WEB SCRAPPING

CS 3080: Python Programming

UCCS University of Colorado Colorado Springs

Web Scrapping

- Web scraping is the term for using a program to download and process content from the Web.
- For example, Google runs many web scraping programs to index web pages for its search engine

Web Scrapping - Modules

- webbrowser
  - *Comes with Python and opens a browser to a specific page.*
- Requests
  - *Downloads files and web pages from the Internet.*
- Beautiful Soup
  - *Parses HTML, the format that web pages are written in.*
- Selenium
  - *Launches and controls a web browser. Selenium is able to fill in forms and simulate mouse clicks in this browser.*

WEBBROWSER MODULE

webbrowser module

- This is about the only thing the webbrowser module can do:

```
import webbrowser

webbrowser.open('http://www.uccs.edu/')
```

- Even so, the open() function does make some interesting things possible

webbrowser module – example

- Write a simple script to automatically launch the map in your browser using the contents of your clipboard.
- This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.
- A google map url looks like
  - *https://www.google.com/maps/search/870+Valencia+St+San+Francisco+CA/*
- So your program can be set to open a web browser to
  - *https://www.google.com/maps/search/your+address+string*
  - *(where your+address+string is the address you want to map).*

webbrowser module – example

Table 11-1: Getting a Map with and Without *mapIt.py*

Manually getting a map	**Using *mapIt.py***
Highlight the address.	Highlight the address.
Copy the address.	Copy the address.
Open the web browser.	Run *mapIt.py*.
Go to *http://maps.google.com/*.	
Click the address text field.	
Paste the address.	
Press ENTER.	

Ideas for similar programs

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather.
- Open several social network sites that you regularly check.

DOWNLOADING FILES FROM THE WEB WITH THE REQUESTS MODULE

Downloading Files from the Web with the requests Module

- The requests module lets you easily download files from the Web without having to worry about complicated issues such as network errors, connection problems, and data compression.
  - *pip install requests*
  - *import requests*

Downloading a Web Page with the requests.get() Function

```
import requests

res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
print(type(res))                    # <class 'requests.models.Response'>
print(res.status_code == requests.codes.ok) # True
print(len(res.text))                # 178981
print(res.text[:250])
# The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare
# This eBook is for the use of anyone anywhere at no cost and with
# almost no restrictions whatsoever. You may copy it, give it away or
# re-use it under the terms of the Proje
```

Checking for Errors

- A simpler way to check for success is to call the **raise_for_status()** method on the Response object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded

```
res = requests.get('http://page.that/does_not_exist’)
res.raise_for_status()
# raise HTTPError(http_error_msg, response=self)
# requests.exceptions.HTTPError: 404 Client Error: Not Found
for url: http://inventwithpython.com/page_that_does_not_exist
```

Checking for Errors

- A simpler way to check for success is to call the **raise_for_status()** method on the Response object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded

```
res = requests.get('http://page.that/does_not_exist’)
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

SAVING DOWNLOADED FILES TO THE HARD DRIVE

Saving Downloaded Files to the Hard Drive

- Even if the page is in plaintext (such as the Romeo and Juliet text you downloaded earlier), you need to write binary data to a file instead of text data in order to maintain the Unicode encoding of the text.
  - *write binary mode: pass the string 'wb' as the second argument to file.open().*
- We can iterate in chunks using the iter_content() of a Response object. Each chunk is of the bytes data type, and you get to specify how many bytes each chunk will contain.
  - *One hundred thousand bytes is generally a good size, so: iter_content(100000).*

Saving Downloaded Files to the Hard Drive

```
res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
res.raise_for_status()
playFile = open('RomeoAndJuliet.txt', 'wb')

for chunk in res.iter_content(100000):
    playFile.write(chunk)

playFile.close()
```

Saving Downloaded Files to the Hard Drive

- Complete process for downloading and saving a file:

1. Call requests.get() to download the content.
2. Call open() with 'wb' to create a new file in write binary mode.
3. Loop over the Response object’s iter_content() method.
4. Call write() on each iteration to write the content to the file.
5. Call close() to close the file.

For more info: http://requests.readthedocs.org/

HTML

HTML

- Hypertext Markup Language (HTML)
- Is the format that web pages are written in.
- HTML beginner tutorials:
  - *http://htmldog.com/guides/html/beginner/*
  - *http://www.codecademy.com/tracks/web/*
  - *https://developer.mozilla.org/en-US/learn/html/*

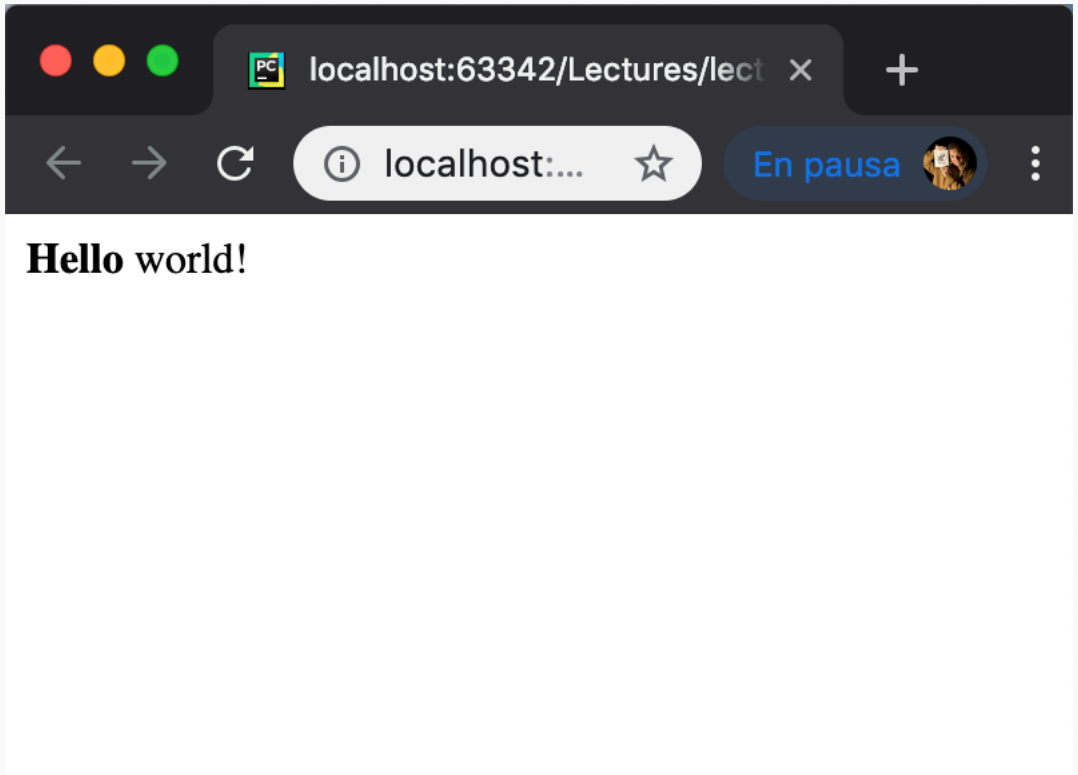
A Quick Refresher

- An HTML file is a plaintext file with the **.html** file extension.
- The text in these files is surrounded by **tags**, which are words enclosed in angle brackets.
- The tags tell the browser how to format the web page.
- A starting tag and closing tag can enclose some text to form an element.
- The **text** (or **inner HTML**) is the content between the starting and closing tags.

A Quick Refresher

helloworld.html

```
<strong>Hello</strong> world!
```

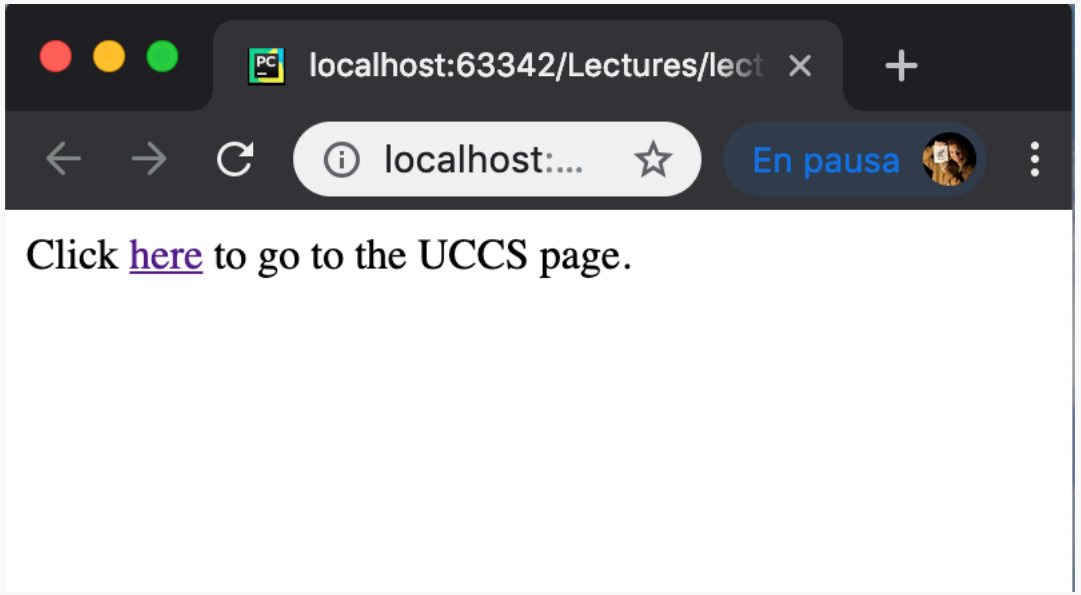


- The opening **** tag says that the enclosed text will appear in bold.
- The closing **** tags tells the browser where the end of the bold text is.

A Quick Refresher

ahref.html

```
Click <a href="http://www.uccs.edu">here</a>
to go to the UCCS page.
```



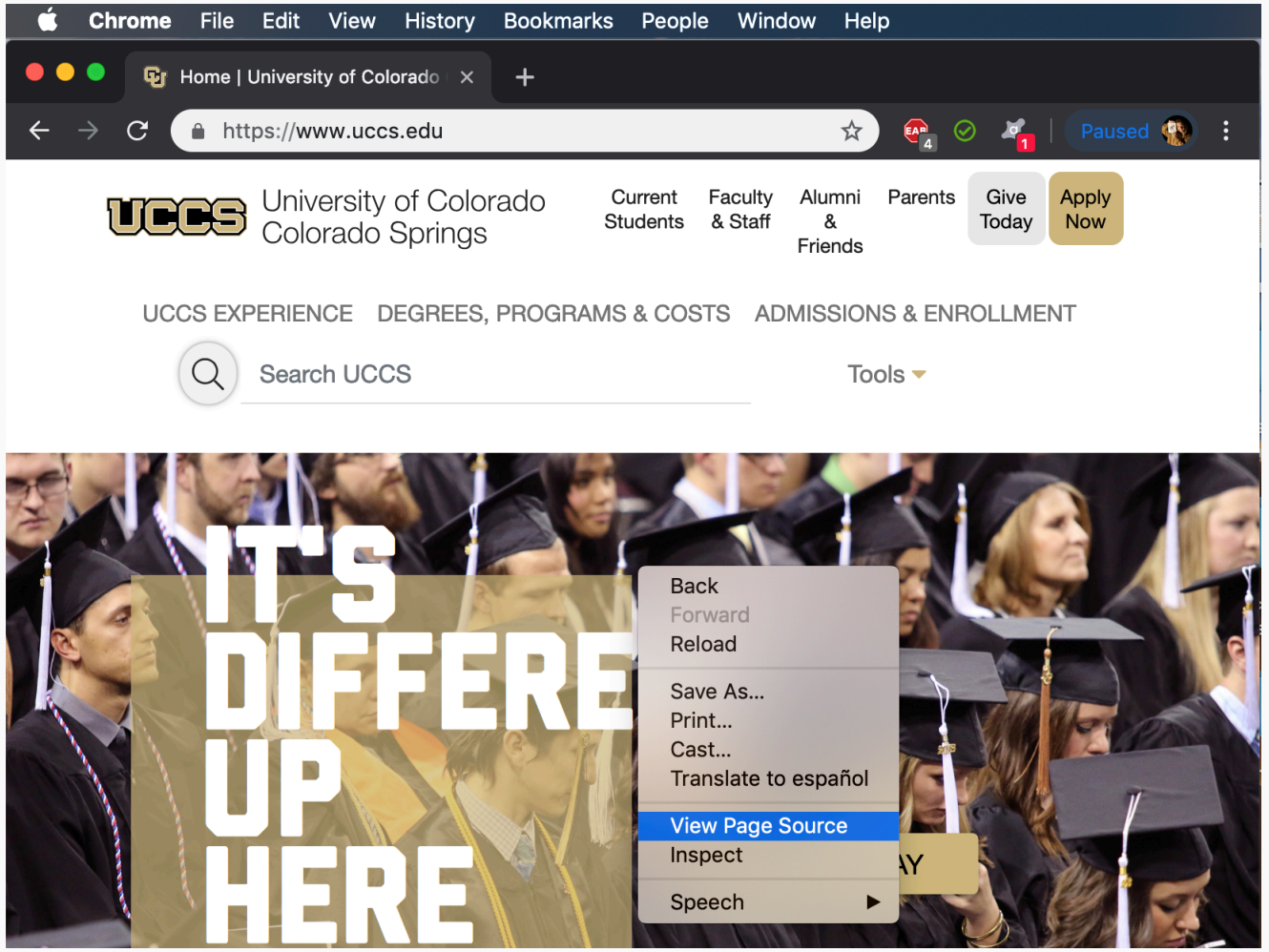
- There are many different tags in HTML.
- Some of these tags have extra properties in the form of attributes within the angle brackets.
- For example, the <a> tag encloses text that should be a link.
- The URL that the text links to is determined by the href attribute.

A Quick Refresher

- Some elements have an **id attribute** that is used to uniquely identify the element in the page.
- You will often instruct your programs to seek out an element by its id attribute,
- Figuring out an element's id attribute using the browser's developer tools is a common task in writing web scraping programs.

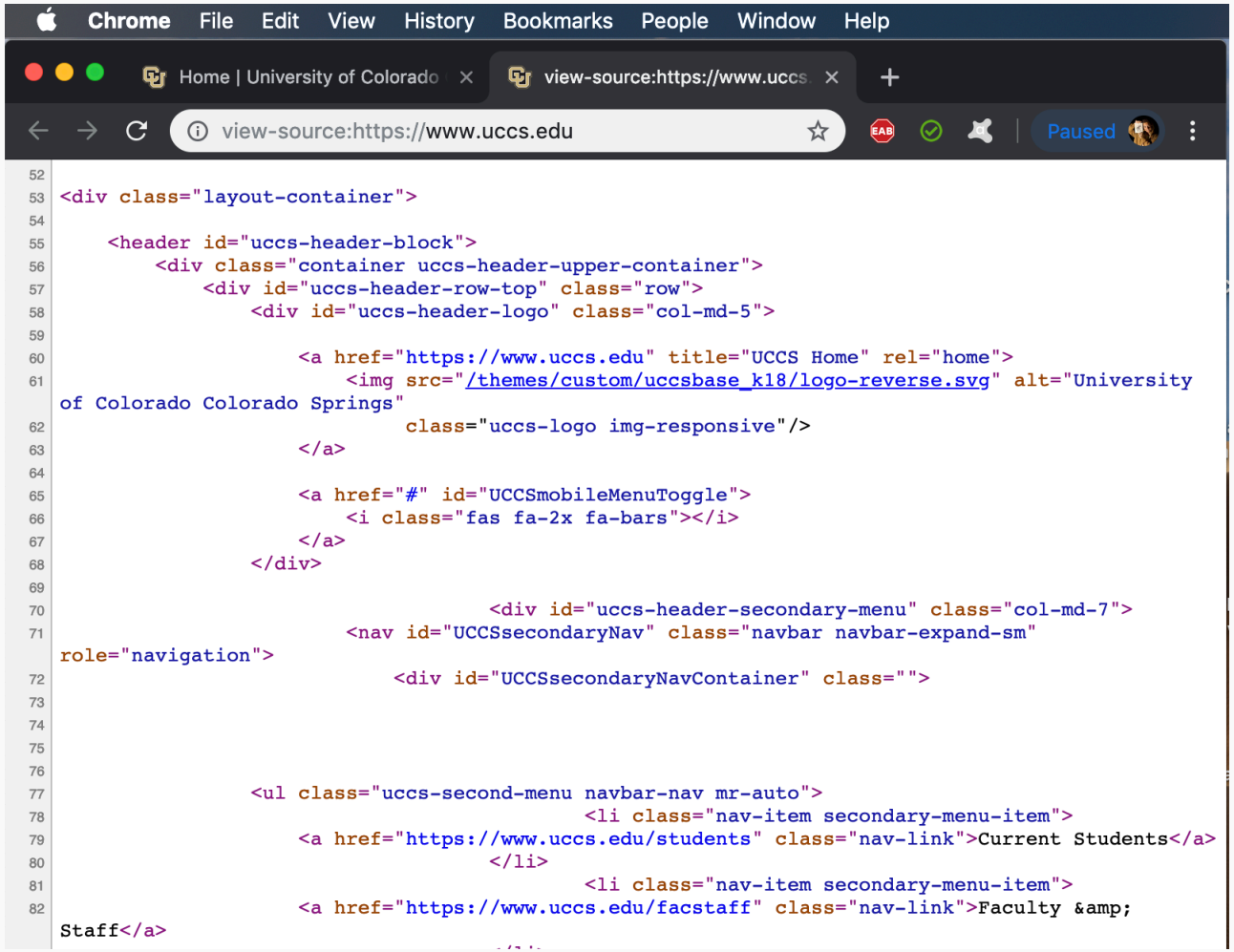
Viewing the Source HTML of a Web Page

- You'll need to look at the HTML source of the web pages that your programs will work with.

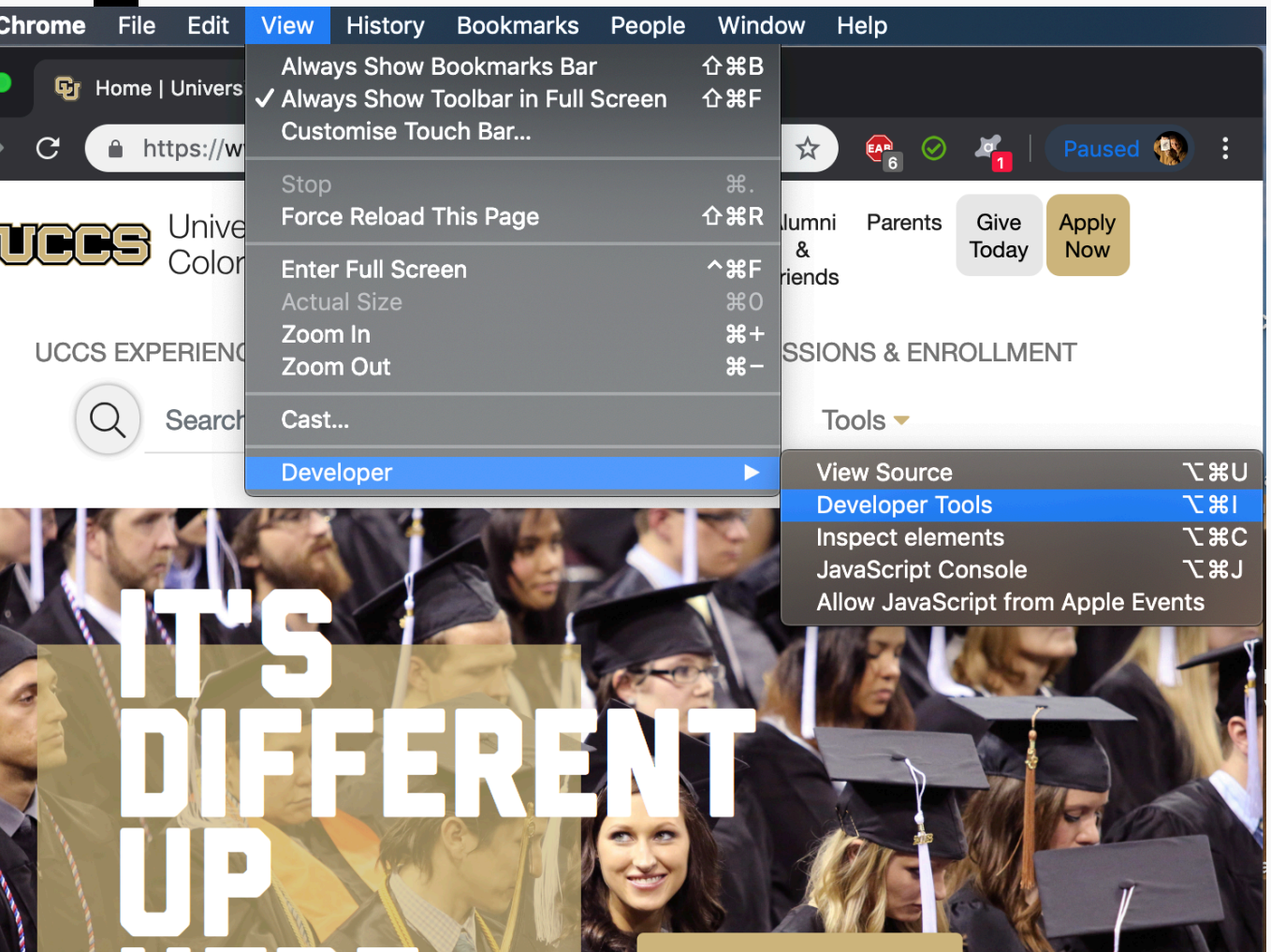


Viewing the Source HTML of a Web Page

- This is the text your browser actually receives.
- The browser knows how to display, or render, the web page from this HTML.
- It's fine if you don't fully understand what you are seeing when you look at the source.
- You won't need HTML mastery to write simple web scraping programs—after all, you won't be writing your own websites. You just need enough knowledge to pick out data from an existing site.



Opening Your Browser's Developer Tools



- Chrome
  - *View > Developer > Developer Tools on both.*
  - *command – option – I on OS X*
  - *Press F12 on Windows*
- Firefox
  - *ctrl-shift-C on Windows and Linux*
  - *command – option – C on OS X*
- Internet Explorer on Windows
  - Press F12
- Safari on OS X
  - Open the Preferences window
  - On the Advanced pane check the Show Develop menu in the menu bar option.
  - After it has been enabled, you can bring up the developer tools by pressing *command – option* – I

Browser Developer Tools

- After enabling or installing the developer tools in your browser, you can **right-click** any part of the web page and select **Inspect** or **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page.
- This will be helpful when you begin to parse HTML for your web scraping programs.

Don’t Use Regular Expressions to Parse HTML

- Locating a specific piece of HTML in a string seems like a perfect case for regular expressions.
- However, I advise you against it.
- There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations in a regular expression can be tedious and error prone.
- A module developed specifically for parsing HTML, such as Beautiful Soup, will be less likely to result in bugs.
- You can find an extended argument for why you shouldn’t to parse HTML with regular expressions at
  - *http://stackoverflow.com/a/1732454/1893164/*

Using the Developer Tools to Find HTML Elements

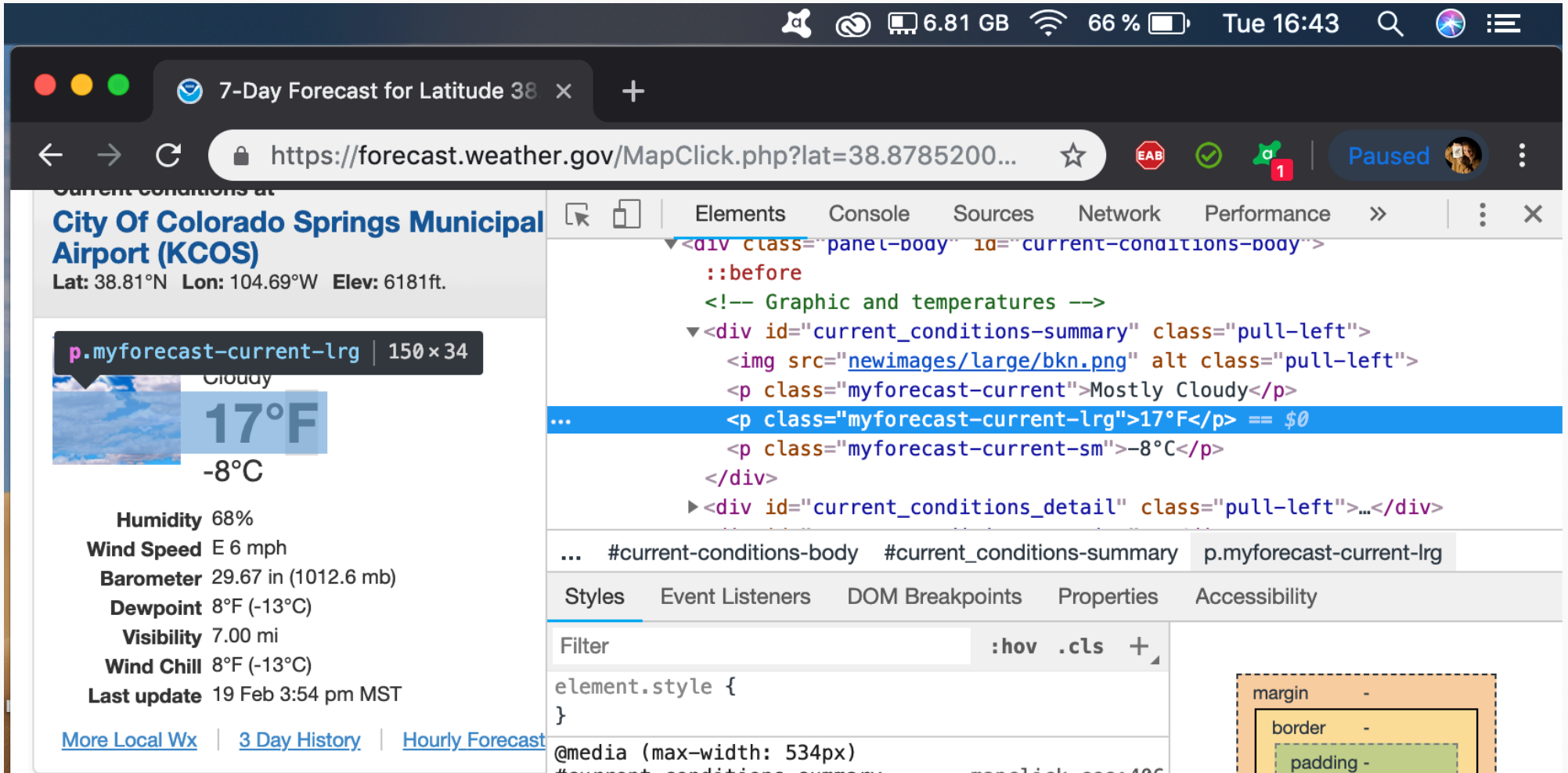
- Once your program has downloaded a web page using the requests module, you will have the page's HTML content as a single string value.
- Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.
  - *This is where the browser's developer tools can help*

Using the Developer Tools to Find HTML Elements

- Say you want to write a program to pull weather forecast data from http://weather.gov/
- You are interested in scraping the temperature information for the 80907 ZIP code.

- Steps:
  - *Go to http://weather.gov/*
  - *Search for 80907 ZIP code*
  - *Inspect the element of the current temperature*
  - *Look for the attribute id or class of that element*

Using the Developer Tools to Find HTML Elements



PARSING HTML WITH THE BEAUTIFULSOUP MODULE

Parsing HTML with the BeautifulSoup Module

- Beautiful Soup is a module for extracting information from an HTML page
- The BeautifulSoup module's name is bs4 (for Beautiful Soup, version 4).
- To install it, you will need to run:
  - *pip install beautifulsoup4*
- While beautifulsoup4 is the name used for installation, to import BeautifulSoup you run
  - *import bs4*

Parsing HTML with the BeautifulSoup Module

```
import bs4
import requests

res = requests.get('http://www.uccs.edu’)
res.raise_for_status()
uccsSoup = bs4.BeautifulSoup(res.text, features="html.parser")
print(type(uccsSoup))
# <class 'bs4.BeautifulSoup'>
```

Parsing HTML with the BeautifulSoup Module

example.html

```
<!-- This is the example.html example file. -->
<html><head><title>The Website Title</title></head>
<body><p>Download the <strong>Python</strong> slides from <a
href="https://canvas.uccs.edu/courses/69263/modules">Canvas</
a>.</p>
<p class="slogan">CS 3080: Python Programming</p>
<p>By <span id="author">Yanyan!!</span></p>
</body></html>
```

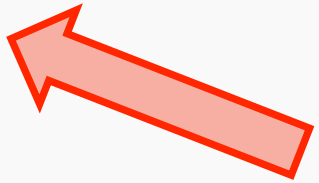
Html file saved in our hard drive!

Parsing HTML with the BeautifulSoup Module

```
import bs4
import requests
import os

os.chdir(os.path.join(".","lectures", "lecture12"))

exampleFile = open('example.html’)
exampleSoup = bs4.BeautifulSoup(exampleFile, features="html.parser")
print(type(exampleSoup))
# <class 'bs4.BeautifulSoup'>
```



- Once you have a BeautifulSoup object, you can use its methods to locate specific parts of an HTML document.

Finding an Element with the select() Method

- You can retrieve a web page element from a BeautifulSoup object by calling the **select()** method and passing a string of a CSS selector for the element you are looking for.

Selector passed to the `select()` method	Will match . . .
`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 an Element with the select() Method

- You can retrieve a web page element from a BeautifulSoup object by calling the **select()** method and passing a string of a CSS selector for the element you are looking for.
- The various selector patterns can be combined to make sophisticated matches. For example**, soup.select('p#author')** will match any element that has an id attribute of author, as long as it is also inside a <p> element.

Finding an Element with the select() Method

- You can retrieve a web page element from a BeautifulSoup object by calling the **select()** method and passing a string of a CSS selector for the element you are looking for.
- The various selector patterns can be combined to make sophisticated matches. For example**, soup.select('p#author')** will match any element that has an id attribute of author, as long as it is also inside a <p> element.
- The select() method will return a list of Tag objects, which is how Beautiful Soup represents an HTML element. The list will contain one Tag object for every match in the BeautifulSoup object's HTML.

Finding an Element with the select() Method

```
exampleFile = open('example.html’)
exampleSoup = bs4.BeautifulSoup(exampleFile, features="html.parser")
elems = exampleSoup.select('#author’)

print(type(elems))                  # <class 'bs4.element.ResultSet’>
print(len(elems))                   # 1
print(type(elems[0]))               # <class 'bs4.element.Tag’>
print(elems[0].getText())           # Yanyan!!
print(str(elems[0]))                # <span id="author">Yanyan!!</span>
print(elems[0].attrs)               # {'id': 'author'}
```

Finding an Element with the select() Method

```
pElems = exampleSoup.select('p’)
print(str(pElems[0]))      # <p>Download the <strong>Python</strong> slides
                           # from <a href="https://canvas...">Canvas</a>.</p>
print(pElems[0].getText())       # Download the Python slides from Canvas.
print(str(pElems[1]))            # <p class="slogan">CS 3080: Python...</p>
print(pElems[1].getText())       # CS 3080: Python Programming
print(str(pElems[2])).    # <p>By <span id="author">Yanyan!!</span></p>
print(pElems[2].getText())       # By Yanyan!!
```

Getting Data from an Element's Attributes

```
spanElem = exampleSoup.select('span')[0]
print(str(spanElem))
# <span id="author">Yanyan!!</span>
print(spanElem.get('id'))
# 'author'
print(spanElem.get('some_nonexistent_addr') == None)
# True
print(spanElem.attrs)
# {'id': 'author'}
```

CONTROLLING THE BROWSER WITH THE SELENIUM MODULE

Controlling the Browser with the selenium Module

- The selenium module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there is a human user interacting with the page.
- Selenium allows you to interact with web pages in a much more advanced way than Requests and Beautiful Soup; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the Web.

Starting a Selenium-Controlled Browser

  - *pip install selenium*
- To use Selenium you will need Firefox, Chrome, etc and probably you will get an error saying you have to install the driver.
- Sometimes the error itself says from where to download the driver. If not, look the error in google and download the driver accordingly.
- Then run this command in the drivers directory to set the driver in PATH:
  - *For firefox:*
    - mv geckodriver /usr/local/bin
  - *For Chrome:*
    - mv chromedriver /usr/local/bin
  - *General guide* *https://selenium-python.readthedocs.io/installation.html*

Starting a Selenium-Controlled Browser

```
from selenium import webdriver


browser = webdriver.Chrome()
print(type(browser))
# <class 'selenium.webdriver.chrome.webdriver.WebDriver’>
browser.get('http://www.uccs.edu')
```

Finding Elements on the Page

Method name	WebElement object/list returned
`browser.find_element_by_class_name(name)` `browser.find_elements_by_class_name(name)`	Elements that use the CSS class *name*
`browser.find_element_by_css_selector(selector)` `browser.find_elements_by_css_selector(selector)`	Elements that match the CSS *selector*
`browser.find_element_by_id(id)` `browser.find_elements_by_id(id)`	Elements with a matching *id* attribute value
`browser.find_element_by_link_text(text)` `browser.find_elements_by_link_text(text)`	<a> elements that completely match the *text* provided
`browser.find_element_by_partial_link_text(text)` `browser.find_elements_by_partial_link_text(text)`	<a> elements that contain the *text* provided
`browser.find_element_by_name(name)` `browser.find_elements_by_name(name)`	Elements with a matching *name* attribute value
`browser.find_element_by_tag_name(name)` `browser.find_elements_by_tag_name(name)`	Elements with a matching tag *name* (case insensitive; an <a> element is matched by `'a'` and `'A'`)

Finding Elements on the Page

- Once you have the WebElement object, you can find out more about it by reading the attributes or calling the methods in Table

Attribute or method	Description
`tag_name`	The tag name, such as `'a'` for an `<a>` element
`get_attribute(`*name*`)`	The value for the element's `name` attribute
`text`	The text within the element, such as `'hello'` in `<span>hello</span>`
`clear()`	For text field or text area elements, clears the text typed into it
`is_displayed()`	Returns `True` if the element is visible; otherwise returns `False`
`is_enabled()`	For input elements, returns `True` if the element is enabled; otherwise returns `False`
`is_selected()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise returns `False`
`location`	A dictionary with keys `'x'` and `'y'` for the position of the element in the page

Clicking the Page

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://www.uccs.edu’)
try:
    linkElem = browser.find_element_by_link_text('Apply Now’)
    linkElem.click()
except:
    print('Was not able to find an element with that text.')
```

Sending Special Keys

```
from selenium.webdriver.common.keys import Keys
```

Attributes	Meanings
Keys.DOWN, Keys.UP, Keys.LEFT, Keys.RIGHT	The keyboard arrow keys
Keys.ENTER, Keys.RETURN	The ENTER and RETURN keys
Keys.HOME, Keys.END, Keys.PAGE_DOWN, Keys.PAGE_UP	The HOME, END, PAGEDOWN, and PAGEUP keys
Keys.ESCAPE, Keys.BACK_SPACE, Keys.DELETE	The ESC, BACKSPACE, and DELETE keys
Keys.F1, Keys.F2, . . . , Keys.F12	The F1 to F12 keys at the top of the keyboard
Keys.TAB	The TAB key

Sending Special Keys

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://www.uccs.edu’)
try:
    linkElem = browser.find_element_by_link_text('Apply Now’)
    linkElem.click()
except:
    print('Was not able to find an element with that text.')
```

Instead of this, what can we do now?

Sending Special Keys

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
browser = webdriver.Firefox()
browser.get('http://www.uccs.edu’)
try:
    linkElem = browser.find_element_by_link_text('Apply Now’)
    linkElem.send_keys(Keys.ENTER)
except:
    print('Was not able to find an element with that text.')
```

Clicking Browser Buttons

- browser.back()
  - *Clicks the Back button.*
- browser.forward()
  - *Clicks the Forward button.*
- browser.refresh()
  - *Clicks the Refresh/Reload button.*
- browser.quit()
  - *Clicks the Close Window button.*

More information on Selenium

- Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript.
  - *http://selenium-python.readthedocs.org/.*