CIS 41B
Advanced Python Programming

The Web

De Anza College

Instructor: Clare Nguyen

Common Web Terminology

- The world wide web was built on 3 main ideas:
  1. HTTP (Hypertext Transfer Protocol): A specification for web clients and servers to communicate
  2. HTML (Hypertext Markup Language): A format to present data
  3. URL (Uniform Resource Locator): A way to uniquely identify a server and a resource on that server.
- A web client:
  - connects to a web server by using HTTP
  - requests a server's resource at a URL
  - receives data in HTML format
- Traditionally web clients are mostly web browsers that work with the HTML format to display the data for the users to view.
- In more recent times many web clients are software applications (called web bots, short for web robots) that are written to retrieve the server data for analysis purpose, but not to display the data.
- As a result, many servers now also provide an API (Application Programming Interface) for web bots to retrieve data.
- The data format of most APIs is JSON (Javascript Object Notation).

Client Server Communication

- The web is a client-server system:
  - Web clients access remote websites to send a *request* for the data.
  - Web servers send a *response* to the request that it receives.
- The clients, servers, and server APIs strive to follow the REST (Representational State Transfer) principle, which states that:
  - Clients and servers communicate through a uniform interface.
    For example, data will be transferred only in standard formats such as HTML or JSON.
  - All requests are stateless.
    Each request / response transaction is independent of the previous transactions. There is no referring back to a previous transaction.
  - Cache-able data are clearly marked.
    The client may cache data that are received in order to improve performance, but at the risk of having stale data. Therefore the server will clearly mark data that can be cached.
  - Servers can be distributed and layered independently of the interface.
    The client does not know or need to know which of the networked servers at a URL will provide the data that it requests.

Components of a Web Page

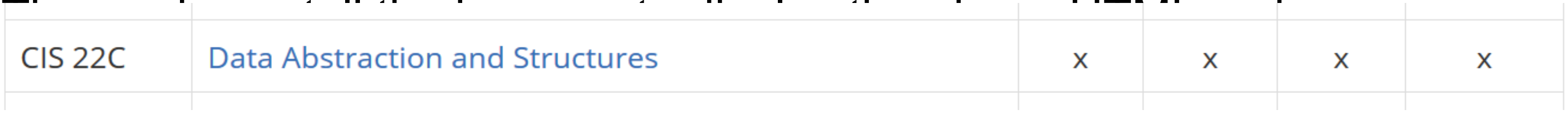
- When we use a browser to visit a web page, the browser makes a request to the server at a particular URL.
- The server responds by sending back files that tell the browser how to display the data.
- The files that the server sends can be in these formats:
  - HTML: the main content of the page
  - CSS: a way to style the page to make it look nicer
  - JS or Javascript: files to make the page interactive
  - Image format: to allow the browser to show pictures and videos
- A browser uses all the files that the server sends in order to produce the web page, but in our Python code we're mainly interested in the data that's in the HTML files.
- The work we do in our code is called web scraping. We extract data that are embedded among the formatting information that are used by the browser.
- The Python script that we write for this work is a web bot.

Web Scraping

- When we web scrape, we extract data from an HTML page.
- The HTML page is made of both data and formatting mark ups that tell the browser how to display the data.
- Here is a part of the HTML for the CIS Class Schedule web page for the CIS 22C class. The data that we want to extract is in green, and the rest are the mark ups:

```
<tr>
  <td><strong>CIS 22C</strong></td>
  <td><a title="View Course Description"
          href="http://ecms.deanza.edu/outlineprogresspublic.html?catalogID=12393">
          Data Abstraction and Structures</a></td>
  <td align="center" valign="middle"><strong>x</strong></td>
  <td align="center" valign="middle"><strong>x</strong></td>
  <td align="center" valign="middle"><strong>x</strong></td>
  <td align="center" valign="middle"><strong>x</strong></td>
</tr>
```

- [illegible]



CIS 22C	Data Abstraction and Structures	x	x	x	x

HTML Basics (1)

- HTML is not a programming language, it's a mark up language that tells the browser how to display data.
- HTML consists of tags that are the markers for the start and end of a block of data or an element that needs to be displayed a certain way.
- Tags are in the format: <tagName> for the start of the block
  </tagName> for the end of the block
- Here's how the basic tags form a simple HTML page:

```
<html>
   <head>
   </head>
   <body>
      <p>
          First paragraph of text
       </p>
       <p>
           Second paragraph
       </p>
   </body>
</html>
```

- The head element contains identification information such as the title, encoding type, links to the CSS and JS files, etc.
- The body element contains the data (the text) that we're interested in.
- The tags are *nested* and therefore are *indented*.
- Indented tags are child tags, and tags that contain child tags are parent tags.
  In this example, the <p> tags are child tags of <body>, and <html> is the parent tag of <body>.

HTML Basics (2)

- Some common tags in the <body> section of the HTML page:

HTML Tags	Functionality
<p> some text </p>	Display "some text" as a paragraph
<a href="URL"> link name </a>	Display a link to another page
<div> … </div>	Create a division or area of the page
<b> some text </b>	Display "some text" in bold
<i> some text </i>	Display "some text" in italics
<table> … </table>	Display a table
<li> some text </li>	Display "some text" as an item in a list

- Here is a complete list of HTML tags.
- Some tags also come with id and class fields, which are used by the CSS files to determine which specific style to apply to the elements.
- The id uniquely identifies an element, and the class is shared by a group of elements that are similar.
- Example:

```
<p class="announcement"  id="meeting1"> Weekly Meeting </p>
```

Download a Web Page: urllib

- There are 2 common ways to download a web page.
- The urllib package has a request module that is the ‘classic’ Python module for accessing HTTP web servers.
- Using the urllib module:

```
import urllib.request
page = urllib.request.urlopen(‘a_URL’)
```

- The returned object, page, is a Response object, it has the content of the HTML page.
- We can access the page:

```
page.read()
```

  which returns the entire content of the HTML page as a byte string.
- A byte string is a sequence of bytes, it’s the raw data form that’s stored on disk. When we call: print(page.read()) the bytes are encoded in ‘utf-8’ format and printed as text characters.
- For file header information: `page.getheaders()` or `page.info()`
  displays the type of data, encoding format, server type, domain name, etc. Here is the complete description of HTTP header fields.
- In case the open request fails, we can use try except to handle the urllib.error.HTTPError exception.

Download a Web Page: Requests (1)

- Unlike urllib, which is a Python package, Requests is a third-party package. Requests is a higher level interface to the web servers than urllib, therefore it is simpler to use than urllib.
- Requests is already in the Anaconda package. To use the Requests module:

```
import requests
page = requests.get('a_URL')
```

The returned object, page, is a Response object.

- We can access data in the Response object:

`page.text`	return the content of the HTML page as a Python string
`page.content`	return the content of the HTML page as a byte string
`page.json()`	return the content of the HTML page in JSON format

- To get the header information: `page.headers['header_field']`
- The 'Content-Type' field of the HTTP header can be important to check before we work with a web page. It shows whether the format of the web page is HTML or JSON or some other type data, which can help us read the downloaded data correctly.

Download a Web Page: Requests (2)

- In addition to the different data formats, web pages use different text encoding scheme due to the different text that they display.
- The most common format for text encoding is UTF-8, which can handle both plain text (ascii) characters and Unicode characters.
- Requests generally does a very good job of reading the web page's <head> information and doing a "best guess" of the text encoding of a page so that it can use the same encoding to display the page text.
- We can query Requests for the encoding that it uses:

  ```
  page.encoding
  ```

  We can also set the encoding if we know what it is:

  ```
  page.encoding = 'utf8'
  ```

- Here is an interesting discussion of how we started with ASCII and arrived at Unicode UTF-8, and here is a detailed discussion of text encoding in Python, which explains in detail how UTF-8 works in Python.

Downloading Constraints

- There can be a few issues when we have a large web page to download.
- It is not possible to store all the content of the web page in a string, and even if it just fits in a string, the string will be long and difficult to work with.
- Requests has an iter_content generator that downloads a portion of the web page at a time. We can use this generator to download the data and store it in a file, which is more easily accessible than a string.

```
with open(filename, 'wb') as outfile :      # wb mode is write byte
   for block in page.iter_content(chunk_size=128):    # or 256, 1024, 2048
       outfile.write(block)                              # write blocks of binary data to file
```

  The downloaded data are bytes of data, so the file open is binary write mode.

- Sometime when the file is large and the network connection is slow, we need to put a timer on the request so that the code is not blocked indefinitely.

```
requests.get(‘a_url', timeout=2)
```

  The timer is in seconds. It is the max number of seconds that elapse since the last server response, it is not the total download time.
  When the timer times out, Requests raises a Timeout exception.
- Last but not least, if the download time is long and there are multiple downloads, then multi-threading or multiprocessing is a good solution.

Requests Exceptions

- Requests get can raise several exceptions while downloading the web page if we ask it to raise the exception by calling the raise_for_status method.
- Here is an example of some common exceptions:

```
try:
    page = requests.get(url, timeout=3)
    page.raise_for_status()    # ask Requests to raise any exception it finds
    # do work, status is no error if we get here

except requests.exceptions.HTTPError as e:
     print ("HTTP Error:", e)
except requests.exceptions.ConnectionError as e:
     print ("Error Connecting:", e)
except requests.exceptions.Timeout as e:
     print ("Timeout Error:", e)
except requests.exceptions.RequestException as e:     # any Requests error
    print (“Request exception: ", e)
```

- In addition, we can ask for the status of the request: where 200 means ‘good’, 404 means ‘not found’, etc.

```
page.status_code
```

- The list of HTTP status code and their meaning can be found here.

Parsing Data (1 of 2)

- Because the data in an HTML page is embedded among the HTML tags, and the tags don’t follow strict syntax, it can be difficult to use basic Python techniques (string methods, regex, etc.) to parse data out of an HTML page.
- Fortunately there is a flexible HTML parser called BeautifulSoup.
- BeautifulSoup is already in the Anaconda package and can be imported into our code:

```
from  bs4  import BeautifulSoup
soup = BeautifulSoup(page.content, "lxml")
```

- BeautifulSoup accepts a Response object byte string, either from urllib or from Requests, and a parser choice. “lxml” is the recommended parser for its speed and flexibility with HTML / XML standards.
- BeaufifulSoup returns a BeautifulSoup object, which has methods that can help us get data from the Response object byte string.
- To print a portion of the page

```
soup.prettify()[0:n]      # from byte 0 to n-1
```

  prettify converts the bytes in the Response object into a list of characters, which we can index to select the portion we want.

Parsing Data (2 of 2)

- To print the data only (without HTML tags):

```
soup.get_text()    or    soup.text
```

- If the text contains characters that can’t be read by our Python code, we can set the encoding:

```
soup.text.encode("utf8“, “ignore”).decode()
                         or
soup.text.encode(“ascii“, “ignore”).decode(“ascii”)
```

  – utf8 is used with most displays
    ascii is used with Python print()
  – The text result is a Python string, which gets encoded to a UTF8 format byte string while ignoring any non-UTF8 character. Then the byte string is decoded back to a Python string.

Parse Data: Find Tags

- An HTML element is a block of text starting with <tag_name> and ending with a matching </tag_name>
- To find the first element with a particular tag:

```
soup.tag_name
```

or

```
soup.find( ‘tag_name’)
```

- To find all elements with a particular tag:

```
for  elem  in soup.find_all(‘tag_name’) :
        print(elem.name)        # print tag names only
```

- find_all can check for a list of tags:

```
for  elem in soup.find_all( [‘tag_name1', ‘tag_name2‘, ‘tag_name3’] ) :
        print(elem)        # print entire element for any of the tags in the list
```

- We can also use regular expression with find_all:

```
for  elem in soup.find_all( re.compile(‘b’) ) :     # any tag name with ‘b’ in it
        print(elem.get_text())          # print only the text part of the tag
```

- find_all is a generator that yields BeautifulSoup objects, the same data type as the object soup.

Parse Data: Find Text and Links

- Sometimes we want to search for data and not a tag.
- To find an exact data text string:

```
soup.find_all( string='search_string' )
```

- To find strings that match a regular expression:

```
soup.find_all( string = re.compile('b', re.I) )      # any text with 'b' in it,
                                                     # case insensitive
```

- To find strings that match a regex and are in a particular tag:

```
soup.find_all( 'a', string = re.compile('b', re.I), )     # any text in tag 'a'
                                                          # where the text contains 'b', case insensitive
```

- To find all links, which is the 'href' field of tag 'a':

```
for link in soup.find_all('a') :          # find all tags 'a',
    print( link['href'] )                 #  print 'href' field of tag 'a'
```

- To access a field within a tag, use the same format as with the 'href' field above:

```
print( tag_name['field_name'] )
```

Parse Data: Using Identifiers

- Since there can be many tags that are the same type, to narrow our search, we can choose tags with a particular CSS class or with a particular ID.
- To find tags with a particular class:

```
soup.find_all( ‘tag_name’, class_= ‘class_name’ )
```

- To find a tag with a particular ID:

```
soup.find_all( id = ‘the_id’ )
```

  This is the most precise search because an ID uniquely identifies a tag, so only one tag will match.

Parse Data: Find CSS Fields

- We can also take advantage of CSS selectors, which is a format that the CSS files used to select HTML elements.
- A CSS selector can be a sequence of tags or a tag with a CSS field:

CSS Selector	What it will select
body p a	'a' tags that are in 'p' tags that are in 'body' tag
p.outer	'p' tags with class = 'outer'
.up .left	Tags with class = 'left' that are in elements where class = 'up'
p#first_elem	'p' tag where id is 'first_elem'

- Note that class names are preceded by **.**
  And ID names are preceded by #
- If a class name contains a space in between words, use a **.** for each space
- To use CSS selector to search: `soup.select( 'p  a')`
  select returns a list of BeautifulSoup objects, just like find_all and find
- To select the first instance of the list only, use select_one()

The Web API

- Instead of writing code to web scrape, if a website provides an API, then it is the recommended way to get to the data easily and always legally.
- A web API is the way that a company or organization shares their data with web clients.
- An API organizes and packs data that are on the server so that the data can be easily downloaded and read by the web clients.
- A web API is useful when data changes quickly, such as with stock prices. We may not have the bandwidth to constantly download data and calculate the result in order to keep up with the latest changes.
- An API is also useful when a large data set needs to be downloaded, such as map data. We don't necessarily want to download and search through a lot of data just to get to one data point, such as a location on the map.
- A web API has a particular URL and several endpoints.
  - The URL lets us get to the API in the same way that we get to a particular website.
  - The endpoint has a particular set of data that we can retrieve from the API.

Using the Web API

- To access a web API, we use Requests with the API's URL:

```
page = requests.get('a_URL')
```

- The downloaded data will most likely be in JSON format, and Requests can convert JSON data into a Python dictionary.

```
resultDict = page.json()
```

- In the resulting dictionary:
  - Each key is a field name and the data is the value of the field.
  - There are no tags to deal with because the API has removed them and has packed the data into a more readable JSON format.

  This makes the Web API an easier way to get data from a website.
- Some organizations will ask users to register and get a key, which is used to access the API.
  Some organizations will ask users to pay to access to their API.
  Many other APIs are free and don't require registration.
- Here are two sample collections of APIs on github and medium

Web Scraping Policy (1)

- Web scraping means going to a website and downloading data that belong to that website.
- Web scraping also involves web crawling, which is the act of following links on a web page to get to other pages of the website. It doesn't take much to find all the 'a' tags' href fields, and then go to each of those links to download more data at those web pages.
- When we write a web scraping application, we create a bot that goes to a website through the back door. We're not visiting the website and viewing the content as the site intends for us to do.
- Therefore it is understandable that companies and organizations want to limit the type of data and who can download the data from their website.
- Most websites have a robots.txt file at their top level URL or their front page. This file specifies what bot or User-agent can access the website. It also has a list of which directories or links are allowed (bots can visit) and which are disallowed (bots cannot visit).
- A programmer who writes a well-behaved bot should always look up the robots.txt file and not write code that goes down a disallowed directory.

Web Scraping Policy (2)

- Even though the robots.txt policy should be followed, not surprisingly there are bad bots that will try to bypass them. Therefore most websites also have a firewall with more advanced techniques at identifying bad bots and blocking them.
- Here is an example discussion of a firewall for web applications.
  It gives reasons for necessary protection against bad bots and it discusses common protection for web services.
  The protections include verifying the IP address of the requestor, setting traps (honeypots) to catch bots but not humans, and setting delays in response time.

Web Scraping Recommendation

- It goes without saying that before we write a web scraper for a particular website, we should read the policy of the site's robot.txt and follow it when we code.
- In addition, our web scraper should access a website at a reasonable rate: 1 access every 5-10 seconds at most, or follow the access delay time if it's shown in the site's robot.txt file.
  If our bot attempts to access the website at a faster rate, even within a short period of time, it can be flagged as a Denial of Service attack (DDoS).
- Whether we write our own web scraper or use one of the web scraping tools, here are some points to consider. Can we can answer 'yes' to all of them?
  - Do I identify myself? (Do I not spoof an IP address?)
  - Do I document the downloading steps?
  - Is my work reproducible?
  - Do I protect and document my sources?
  - Do I give credit to my sources?
- Here is a discussion of the ethics of web scraping.

Going further…

- In this module we explored downloading data from HTML pages, but Requests can also download files, images, videos that are on the web pages.
- Web scrapers are controversial because many of them are misused, but they are an effective way to mine data or collect information from the web.
- There are many web scraping tools such as cURL, Selenium, Node.js, jQuery, etc. Most notably in Python is Scrapy, a tool that lets us create and customize a web scraper, all in Python.

Up next: Database