Web Scraping

Unleash your Internet Viking

Andrew Collier

PyCon 2017

andrew@exegetic.biz
https://twitter.com/DataWookie
https://github.com/DataWookie

toptal

Scraping

toptal

What is Scraping?

- Retrieving *selected* information from web pages.
- Storing that information in a (un)structured format.

toptal

Why Scrape?

As opposed to using an API:

- web sites (generally) better maintained than APIs;
- many web sites don't expose an API; and
- APIs can have restrictions.

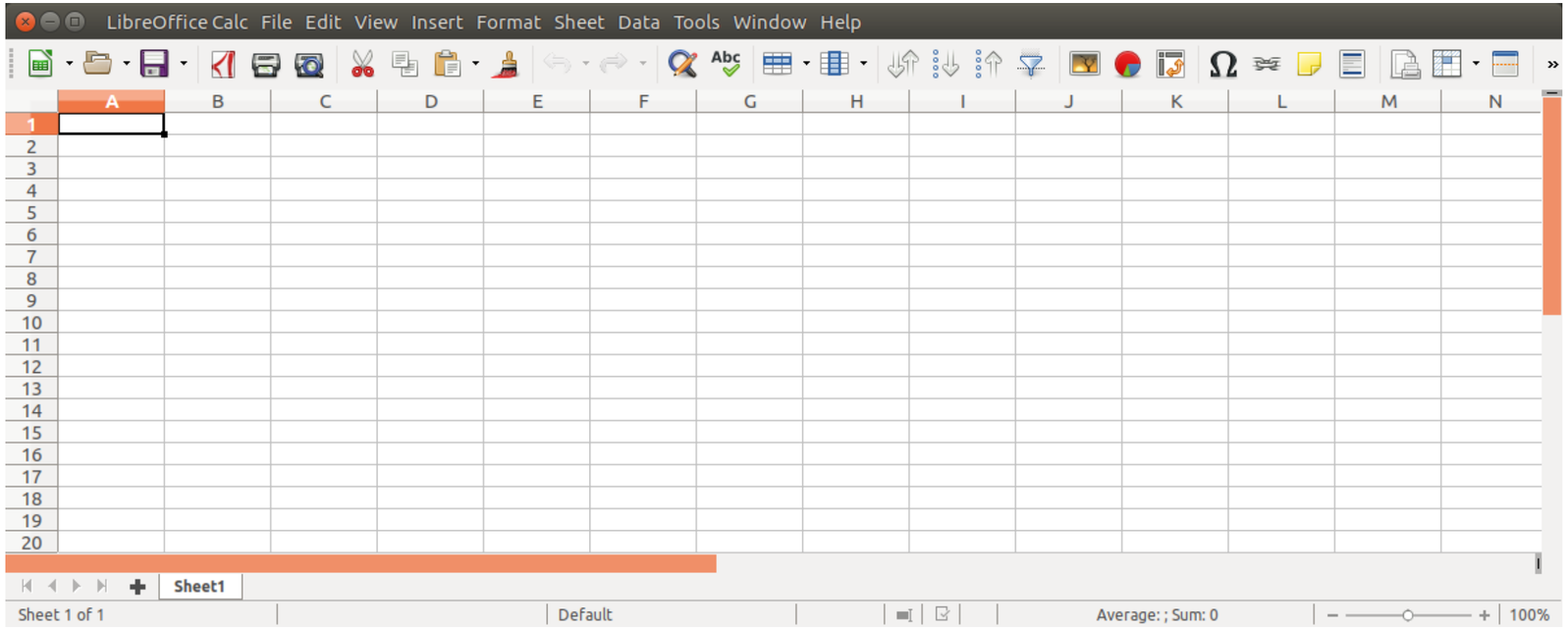
Other benefits:

- anonymity;
- little or no *explicit* rate limiting and
- **any content on a web page can be scraped**.

toptal

Manual Extraction

Let's be honest, you could just copy and paste into a spreadsheet.



As opposed to manual extraction, web scraping is...

- vastly more targeted
- less mundane and
- consequently less prone to errors.

toptal

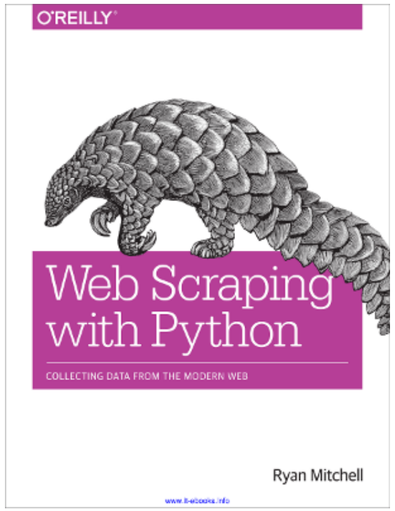
Crawling versus Scraping

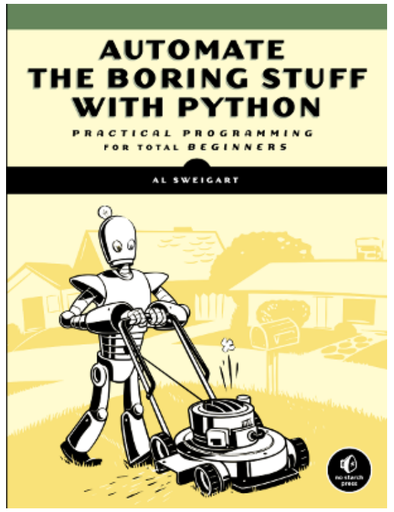
A web crawler (or "spider")

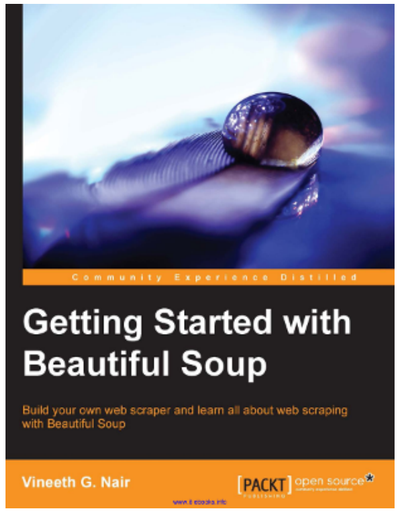
- systematically browses a series of pages and
- follows new URLs as it finds them.

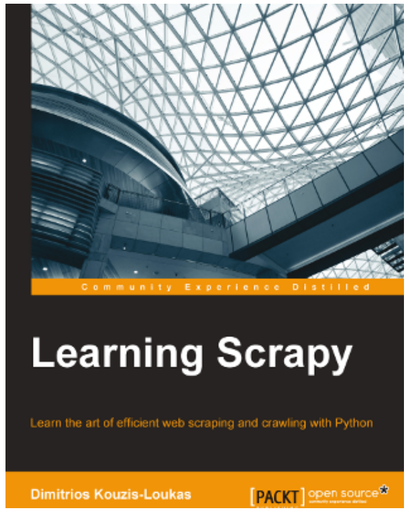
It essentially "discovers" the structure of a web site.

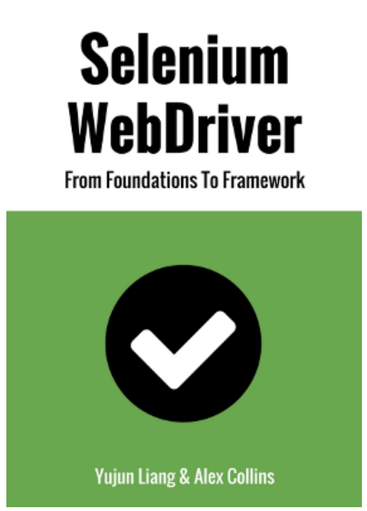
Resources

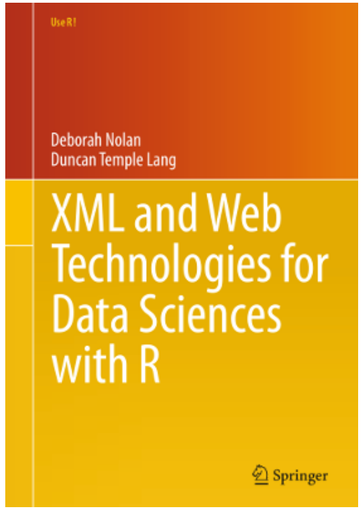


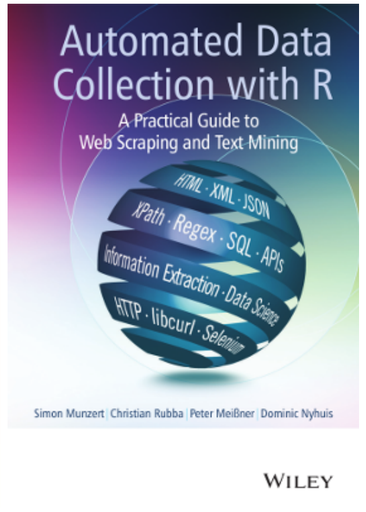












toptal

Anatomy of a Web Site: HTML

toptal

What is HTML?

HTML...

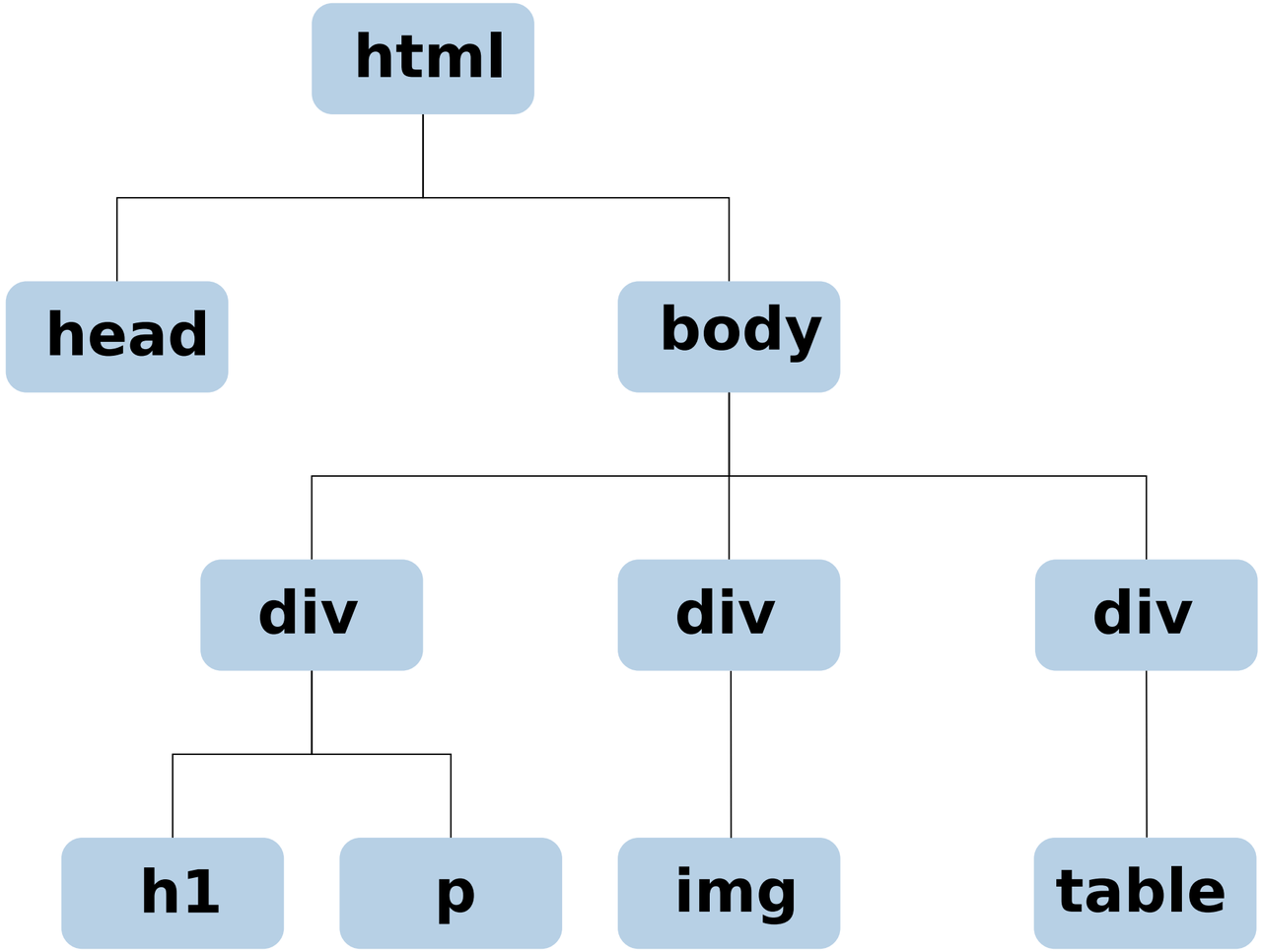
- stands for "Hyper Text Markup Language";
- is the standard markup language for creating web pages;
- describes the structure of web pages using tags.

toptal

A Sample HTML Document

```
<!DOCTYPE html> <!-- This is a HTML5 document. -->
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <h1>Main Heading</h1>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
        tempor incididunt ut labore et dolore magna aliqua.</p>
        <h2>First Section</h2>
        <p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi
        ut aliquip ex ea commodo consequat.</p>
        <h2>Second Section</h2>
        <p>Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
        dolore eu fugiat nulla pariatur.</p>
    </body>
</html>
```

Hypothetical Document Tree



toptal

HTML Tags

HTML tags are

- used to label pieces of content but
- are not visible in the rendered document.

Tags are enclosed in angle brackets and (almost always) come in pairs.

```
<tag>content</tag>
```

- `<tag>` - opening tag
- `</tag>` - closing tag

Tags define structure but not appearance.

HTML Tags - Document Structure

```
<html>
    <head>
        <!-- Meta-information goes here. -->
    </head>
    <body>
        <!-- Page content goes here. -->
    </body>
</html>
```

- `<html>` - the root element
- `<head>` - document meta-information
- `<body>` - document visible contents

toptal

HTML Tags - Headings

```
<h1>My Web Page</h1>
```

- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`

toptal

HTML Tags - Links

The *anchor* tag is what makes a WWW into a web, allowing pages to link to one another.

```
<a href="https://www.google.co.za/">Google</a>
```

- The tag content is the anchor text.
- The `href` attribute gives the link's destination.

HTML Tags - Lists

```
<ol>
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>
```

Lists come in two flavours:

- ordered, `<ol>`, and
- unordered, `<ul>`.

toptal

HTML Tags - Tables

```
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Bob</td>
    <td>50</td>
  </tr>
  <tr>
    <td>Alice</td>
    <td>23</td>
  </tr>
</table>
```

A table is

- enclosed in a `<table>` tag;
- broken into rows by `<tr>` tags;
- divided into cells by `<td>` and `<th>` tags.

HTML Tags - Images

```
<img src="http://via.placeholder.com/350x150" alt="Placeholder" width="350" height="150">
```

Mandatory attributes:

- `src` - link to image (path or URL).

Optional attributes:

- `alt` - text to be used when image can't be displayed;
- `width` - width of image;
- `height` - height of image.

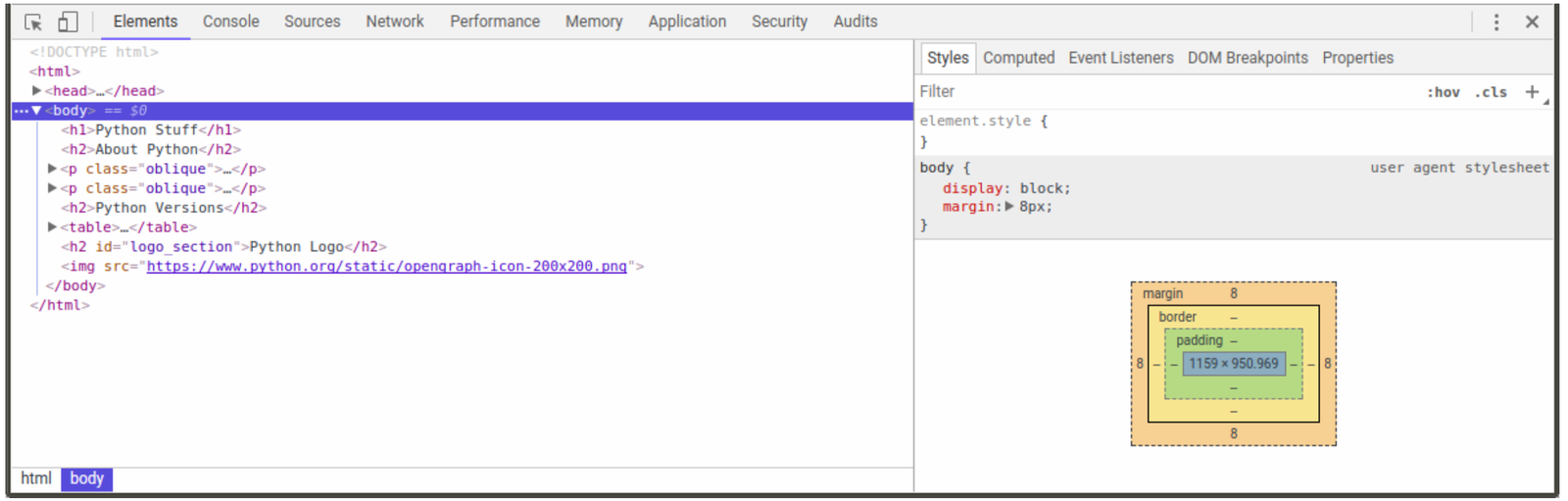
HTML Tags - Non-Semantic

The `<div>` and `<span>` tags give structure to a document without attaching semantic meaning to their contents.

- `<div>` - block
- `<span>` - inline

Developer Tools

Modern browsers have tools which allow you to interrogate most aspects of a web page.



To open the Developer Tools use Ctrl + Shift + I

A Real Page

Take a look at the page for Web scraping on Wikipedia.

To inspect the page structure, open up Developer Tools.

Things to observe:

- there's a lot going on in `<head>` (generally irrelevant to scraping though!);
- most of structure is defined by `<div>` tags;
- many of the tags have `id` and `class` attributes.

Exercise: A Simple Web Page

Create a simple web page with the following elements:

1. A `<title>`.
2. A `<h1>` heading.
3. Three `<h2>` section headings.
4. In the first section, create two paragraphs.
5. In the second section create a small table.
6. In the third section insert an image.

Anatomy of a Web Site: CSS

toptal

Adding Styles

Styles can be embedded in HTML or imported from a separate CSS file.

```
<head>
    <!-- Styles embedded in HTML. -->
    <style type="text/css">
    body {
        color:red;
    }
    </style>

    <!-- Styles in a separate CSS file. -->
    <link rel="stylesheet" href="styles.css">
</head>
```

toptal

CSS Rules

A CSS rule consists of

- a selector and
- a declaration block consisting of *property name: value;* pairs.

For the purposes of web scraping the selectors are paramount.

A lexicon of selectors can be found here.

Style by Tag

Styles can be applied by tag name.

```
/* Matches all <p> tags. */
p {
    margin-top:     10px;
    margin-bottom:  10px;
}

/* Matches all <h1> tags. */
h1 {
    font-style:     italic;
    font-weight:    bold;
}
```

toptal

Style by Class

Classes allow a greater level of flexibility.

```
/* Matches all tags with class "alert". */
.alert {
    color:          red;
}

/* Matches <p> tags with class "alert". */
p.alert {
    font-style:     italic;
}
```

```
<h1 class="alert">A Red Title</h1>

<p class="alert">A paragraph with alert. This will have italic font and be coloured red.</p>

<p>Just a normal paragraph.</p>
```

toptal

Style by Identifier

An identifier can be associated with *only one* tag.

```
#main_title {
    color:          blue;
}
```

```
<h1 id="main_title">Main Title</h1>
```

toptal

Combining Selectors: Groups

```
/* Matches both <ul> and <ol>. */
ul,
ol {
    font-style:     italic;
}

/* Matches both <h1> and <h2>, as well as <h3> with class 'info'. */
h1,
h2,
h3.info {
    color:          blue;
}
```

Combining Selectors: Children and Descendants

Descendant selectors:

```
/* Matches both
 *
 * <div class="alert"><p></p></div>
 *
 * and
 *
 * <div class="alert"><div><p></p></div></div>.
 */
.alert p {
}
```

Child selectors (indicated by a `>`):

```
/* Matches
 *
 * <div class="alert"><p></p></div>
 *
 * but it won't match
 *
 * <div class="alert"><div><p></p></div></div>.
 */
.alert > p {
}
```

toptal

Combining Selectors: Multiple Classes

```
/* Matches
 *
 * <p class="hot wet"></p>
 *
 * but it won't match
 *
 * <p class="hot"></p>.
 */
.hot.wet {
}
```

Learn more about these combinations here.

Pseudo Elements

These are (arguably) the most common:

- `:first-child`
- `:last-child`
- `:nth-child()`

```
/* Matches <p> that is first child of parent. */
p:first-child {
}
/* Matches <p> that is third child of parent. */
p:nth-child(3) {
}
```

These are particularly useful for extracting particular elements from a list.

Attributes

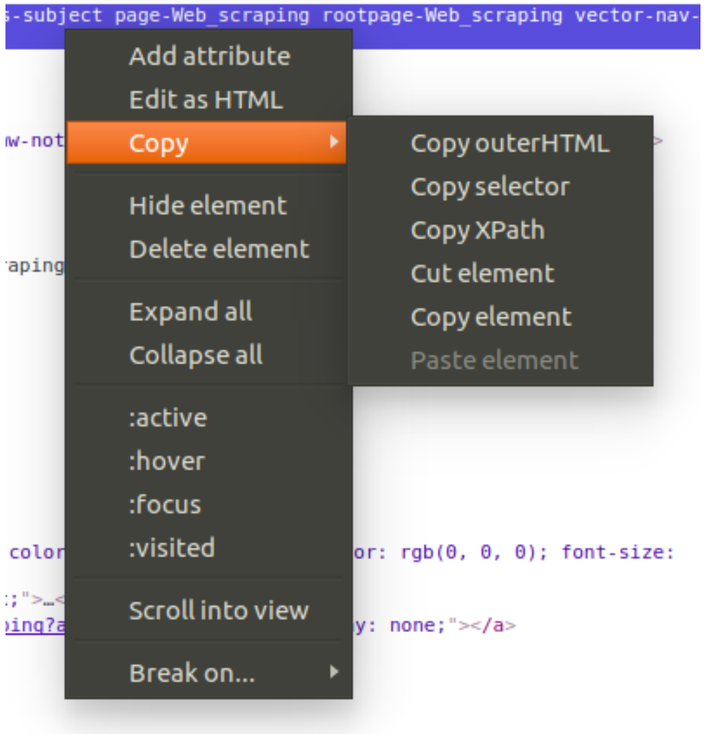
```
/* Matches <a> with a class attribute. */
a[class] {
}

/* Matches <a> which links to Google.
 *
 * There are other relational operators. For example:
 *
 * ^= - begins with
 * $= - ends with
 * *= - contains
 */
a[href="https://www.google.com/"] {
}
```

toptal

Selectors from Developer Tools

In Developer Tools right-click on any element.



SelectorGadget

SelectorGadget is a Chrome extension which helps generate CSS selectors.

- green: chosen element(s)
- yellow: matched by selector
- red: excluded from selector

toptal

Exercise: Style a Simple Web Page

Using the simple web page that we constructed before, do the following:

1. Make the `<h1>` heading blue using a tag name selector.
2. Format the contents of the `<p>` tags in italic using a class selector.
3. Transform the third `<h2>` tag to upper case using an identifier.

Anatomy of a Web Site: XPath

XPath is another way to select elements from a web page.

It's designed for XML but works for HTML too.

XPath can be used in both Developer Tools and SelectorGadget.

Whether you choose XPath or CSS selectors is a matter of taste.

CSS

```
#main > div.example > div > span > span:nth-child(2)
```

XPath

```
//*[@id="main"]/div[3]/div/span/span[2]
```

toptal

Anatomy of a Web Site: Files

toptal

robots.txt

The `robots.txt` file communicates which portions of a site can be crawled.

- It provides a hint to crawlers (which might have a positive or negative outcome!).
- It's advisory, not prescriptive. Relies on compliance.
- One `robots.txt` file per subdomain.

More information can be found here.

```
# All robots can visit all parts of the site.
User-agent: *
Disallow:

# No robot can visit any part of the site.
User-agent: *
Disallow: /

# Google bot should not access specific folders and files.
User-agent: googlebot
Disallow: /private/
Disallow: /login.php

# One or more sitemap.xml files.
#
Sitemap: https://www.example.com/sitemap.xml
```

sitemap.xml

The `sitemap.xml` file provides information on the layout of a web site.

- Normally located in root folder.
- Can provide a useful list of pages to crawl.
- Should be treated with caution since if not automated then often out of date.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://www.example.com/index.html</loc>
        <lastmod>2017-02-01</lastmod>
        <changefreq>monthly</changefreq>
        <priority>0.8</priority>
    </url>
    <url>
        <loc>http://www.example.com/contact.html</loc>
    </url>
</urlset>
```

Important tags:

- `<url>` - Parent tag for an URL (mandatory).
- `<loc>` - Absolute URL of a page (mandatory).
- `<lastmod>` - Date of last modification (optional).
- `<changefreq>` - Frequency with which content changes (optional).
- `<priority>` - Relative priority of page within site (optional).

urllib: Working with URLs

The `urllib` module has various utilities for dealing with URLs.

Sub-Modules

It's divided into three major sub-modules:

- `urllib.parse` - for parsing URLs
- `urllib.request` - opening and reading URLs
- `urllib.robotparser` - for parsing `robots.txt` files

There's also `urllib.error` for handling exceptions from `urllib.request`.

Notebook: urllib

toptal

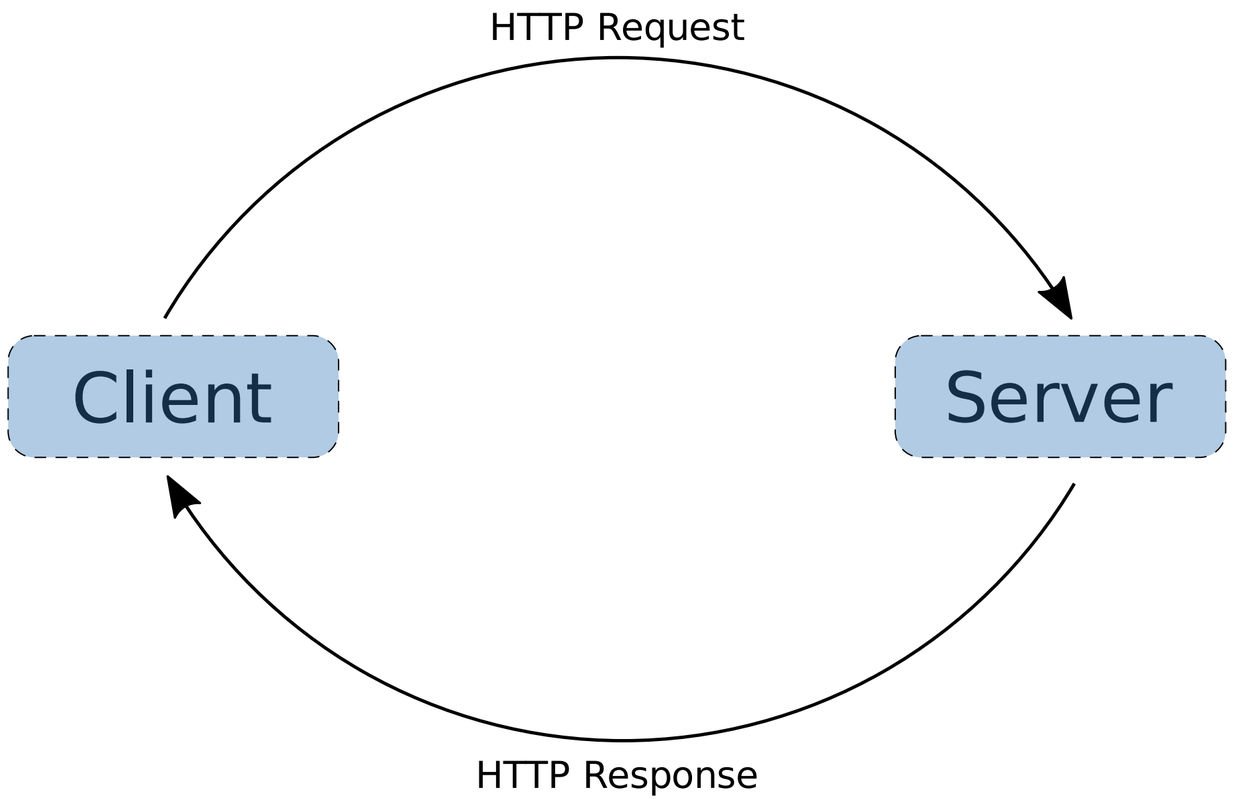
requests: HTTP for Humans

The `requests` package makes HTTP interactions easy.

It is not part of base Python. Read the documentation here.

toptal

HTTP Requests



toptal

Important request types for scraping: GET and POST.

Functions

The `requests` module has functions for each of the HTTP request types.

Most common requests:

- `get()` - retrieving a URL
- `post()` - submitting a form

Other requests:

- `put()`
- `delete()`
- `head()`
- `options()`

GET

A GET request is equivalent to simply visiting a URL with a browser.

Pass a dictionary as `params` argument.

For example, to get 5 matches on "web scraping" from Google:

```
>>> params = {'q': 'web scraping', 'num': 5}
>>> r = requests.get("https://www.google.com/search", params=params)
```

Check `Response` object.

```
>>> r.status_code
200
>>> r.url
'https://www.google.com/search?num=5&q=web+scraping'
```

POST

A POST request results in information being stored on the server. This method is most often used to submit forms.

Pass a dictionary as `data` argument.

Let's sign John Smith up for the OneDayOnly newsletter.

```
>>> payload = {
...     'firstname': 'John',
...     'lastname': 'Smith',
...     'email': 'john.smith@gmail.com'
... }
>>> r = requests.post("https://www.onedayonly.co.za/subscribe/campaign/confirm/", data=payloa
d)
```

Response Objects

Both the `get()` and `post()` functions return `Response` objects.

A `Response` object has a number of useful attributes:

- `url`
- `status_code`
- `headers` - a dictionary of headers
- `text` - response as text
- `content` - response as binary (useful for non-text content)
- `encoding`

Also some handy methods:

- `json()` - decode JSON into dictionary

HTTP Status Codes

HTTP status codes summarise the outcome of a request.

These are some of the common ones:

2xx Success

- `200` - OK

3xx Redirect

- `301` - Moved permanently

4xx Client Error

- `400` - Bad request
- `403` - Forbidden
- `404` - Not found

5xx Server Error

- `500` - Internal server error

HTTP Headers

HTTP headers appear in both HTTP request and response messages. They determine the parameters of the interaction.

These are the most important ones for scraping:

Request Header Fields

- `User-Agent`
- `Cookie`

You can modify request headers by using the `headers` parameter to `get()` or `post()`.

Response Header Fields

- `Set-Cookie`
- `Content-Encoding`
- `Content-Language`
- `Expires`

HTTPBIN

This is a phenomenal tool for testing out HTTP requests.

Have a look at the range of endpoints listed on the home page. These are some that we'll be using:

- http://httpbin.org/get - returns GET data
- http://httpbin.org/post - returns POST data
- http://httpbin.org/cookies - returns cookie data
- http://httpbin.org/cookies/set - sets one or more cookies

For example:

```
>>> r = requests.get("http://httpbin.org/get?q=web+scraping")
>>> print(r.text)
{
  "args": {
    "q": "web scraping"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "origin": "105.184.228.131",
  "url": "http://httpbin.org/get?q=web+scraping"
}
```

Notebook: requests

toptal

Parsing HTML: Regex

> *Let's say you have a problem, and you decide to solve it with regular expressions.*
>
> *Well, now you have two problems.*

You can build a Web Scraper using regular expressions but

- it won't be easy and
- it'll probably be rather fragile.

toptal

Parsing HTML: LXML

LXML is a wrapper for libxml2 written in C.

It's super fast.

But very low level, so not ideal for writing anything but the simplest scrapers.

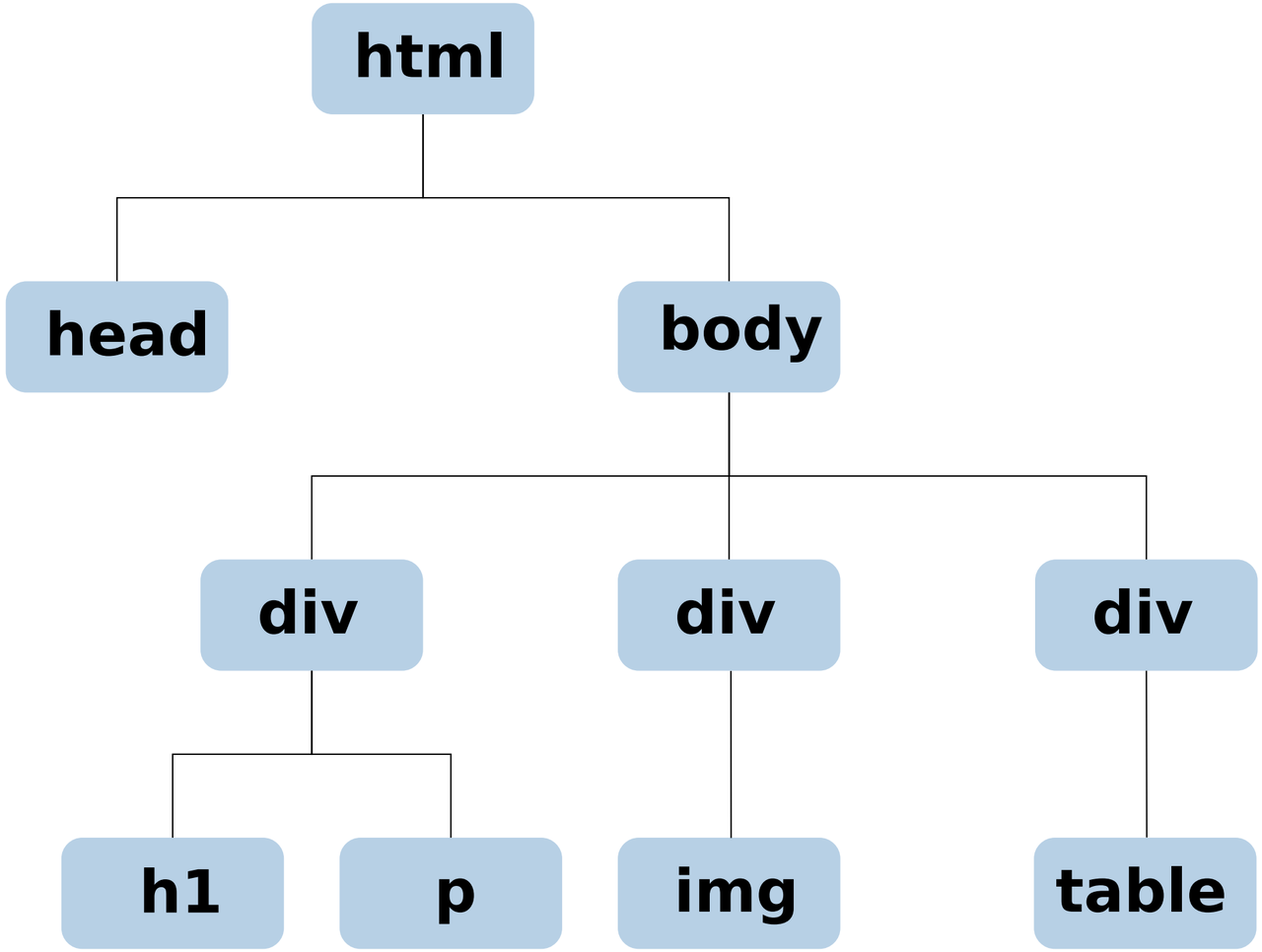
Elements

Document tree (and parts thereof) are represented by `Element` objects.

Makes recursive parsing very simple. Same operation for

- search on entire document and
- search from within document.

Document Tree Structure



toptal

Example: Search of Private Property

toptal

Exercise: Deals from OneDayOnly

1. Retrieve today's deals from OneDayOnly.
2. Scrape brand, name and price for each deal.

Beautiful Soup

> *You didn't write that awful page.*
> *You're just trying to get some data out of it.*
> *Beautiful Soup is here to help.*

Beautiful Soup makes parsing a web page simple.

Objects

Beautiful Soup has two key classes:

- `BeautifulSoup`
- `Tag`

Notebook: Beautiful Soup

Example: Wikipedia HTML Entity

Scrape the table of HTML entities on Wikipedia.

Exercise: Race Results

Scrape results table from Race Results.

Preparation

1. Start from http://bit.ly/2y8nJDA.
2. Select a race.
3. Find POST request parameters (read http://bit.ly/2y8nJDA).
4. Find POST request URL (not the same as URL above!).

Scraper

Write a scraper which will:

1. Submit POST request for selected race.
2. Parse the results.
3. Write to CSV file.

Hints

- *This is more challenging because the HTML is poorly formed.*
- *Grab all the table cells and then restructure into nested lists.*

toptal

Scrapy

Scrapy is a framework for creating a robot or spider which will recursively traverse pages in a web site.

CLI Options

Scrapy is driven by a command line client.

```
$ scrapy -h
Scrapy 1.4.0 - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  bench         Run quick benchmark test
  fetch         Fetch a URL using the Scrapy downloader
  genspider     Generate new spider using pre-defined templates
  runspider     Run a self-contained spider (without creating a project)
  settings      Get settings values
  shell         Interactive scraping console
  startproject  Create new project
  version       Print Scrapy version
  view          Open URL in browser, as seen by Scrapy

  [ more ]      More commands available when run from project directory

Use "scrapy <command> -h" to see more info about a command
```

Scrapy Shell

The Scrapy shell allows you to explore a site interactively.

```
$ scrapy shell
[s] Available Scrapy objects:
[s]   scrapy     scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler    <scrapy.crawler.Crawler object at 0x7fc1c8fe6518>
[s]   item       {}
[s]   settings   <scrapy.settings.Settings object at 0x7fc1cbfda198>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects
[s]   fetch(req)                  Fetch a scrapy.Request and update local objects
[s]   shelp()           Shell help (print this help)
[s]   view(response)    View response in a browser
In [1]:
```

toptal

Interacting with the Scrapy Shell

```
In [1]: fetch("http://quotes.toscrape.com/")
2017-09-19 17:24:42 [scrapy.core.engine] INFO: Spider opened
2017-09-19 17:24:43 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.toscrape.com/
>
```

We can open that page in a browser.

```
In [2]: view(response)
```

And print the page content.

```
In [3]: print(response.text)
```

We can use CSS or XPath to isolate tags and extract their content.

```
In [4]: response.css("div:nth-child(6) > span.text::text").extract_first()
Out[4]: '“Try not to become a man of success. Rather become a man of value.”'
```

```
In [5]: response.css("div:nth-child(6) > span:nth-child(2) > a::attr(href)").extract_first()
Out[5]: '/author/Albert-Einstein'
```

Note that we have used the `::text` and `::attr()` filters.

Exercise: Looking at Lawyers

Explore the web site of Webber Wentzel.

1. Open the link above in your browser.
2. Select a letter to get a page full of lawyers.
3. Fetch that page in the Scrapy shell.
4. Use SelectorGadget to generate the CSS selector for one of the lawyer's email addresses.
5. Retrieve the email address using the Scrapy shell.
6. Retrieve the email addresses for all lawyers on the page.

Hints

- *Use an attribute selector to pick out the links to email addresses.*

toptal

Creating a Project

After the exploratory phase we'll want to automate our scraping.

We're going to scrape http://quotes.toscrape.com/.

```
$ scrapy startproject quotes
```

```
$ tree quotes
quotes/
├── quotes
│   ├── __init__.py
│   ├── items.py                    # Item definitions
│   ├── middlewares.py
│   ├── pipelines.py                # Pipelines
│   ├── __pycache__
│   ├── settings.py                 # Settings
│   └── spiders                     # Folder for spiders
│       ├── __init__.py
│       └── __pycache__
└── scrapy.cfg                      # Configuration

4 directories, 7 files
```

Creating a Spider

Spiders are classes which specify

- how to follow links and
- how to extract information from pages.

Find out more about spiders here.

```
$ cd quotes
$ scrapy genspider Quote quotes.toscrape.com
Created spider 'Quote' using template 'basic' in module:
  quotes.spiders.Quote
```

This will create `Quote.py` in the `quotes/spiders` folder.

Spider Class

This is what `Quote.py` looks like.

```
import scrapy

class QuoteSpider(scrapy.Spider):
    name = 'Quote'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        pass
```

It defines these class attributes:

- `allowed_domains` - links outside of these domains will not be followed; and
- `start_urls` - a list of URLs where the crawl will start.

The `parse()` method does most of the work (but right now it's empty).

You can also override `start_requests()` which yields list of initial URLs.

Anatomy of a Spider

URLs

Either

- define `start_urls` or
- override `start_requests()`, which must return an iterable of `Request` (either a list or generator).

```
    def start_requests(self):
        pass
```

These will form the starting point of the crawl. More requests will be generated from these.

Parsers

Define a `parse()` method which

- accepts a `response` parameter which is a `TextResponse` (holds page contents);
- extract the required data and
- finds new URLs, creating new `Request` objects for each of them.

Starting the Spider

```
$ scrapy crawl -h
Usage
=====
  scrapy crawl [options] <spider>

Run a spider

Options
=======
--help, -h              show this help message and exit
-a NAME=VALUE           set spider argument (may be repeated)
--output=FILE, -o FILE  dump scraped items into FILE (use - for stdout)
--output-format=FORMAT, -t FORMAT
                        format to use for dumping items with -o

Global Options
--------------
--logfile=FILE          log file. if omitted stderr will be used
--loglevel=LEVEL, -L LEVEL
                        log level (default: DEBUG)
--nolog                 disable logging completely
--profile=FILE          write python cProfile stats to FILE
--pidfile=FILE          write process ID to FILE
--set=NAME=VALUE, -s NAME=VALUE
                        set/override setting (may be repeated)
--pdb                   enable pdb on failure
```

We'll kick off our spider as follows:

```
$ scrapy crawl Quote
```

Exporting Data

Data can be written to a range of media:

- standard output
- local file
- FTP
- S3.

Scrapy can also export data in a variety of formats using Item Exporters.

But if you don't need anything fancy then this can be done from command line.

```
$ scrapy crawl Quote -o quotes.csv -t csv            # CSV
$ scrapy crawl Quote -o quotes.json -t json          # JSON
```

Or you can configure this in `settings.py`.

Find out more about feed exports here.

Settings

Modify `settings.py` to configure the behaviour of the crawl and scrape. Find out more here.

Throttle Rate

```
CONCURRENT_REQUESTS_PER_DOMAIN = 1
DOWNLOAD_DELAY = 3
```

Output Format

```
FEED_FORMAT = "csv"
FEED_URI = "quotes.csv"
```

Pipelines

Every scraped item passes through a pipeline which can apply a sequence of operations.

Example operations:

- validation
- remove duplicates
- export to file or database
- take screenshot
- download files and images.

toptal

Templates

A project is created from a template.

Templates are found in the `scrapy/templates` folder in your Python library.

You can create your own templates which will be used to customise new projects.

The Cookiecutter project is also great for working with project templates.

Scrapy Classes

Request

A `Request` object characterises the query submitted to the web server.

- `url`
- `method` - the HTTP request type (normally either `GET` or `POST`) and
- `headers` - dictionary of headers.

Response

A `Response` object captures the response returned by the web server.

- `url`
- `status` - the HTTP status
- `headers` - dictionary of headers
- `urljoin()` - construct an absolute URL from a relative URL.

TextResponse

A `TextResponse` object inherits from `Response`.

- `text` - response body
- `encoding`
- `css()` or `xpath()` - apply a selector

toptal

Example: Quotes to Scrape

toptal

Exercise: Catalog of Lawyers

Scrape the employee database of Webber Wentzel.

Hints

- You might find `string.ascii_uppercase` useful for generating URLs.
- It might work well to follow links to individual profile pages.
- *Limit the number of concurrent requests to 2.*

Exercise: Weather Buoys

Data for buoys can be found at http://www.ndbc.noaa.gov/to_station.shtml.

For each buoy retrieve:

- identifier and
- geographic location.

Limit the number of concurrent requests to 2.

Example: Slot Catalog

Scrape the information for slots games from https://slotcatalog.com/.

Hints

- *Limit the number of concurrent requests to 2.*
- *Limit the number of pages scraped.*

```
$ scrapy crawl -s CLOSESPIDER_ITEMCOUNT=5 slot
```

Creating a CrawlSpider

Setting up the 'horizontal' and 'vertical' components of a crawl can be tedious.

Enter the CrawlSpider, which makes this a lot easier.

It's beyond our scope right now though!

Selenium

When do You Need Selenium?

When scraping web sites like these:

- FinishTime
- takealot (doesn't rely on JavaScript, but has other challenges!)

toptal

Notebook: Selenium

Example: takealot

1. Submit a search.
2. Show 50 items per page in results.
3. Sort results by ascending price.
4. Scrape the name, link and price for each of the items.

Exercise: Sports Betting

NetBet relies heavily on JavaScript. So conventional scraping techniques will not work.

Write a script to retrieve today's Horse Racing odds.

1. Click on Horse Racing menu item.
2. Select a course and time. Press View. Behold the data!
3. Turn off JavaScript support in your browser. Refresh the page... You're going to need Selenium!
4. Turn JavaScript back on again. Refresh the page.

Once you've got the page for a particular race, find the selectors required to scrape the following information for each of the horses:

- Horse name
- Trainer and Jockey name
- Weight
- Age
- Odds.

Hints

- *The table you are looking for can be selected with* `table.oddsTable`.
- *The first row of the table needs to be treated differently.*

toptal

Where to Now?

toptal

Crawling at Scale

When your target web site is sufficiently large the actual scraping is less of a problem than the infrastructure.

Do the Maths

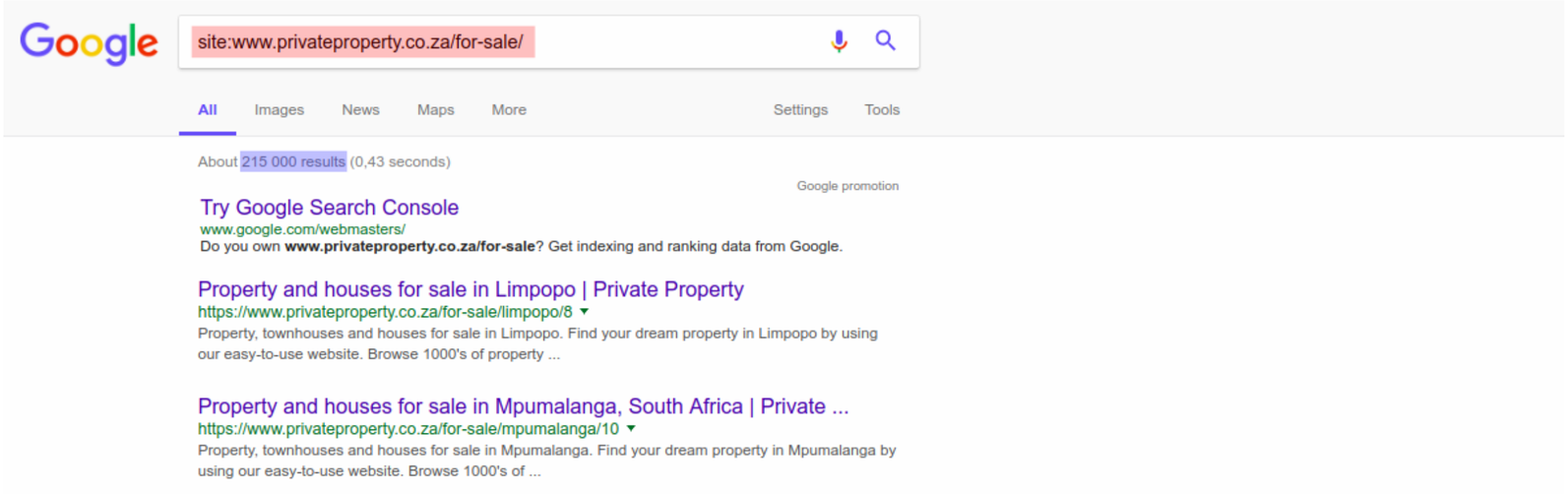
How long does it take you to scrape a single page?

How many pages do you need to scrape?

Crawling: Site Size

Google is arguably the largest crawler of web sites.

A Google `site:` search can give you an indication of number of pages.



toptal

Multiple Threads

Your scraper will spend a lot of time waiting for network response.

With multiple threads you can keep your CPU busy even when waiting for responses.

Remote Scraping

Setting up a scraper on a remote machine is an efficient way to

- handle bandwidth;
- save on local processing resources;
- scrape even when your laptop is turned off and
- send requests from a new IP.

Use the Cloud

An AWS Spot Instance can give you access to a powerful machine and a great network connection.

But terminate your instance when you are done!

Avoiding Detection

Many sites have measures in place to prevent (or at least discourage) scraping.

User Agent String

Spoof `User-Agent` headers so that you appear to be "human".

Find out more about your browser's `User-Agent` here.

Frequency

Adapt the interval between requests.

```
>>> from numpy.random import poisson
>>> import time
>>> time.sleep(poisson(10))
```

Vary your IP

Proxies allow you to effectively scrape from multiple (or at least other) IPs.

Making it Robust

Store Results Immediately (if not sooner)

Don't keep results in RAM. Things can break. Write to disk ASAP.

Flat file is good.

Database is better.

Plan for Failure

1. Cater for the following issues:
    - 404 error
    - 500 error
    - invalid URL or DNS failure.
2. Handle exceptions.

Nothing worse than finding your scraper has been sitting idle for hours.

Sundry Tips

Use a Minimal URL

Strip unnecessary parameters off the end of a URL.

Maintain a Queue of URLs to Scrape

Stopping and restarting your scrape job is not a problem because you don't lose your place.

Even better if the queue is accessible from multiple machines.

Data Mashup

One of the coolest aspects of Web Scraping is being able to create your own set of data.

You can...

- use these data to augment existing data; or
- take a few sets of scraped data and merge them to form a *data mashup*.

Scraping FTW! ☺

Have Fun.

toptal