Lecture 11: Data Acquisition (Part 2) - Web Scraping Basics

Data Science, DST, UIC

Many data science projects start with the first step of obtaining an appropriate dataset. In some cases (the *ideal situation*), a dataset is readily provided by a business partner, company's data warehouse, or can be bought or obtained in a structured format by external data providers; but many truly interesting projects start from collecting a treasure trove of information from the same place as humans do: **the web**.

1. Introduction

Web scraping (also called *web harvesting*, *web data extraction*, or even *web data mining*), can be defined as **the construction of an agent to download, parse, and organize data from the web in an automated manner.**

When surfing the web using a normal web browser, you've probably encountered multiple sites where you considered the possibility of gathering, storing, and analyzing the data presented on the site's pages. Especially for data scientists, whose *raw material* is data, the web exposes a lot of interesting opportunities:

- There might be an interesting table on a webpage you want to retrieve to perform some statistical analysis.
- You might be wondering about doing social network analytics using profile data found on a web forum.
- It might be interesting to monitor a news site for trending new stories on a particular topic of interest.
- Perhaps you want to get a list of reviews from a movie site to perform text mining, create a recommendation engine, or build a predictive model to spot fake reviews.

The web contains lots of interesting data sources that provide a treasure trove for all sorts of interesting things. Although many websites nowadays provide Application Programming Interface (API) that provides a means for the outside world to access their data repository in a structured way, web scraping is still important for various reasons:

- The website you want to extract does not provide an API.
- The API provided is not free.
- The API rate is limited, i.e., you can only access it a number of certain times per second, per day,...
- The API does not expose all the data you wish to obtain (whereas the website does).

  In all of these cases, the usage of web scraping might come in handy. The fact remains that if you can view some data in your web browser, you will be able to access and retrieve it through a program.

2. HTTP Request & Response

We now introduce one of the core building blocks that makes up the web: the **HyperText Transfer Protocol (HTTP)**, after having provided a brief introduction to computer networks in general. We then introduce the Python requests library, which we'll use to perform HTTP requests and effectively start retrieving websites with Python.

Whenever you surf the web, a whole series of networking protocols is being kicked into gear to set up connections to computers all over the world and retrieve data, all in a matter of seconds. Consider, for instance, the following series of steps that gets executed by your web browser once you navigate to a

website, say `www.google.com`:

- You enter `www.google.com` into your web browser, which needs to figure out the IP address for this site. The web provides a mechanism to translate domain names like `www.google.com` to an IP address (through DNS).
- Your browser can now establish a connection to `172.217.17.68`, Google's web server. Several protocals are combined to construct a complex message, including a message formatted by HTTP protocal which is used to request and receive web pages.
- Google's web server now sends back an HTTP reply, containing the contents of the page we want to visit. In most cases, this textual content is formatted using HTML.Note that a web page will oftentimes contain pieces of content for which the web browser will initiate new HTTP requests. In case the received page instructs the browser to show an image, for example, the browser will fire off another HTTP request to get the contents of the image.
- From this (oftentimes large) bunch of text, our web browser can set off to render the actual page, that is, making sure that everything appears neatly on screen as instructed by the HTML content.

The core component in the exchange of messages between web browsers and servers consists of a **HyperText Transfer Protocol (HTTP) request message** to a web server, followed by an **HTTP response** (also oftentimes called an HTTP reply), which can be rendered by the browser. Let us now take a look at what an HTTP request and reply look like.

2.1 HTTP Request

The following code fragment shows a full HTTP request message as executed by a web browser for searching https://www.uic.edu.cn (https://www.uic.edu.cn):

```
GET / HTTP/1.1

Host: www.uic.edu.cn

Connection: keep-alive

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8

Accept-Encoding: gzip, deflate, br

Accept-Language: en-US,en;q=0.9

Cookie: _pk_id.4.0073=468d1f7ff8b05150.1550718584.3.1551076516.1551076510.
```

- `GET / HTTP/1.1` is request line. It contains the HTTP method we want to execute (`GET` in this example), the URL we want to retrieve (`/`) and the HTTP version(`HTTP/1.1`). Don't worry too much about the `GET` method. HTTP has a number of verbs (that we'll discuss later on). For now, it is important to know that `GET` means this: "get the contents of this URL for me." Every time you enter a URL in your address bar and press enter, your browser will perform a `GET` request. Next up are the request headers, each on their own line. Each header includes a name followed by a colon and the actual value of the header.
  - `Host` is a standardized and mandatory header in HTTP 1.1 and higher, indicating from which domain name the server should retrive the page.
  - `Connection:keep-alive` signposts to the server that it should keep the connection open for subsequence requests if it can.
  - `Upgrade-Insecure-Requests:1` tells the server that it supports the upgrade mechanisms of upgrade-insecure-requests.
  - `User-Agent` contains a large text value through which the browser informs the server what it is and which version it is running as.

- `Accept` tells the server which forms of content the browser prefers to get back
- `Accept-Encoding` tells the server that the browser is also able to get back compressed content.
- `Cookie` shows the cookie list.

2.2 HTTP Response

The following shows the HTTP response:

```
HTTP/1.1 200 OK

Date: Sat, 17 Apr 2021 05:18:49 GMT

Server: VWebServer

Expires: Sat, 17 Apr 2021 05:28:49 GMT

Cache-Control: private, max-age=600

Vary: User-Agent

Last-Modified: Sat, 17 Apr 2021 04:48:46 GMT

Accept-Ranges: bytes

Content-Encoding: gzip

Content-Length: 12623

Content-Type: text/html

Content-Language: zh-CN

<CR><LF>

<html>

... Contents ..

</html>
```

- `HTTP/1.1 200 OK` indicates the status result of the request. It opens by listing the HTTP version the server underdstands, followed by a status code (`200`) and a status message (`OK`). If all goes well, the status will be 200. There are a number of agreed-upon HTTP status codes that we'll take a closer look at later on, but you're probably also familiar with the `404` status message, indicating that the URL listed in the request could not be retrieved, that was *not found* on the server.

Next up are, again, a number of headers, now coming from the server. Here, the server includes its current data and version in its headers. Another important header here is `Content-Type` as it will provide browsers with information regarding what the content included in the reply looks like. Following the headers is a blank `<CR><LF>` line and an optional message body containing the actual content of the reply.

2.3 Python `requests` library

Python provides built-in libraries for standard networking functionality, making sure that we neatly format HTTP request messages and are able to parse the incoming responses. Here, we will use `requests` (see http://docs.python-requests.org/ (http://docs.python-requests.org/)) library to deal with HTTP, which is an elegant and simple HTTP library for Python, built “for human beings.” Following is a simple web scraping program:

In [1]:
```
import requests
url = 'http://stuweb.uic.edu.cn/~ruimeng/basichttp/'
header = {'User-Agent':'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
          ' (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36'}
r = requests.get(url,headers = header, timeout=10)
print (r.text)
```

```
Hello from the Web! I love DS!
```

Try opening this web page in your browser. You'll see "Hello from the web!" appear on the page. This is what we extract using Python.

- We use the `requests.get` method to perform an `HTTP GET` request to the provided URL
- The `requests.get` method returns a response. Response python object containing lots of information regarding the HTTP reply that was retrieved.
- `r.text` contains the HTTP response content body in a textual form. Here, the HTTP response body simple contains the content `Hello from the web! I love DS!`.

In [2]:
```
# What is the request url?
print(r.url)
# Which HTTP status code did we get back from the server?
print(r.status_code)
# What is the textual status code?
print(r.reason)
# What were the HTTP response headers?
print(r.headers)
# The request information is saved as a Python object in r.request:
print(r.request)
# What were the HTTP request headers?
print(r.request.headers)
# The HTTP response content:
print(r.content)
```

```
http://stuweb.uic.edu.cn/~ruimeng/basichttp/ (http://stuweb.uic.edu.cn/~ruimeng/basichttp/)
200
OK
{'Date': 'Sun, 17 Oct 2021 11:17:49 GMT', 'Server': 'Apache/2.2.15 (CentOS)', 'Last-Modifie
d': 'Wed, 20 Mar 2019 01:22:30 GMT', 'ETag': '"6a004a-1f-5847c7494d3bc"', 'Accept-Ranges': 'b
ytes', 'Content-Length': '31', 'Connection': 'close', 'Content-Type': 'text/html; charset=UTF
-8'}
<PreparedRequest [GET]>
{'User-Agent': 'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHT
ML, like Gecko) Chrome/94.0.4606.81 Safari/537.36', 'Accept-Encoding': 'gzip, deflate', 'Acce
pt': '*/*', 'Connection': 'keep-alive'}
b'Hello from the Web! I love DS!\n'
```

- The headers attribute of the request.Response object (`r`) returns a dictionary of the headers the server included in its HTTP reply. This server reports its data, server version, and also provides the "Content-Type" header.
- Since an HTTP request message also includes headers, we can access the headers attribute for this object as well to get a dictionary representing the headers that were included by requests. Note that requests politely reports its "User-Agent" by default. In addition, requests can take care of compressed pages automatically as well, so it also includes an "Accept-Encoding" header to signpost this. Finally, it includes an "Accept" header to indicate that "any format you have can be sent back" and can deal with "keep-alive" connections as well. Later on, however, we'll see cases where we need to override requests' default request header behavior.

2.4 Query Strings: URLs with Parameters

There's one more thing we need to discuss regarding the basic working of HTTP: URL parameters. You've probably encountered this sort of URL many times when surfing the web, for example:

- https://www.google.com/search?q=web+scraping (https://www.google.com/search?q=web+scraping)
- https://www.baidu.com/s?wd=UIC (https://www.baidu.com/s?wd=UIC)

The optional `?⋯` part in URLs is called the `query string`, and it is meant to contain data that does not fit within a URL's normal hierarchical path structure. Web servers are smart pieces of software. When a server receives an HTTP request for such URLs, it may run a program that uses the parameters included in the query string — the `URL parameters` — to render different content, e.g., compare https://www.baidu.com/s?wd=web%20scraping (https://www.baidu.com/s?wd=web%20scraping) with https://www.baidu.com/s?wd=data%20science (https://www.baidu.com/s?wd=data%20science). Query strings in URLs should adhere to the following conventions:

- A query string comes at the end of a URL, starting with a single question mark, `?`.
- Parameters are provided as key-value pairs and separated by an ampersand, `&`. (E.g., https://www.google.com/search?q=web+scraping&safe=active (https://www.google.com/search?q=web+scraping&safe=active))
- The key and value are separated using an equals sign, `=`.
- Since some characters cannot be part of a URL or have a special meaning (the characters `/`, `?`, `&`, and `=` for instance), URL `encoding` needs to be applied to properly format such characters when using them inside of a URL.

Let's take a look at how to deal with URL parameters in requests. The easiest way to deal with these is to include them simply in the URL itself:

In [3]:
```
import requests


url = 'http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=DS'

r = requests.get(url, headers= header)
print(r.text)
# Will show: I don't have any information on "DS"
```

```
I don't have any information on DS.
```

In some cases, requests will try help you out and encode some characters for you:

In [4]:
```
url = 'http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=a query with spaces'
r = requests.get(url)
# Parameter will be encoded as 'a%20query%20with%20spaces'
# You can verify this be looking at the prepared request URL:
print(r.request.url)
# Will show [...]/paramhttp/?query=a%20query%20with%20spaces
print(r.text)
# Will show: I don't have any information on "a query with spaces"
```

```
http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=a%20query%20with%20spaces (http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=a%20query%20with%20spaces)
I don't have any information on a query with spaces.
```

However, sometimes the URL is too ambiguous for requests to make sense of it:

In [5]:
```
url = 'http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=complex?&'
# Parameter will not be encoded
r = requests.get(url)
# You can verify this be looking at the prepared request URL:
print(r.request.url)
# Will show [...]/paramhttp/?query=complex?&
print(r.text)
# Will show: I don't have any information on "complex?"
```

```
http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=complex?& (http://stuweb.uic.edu.hk/~ruimeng/paramhttp/?query=complex?&)
I don't have any information on complex?.
```

In this case, requests is unsure whether you meant "?&" to belong to the actual URL as is or whether you wanted to encode it. Hence, requests will do nothing and just request the URL as is. On the server-side, this particular web server is able to derive that the second question mark ("?") should be part of the URL parameter (and should have been properly encoded, but it won't complain), though the ampersand "&" is too ambiguous in this case. Here, the web server assumes that it is a normal separator and not part of the URL parameter value.

So how then, can we properly resolve this issue? A first method is to use the `url-lib.parse` functions, `quote` and `quote_plus`. The former is meant to encode special characters in the path section of URLs and encodes special characters using percent "%XX" encoding, including spaces. The latter does the same, but replaces spaces by plus signs, and it is generally used to encode query strings:

In [6]:
```
import requests
from urllib.parse import quote, quote_plus
raw_string = 'a query with /, spaces and?&'
print(quote(raw_string))
print(quote_plus(raw_string))
```

```
a%20query%20with%20/%2C%20spaces%20and%3F%26
a+query+with+%2F%2C+spaces+and%3F%26
```

The quote function applies percent encoding, but leaves the slash (/) intact (as its default setting, at least) as this function is meant to be used on URL paths. The quote_ plus function does apply a similar encoding, but uses a plus sign (+) to encode spaces and will also encode slashes.

3. Understanding HTML

So far we have discussed the basics of HTTP and how you can perform HTTP requests in Python using the requests library. However, since most web pages are formatted using the Hypertext Markup Language (HTML), we need to understand how to extract information from such pages.

3.1 Hypertext Markup Language (HTML)

The key to understanding any scraping is looking at the HTML and understanding how you want to pull your data out. We use the imdb webpage as an example:

User-Agent:

- A user agent is a string that a browser or app sends to each website you visit. A typical user agent string contains details like – the application type, operating system, software vendor or software version of the requesting software user agent. Web servers use this data to assess the capabilities of your computer, optimizing a page's performance and display. User Agents are sent as a request header called "User-Agent".

- There are some webservers that will reject a request if the User-Agent is unknown/unrecognized. Then you have to provide a custom User-Agent that mimics a well-known value that the webserver will accept. You can open your browser and check the User-Agent information in your request header. Following is an example.
- If you are blocked or banned while scraping a website, you can usually get past those bans by sending random user agents instead of sticking to one, and make requests through a set of IP Addresses or reduceing crawling rate. Refer to https://www.scrapehero.com/how-to-rotate-proxies-and-ip-addresses-using-python-3/ (https://www.scrapehero.com/how-to-rotate-proxies-and-ip-addresses-using-python-3/) for details

In [7]:
```
url = 'https://www.imdb.com/title/tt0944947/'

r = requests.get(url, headers = header)
r.text
```

Out[7]:

```
'<!DOCTYPE html><html lang="en-US" xmlns:og="http://opengraphprotocol.org/schema/" xmlns:fb="http://www.facebook.com/2008/fbml"><head><script></script><script>if(typeof uet === \'function\'){ uet(\'bb\', \'LoadAds\', {wb: 1}); }</script><script>if(typeof uet === \'function\'){ uet(\'be\', \'LoadAds\', {wb: 1}); }</script><script>if(typeof uex === \'function\'){ uex(\'ld\', \'LoadAds\', {wb: 1}); }</script><script>if(typeof uet === \'function\'){ uet(\'bb\', \'LoadStyledComponents\', {wb: 1}); }</script><style data-styled="" data-styled-version="5.2.1">.drxdwu{width:100%;}/*!sc*/\ndata-styled.g28[id="MainColumnStyledEditorialSingle__StyledSlateCard-sc-11k6l2x-0"]{content:"drxdwu,"}/*!sc*/\n.hdblxT{width:100%;}/*!sc*/\n@media screen and (min-width:600px){.hdblxT{width:63%;}}/*!sc*/\ndata-styled.g29[id="MainColumnStyledEditorialSingle__SlateCardContainer-sc-11k6l2x-1"]{content:"hdblxT,"}/*!sc*/\n.ilrSAi{display:-webkit-box;display:-webkit-flex;display:-ms-flexbox;display:flex;-webkit-flex-direction:column;-ms-flex-direction:column;flex-direction:column;padding-right:1rem;padding-right:var(--ipt-pageMargin,1rem);padding-left:1rem;padding-left:var(--ipt-pageMargin,1rem);}/*!sc*/\n.ilrSAi.ilrSAi{margin-bottom:.75rem;margin-bottom:var(--ipc-pageSection-bottomMargin);}/*!sc*/\ndata-styled.g34[id="MainColumnStyledEditorialSingle__EditorialContainer-sc-11k6l2x-6"]{content:"ilrSAi,"}/*!sc*/\n.feEynz{display:-webkit-box;display:-webkit-flex;display:-ms-flexbox;display:flex;-webkit-flex-direction:column;-ms-flex-direction:column;flex-direction:column;margin-top:1rem;}/*!sc*/\n@media screen and (min-width:600px){.feEynz{width:37%;margin-right:1rem;margin-top:0;}}/*!sc*/\ndata-styled.g35[id="M
```


The returned content is a web page which looks like a textual mess. We need to convert it into nicely formatted pages and dig out useful information. Just as with HTTP, python provides a powerful library that can help us navigate this textual mess. And, contrary to a web browser, we're not interested in fetching out a complete page's content and rendering it, but only in extracting those pieces we're interested in.

HTML Tags

Hypertext Markup Language (HTML) is the standard markup language for creating web pages. HTML provides the building blocks to provide structure and formatting to documents. This is provided by means of a series of `tags`. HTML tags often come in pairs and are enclosed in angled brackets, with `<tagname>` being the opening tag and `</tagname>` indicating the close tag. Following is a simple HTML document:

```
<html>
   <body>
       <h1> My First Heading </h1>
       <p> My first paragraph.</p>
   </body>
 </html>
```

- The text between `<html>` and `</html>` describes the web page
- The text between `<body>` and `</body>` is the visible page content
- The text between `<h1>` and `</h1>` is displayed as a heading
- The text between `<p>` and `<p>` is displayed as a paragraph

Some commonly used tags are the following:

- `<p>...</p>` to enclose a paragraph;
- `<br>` to set a line break;
- `<table>...</table>` to start a table block, inside; `<tr>...<tr/>` is used for the rows; and `<td>...</td>` cells;
- `<img>` for images;
- `<h1>...</h1>` to `<h6>...</h6>` for headers;
- `<div>...</div>` to indicate a “division” in an HTML document, basically used to group a set of elements;
- `<a>...</a>` for hyperlinks;
- `<ul>...</ul>`, `<ol>...</ol>` for unordered and ordered lists respectively; inside of these, `<li>...</li>` is used for each list item.

HTML elements & Attributes

An **HTML element** is everything from the start tag to the end tag. Tags that come in pairs have content, which is called element content.

Start Tag	Element Content	End Tag
`<p>`	This is a paragragh	`</p>`
`<a>`	click here	`</a>`
` `		

HTML elements can have **attributes**:

- Attributes provide additional information about an element
- Attributes are always specified in the start tag
- Attributes come in name/value pairs like: name="value“

For instance: `<a href=“http://www.google.com”> click here </a>` will redirect the user to Google’s home page when the link is clicked. The `href` attribute hence indicates the web address of the link. For an image tag, which doesn’t come in a pair, the `src` attribute is used to indicate the URL of the image the browser should retrieve, for example, `<img src=“figures/image.jpg”>`.

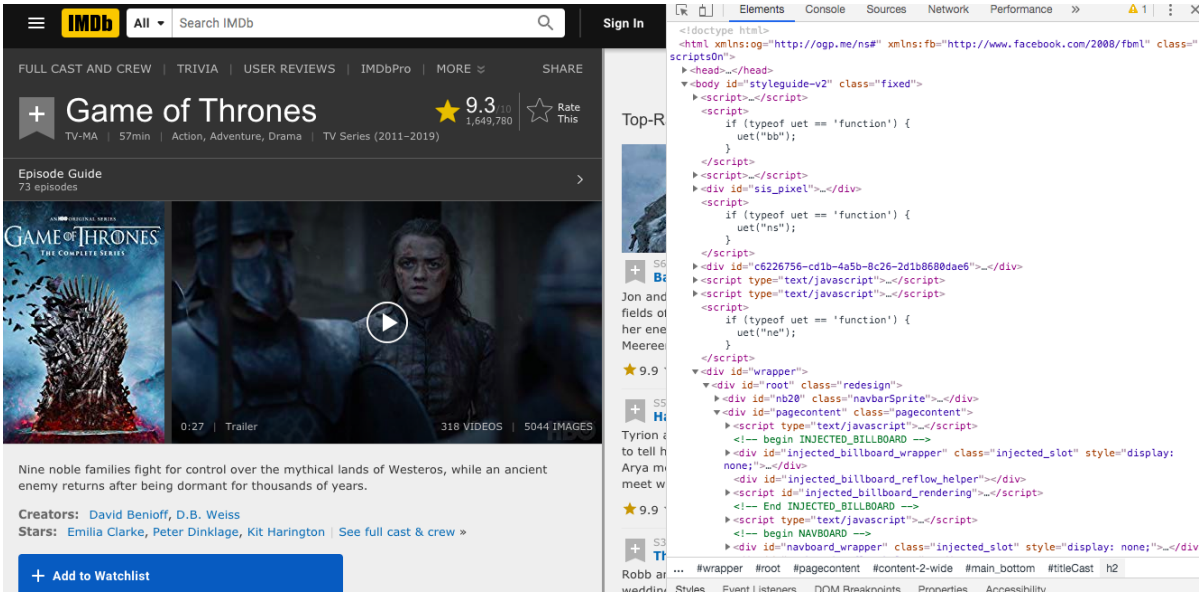
Browser as a Development Tool

Before we continue, we want to provide you with a few tips that will come in handy while building web scrapers. Most modern web browsers nowadays include a toolkit of powerful tools you can use to get an idea of what’s going on regarding HTML, for the following, we use Google Chrome for illustration.

First of all, it is helpful to know how you can take a look at the underlying HTML of this page. To do so, you can right-click on the page and press **View page source**. A new page will open containing the raw HTML contents for the current page (the same content as what we got back using **r.text**).

```
<!DOCTYPE html>
<html
    xmlns:og="http://ogp.me/ns#"
    xmlns:fb="http://www.facebook.com/2008/fbml">
    <head>

<script type='text/javascript'>var ue_t0=ue_t0||+new Date();</script>
<script type='text/javascript'>
window.ue_ihb = (window.ue_ihb || window.ueinit || 0) + 1;
if (window.ue_ihb === 1) {

var ue_csm = window,
    ue_hob = +new Date();
(function(d){var e=d.ue=d.ue||{},f=Date.now||function(){return+new Date};e.d=function(b){return f()-(b?0:d.ue_t0)};e.stub=function(b,a){if(!b[a]){var c=
[];b[a]=function(){c.push([c.slice.call(arguments),e.d(),d.ue_id])};b[a].replay=function(b){for(var
a;a=c.shift();)b(a[0],a[1],a[2])};b[a].isStub=1}};e.exec=function(b,a){return function(){try{return b.apply(this,arguments)}catch(c){ueLogError(c,
{attribution:a||"undefined",logLevel:"WARN"})}}}})(ue_csm);


    var ue_err_chan = 'jserr';
(function(d,e){function h(f,b){if(!(a.ec>a.mxe)&&f){a.ter.push(f);b=b||{};var
c=f.logLevel||b.logLevel;c&&c!==k&&c!==m&&c!==n&&c!==p||a.ec++;c&&c!=k||a.ecf++;b.pageURL=""+(e.location?
e.location.href:"");b.logLevel=c;b.attribution=f.attribution||b.attribution;a.erl.push({ex:f,info:b})}}function l(a,b,c,e,g)
{d.ueLogError({m:a,f:b,l:c,c:""+e,err:g,fromOnError:1,args:arguments},g?{attribution:g.attribution,logLevel:g.logLevel}:void 0);return!1}var
k="FATAL",m="ERROR",n="WARN",p="DOWNGRADED",a={ec:0,ecf:0,
pec:0,ts:0,erl:[],ter:[],mxe:50,startTimer:function(){a.ts++;setInterval(function()
{d.ue&&a.pec<a.ec&&d.uex("at");a.pec=a.ec},1E4)}};l.skipTrace=1;h.skipTrace=1;h.isStub=1;d.ueLogError=h;d.ue_err=a;e.onerror=l})(ue_csm,window);


var ue_id = '1CVVWTC9JDTXMMQFMHR4',
    ue_url,
    ue_navtiming = 1,
    ue_mid = 'A1EVAM02EL8SFB',
    ue_sid = '143-0218484-0807223',
    ue_sn = 'www.imdb.com',
    ue_furl = 'fls-na.amazon.com',
    ue_surl = 'https://unagi-na.amazon.com/1/events/com.amazon.csm.nexusclient.prod',
    ue_int = 0,
    ue_fcsn = 1,
    ue_urt = 3,
    ue_rpl_ns = 'cel-rpl',
    ue_ddq = 1,
    ue_fpf = '//fls-na.amazon.com/1/batch/1/OP/A1EVAM02EL8SFB:143-0218484-0807223:1CVVWTC9JDTXMMQFMHR4$uedata=s:',
    ue_sbuimp = 1,

    ue_swi = 1;
```

Additionally, you can right click and select `Inspect` to open the Chrome Developer Tools, which contains a lot of helpful tools for web scrapers. You should get a screen like the one shown as follows:



The Developer Tools pane is organized by means of a series of tabs, of which `Elements` and `Network` will come in most helpful.

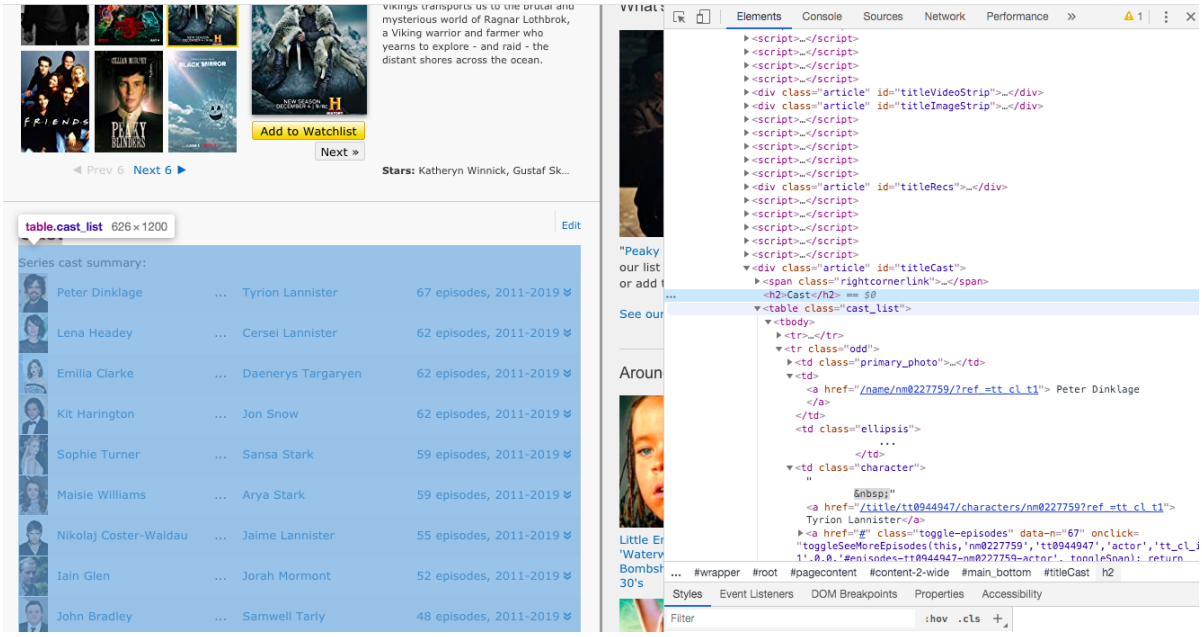
Let's start by taking a look at the `Network` tab. You should see a red `recording` icon in the toolbar indicating that Chrome is tracking network requests. Refresh the webpage and look at what happens in the Developer Tools pane:

- Chrome starts logging all requests it is making, starting with an HTTP request for the page itself at the top.
- Note that your web browser is also making lots of other requests to actually render the page, most of them to fetch image data (`Type: png`). By clicking a request, you can get more information about it.

Moving on to the `Elements` tab, we see a similar view as what we see when viewing the page's source, though now neatly formatted as a tree-based view, with little arrows that we can expand and collapse.

- What is particularly helpful here is the fact that you can hover over the HTML tags in the Elements tab, and Chrome will show a transparent box over the corresponding visual representation on the web

page itself. This can help you to quickly find the pieces of content you're looking for.
- Alternatively, you can right-click any element on a web page and press "Inspect" to immediately highlight its corresponding HTML code in the Elements tab.



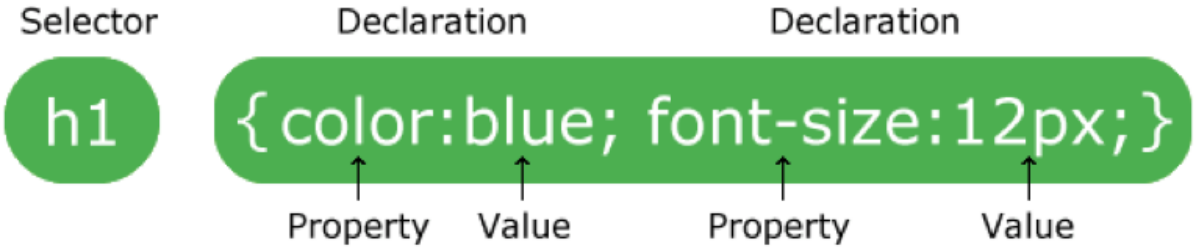
Next, note that any HTML element in the Elements tab can be right-clicked. `Copy`, `Copy selector` and `Copy XPath` are particularly useful, which we're going to use quite often later on. You'll even see that you can edit the HTML code in real time (the web page will update itself to reflect your edits). But these changes are of course only local. They don't do anything on the web server itself and will be gone once you refresh the page, though it can be a fun way to experiment with HTML.

Cascading Style Sheet

CSS stands for Cascading Style Sheets. CSS describes how HTML elements are to be displayed, including the design, layout and variations in display for different devices and screen sizes.

A CSS rule, e.g., `p {color:red; text-align:center; }` , has two main parts:

- A selector : points to the HTML element you want to style.
- Declaration block: contains one or more declarations separated by semicolons. Each declaration include a CSS property name and a value, separated by a colon.



CSS selectors define the patterns used to "select" the HTML elements you want to style. They are used to "find" (or select) HTML elements based on their

- element name: specify the name of the start tag
- id:
  - The id of an element should be unique within a page, so the id selector is used to select one unique element
  - To select an element with a specific id, write a hash (#) character, followed by the id of the element

- clsss: To select elements with a specific class, write a period (.) character, followed by the name of the class

Following is a simple example using different CSS selectors:

```
<html>
    <head>
    <style>
        p {color: blue; text-align: right }
           .center {text-align: center}
           .left {text-align: left}
           #para1 {color : red; }
    </style>
    </head>
    <body>
        <p> This is a paragraph!</p>
        <p id= "para1"> Another paragraph!</p>
        <p class="center">I belong to center class!</p>
        <p class="left">I belong to right class!</p>
    </body>
</html>
```

The `BeautifulSoup` Library

We're now ready to start working with HTML pages using Python. Recall the following lines of code:

In [8]:
```
url = 'https://www.imdb.com/title/tt0944947/'
r = requests.get(url)
html_contents = r.text
```

Python provide a `BeautifulSoup` library to deal with the HTML contained in html_contents.

In [9]:
```
from bs4 import BeautifulSoup
html_soup = BeautifulSoup(html_contents, "html.parser")
```

Beautiful Soup's main task is to take HTML content and transform it into a tree-based representation. Once you've created a BeautifulSoup object, there are two methods you'll be using to fetch data from the page:

- `find(name, attrs, recursive, string, **keywords)` : returns the first child of this Tag matching the given criteria.
- `find_all(name, attrs, recursive, string, limit, **keywords)` : returns a list of Tag objects that match the given criteria.

Both methods look very similar indeed, with the exception that find_all takes an extra limit argument.

- The `name` argument defines the tag names you wish to "find" on the page. You can pass a string, or a list of tags. Leaving this argument as an empty string simply selects all elements.
- The `attrs` argument takes a Python dictionary of attributes and matches HTML elements that match those attributes.
- The `recursive` argument is a Boolean and governs the depth of the search.
  - If set to `True` , the default value, the find and find_all methods will look into children, children's children, and so on... for elements that match your query.
  - If it is `False` , it will only look at direct child elements.
- The `string` argument is used to perform matching based on the text content of elements.
- The `limit` argument is only used in the find_all method and can be used to limit the number of elements that are retrieved.

- `**keywords` is kind of a special case. Basically, this part of the method signature indicates that you can add in as many extra named arguments as you like, which will then simply be used as attribute

In [10]:
```
print("Find tags with name 'h1':")
print(html_soup.find('h1'))
print("Find all tags with id attribute value 'titleVideoStrip':")
print (html_soup.find('',{'id':'titleVideoStrip'}))
print("Find all tags with name 'h1' or 'h2':")
for found in html_soup.find_all(['h1','h2']):
    print(found)
```

```
Find tags with name 'h1':
<h1 class="TitleHeader__TitleText-sc-1wu6n3d-0 dxSWFG" data-testid="hero-title-block__title" textlength="15">Game of Thrones</h1>
Find all tags with id attribute value 'titleVideoStrip':
None
Find all tags with name 'h1' or 'h2':
<h1 class="TitleHeader__TitleText-sc-1wu6n3d-0 dxSWFG" data-testid="hero-title-block__title" textlength="15">Game of Thrones</h1>
<h2 class="MainColumnStyledEditorialSingle__Title-sc-11k6l2x-9 MainColumnStyledEditorialSingle__TitleStackedOnContent-sc-11k6l2x-10 ivMuNr gpedaY">Streaming Picks for a Post-Westeros World</h2>
<h2 class="MainColumnStyledEditorialSingle__Title-sc-11k6l2x-9 MainColumnStyledEditorialSingle__TitleInlineWithContent-sc-11k6l2x-11 ivMuNr ikMByw">Streaming Picks for a Post-Westeros World</h2>
```

From the example, we can observe that both `find` and `find_all` return Tag objects. Using these, there are a number of interesting things you can do:

- Access the `name` attribute to retrieve the tag name.
- The `children` attribute does the same but provides an iterator instead; the `descendants` attribute also returns an iterator, including all the tag's descendants in a recursive manner.
- Go "up" the HTML tree by using the `parent` and `parents` attributes. To go sideways (i.e., find next and previous elements at the same level in the hierarchy), `next_sibling`, `previous_sibling` and `next_siblings`, and `previous_siblings` can be used.
- Access the attributes of the element through the `attrs` attribute of the Tag object. For the sake of convenience, you can also directly use the Tag object itself as a dictionary.
- Use the `text` attribute to get the contents of the Tag object as clear text (without HTML tags).

Following code illustrates these concepts:

In [11]:
```
from bs4 import BeautifulSoup
import requests

url = 'https://www.imdb.com/title/tt0944947/'
r = requests.get(url, headers = header)
html_contents = r.text

html_soup = BeautifulSoup(html_contents, "html.parser")

# Find first tag with specific class attribute
first_tag = html_soup.find('li',{'class':'ipc-metadata-list__item'})
print(first_tag.name)
print("tag.parent:",first_tag.parent.name)
print("first_tag.text:",first_tag.text)
print("first_tag.get_text():",first_tag.get_text()) #Does the same with .text
print("first_tag.attrs:",first_tag.attrs)
print(first_tag.attrs['class'])
print(first_tag['class']) # Does the same
print(first_tag.get('class')) # Does the same
print('------------ Did you know? ------------')
# Find the first three cite elements with a text-block class
doyouknow = html_soup.find_all('li', class_='ipc-metadata-list__item ipc-metadata-list__item--s
for case in doyouknow:
    print("case start..\n")
    heading = case.find('a').text
    content = case.find('div').text
    print(heading)
    print(content)
```

```
li
tag.parent: ul
first_tag.text: CreatorsDavid BenioffD.B. Weiss
first_tag.get_text(): CreatorsDavid BenioffD.B. Weiss
first_tag.attrs: {'role': 'presentation', 'class': ['ipc-metadata-list__item'], 'data-test
id': 'title-pc-principal-credit'}
['ipc-metadata-list__item']
['ipc-metadata-list__item']
['ipc-metadata-list__item']
------------ Did you know? ------------
case start..

Trivia
According to Kit Harington (Jon Snow), his performance in the rejected pilot episode was s
o bad that the creators often jokingly threaten to release scenes of it on the internet if
he complains too much.
case start..

Quotes
```

Note that

- If you find yourself traversing a chain of tag names as follows: `tag.find('div').find('table').find('thead').find('tr')` It might be useful to keep in mind that Beautiful Soup also allows us to write this in a shorthand way: `tag.div.table.thead.tr`
- Similarly, the following line of code: `tag.find_all('h1')` is the same as calling: `tag('h1')`.

Let us now try to work out the following use case. You'll note that our Game of Thrones page has a well-maintained table listing the actor/actress name, character name and how many episodes in total does that character show up. Let's try to fetch all of this data at once using what we have learned:

In [12]:
```
from bs4 import BeautifulSoup
import requests
import pandas as pd
url = 'https://www.imdb.com/title/tt0944947/fullcredits?ref_=tt_cl_sm'
r = requests.get(url, headers = header)
html_contents = r.text
html_soup = BeautifulSoup(html_contents, "html.parser")

# use lists to hold the attributes
actors = []
characters = []
episodes = []

table = html_soup.find('table', class_='cast_list')

for row in table.find_all('tr')[1:30]: # ignore the first row
    cols = row.find_all('td')
    if(len(cols)<4): # handle the extra tr
        continue
    if (len(cols[3].find_all('a'))<2):
        continue
    actors.append(cols[1].find('a').text.strip()) # get the actor name

    chara = cols[3].find_all('a')[0].text.strip() # get the character
    epi = cols[3].find_all('a')[1].text.strip()  # get the episode info
    characters.append(chara)
    episodes.append(epi)

pd.DataFrame({'Actor':actors, 'Character':characters,'Episodes':episodes})
```

Out[12]:

	Actor	Character	Episodes
0	Peter Dinklage	Tyrion Lannister	67 episodes, 2011-2019
1	Lena Headey	Cersei Lannister	62 episodes, 2011-2019
2	Emilia Clarke	Daenerys Targaryen	62 episodes, 2011-2019
3	Kit Harington	Jon Snow	62 episodes, 2011-2019
4	Sophie Turner	Sansa Stark	59 episodes, 2011-2019
5	Maisie Williams	Arya Stark	59 episodes, 2011-2019
6	Nikolaj Coster-Waldau	Jaime Lannister	55 episodes, 2011-2019
7	Iain Glen	Jorah Mormont	52 episodes, 2011-2019
8	John Bradley	Samwell Tarly	48 episodes, 2011-2019
9	Alfie Allen	Theon Greyjoy	47 episodes, 2011-2019
10	Conleth Hill	Lord Varys	46 episodes, 2011-2019
11	Liam Cunningham	Davos Seaworth	42 episodes, 2012-2019
12	Gwendoline Christie	Brienne of Tarth	42 episodes, 2012-2019
13	Aidan Gillen	Petyr 'Littlefinger' Baelish	41 episodes, 2011-2017
14	Isaac Hempstead Wright	Bran Stark	40 episodes, 2011-2019

4. HTML Forms and POST Request

We've already seen most of the core building blocks that make up the modern web: HTTP, HTML, and CSS. However, we're not completely finished with HTTP yet. So far, we've only been using one of HTTP's request **verbs** or **methods**: `GET` . This part will introduce you another method, `POST` , which is commonly used to submit web forms. Then, we introduce an example of web scrapers on real-life website.

1.1 HTML Forms

We've already seen one way how web browsers (and you) can pass input to a web server, that is, by simply including it in the requested URL itself, either by including URL parameters or simply by means of the URL path itself, e.g., https://www.google.com/search?q=web+scraping&oq=web+scraping (https://www.google.com/search?q=web+scraping&oq=web+scraping). However, one can easily argue that this way of providing input is not that user friendly. Imagine that we'd want to buy some tickets for a concert, and that we'd be asked to send our details to a web server by including our name, e-mail address, and other information as a bunch of URL parameters. Not a very pleasant idea indeed! In addition, URLs are (by definition) limited in terms of length, so in case we want to send lots of information to a web server, this "solution" would also fail to work.

Websites provide a much better way to facilitate providing input and sending that input to a web server, one that you have no doubt already encountered: web forms. Whether it is to provide a "newsletter sign up" form, a "buy ticket" form, or simply a "login" form, **web forms** are used to collect the appropriate data. The way how web forms are shown in a web browser is simply by including the appropriate tags inside of HTML. That is, each web form on a page corresponds with a block of HTML code enclosed in `<form>` tags:

```
<form> ..... </form>
```

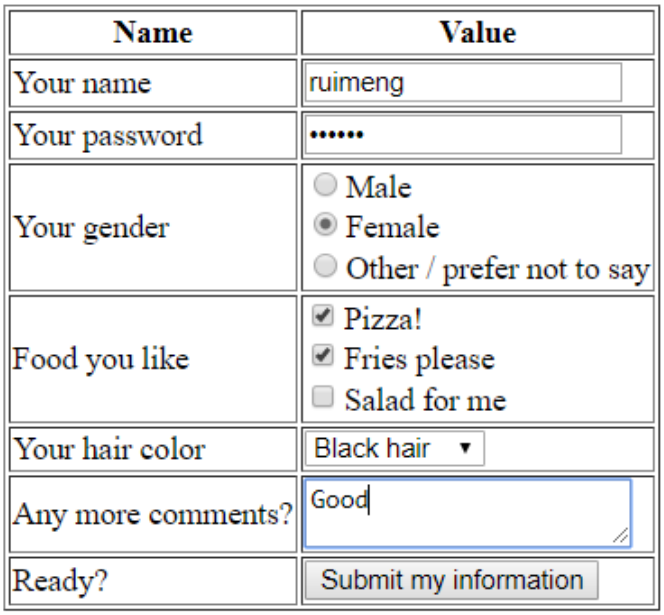
Inside of the form tags, there are a number of tags that represent the form fields themselves. Most of these are provided through an `<input>` tag, with the `type` attribute specifying what kind of field it should represent:

Tag	Description
`<input type="text">`	Simple text fields
`<input type="password">`	Password entry fields
`<input type="button">`	General purpose buttons
`<input type="reset">`	Reset button (when clicked, the browser will reset all form values)
`<input type="Submit">`	Submit button
`<input type="checkbok">`	Check boxes
`<input type="radio">`	Radio boxes

Apart from these, you'll also find pairs of tags being used (`<input>` does not come with a closing tag):

Tag	Description
`<select>...</select>`	Drop-down lists. Within these, every choice is defined by using `<option>...</option>`
`<textarea>...</textarea>`	Large text entry fields
`<buttion>...</button>`	Another way to define buttons

Navigate to http://stuweb.uic.edu.hk/~ruimeng/Form/basic_form.html (http://stuweb.uic.edu.hk/~ruimeng/Form/basic_form.html) to see basic web form in action. If you take a look at the source page, you will notice that some tags have "name" and "value" attribtues, e.g., `<input type="radio" name="gender" value="M">` . These attributes will be used by your web browser once a web form is "submitted." To do so, try pressing the `Submit my information` button on the example web page. Notice that upon submitting a form, your browser fires of a new HTTP request and includes the entered information in its request. In this simple form, a simple `HTTP GET` request is being used, basically converting the fields in the form to key-value URL parameters, e.g., after clicking the submit button, we will see the following url.



Name	Value
Your name	ruimeng
Your password	••••••
Your gender	○ Male ◉ Female ○ Other / prefer not to say
Food you like	☑ Pizza! ☑ Fries please ☐ Salad for me
Your hair color	Black hair ▾
Any more comments?	Good
Ready?	Submit my information

http://stuweb.uic.edu.hk/~ruimeng/Form/basic_form.html?

1.2 POST Request

As we mentioned, its not a good idea to simply allow for such requests to come in as HTTP GET requests. Now, we introduce another type of request that your browser will often use in case you wish to submit information to a web server: the `POST` request. To introduce it, navigate to the page over at http://stuweb.uic.edu.hk/~ruimeng/Form/post_form.html (http://stuweb.uic.edu.hk/~ruimeng/Form/post_form.html).

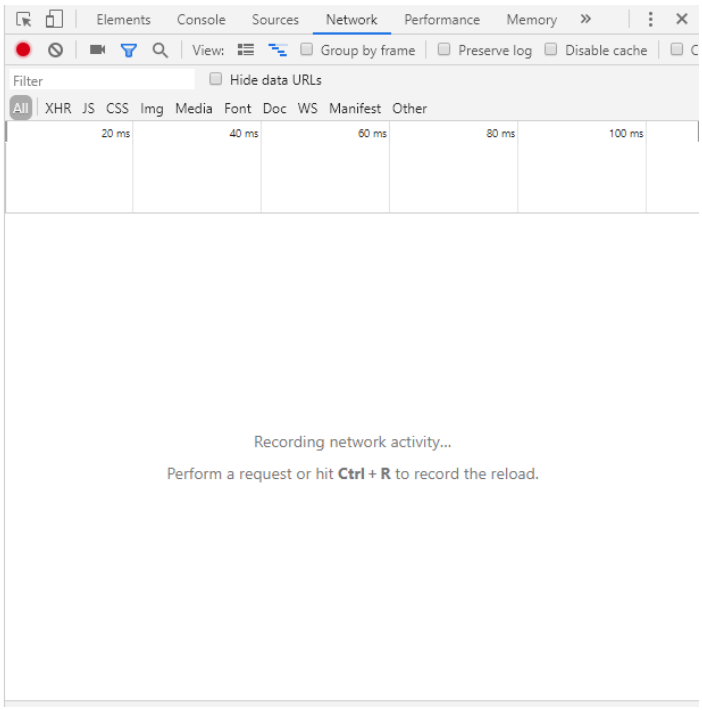
Notice that this page looks exactly the same as the one from above, except for one small difference in the HTML source: there's an extra `method` attribute for the `<form>` tag now:

```
<form method="POST"> ..... </form>
```

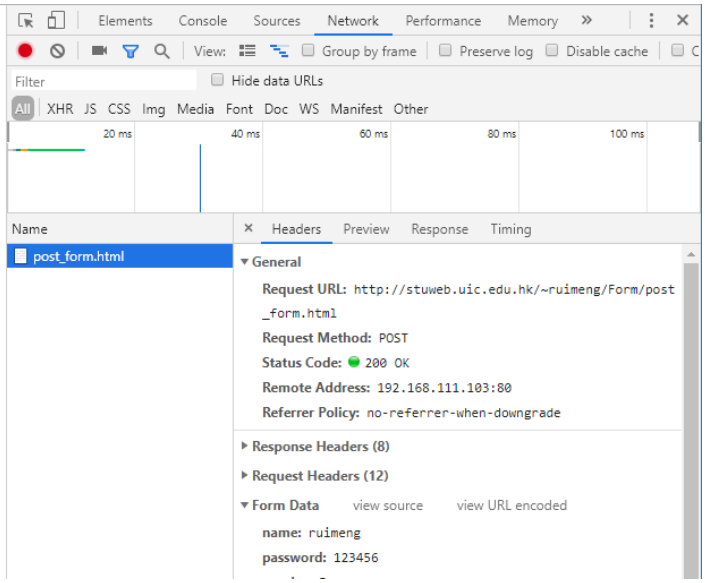
- The default value for the `method` attribute is `get`, basically instructing your browser that the contents of this particular form should be submitted through an HTTP GET request.
- When set to `post`, however, your browser will be instructed to send the information through an HTTP POST request. Instead of including all the form information as URL parameters, the POST HTTP request includes this input as part of the HTTP request body instead.

Try this by pressing `Submit my information` while making sure that Chrome's Developer Tools are monitoring network requests. If you inspect the HTTP request, you'll notice that the request method is now set to `POST` and that Chrome includes an extra piece of information called `Form Data` to show you which information was included as part of the HTTP request body.

Name	Value
Your name	ruimeng
Your password	••••••
Your gender	○ Male ◉ Female ○ Other / prefer not to say
Food you like	☑ Pizza! ☑ Fries please ☐ Salad for me
Your hair color	Black hair ▾
Any more comments?	Good
Ready?	Submit my information



Name	Value
Your name	ruimeng
Your password	••••••••••
Your gender	○ Male ○ Female ○ Other / prefer not to say
Food you like	☐ Pizza! ☐ Fries please ☐ Salad for me
Your hair color	Black hair ▾
Any more comments?	
Ready?	Submit my information



In [13]:

```
import requests
url = 'http://stuweb.uic.edu.hk/~ruimeng/PostForm/'
# First perform a GET request
# r = requests.get(url)
# Followed by a POST request
formdata = {
'name': 'ruimeng',
'password': '123456',
'gender': 'F',
'food[]':['Pizza','Fries please'],
'haircolor': 'black',
'comments': 'Good'
}
r = requests.post(url, data=formdata)
print(r.text)
```

```
<html>
	<body>

		<form method="POST" action="/~ruimeng/PostForm/index.php">

			<table border="1">
				<tr><th>Name</th><th>Value</th></tr>

				<tr><td>Your name</td>
					<td><input type="text" name="name"></td></tr>
				<tr><td>Your password</td>
					<td><input type="password" name="password"></td></tr>

				<tr><td>Your gender</td>
					<td><input type="radio" name="gender" value="M">Male<
br>
						<input type="radio" name="gender" value="F">F
emale<br>
						<input type="radio" name="gender" value="N">O
ther / prefer not to say</td></tr>

				<tr><td>Food you like</td>
					<td><input type="checkbox" name="food[]" value="Pizz
a">Pizza!<br>
						<input type="checkbox" name="food[]" value="F
ries please">Fries please<br>
						<input type="checkbox" name="food[]" value="S
alad for me">Salad for me</td></tr>

				<tr><td>Your hair color</td>
					<td>
						<select name="haircolor">
							<option value="black" selected>Black
hair</option>
							<option value="brown">Brown hair</opt
ion>
							<option value="blonde">Blonde hair</o
ption>
							<option value="other">Other</option>
						</select>
					</td></tr>

				<tr><td>Any more comments?</td>
					<td>
						<textarea name="comments"></textarea>
					</td></tr>

				<tr><td>Ready?</td>
					<td>
						<input type="submit" value="Submit my informa
tion">
					</td></tr>
			</table>

		</form>
		<h2>Thanks for submitting your information <br></h2>Name:ruimeng<br>Gender:F<
br>Like Food:Pizza,Fries please<br>Hair:black<br>Comments:Good<br>
	</body>
</html>
```

The data argument is supplied as a Python dictionary object representing name-value pairs. Take some time to play around with this example to see how data for the various input elements is actually submitted.

Note that in our example above, we're still *polite* in the sense that we first execute a normal GET request before sending the information through a POST, though this is not even required. We can simply comment

5 Web Scraping Example

Now, let's look at an example which are more data science oriented use cases.

- We're going to start simple by scraping a list of reviews for episodes of a TV series, using IMDB (the Internet Movie Database). We'll use Game of Thrones as an example, the episode list for which can be found at http://www.imdb.com/title/tt0944947/episodes (http://www.imdb.com/title/tt0944947/episodes). Note that IMDB's overview is spread out across multiple pages (per season or per year).

As introduced in the section "Query Strings: URLs with Parameters", we can have query string in the url. In the following example, we use another way to perform such query by the usage of `params` argument in the `request.get` method. You can simply pass a Python dictionary with your non-encoded URL parameters and the requests will take care of encoding them for you.

If you click the url (http://www.imdb.com/title/tt0944947/episodes) (http://www.imdb.com/title/tt0944947/episodes)), you will find that that IMDB's overview is spread out across multiple pages(per season or per year), so we iterate over the seasons, and the season number can be passed as a query string, e.g., https://www.imdb.com/title/tt0944947/episodes?season=6 (https://www.imdb.com/title/tt0944947/episodes?season=6).

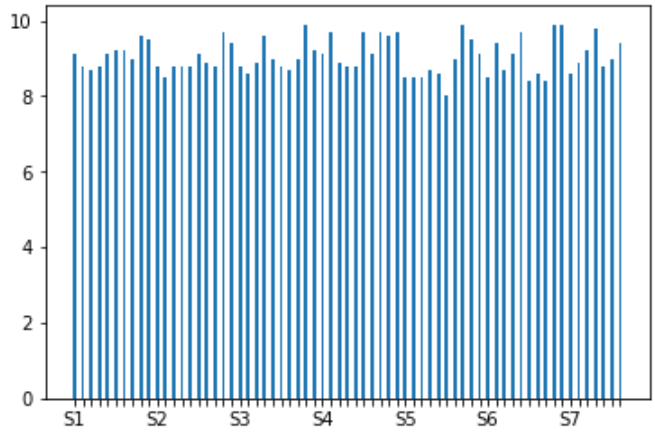
You may encounter the exception: `ConnectionError: HTTPConnectionPool(host='xxx.xx.xxx.xxx', port=xxxx): Max retries exceeded with url:xx` if using `requests` to establish multiple connections without closing. You can use the following example code to avoid such exception.

```
requests.adapters.DEFAULT_RETRIES = 5
    s = requests.session()
    s.keep_alive = False
    url = 'xx'
    r = s.get(url,params=xx)
```

In [14]:

```
import requests
import matplotlib.pyplot as plt
from bs4 import BeautifulSoup
url = 'http://www.imdb.com/title/tt0944947/episodes'
episodes = []
ratings = []
# Go over seasons 1 to 7
for season in range(1, 8):
    # query string in URL by using params argument and pass a dictionary
    requests.adapters.DEFAULT_RETRIES = 5
    s = requests.session()
    s.keep_alive = False
    r = s.get(url, params={'season': season})
    soup = BeautifulSoup(r.text, 'html.parser')
    listing = soup.find('div', class_='eplist')
    for epnr, div in enumerate(listing.find_all('div', recursive=False)):
        episode = "{0}.{1}".format(season, epnr + 1)
        rating_el = div.find(class_='ipl-rating-star__rating')
        rating = float(rating_el.get_text(strip=True))
        print('Episode:', episode, '-- rating:', rating)
        episodes.append(episode)
        ratings.append(rating)

episodes = ['S' + e.split('.')[0] if int(e.split('.')[1]) == 1 else '' for e in episodes]
# ['S1','','','','',''......., 'S2','','',....]

# We can then plot the scraped ratings using “matplotlib,” a well-known plotting library for P
%matplotlib inline
plt.figure()
positions = [a*2 for a in range(len(ratings))] # x coordinates of the bars
plt.bar(positions, ratings, align='center');
plt.xticks(positions, episodes); # x coordinates, labels
# plt.show()
```

Episode: 1.1 -- rating: 9.1
Episode: 1.2 -- rating: 8.8
Episode: 1.3 -- rating: 8.7
Episode: 1.4 -- rating: 8.8
Episode: 1.5 -- rating: 9.1
Episode: 1.6 -- rating: 9.2
Episode: 1.7 -- rating: 9.2
Episode: 1.8 -- rating: 9.0
Episode: 1.9 -- rating: 9.6
Episode: 1.10 -- rating: 9.5
Episode: 2.1 -- rating: 8.8
Episode: 2.2 -- rating: 8.5
Episode: 2.3 -- rating: 8.8
Episode: 2.4 -- rating: 8.8
Episode: 2.5 -- rating: 8.8
Episode: 2.6 -- rating: 9.1
Episode: 2.7 -- rating: 8.9
Episode: 2.8 -- rating: 8.8
Episode: 2.9 -- rating: 9.7
Episode: 2.10 -- rating: 9.4
Episode: 3.1 -- rating: 8.8
Episode: 3.2 -- rating: 8.6
Episode: 3.3 -- rating: 8.9
Episode: 3.4 -- rating: 9.6
Episode: 3.5 -- rating: 9.0
Episode: 3.6 -- rating: 8.8
Episode: 3.7 -- rating: 8.7
Episode: 3.8 -- rating: 9.0
Episode: 3.9 -- rating: 9.9
Episode: 3.10 -- rating: 9.2
Episode: 4.1 -- rating: 9.1
Episode: 4.2 -- rating: 9.7
Episode: 4.3 -- rating: 8.9
Episode: 4.4 -- rating: 8.8
Episode: 4.5 -- rating: 8.8
Episode: 4.6 -- rating: 9.7
Episode: 4.7 -- rating: 9.1
Episode: 4.8 -- rating: 9.7
Episode: 4.9 -- rating: 9.6
Episode: 4.10 -- rating: 9.7
Episode: 5.1 -- rating: 8.5
Episode: 5.2 -- rating: 8.5
Episode: 5.3 -- rating: 8.5
Episode: 5.4 -- rating: 8.7
Episode: 5.5 -- rating: 8.6
Episode: 5.6 -- rating: 8.0
Episode: 5.7 -- rating: 9.0
Episode: 5.8 -- rating: 9.9
Episode: 5.9 -- rating: 9.5
Episode: 5.10 -- rating: 9.1
Episode: 6.1 -- rating: 8.5
Episode: 6.2 -- rating: 9.4
Episode: 6.3 -- rating: 8.7
Episode: 6.4 -- rating: 9.1
Episode: 6.5 -- rating: 9.7
Episode: 6.6 -- rating: 8.4
Episode: 6.7 -- rating: 8.6
Episode: 6.8 -- rating: 8.4
Episode: 6.9 -- rating: 9.9
Episode: 6.10 -- rating: 9.9
Episode: 7.1 -- rating: 8.6
Episode: 7.2 -- rating: 8.9
Episode: 7.3 -- rating: 9.2
Episode: 7.4 -- rating: 9.8
Episode: 7.5 -- rating: 8.8
Episode: 7.6 -- rating: 9.0
Episode: 7.7 -- rating: 9.4


10
8
6
4
2
0
S1
S2
S3
S4
S5
S6
S7