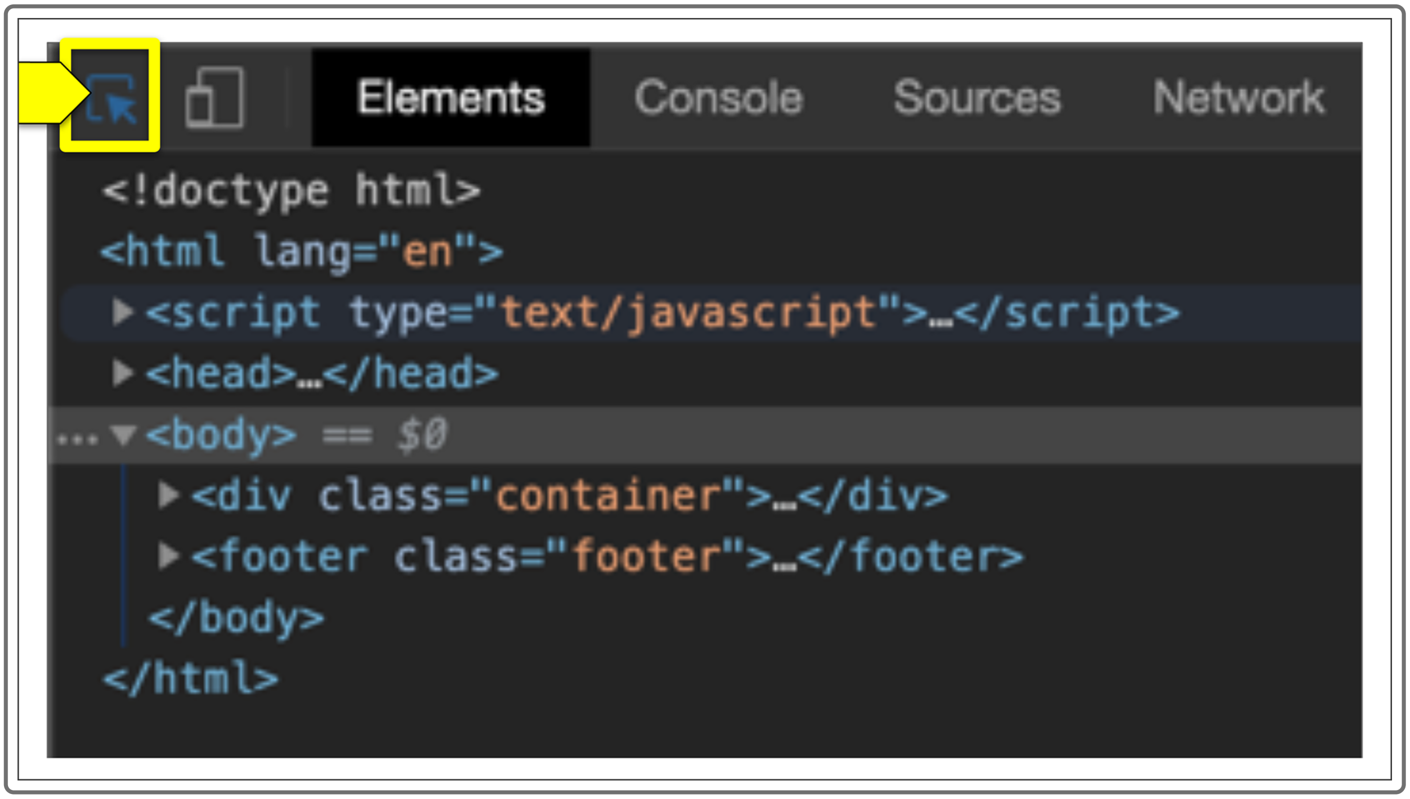
"There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."
by Albert Einstein (about)
Tags: inspirational life live miracle miracles

Top Ten tags
love inspirational life humor books reading friendship friends truth simile

Before doing the coding, let's use DevTools to review the details of the "Top Ten tags" line. Right-click anywhere on the webpage, and then click Inspect. Note that in the DevTools panel, we can use a shortcut to choose an element on the page instead of searching through the tags.

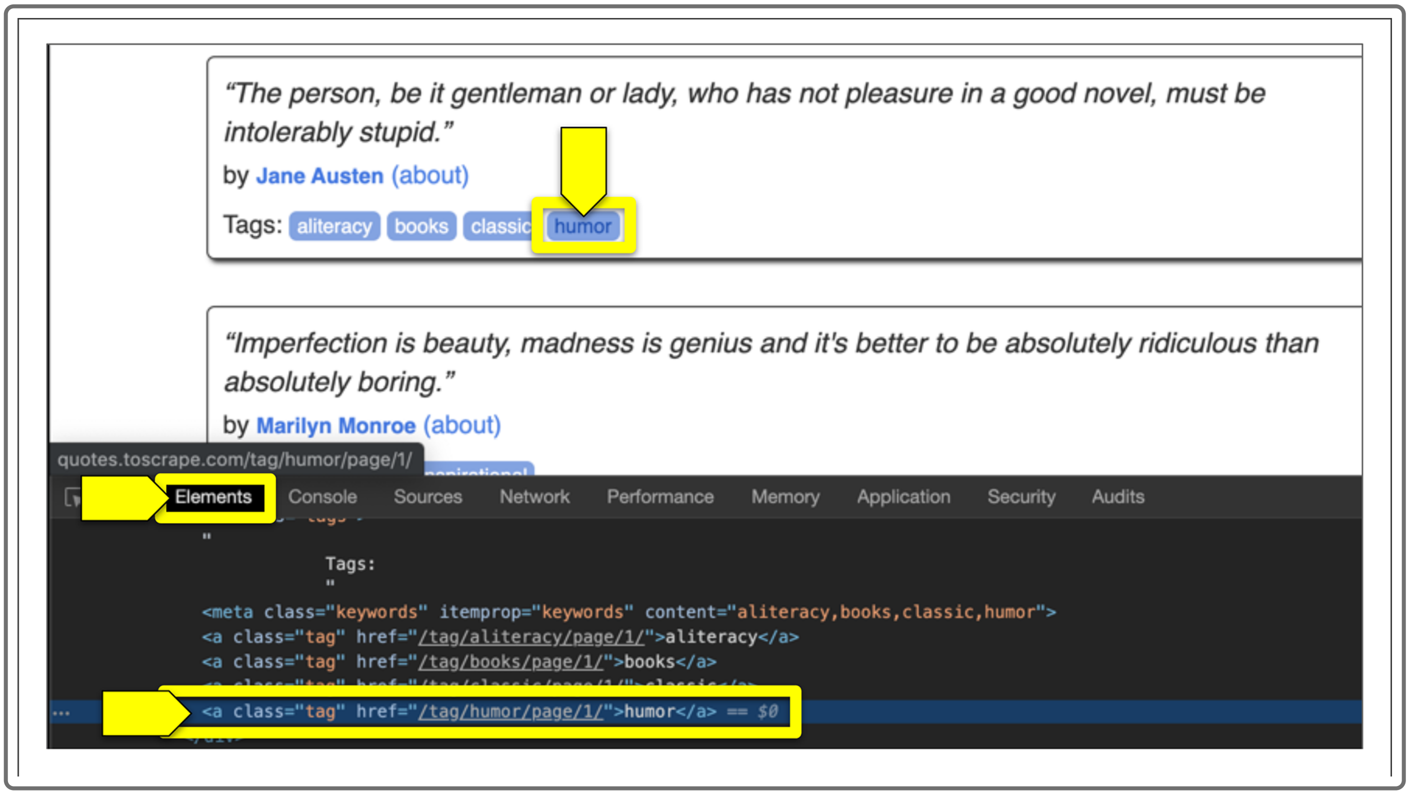
To do so, complete the following steps:

1. Click the "Select an element in the page to inspect it" button (which has an arrow icon and appears on the left side of the panel menu).



```
<!doctype html>
<html lang="en">
 ▶<script type="text/javascript">…</script>
 ▶<head>…</head>
…▼<body> == $0
   ▶<div class="container">…</div>
   ▶<footer class="footer">…</footer>
  </body>
</html>
```

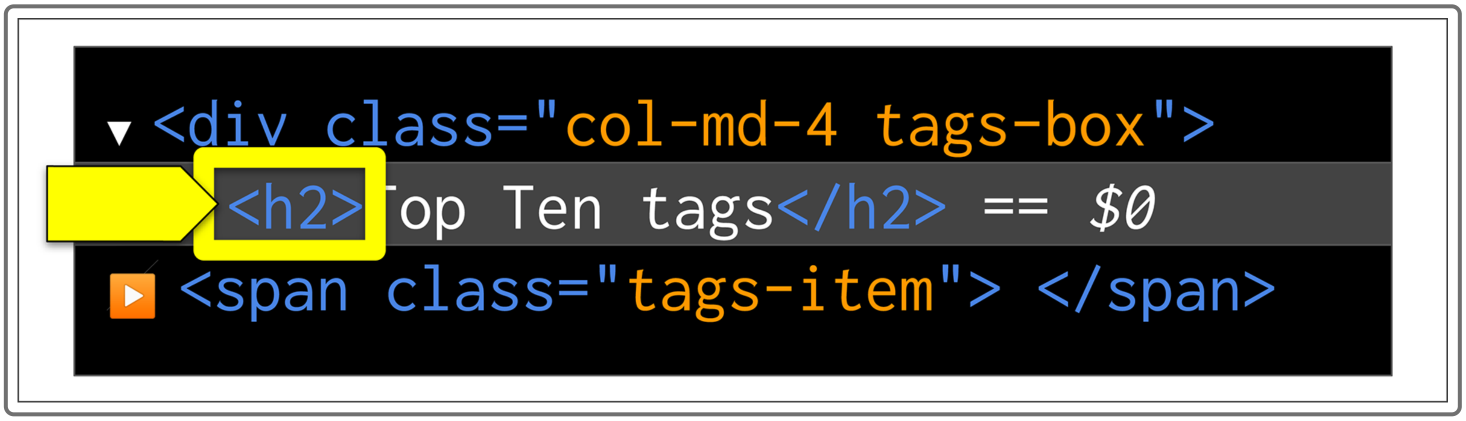
2. On the webpage, click the element that you want, like the humor tag. DevTools goes to the line of code where the humor tag is nested.


"The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."
by Jane Austen (about)
Tags: aliteracy books classic humor
"Imperfection is beauty, madness is genius and it's better to be absolutely ridiculous than absolutely boring."
by Marilyn Monroe (about)
quotes.toscrape.com/tag/humor/page/1/
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits

```
            "
                Tags:
                "
            <meta class="keywords" itemprop="keywords" content="aliteracy,books,classic,humor">
            <a class="tag" href="/tag/aliteracy/page/1/">aliteracy</a>
            <a class="tag" href="/tag/books/page/1/">books</a>
            <a class="tag" href="/tag/classic/page/1/">classic</a>
…           <a class="tag" href="/tag/humor/page/1/">humor</a> == $0
```

That was quick! But be aware that sometimes, we still have to drill down through the tags to find the ones that we want. In any case, the ability to select elements directly on the webpage immensely reduces the time that we spend to find our elements.

In the DevTools panel, notice that our text is nested within the `<h2>Top Ten tags</h2>` tag. Our shortcut has taken us to the `<h2 />` tag holding all the text that we want.



```
▼ <div class="col-md-4 tags-box">
    <h2>Top Ten tags</h2> == $0
  ▶ <span class="tags-item"> </span>
```

This process would work well if only one `h2` element existed on the page that we're scraping. In that case, we could scrape it once and be done. But, what if multiple `h2` elements exist, and we want to scrape some or all of them? In that case, we need to use a different approach.

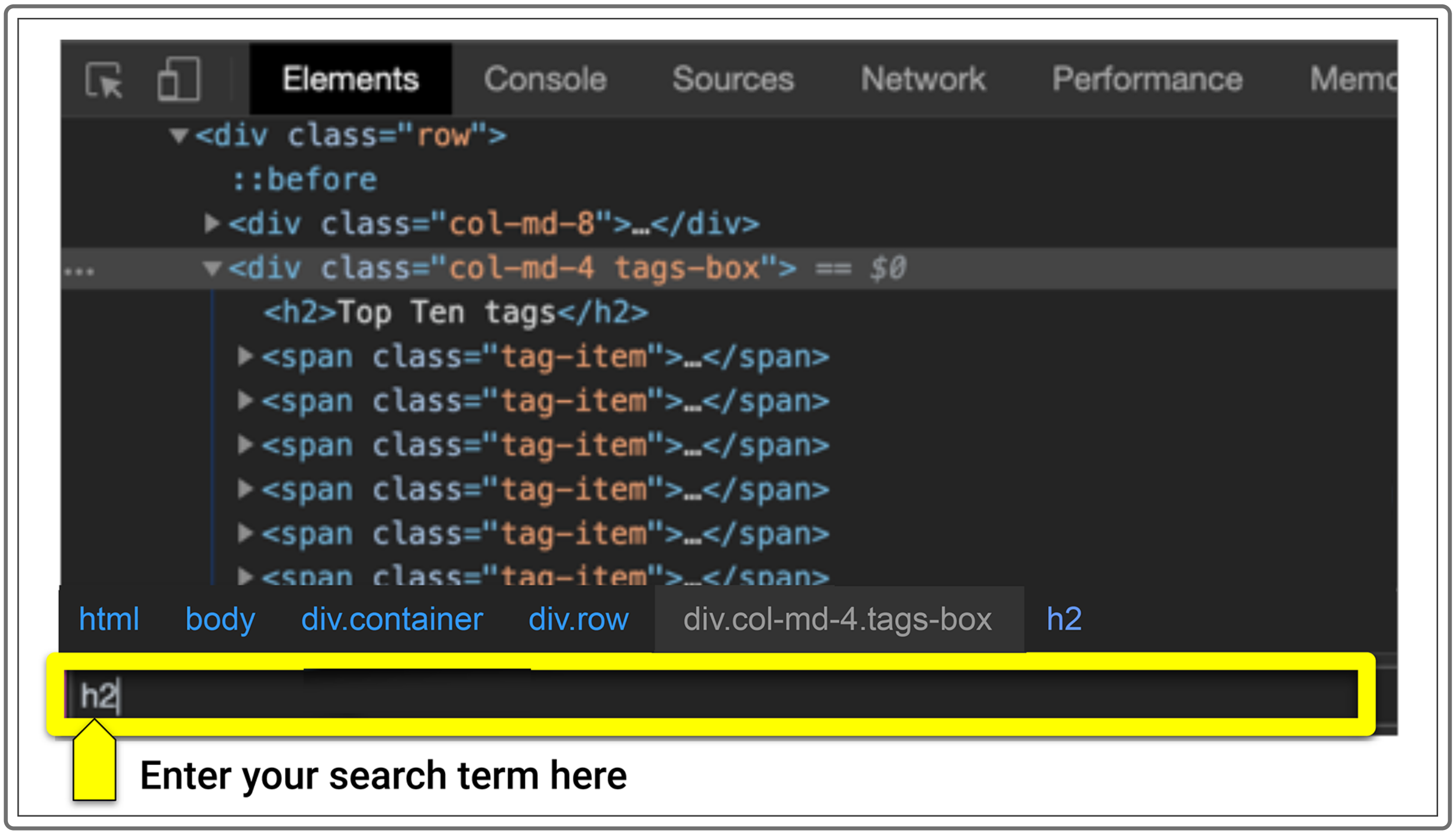
## Search for Elements

We've been able to select a component on the page via a shortcut—specifically, by using the "Select an element in the page to inspect it" button in DevTools. But sometimes, we need to know how many times a particular type of tag exists on the page. For example, say that the title we want to scrape exists in an `<h2 />` tag but that several `<h2 />` tags exist on the page. In that case, we need to tailor our code to target the one that we want. To do so, we can use DevTools to search for elements in the HTML code, as we learned how to do in an earlier lesson.

With DevTools still active, press Ctrl+F (on Windows) or Command+F (on macOS) to display the Find box.

Next, we want to search for all the `<h2 />` tags in the document. Then, we'll know if we need to make our search more specific by adding a `class` or an `id` attribute.

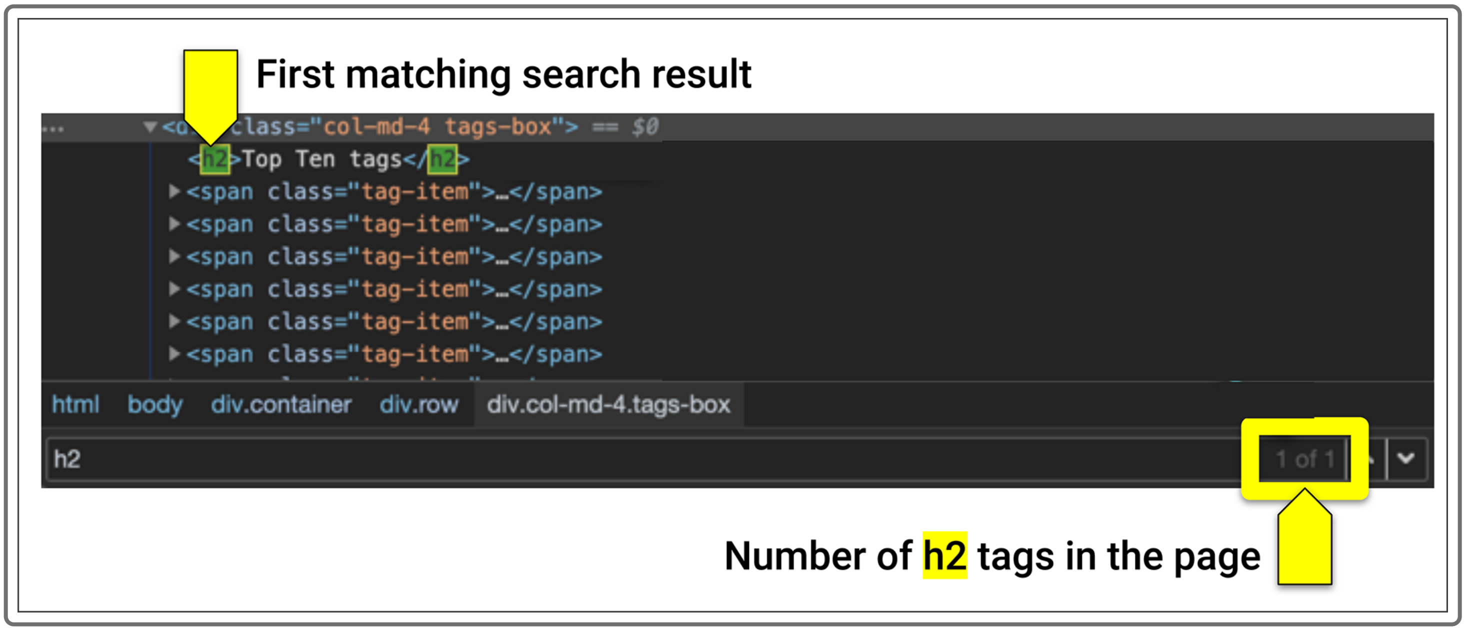
To do so, in the Find box, type "h2" (without the quotation marks), and then press Enter on your keyboard.



```
▼<div class="row">
   ::before
  ▶<div class="col-md-8">…</div>
  ▼<div class="col-md-4 tags-box"> == $0
     <h2>Top Ten tags</h2>
    ▶<span class="tag-item">…</span>
    ▶<span class="tag-item">…</span>
    ▶<span class="tag-item">…</span>
    ▶<span class="tag-item">…</span>
    ▶<span class="tag-item">…</span>
    ▶<span class="tag-item">…</span>
```

```
h2
```

Two things happen. First, the first `h2` tag gets highlighted in the HTML code. Second, at the end of the Find box, the number of `h2` tags in the document displays. Specifically, "1 of n" displays, where n is the number `h2` tags in the document.



```
▼<div class="col-md-4 tags-box"> == $0
   <h2>Top Ten tags</h2>
  ▶<span class="tag-item">…</span>
  ▶<span class="tag-item">…</span>
  ▶<span class="tag-item">…</span>
  ▶<span class="tag-item">…</span>
  ▶<span class="tag-item">…</span>
  ▶<span class="tag-item">…</span>
```

```
h2
```

Notice that in this case, "1 of 1" displays. This means that only one `<h2 />` tag exists in this document. So, we know that we can scrape based on `h2` without including a `class` or `id` attribute. In most cases, we will need to add such an attribute, so we'll practice that later in this lesson.

## Scrape the Title

At last, we can scrape the "Top Ten tags" title. To begin, in the next cell in your Jupyter notebook, enter and run the following code:

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

The preceding code tells Splinter to visit the site that the `url` variable specifies. (The `url` variable contains the address of the site.)

Next, we'll use Beautiful Soup to parse the HTML code. To do so, in the next cell, enter and run the following code:

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

**NOTE**

Remember that we're actively using two Python libraries. We're using Splinter to visit the website and extract its HTML code. We're then using Beautiful Soup to parse the HTML code.

We've now parsed all the HTML code on the page. This means that Beautiful Soup has examined the components on the page and can access them. Specifically, Beautiful Soup has parsed the HTML text and stored it as an object.

In the preceding code, we use `html.parser` to parse the information, but other options also exist.

DEEP DIVE ▼

Next, we want to find the title and extract it. To do so, in the next cell, enter and run the following code:

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

In the preceding code, the first `print` statement displays `<h2>Top Ten tags</h2>`. And, the second print statement displays `Top Ten tags`. To understand how we got those, let's go over the first two lines of code:

1. On the `h2 = html_soup.find('h2')` line, we use the `html_soup` object that we created earlier and call its `find` method to search for the `<h2 />` tag. Printing the result thus produces `<h2>Top Ten tags</h2>`.
2. On the `title = h2.text` line, we extract just the text from the `<h2 />` tag by adding `.text` to the end of the code. This extracts the `text` attribute, so printing it produces only the title text of `Top Ten tags`.

**NOTE**

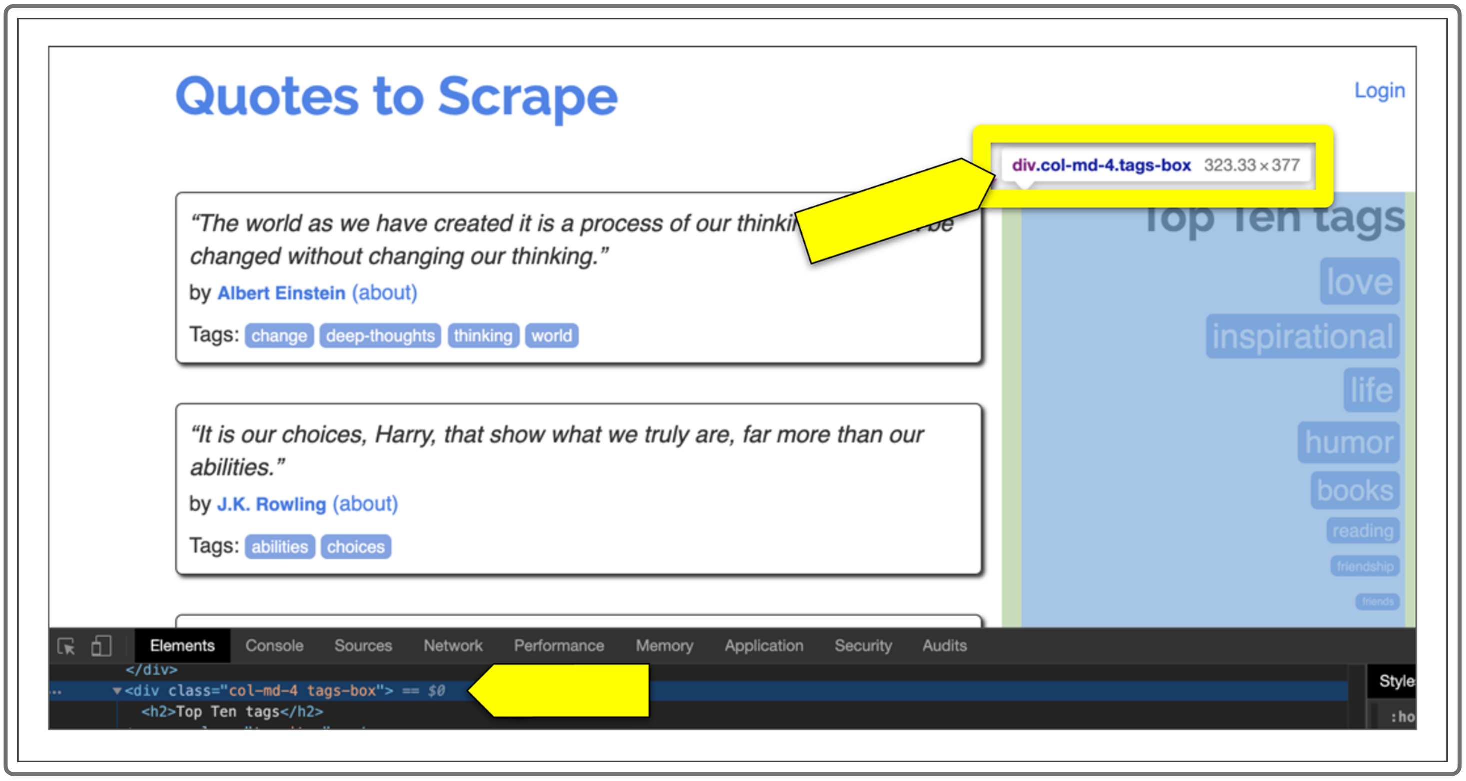
We could also directly access the title text by using `title = html_soup.find('h2').text`.

Congratulations! You've completed your first real scrape.

Next, we'll practice scraping again. But this time, we'll use Splinter to scrape all the tags that are associated with the title we just extracted.

## Scrape All the Tags

Now, we want to extract all 10 of the Top Ten tags from the webpage. To do so, use the "Select an element in the page to inspect it" button to highlight the `<div />` container that holds all the tags.



"The world as we have created it is a process of our thinki be changed without changing our thinking."
by Albert Einstein (about)
Tags: change deep-thoughts thinking world

"It is our choices, Harry, that show what we truly are, far more than our abilities."
by J.K. Rowling (about)
Tags: abilities choices

```
  </div>
▾<div class="col-md-4 tags-box"> == $0
    <h2>Top Ten tags</h2>
```

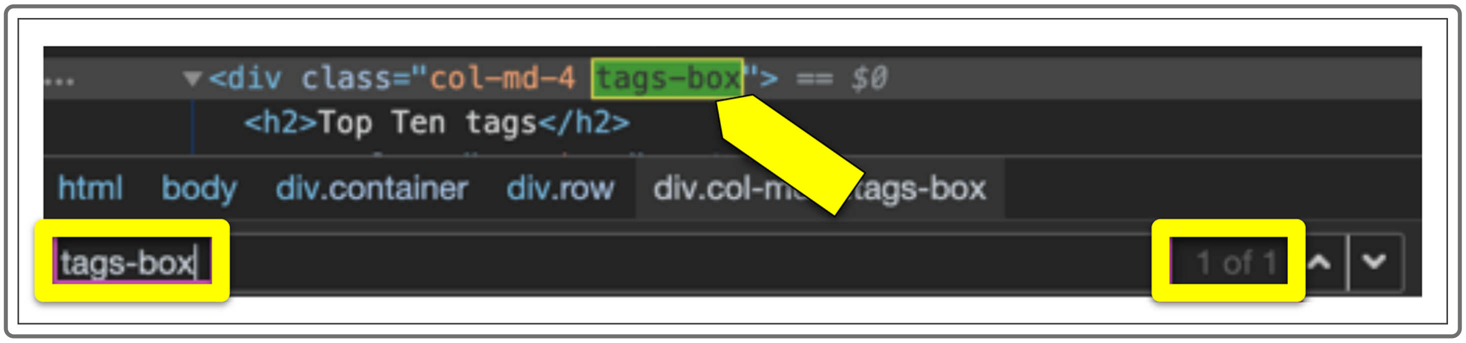
Notice that the opening tag for this division is `<div class="col-md-4 tags-box">`. This means that this division has two classes: `col-md-4` and `tags-box`.

**NOTE**

An HTML element can belong to multiple classes.

The `col-md-4` class is a Bootstrap feature. **Bootstrap** is an HTML and CSS library that simplifies building websites. It uses a grid system to divide a page into 12 columns of equal width. In this case, `col-md-4` means that the box containing "Top Ten tags" takes up four columns. The main quotes section takes up the remaining eight columns. Websites that use Bootstrap commonly use this class, but many others exist.

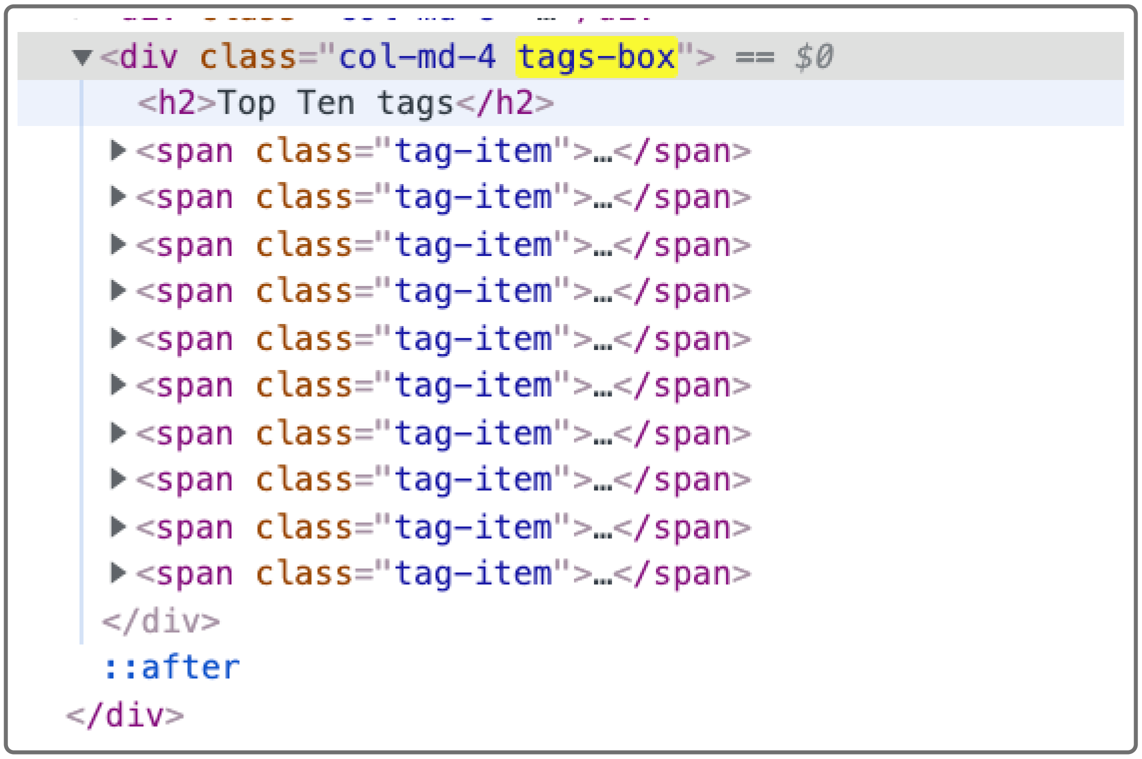
The other class, `tags-box`, seems specific to this website, but we want to confirm that. To do so, use the DevTools Find box to search for it.



```
<div class="col-md-4 tags-box"> == $0
  <h2>Top Ten tags</h2>
```

Notice that searching for “tags-box” returns only one result: our `<div class="col-md-4 tags-box">` tag. This means that `tags-box` is unique in the HTML code, so we can use it to find specific data.

Next, we want to examine the content of this `div` element. To do so, in DevTools, expand the `<div class="col-md-4 tags-box">` line.



```
▼<div class="col-md-4 tags-box"> == $0
    <h2>Top Ten tags</h2>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
   ▶<span class="tag-item">…</span>
  </div>
  ::after
</div>
```

Notice that we get the `<h2>Top Ten tags</h2>` line followed by a list of `<span />` tags, each with a class of `tag-item`. Expand some of the `span` elements to review their contents. If you observe `<a />` tags that contain the names of the Top Ten tags, you're in the right place.

Because the list that displays on the webpage contains 10 items, let's use the DevTools search function to verify the list item count. This is one more way to check that we'll scrape the correct data. To do so, search for “tag-item” (without the quotation marks), and then note the number of returned results. This number should indeed be 11.

Splendid! We can now scrape all the tags by using a `for` loop. To do so, in the next cell of your notebook, enter and run the following code:

```
# 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)
```

Let's go over the preceding code:

1. The first line, `tag_box = html_soup.find('div', class_='tags-box')`, assigns the results of a search to a new variable named `tag_box`. The search is for `<div />` tags that have a class of `tags-box`.
    - This search occurs in the HTML code that we parsed and stored in the `html_soup` variable earlier.
    - The `class_` argument contains an underscore (`_`). That's because the word `class` is reserved for other uses in Python..

2. The second line, `tags = tag_box.find_all('a', class_='tag')`, drills down further into the data in `tag_box`.
    - This line uses the `find_all` method to search within `tag_box`. This time, we search through the parsed results that are stored in our `tag_box` variable to find all the anchor (`a`) elements that belong to the `tag` class.
    - We use `find_all` because we want to capture all the results rather than a single or specific one.

NOTE

The Beautiful Soup `find` method returns the first search result. The `find_all` method returns all the results.

3. We added a `for` loop. This loop cycles through the tags in the `tags` variable, extracts the HTML code from each, and then prints only the text of each tag.

## End the Automated Browsing Session

We can now end the automated browsing session. Remember that until we do so, the browser continues to listen for instructions and use the computer's resources. Leaving an automated browser active when you don't need it might unnecessarily strain its memory or battery.

To end the automated browsing session, in the next cell in your Jupyter notebook, enter and run the following code:

```
browser.quit()
```

In this lesson, you used an automated browser to scrape a real website. You used DevTools to search for elements, and you then used Beautiful Soup to extract data from those elements. In the next lesson, you'll further your skills by automating browser actions, like clicking buttons.