ITEC 1150 Week 13
Chapter 12
WEB REQUESTS & SCRAPING

Fetching Data From the Internet

WE CAN DOWNLOAD ANYTHING OFF THE INTERNET, SAVE IT, PRINT IT, GET DATA OUT OF IT, AND SO ON.

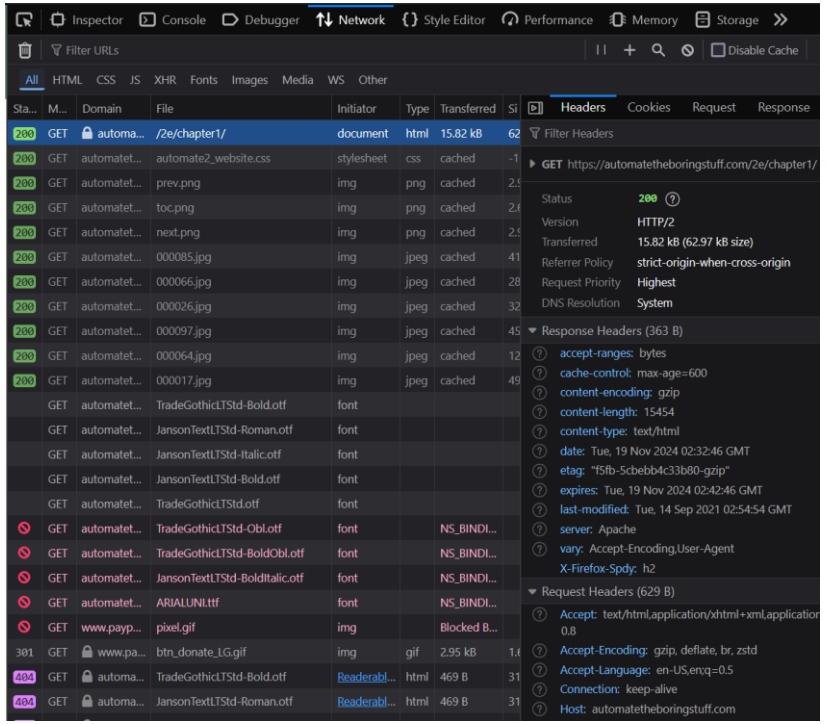
THIS IS A FOUNDATIONAL PART OF HOW MANY LARGE APPLICATIONS ARE BUILT TODAY.

Browser Developer Tools

- We can use our browser to see what is happening in the background when we load a web page. Open the developer tools for your browser:
  - Firefox: from the menu in the upper-right corner, go to More Tools → Web Developer Tools
  - Chrome & Edge: from the menu in the upper-right corner, go to More Tools → Developer Tools
  - For either browser, you can use the keyboard shortcut `ctrl + shift + I` (`cmd + opt + I` on Mac).

Browser Developer Tools (cont.)

- When you open the developer tools, it will take up part of the browser window, and contains a number of different tabs, which allow you to examine details about the web page, such as the structure, what storage it is using on your computer, what network requests the page has made, and so forth.
- Let's look at one of the network requests from loading automatetheboringstuff.com.



Anatomy Of A Web Request

- Unless a web page is old-fashioned and contains no images, multiple network requests are made to load all of the different elements that make up the page. Here's a little data from the developer tools about the first one request made when loading the Chapter 12 page of the book:

```
GET https://automatetheboringstuff.com/2e/chapter1/

Status: 200
```

- The URL is recognizable; `GET` is what we're telling the webserver we're doing. (There are other verbs that can be used, most of which are for submitting data, like a form. We're only concerned with `GET` in this course.)
- The status of `200` means the request was successful. Can you guess what the status would be if we went to a page that didn't exist?

The Requests Module

- Now that we know a little bit about web requests, let's look at how we can make them from Python.
- The **`requests`** module is a simple way to download files & data. Install it as we did with PyInputPlus (see chapter 8 if you need a refresher).
- Add this to your exercise file:

```
1. import requests # add requests to your import statement
2. response = requests.get('https://automatetheboringstuff.com/files/rj.txt')
```

- The method `requests.get()` fetches whatever is at the URL provided.
- What it returns (stored here in the `response` variable) contains the data from the web page (e.g., the HTML or whatever else the web server responds with), but also contains a lot of other information about the request, such as the status.

The Requests Module (cont.)

- Now add the following three lines to your exercise file:

```
1. print(type(response)) # returns the type of the response variable
2. print(response.status_code) # check out all the HTTP response status codes on python.org
3. print(response.status_code == requests.codes['OK']) # easy way to check if HTTP response is OK
```

- The status code is always returned by a web server. We've already talked about 200 and you probably know 404, which means "not found"; there are many others, grouped into categories:
  1. Informational responses (100 – 199)
  2. Successful responses (200 – 299)
  3. Redirection messages (300 – 399)
  4. Client error responses (400 – 499)
  5. Server error responses (500 – 599)

For a full list and details, see https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

- We will only be concerned with checking for errors, which is easy to do.

Checking For Errors

- It's a good idea to always wrap calls to `requests` in `try/except` blocks.
- On the previous slide, we checked the status of the response, but that's not sufficient for error handling—if the website is down or the domain is incorrect, an exception will be raised before anything gets put in the response!
- As an alternative to checking the response status, the following line will raise an exception for any error statuses:

```
1. response.raise_for_status()
```

- If you use this, you **must** use a `try/except` block around it.

The Requests Module – Getting Content

- A full sample of downloading a text file and printing some of the result:

```
1.  import requests # add requests to your import statement
2.  try:
3.      response = requests.get('https://automatetheboringstuff.com/files/rj.txt') # downloads file at URL
4.      print(type(response)) # returns the type of the response variable
5.      print(response.status_code) # check out all the HTTP response status codes on python.org
6.      print(response.status_code==requests.codes['OK']) # easy way to check if HTTP response is OK
7.      response.raise_for_status() # check for errors
8.      print(len(response.text)) # shows the total length of the text file
9.      print(response.text[0:250]) # print part of the text file [startindex:stopindex]
10. except:
11.     print('There was an error fetching the resource at https://automatetheboringstuff.com/files/rj.txt ')
```

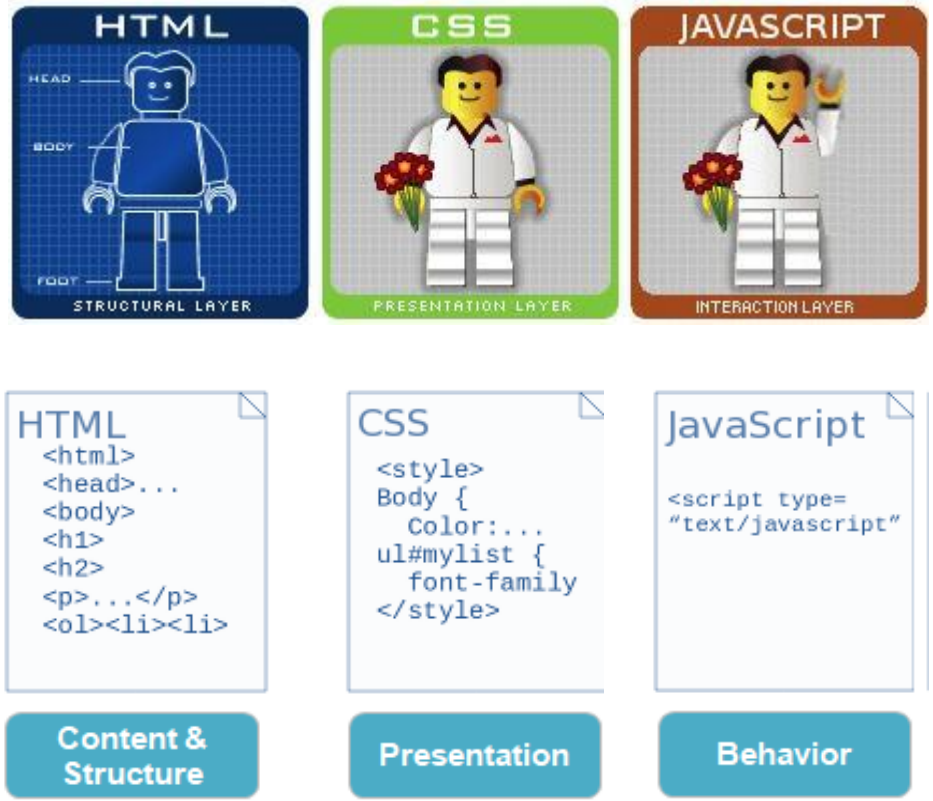
- We successfully downloaded a text file and printed a portion of the content.

Web Fundamentals

NOW THAT WE KNOW HOW TO FETCH A WEB RESOURCE, WHAT'S INSIDE OF IT AND WHAT CAN WE DO WITH IT?

The Types of Content That Make Up A Web Page

- HTML (Hyper-Text Markup Language) defines the *content* and *structure* of a web page
- CSS (Cascading Style Sheets) define how the structure and content are *presented* in the browser.
- JavaScript defines how a page *behaves*.
- We'll only have to worry about the HTML in this course, and fortunately, web servers will usually return the HTML first, if there is any.



Getting Data From Web Pages

- Getting data from text files is simple enough, but getting data out of an HTML page is more complicated.
- The image to the right shows just a bit of the HTML from a weather forecast site; trying to get data out of that is extremely difficult without using another module.
- It may look like a job for regexes, but it is **not**. See this famous top-voted answer to the subject of regex and HTML:

  https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

```
        </div>
        <!-- /current-conditions-body -->
    </div>
<!-- /Current Conditions -->
</div>

<!-- 7-Day Forecast -->
<div id="seven-day-forecast" class="panel panel-default">
    <div class="panel-heading">
    <b>Extended Forecast for</b>
    <h2 class="panel-title">
                Minneapolis MN    </h2>
    </div>
    <div class="panel-body" id="seven-day-forecast-body">
            <div id="seven-day-forecast-container"><ul id="seven-day-forecast-list" class="list-u
<div class="tombstone-container">
<p class="period-name">This<br>Afternoon</p>
<p><img src="newimages/medium/sct.png" alt="This Afternoon: Mostly sunny, with a high near 72. We
<div class="tombstone-container">
<p class="period-name">Tonight<br><br></p>
<p><img src="DualImage.php?i=hi_nshwrs&j=nbkn&ip=10" alt="Tonight: A 10 percent chance of showers
<div class="tombstone-container">
<p class="period-name">Thursday<br><br></p>
<p><img src="DualImage.php?i=shra&j=shra&ip=20&jp=50" alt="Thursday: A 50 percent chance of showe
<div class="tombstone-container">
<p class="period-name">Thursday<br>Night</p>
<p><img src="newimages/medium/hi_nshwrs30.png" alt="Thursday Night: A 30 percent chance of shower
<div class="tombstone-container">
<p class="period-name">Friday<br><br></p>
<p><img src="DualImage.php?i=few&j=wind_skc" alt="Friday: Sunny, with a high near 61. Breezy, wit
<div class="tombstone-container">
<p class="period-name">Friday<br>Night</p>
<p><img src="newimages/medium/nfew.png" alt="Friday Night: Mostly clear, with a low around 40. No
<div class="tombstone-container">
<p class="period-name">Saturday<br><br></p>
<p><img src="newimages/medium/few.png" alt="Saturday: Sunny, with a high near 74. Southeast wind
<div class="tombstone-container">
<p class="period-name">Saturday<br>Night</p>
<p><img src="newimages/medium/hi_nshwrs20.png" alt="Saturday Night: A 20 percent chance of shower
<div class="tombstone-container">
<p class="period-name">Sunday<br><br></p>
<p><img src="newimages/medium/few.png" alt="Sunday: Sunny, with a high near 74. North northwest w
<script type="text/javascript">
// equalize forecast heights
$(function () {
    var maxh = 0;
    $(".forecast-tombstone .short-desc").each(function () {
        var h = $(this).height();
        if (h > maxh) { maxh = h; }
    });
    $(".forecast-tombstone .short-desc").height(maxh);
});
</script>    </div>
</div>

<!-- Everything between 7-Day Forecast and Footer goes in this row -->
<div id="floatingDivs" class="row">
```

Beautiful Soup For HTML Parsing

- The program below uses a new module called BeautifulSoup to **parse** the HTML in the page so we can access it with Python functions instead of trying to parse it ourself.
- Install the module as we did with PyInputPlus (see chapter 8 if you need a refresher).
- **Important note: Install the module "beautifulsoup4"! The module "beautifulsoup" is older and will not work!**

```
1. import requests
2. from bs4 import BeautifulSoup as soup
3. response = requests.get('https://automatetheboringstuff.com/2e/chapter12/')
4. parser = soup(response.text, 'html.parser') # build an html parser for the text of the response
5. programs = parser.select('.programs') # What's .programs?
6. for program in programs:
7.     print('--------------')
8.     print(program.text)
```

Beautiful Soup And CSS Selectors

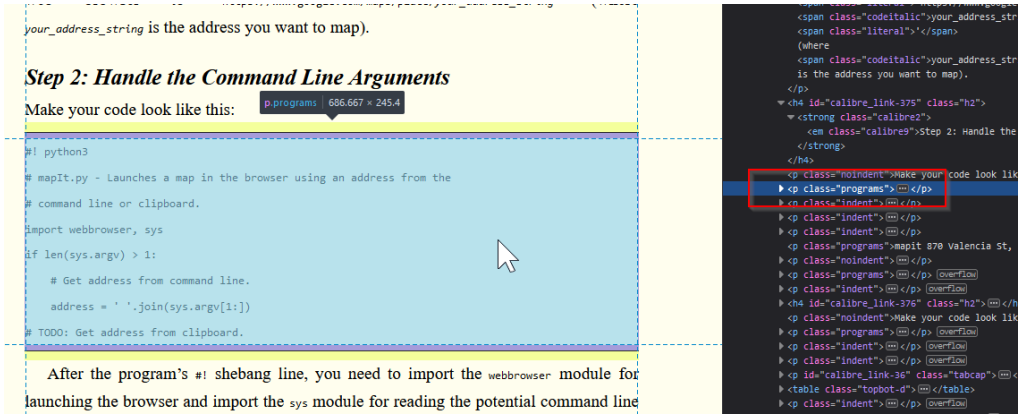
```
1. parser = soup(response.text, 'html.parser')
2. programs = parser.select('.programs')
3. for program in programs:
4.     print('--------------')
5.     print(program.text)
```

- The `select()` method of the parser can be used to find elements via what are called *CSS Selectors*, as shown on the right.
- It will return a list of elements that you can check for length and loop through like any other list.
- The `text` property of each element contains what we see in the browser; try just printing `program` instead of `program.text` to see the difference.

`soup.select('div')`	All elements named `<div>`
`soup.select('#author')`	The element with an `id` attribute of author
`soup.select('.notice')`	All elements that use a `CSS class` attribute named notice
`soup.select('div span')`	All elements named `<span>` that are within an element named `<div>`
`soup.select('div > span')`	All elements named `<span>` that are *directly* within an element named `<div>`, with no other element in between
`soup.select('input[name]')`	All elements named `<input>` that have a name attribute with any value
`soup.select('input[type="button"]')`	All elements named `<input>` that have an attribute named type with value button

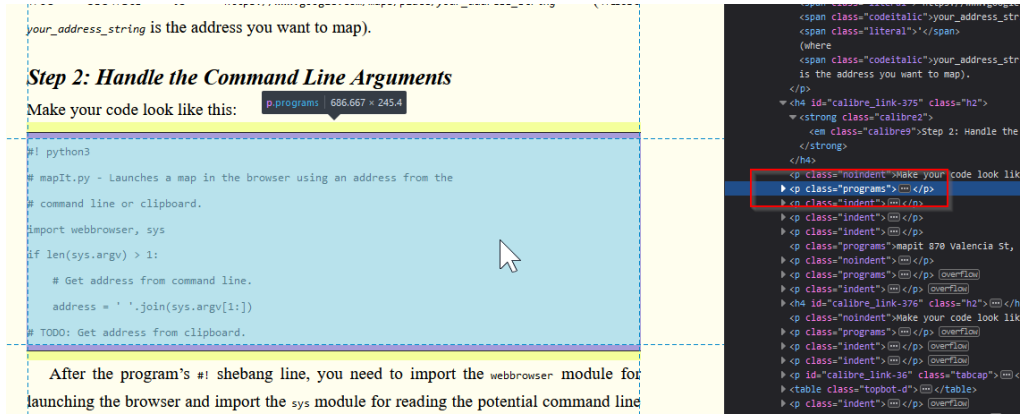
Finding The Content We Want In The Page

- In the program on the previous slide, the line `programs = parser.select('.programs')` selected the content we wanted, but how to know that the program listings were in something called `.programs`?
- The browser Developer Tools can help!
- The element selector is in the top left corner. Click on it, and then hover over different parts of a web page.



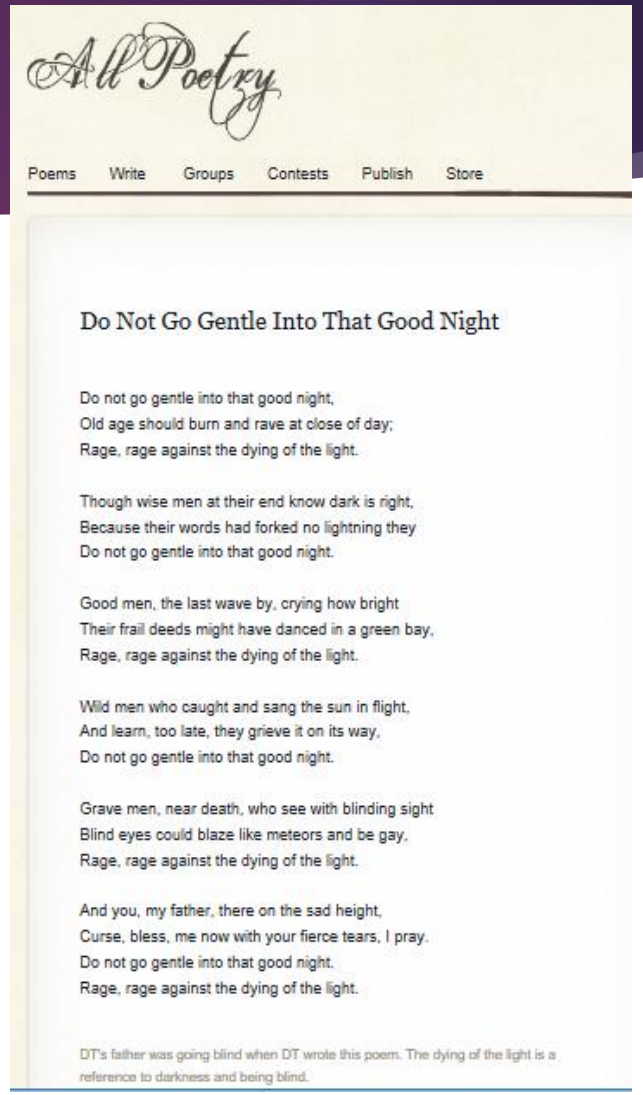
The "class" CSS Selector

- You'll see something like the screenshot on the right, where we've selected a paragraph (<p>) with a *class* named "programs".
- CSS Selectors are a complicated topic; you will only need to know how to select items by CSS Class for the lab.



Sample Program By Former ITEC Student

- Let's look at the D2L resource file demo12_web_scraping2.py
- Put it in your chapter 12 project folder, run it a few times, and enter different options
- Look at the results
- Find the file saved by the program (in your CWD)
- Then, compare to the actual web page shown at right



Next Week

- This week, we looked at how to fetch a web page and read content out of the web page.
- The content we looked at was from pages designed to be viewed in a browser; next week, we'll look at fetching data meant to be used in programs!