Web-Scraping

Sean Sall

June 8th, 2016

Objectives

- Understand the basics of the client-server relationship in a web framework
- Explain the basic concepts of HTML and CSS
- Learn how to use Python to pull elements from a web-page
- Learn how to use Python to pull elements from an API (afternoon assignment)

Agenda

- World-Wide-Web v. Internet
- Client-Server relationship
- HTML and CSS Basics
- Manual Web-Scraping
  - `requests` library
  - `bs4` library (and `BeautifulSoup`)
- Web-scraping through API's (assignment)

Why does this matter?

- Any time that you want data from the web and there isn't a clickable link to a `.csv`, `.zip`, etc., you'll have to pull it either manually or from an API using Python
- There's a lot of data up for grabs on the web, and after today it'll all be at the tips of your fingers (subject to legal issues)

Intro to the Web

World-Wide-Web v. Internet I

- The world-wide-web is **technically** different from the internet.
- The **world-wide-web** (`www`) is basically a system of Internet servers where information (documents) is hosted
- The **internet** is what connects them all

World-Wide-Web v. Internet: A Visual

- Another way of looking at this is that the **world-wide-web** is a collection of islands existing all over the globe, and the internet is a collection of bridges connecting those islands

World-Wide-Web URL's I

- The web uses **U**niform **R**esource **L**ocaters (URL's) to specify the location of a document within the internet
- Each URL has a couple of different parts:

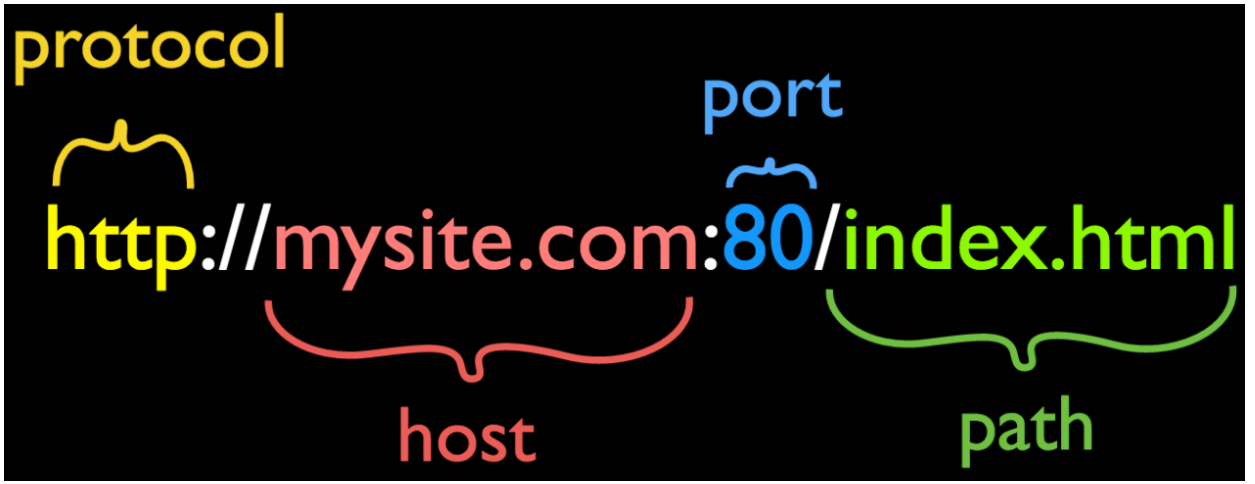


Figure 2:Parts of a URL

World-Wide-Web URL's II

- **Protocol** - specifies how the web server should interpret the information you are sending it
  - In the browser, we can often leave this part out and it will automatically get filled in... but this is typically **not** the case when web-scraping
- **Host** - points to the domain name of the web server you want to communicate with, and is associated with a specific IP address
- **Port** - holds additional information used to connect with the host
  - if we think of the host holding the street address we want to go to, then the port holds the apartment number
- **Path** - indicates where on the server the file you're requesting lives

Server-Client relationships

- At any point in time, any person or computer connected to the internet can be either a server or client (definitions follow)
- The **client** is the requesting party (e.g. requesting some info., like a web page)
- The **server** is the party providing that information (e.g. a web page), and responding to requests from a client

Server-Client relationships I

- If we visit `www.example.com` in our browser, then we are the client and `www.example.com` is the server
- This interaction starts with the client (us) issuing a `get` request to the server,indicating that it would like some specific piece of information

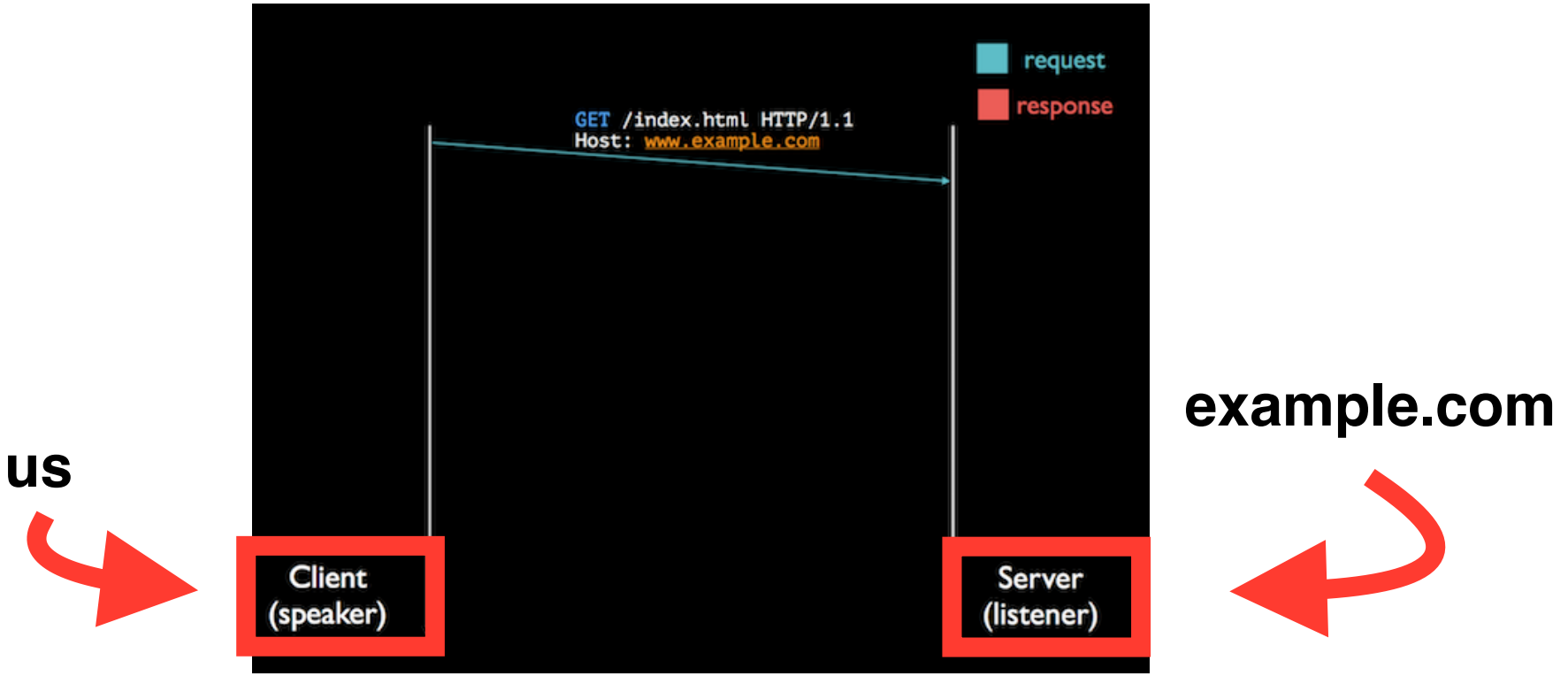


Figure 3:Get Request

Server-Client relationships II

- Once the server gets this `request`, it will send back a `response` with the information requested in the `body` (and a header with a status code in it)

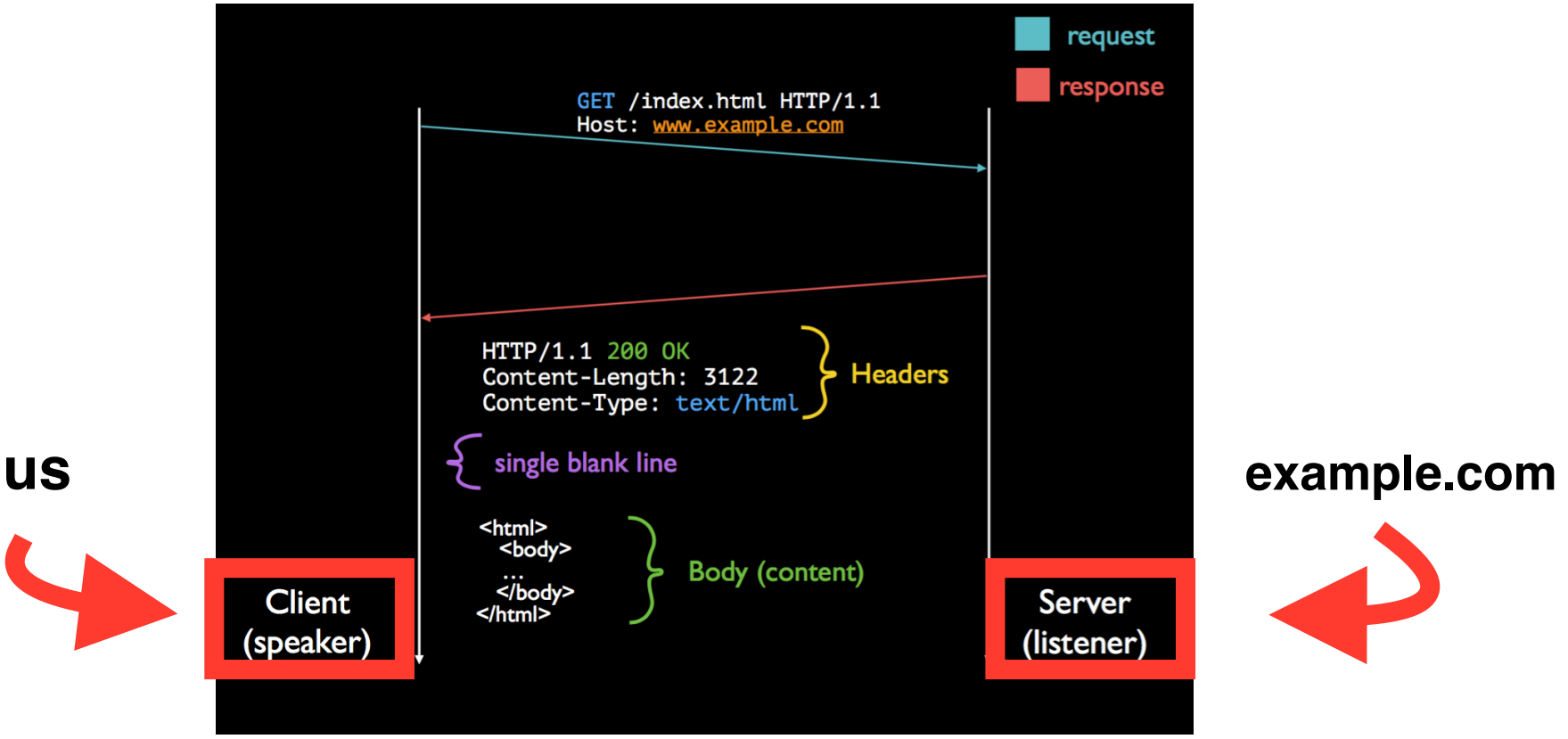


Figure 4:Response

Server-Client relationships III

- Future requests will happen in a similar way (below is an example of an **A**synchronous **J**avaScript **A**nd **X**ML (AJAX) request)

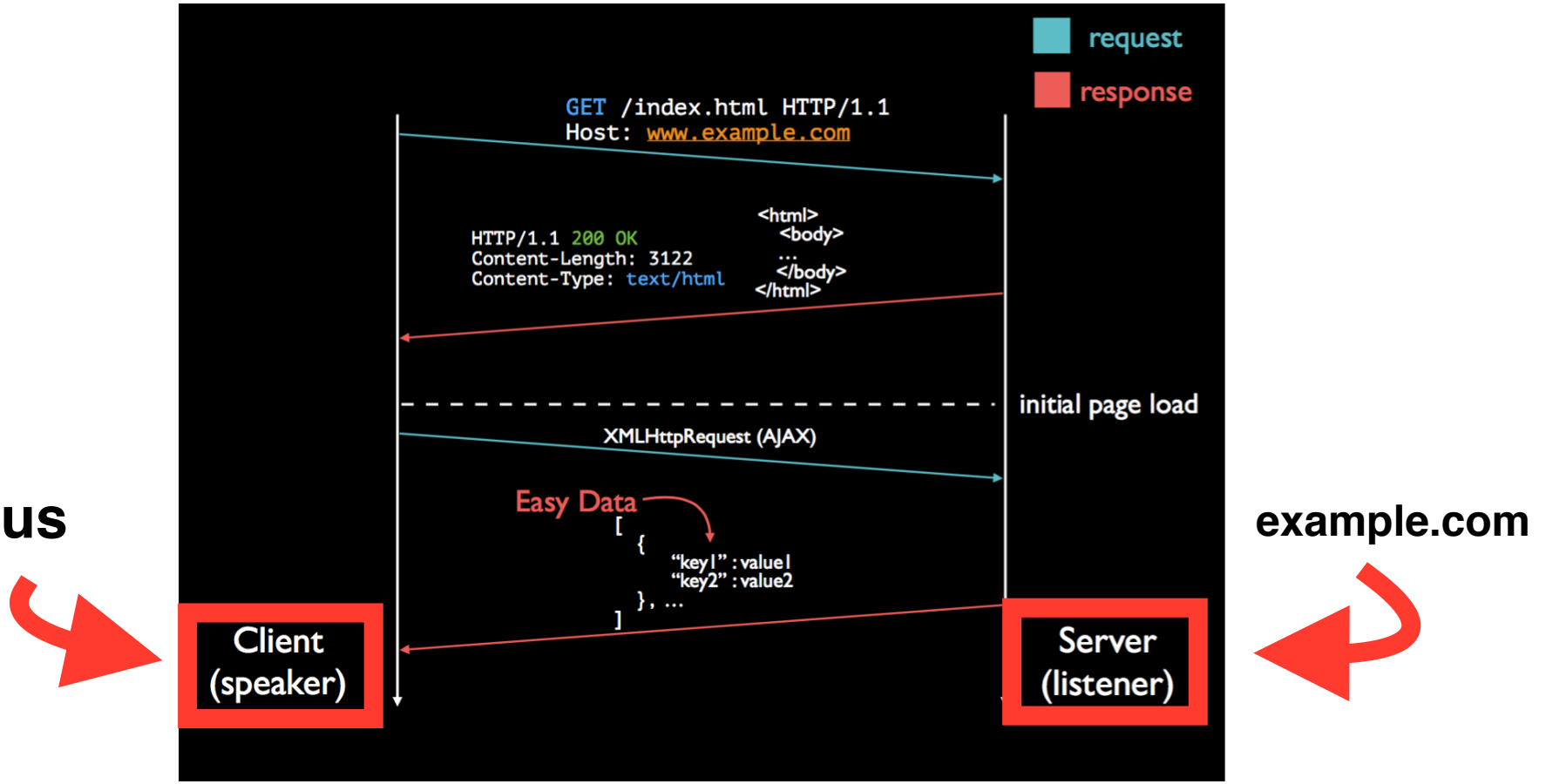


Figure 5:Ajax Request

Intro to HTML and CSS

HyperText Markup Language (HTML)

- The majority of web pages are written in HyperText Markup Language (HTML)
- HTML is written using **HTML tags**, where each tag describes different document content

Popular HTML tags

- Popular tags include

Tag	Description
`<h1> - <h6>`	Heading
`<p>`	Paragraph
`<i>`	Italic
`<b>`	Bold
`<a>`	Anchor (links)
`<ul> & <li>`	Unordered List & List Item
`<blockquote>`	Blockquote
`<img>`	Image
`<div>`	Division

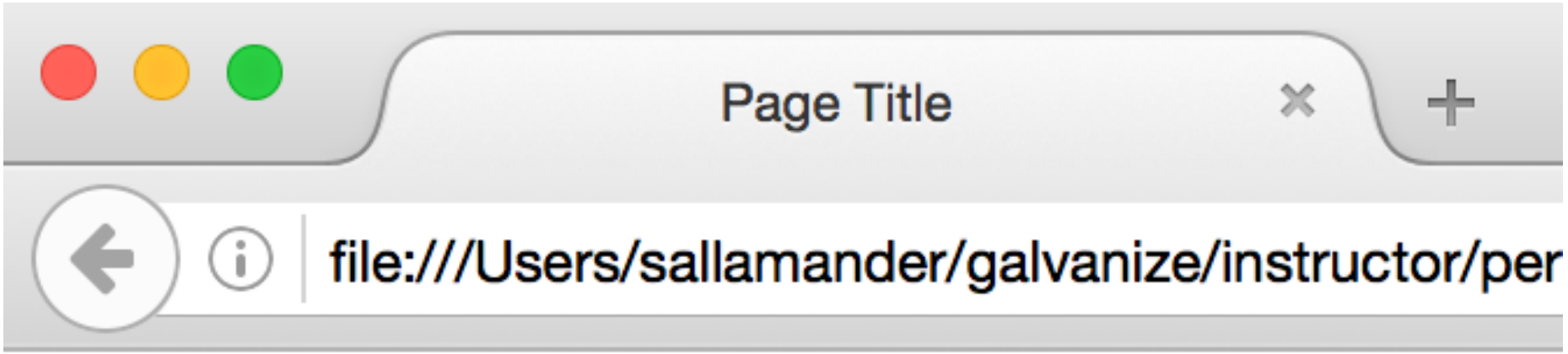
Example HTML Page

- An example HTML Page might look like the following:

```
<!DOCTYPE html>
<title>Page Title</title>

<h1>This is a heading</h1>
<p>This is a paragraph.</p>
```

Example HTML Page Visual

- If we looked at how that page rendered in the browser, we'd see this:



This is a heading

This is a paragraph.

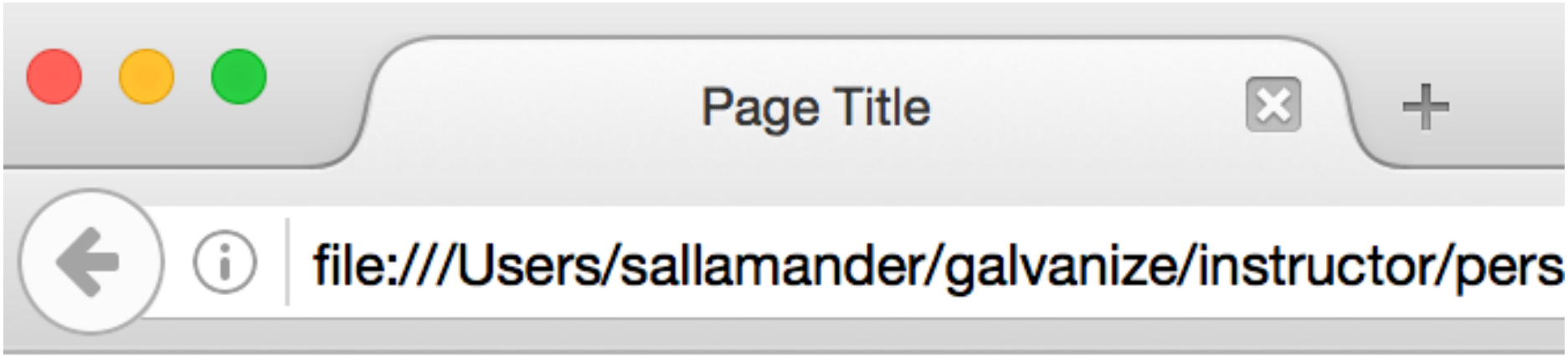
Style Attributes

- To actually get our web page to look nice, we can add some style to it. One way to do this is to simply place a style **attribute** within one of the **html tags**:

```
<!DOCTYPE html>
<title>Page Title</title>

<h1 style="color:red;">This is a heading</h1>
<p style="color:blue;">This is a paragraph.</p>
```

Style Visual

- With that change, our web page would now look like the following:



This is a heading

This is a paragraph.

Cascading Style Sheets Best Practice

- If we had a whole website, where we wanted certain text to be red, or blue, then typing the style attributes in like that would be a pain (it's also not best practice to do this)
- Common best practice is to use **css selectors**, which are associated with specific tags and allow you to put style attributes on those tags in a separate file, called a **cascading style sheet**
  - We'd use a **class** or **id** in the raw HTML, and then style the attributes associated with that **class** or **id**

Cascading Style Sheets example

HTML file

```
<!DOCTYPE html>
<link rel="stylesheet" type="text/css" href="mystyle.css">
<title>Page Title</title>

<h1 class="red_text">This is a red heading using a class.</h1>
<p id="blue_text">This is a blue paragraph using an ID.</p>
```

CSS file

```
.red_text {
    color: red
}

#blue_text {
    color: blue
}
```

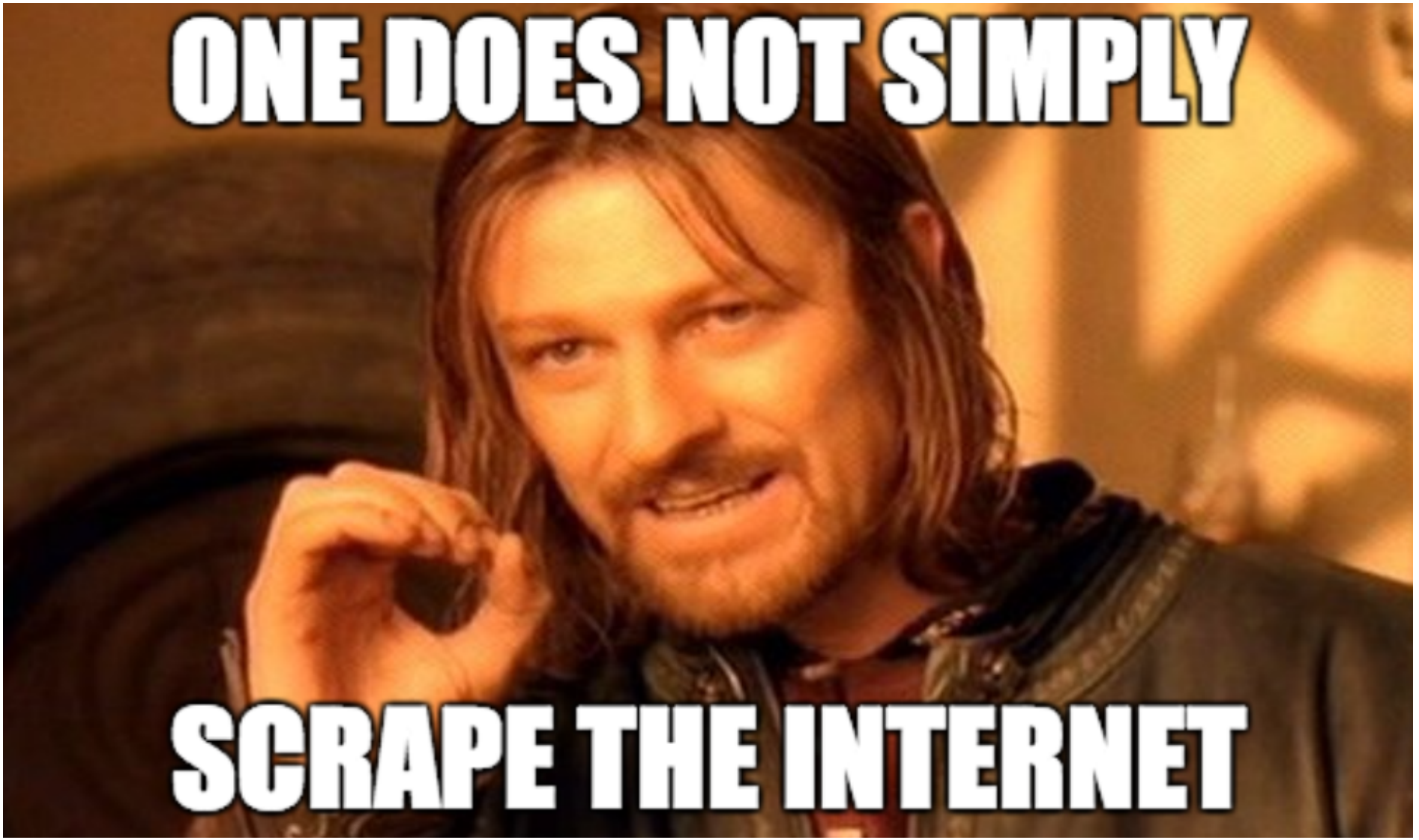
Cascading Style Sheets Visual

- Our web page would then look like the following:

Figure 8:CSS Stylesheets

HTML and CSS in the Wild, and Web-Scraping

Scraping is Hard

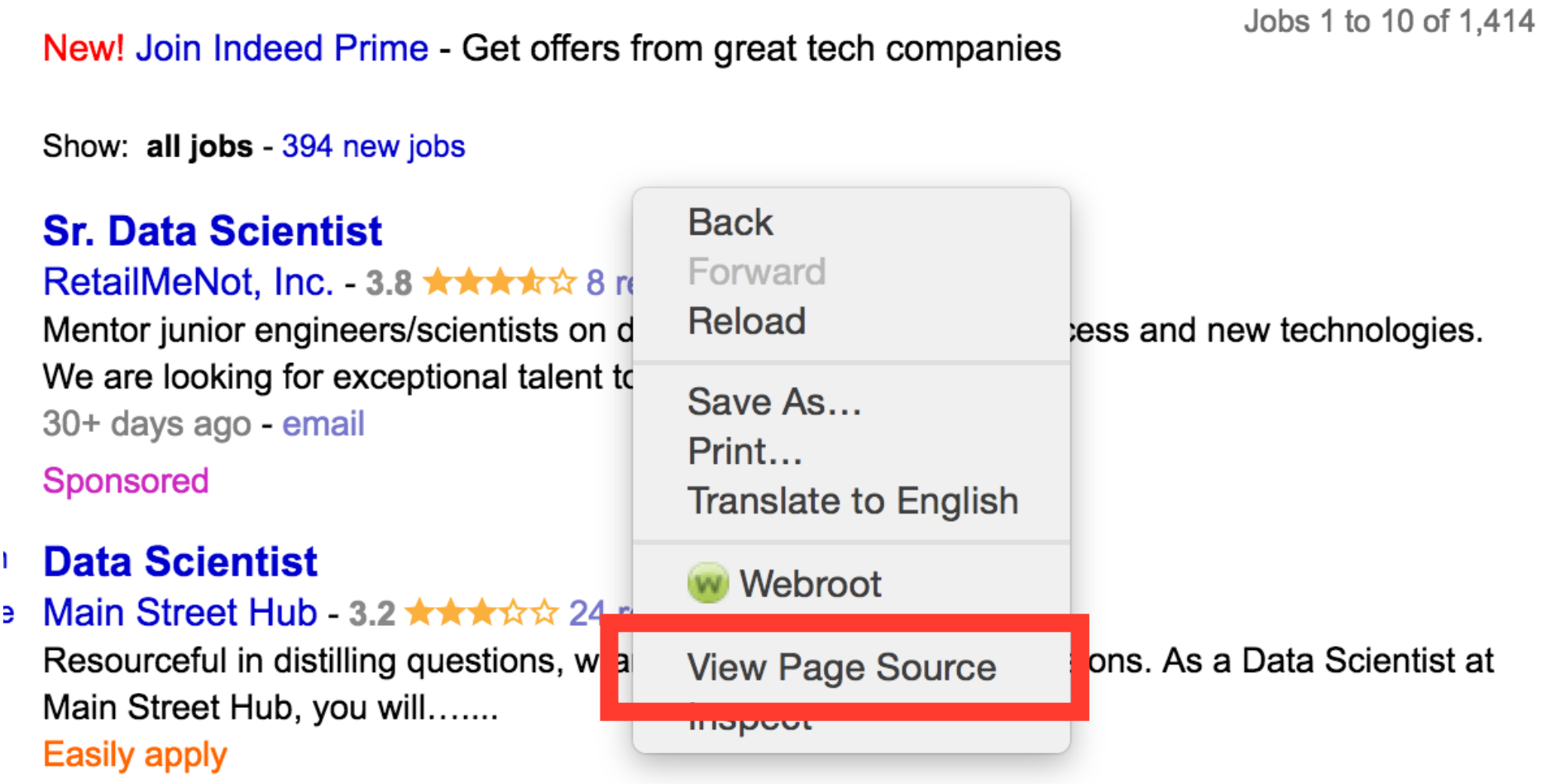


Web-Scraping at a High Level

- In practice, when we are scraping we'll issue a `get` request for an entire page's HTML, and then pull specific elements by either their **html tags** or **css selectors**

Using Page Source

- To see all of the HTML on a wage page, right click and use the `View Page Source`:



Resulting HTML

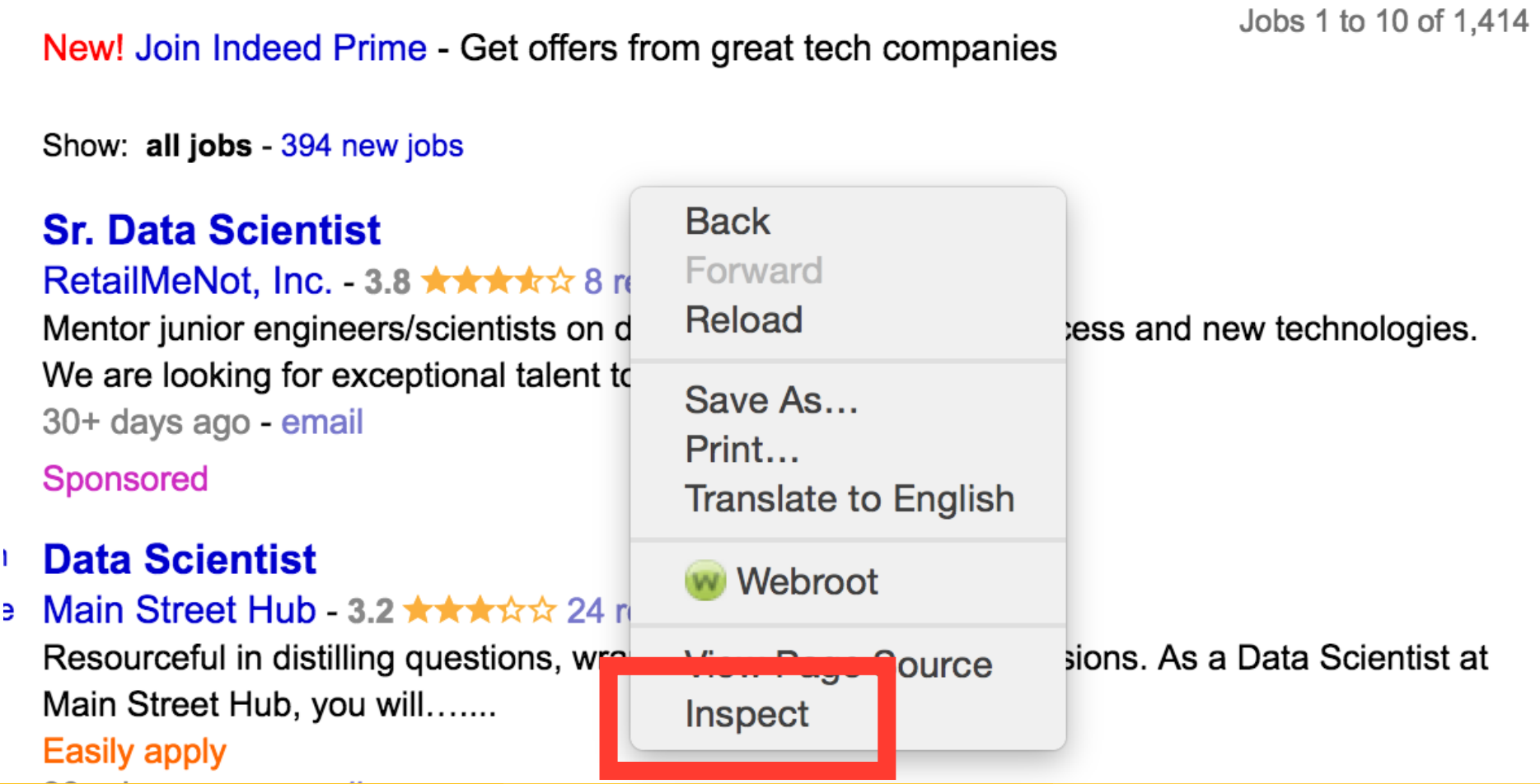
- The Page Source would show us the following:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=UTF-8">
<!-- pll --><script type="text/javascript" src="/s/6adbd7b/en_US.js"></script>
<link href="/s/a6a334e/jobsearch_all.css" rel="stylesheet" type="text/css">
<link rel="alternate" type="application/rss+xml" title="Data Science Jobs, Employment in Austin, TX" href="http://rss.indeed.com/rss?q=data+science&l=Austin%2C+TX">
<link rel="alternate" media="handheld" href="/m/jobs?q=data+science&l=Austin%2C+TX">
<script type="text/javascript">

    window['closureReadyCallbacks'] = [];

    function call_when_jsall_loaded(cb) {
        if (window['closureReady']) {
            cb();
        } else {
            window['closureReadyCallbacks'].push(cb);
        }
    }
</script>

<script type="text/javascript" src="/s/eface3a/jobsearch-all-compiled.js"></script>
            <script type="text/javascript">
var pingUrlsForGA = [];

```

Figure 11:Actual Page Source

Using Inspect or Inspect Element

- To view the HTML of a single element, right click and use the `Inspect` or `Inspect Element` (Chrome or Firefox dependent):



Resulting HTML from `Inspect` or `Inspect Element`

- The `Inspect` (or `Inspect Element`) would then show us the HTML corresponding to the individual element you chose to look at:

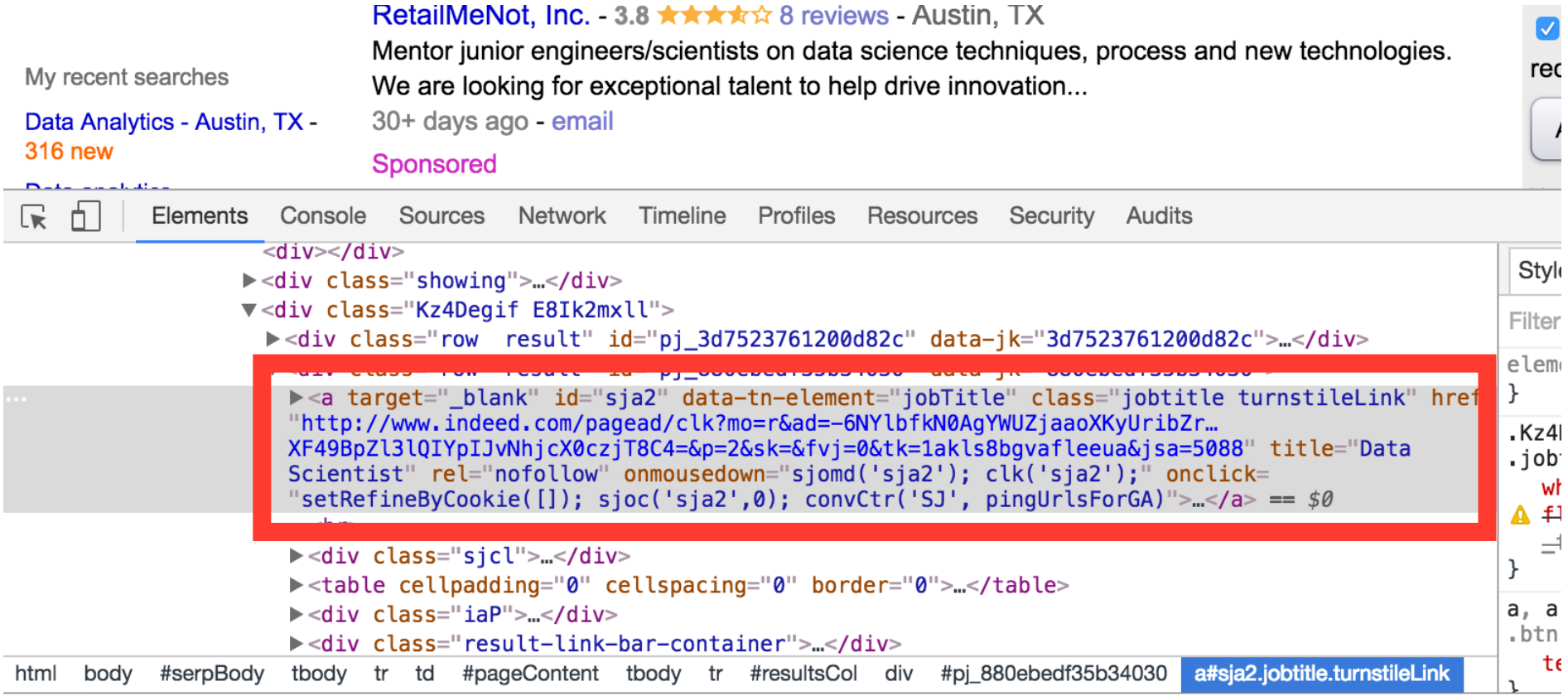


Figure 13:Individual Element

Grabbing individual elements

- We'll then examine the HTML corresponding to the individual element we want to scrape for, and then use either it's **html tag** or **css selectors** to parse it out in Python
- That leaves us with the following workflow:
  1. Find the page that you want to scrape or grab information from
  2. Find the element(s) that you want to grab
  3. Use inspect element to figure out what **html tag** or **css selectors** to use to grab the element(s)
  4. Use Python to actually grab the element(s)

Using Python and BeautifulSoup I

- To use Python, we'll first start by issuing a `get` request on the URL (using the `requests` library), and then use `BeautifulSoup` to parse the HTML

```
import requests
from bs4 import BeautifulSoup

req = requests.get('www.example.com')
html = BeautifulSoup(req.content, 'html.parser')
```

Using Python and BeautifulSoup II

- Next, we use methods on what's returned from `BeautifulSoup` to actually grab content from the page:

```
html.find('a') # Returns the **first** 'a' tag.
html.find_all('a') # Returns **all** 'a' tags.

html.find('a').text # Returns the text of the
                    # **first** 'a' tag.
```

Note: There are many ways of finding the same elements on the page.

Major selecting methods

- `.select()` - This method allows us to select tags using CSS selectors
- `.find_all()` - This method allows us to select all tags matching certain parameters and returns them as a list. For example, `soup.find_all('div')` would return a list of all div tags in the original soup object while `soup.find_all('div', attrs={'class': 'content'})` would return only the div tags that also had `class=content`.
- `.find()` - This method is the exact same as calling `soup.find_all(limit=1)`. Rather than returning a list, it only returns the first match that it finds.