CIS192 Python Programming

HTTP Requests and HTML Parsing

Robert Rand

University of Pennsylvania

March 30, 2016

Outline

The Internet in One Slide

- Network of computers that communicate via Internet protocol (IP)
- Internet service providers (ISP) direct traffic via Routing tables
- IP addresses say which computer(s) should receive a message
- 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

Outline

Making a Request

- First you must install requests `pip install requests`
- `r = requests.get(url)` will make an HTTP GET request
  - Returns a `Response` object
- `requests.{head|post|put|delete}(url)` as well
- `r.text` is the body of the response
  - requests attempts to decode the body for you
- `r.content` is the raw body of the response
  - Use this if requests guessed the wrong encoding
- `r.headers` is the header of the response
  - Lots of extra details that you can usually ignore

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:

```
if r.status_code ...:
    raise Exception
```

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:.../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

Outline

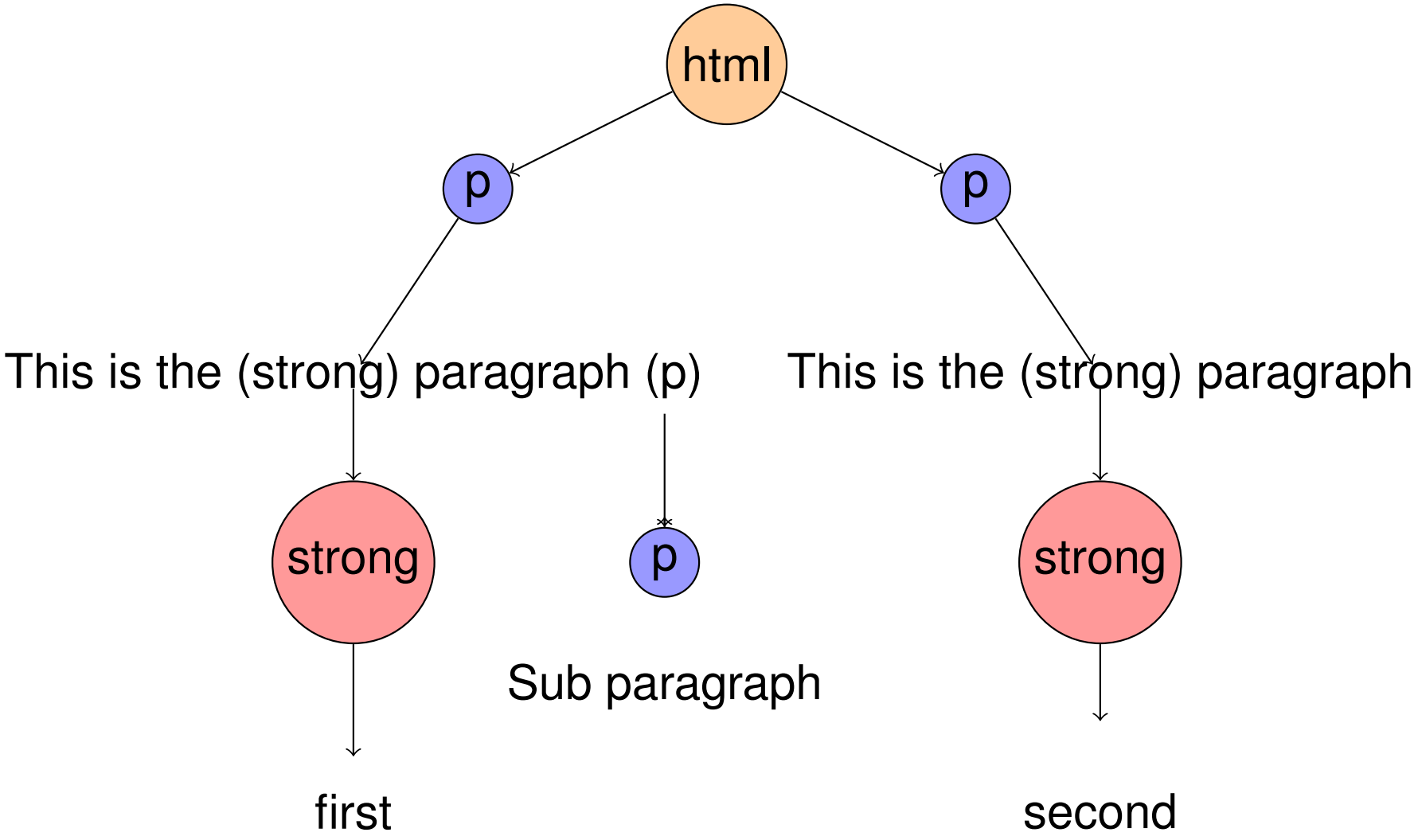
HyperText Markup Language

- 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

HTML Example

```
<html>
  <p>
    This is the <strong>first</strong> paragraph
    <p> Sub paragraph </p>
  </p>
  <p> This is the <strong>second</strong> paragraph
  </p>
</html>
```

HTML Example



Outline

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

- Install it – `pip install beautifulsoup4`
- import it `from bs4 import BeautifulSoup`
- Create the tree from a string or file handle
  - `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> 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]` $\iff$ `t.attrs[key]`
- `t.text` will give a string of all text in the subtree rooted at t
- `t.string` returns a NavigableString
  - If t has exactly one child `and`
  - That child is a non-empty string

NavigableString Objects

- NavigableStrings support all operations of regular strings (`str`)
  - `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 elements 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)`