CIS192 Python Programming
HTTP & HTML & JSON

Harry Smith

University of Pennsylvania

November 1, 2017
Outline

1. HTTP Requests
   - HTTP
   - Requests

2. HTML Parsing
   - HTML
   - Beautiful Soup
What’s HTTP?

- The *protocol* that directs how exactly we send resources back and forth on the web.
- A protocol is a set of rules that determines which messages can be exchanged, and which messages are appropriate replies.
- Two roles: server and client
The Internet, TL;DR

- Network of computers that communicate via Internet protocol (IP)
- Internet service providers (ISP) direct traffic
- IP addresses are computers’ ‘mailing addresses’
- A Uniform Resource Locator (URL) refers to an IP address
- Domain Name System (DNS) resolves URLs to IP addresses
- HyperText Transfer Protocol (HTTP) is a way to talk via IP
Types of Requests

- **GET**: retrieve a representation of the specified resource
  - Should not modify the state of the server

- **HEAD**: a GET request but without the body (only the header)

- **POST**: Supply the resource with the content of the POST
  - The resource is an entity that can process data
  - The content of the POST is the data to be processed

- **PUT**: Store this data at the resource
  - Defines what the contents of the URI should be
  - A GET to the resource should return what was PUT

- **DELETE**: deletes the resource
Making a Request

- **First you must install** `requests`  
  ▶ `pip3 install requests`
- `r = requests.get(url)` **will make an HTTP GET request**  
  ▶ Returns a `Response` object
- `requests.{head|post|put|delete}(url)`
- `r.text` **is the body of the response**
- `r.headers` **is the header of the response** (Extra details, can usually ignore them)
HTTP Response Codes

- `r.status_code` is the HTTP status code of the response
- **1xx**: Informational. Not the actual response but not an error
- **2xx**: Everything is good
- **3xx**: Redirection. Need to make a new request
- **4xx**: Client Error: Didn’t ask right, not allowed, doesn’t exist
- **5xx**: Server Error: Might be your fault but probably not

Requests handles 1xx and 3xx for you. Can see in `r.history`

- `r.raise_for_status()` will raise an error for 4xx or 5xx
  - Prefer over:
    ```python
define 
if r.status_code ...:
    raise Exception
```
# HTTP Response Codes for Dummies

<table>
<thead>
<tr>
<th>Status Code Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20x</td>
<td>Cool.</td>
</tr>
<tr>
<td>30x</td>
<td>ask that dude over there.</td>
</tr>
<tr>
<td>40x</td>
<td>you f________ up.</td>
</tr>
<tr>
<td>50x</td>
<td>we f________ up.</td>
</tr>
</tbody>
</table>

@DanaDanger
Arguments to GET and POST

- Parameters to a GET request go in the URL’s query string
  - 'http://www.example.com/test?a1=v1&a2=v2'
  - GETs from the test page with a1=v1 and a2=v2
  - `requests.get('http://www.example.com/test', params=p)`
  - If `p = {'a1':v1, 'a2':v2}` the above are the same

- POST request data can be passed as a `dict`
  - `r = requests.post(url, data=d)`

- GET and POST also support a headers `dict` as a kwarg
APIs

- Application Programming Interface
  - Specifies how software components should interact
- On the web, lots of services/websites that provide data in a structured way to analyze!
  - Facebook
  - Google (Maps, Calendar, YouTube...)
  - Twitter
  - Yelp
  - Insert your favorite website/app here

- Usually represented in JSON
  - JavaScript Object Notation
  - A standard lightweight data format, language-agnostic
  - Use `json` module to parse JSON strings
Outline

1 HTTP Requests
   - HTTP
   - Requests

2 HTML Parsing
   - HTML
   - Beautiful Soup
HTML is a standardized way of specifying the contents of a page
It's composed of elements (<tags>) with contents and attributes
<tag attribute="val">content</tag>
Tags are supposed to specify semantics not style
  ▶ <p>A paragraph</p> Semantic grouping of page
  ▶ <b>bold</b> Style of text. Better to use <strong> or CSS
The tags form a tree with <html> at the root
This is the first paragraph
Sub paragraph
This is the second paragraph
This is the (strong) paragraph (p)

This is the (strong) paragraph

strong

Sub paragraph

first

second
Goals of Beautiful Soup

- Make searching through HTML easy (Beautiful)
  - Build the tree from the raw text
  - Provided methods for moving around the tree
  - Provide methods for finding sets of elements

- Handle poorly formatted HTML (Tag Soup)
  - Historically browsers have been lenient with HTML
  - Un-closed tags and badly nested tags are common
    - <html><p>first</p>second</html>
    - <strong><p></strong></p> ??
Using Beautiful Soup

- **Installation:** `pip3 install beautifulsoup4`
- **Importing:** `from bs4 import BeautifulSoup`
- **Create the tree from a string or file handle**
  - `soup = BeautifulSoup(r.text)`
  - `soup = BeautifulSoup(html_string)`
  - `soup = BeautifulSoup(open('html_file', 'r'))`
- `soup.<tag>` returns the first element with that tag
  - `soup.p` returns the first paragraph
  - If there are no `<tag>`s, returns `None`
- The object `soup.<tag>` returns has type: `bs4.element.Tag`
Tag Objects

- A tag represents `<tag attribute="val">content</tag>`
- `t.name` is the value within <> (tag in this case)
- `t['attribute']` looks up attribute in a dictionary
- `t[key] ⇐⇒ t.attrs[key]`
- `t.text` will give a string of all text in the subtree rooted at `t`
- `t.string` returns a NavigableString
  - Only if `t` has exactly one child and that child is a non-empty string
NavigableString Objects

- NavigableStrings support all operations of regular strings (`str`)
  ```python
  tag.string.split(',
  ```
- Additionally, it knows where it is in the tree.
- You can move to a parent or sister tag
- Details of moving around are basically the same as Tags
Moving Around

- `t.<tag>` gets the first matching element below `t` in the tree
- `t.children` is an iterator over an element's immediate children
- `t.descendants` is an iterator over all elements under `t`
  - Pre-order traversal
- `t.strings` is an iterator over all navigable strings under `t`
- `t.parent` is the parent of `t` in the tree
- `t.(next_/previous_)sibling` move to adjacent nodes
- `t.(next_/previous_)element` generalizes to the next node in the pre-order traversal
Searching the Tree

- Can search by matching with the following filters:
  - Literal strings
  - Compiled regular expressions
  - any string in a list
  - a function that returns True for tags you want
  - True matches everything

- `t.find_all(filter)` returns all descendants with names that match
- `t.find(...)` is like `t.find_all(...)` but only first match
- `kwargs` match attributes against filters
- `t.find(text=filter)` matches against the `.text` of a tag
- `t.find_(parents/next_siblings/all_next/previous)`
- To use Python keywords, append an `_`
  - `t.find(class_=filter)`
Review of JSON

- Many Web APIs transmit data in JSON
- JSON → JavaScript Object Notation

Data Types:
  - Numbers: 25, 167.6
  - Strings: "firstName"
  - Boolean: true, false
  - List: [25, "firstName", true]
  - Dictionary with String keys: {"fst": 1, "snd": 2}
  - Empty Value: null

Always wrap your JSON in a top-level dictionary
  - {"data": original_JSON}
  - JavaScript Bug allows top-level arrays to be hacked
JSON in Python

- The JSON standard library: `import json`
- `json.dumps(obj)` returns a JSON string of `obj`
- `json.dump(obj, f_handle)` writes the JSON to the file
- `json.loads(s)` returns a Python object from a JSON string
- `json.load(f_handle)` returns Python object from a file
- Flask has JSON: `from flask import jsonify, json`
  - use Flask’s `json.dumps()`/`loads()`
  - `return jsonify(d)` sends a JSON response from a `dict`
  - Takes care of details like headers and encoding
- `requests` has JSON
  - `r = request.get(...)`
  - `r.json()` parses out a Python object