

# Web Scraping & Text Mining

Paulo Serôdio

Postdoctoral Researcher  
School of Economics  
Universitat de Barcelona

May 14, 2018



# Introduction

# Introduction



# Introduction



IPE Institutions and Political Economy  
RG Research Group

# Introduction



IPE Institutions and Political Economy  
RG Research Group



# Introduction



IPE Institutions and Political Economy  
RG Research Group



# Your background

# Course Materials & Structure

<http://www.pauloserodio.com/eui2018>

# WEB SCRAPING I

# Ethics

## On the ethics of web scraping and data journalism

- If an institution publishes data on its website, this data should automatically be public
- If a regular user can't access the data, we shouldn't try to get it (that would be hacking)
- Always read the user terms and conditions
- Always check the robots.txt file, which states what is allowed to be scraped

# Disclaimer

- 1 you take all the responsibility for your web scraping work
- 2 take all copyrights of a country's jurisdiction into account
- 3 if you publish data, do not commit copyright fraud
- 4 if in doubt, ask the author/creator/provider of data for permission – if your interest is entirely scientific, chances aren't bad that you get data
- 5 consult current jurisdiction, e.g. on <http://blawgsearch.justia.com> or from a lawyer specialized on internet law

## Basic rules

- 1 You should check a site's terms and conditions before you scrape them. It's their data and they likely have some rules to govern it.
- 2 Be nice - A computer will send web requests much quicker than a user can. Make sure you space out your requests a bit so that you don't hammer the site's server.
- 3 Scrapers break - Sites change their layout all the time. If that happens, be prepared to rewrite your code.
- 4 Web pages are inconsistent - There's sometimes some manual clean up that has to happen even after you've gotten your data.

# Robots.txt rules

1. Allow full access

User-agent: \*

Disallow:

2. Block all access

User-agent: \*

Disallow:/

3. Partial Access

User-agent: \*

Disallow:/folder/

User-agent: \*

Disallow:/file.html

## Robots.txt rules

4. Crawl rate limiting

Crawl-delay: 11

5. Visit time

Visit-time: 0400-0845

6. Request Rate (pages per second)

Request-rate: 1/10

# Scraping the web: two approaches

Two different approaches:

1. **Screen scraping**: extract data from source code of website, with html parser and/or regular expressions (`rvest` package in R);
2. **Web APIs** (application programming interfaces): a set of structured http requests that return JSON or XML data
  - `httr` package to construct API requests
  - Packages specific to each API: `weatherData`, `WDI`, `Rfacebook`...
  - Check CRAN Task View on Web Technologies and Services for examples

# good scraping practices

1. Respect the hosting site's wishes:
  - Check if an API exists or if data are available for download
  - Keep in mind where data comes from and give credit (and respect copyright if you want to republish the data!)
2. Limit your bandwidth use:
  - Wait one or two seconds after each hit
  - Scrape only what you need, and just once (e.g. store the html file in disk, and then parse it)
3. When using APIs, read documentation
4. Is there a batch download option?
  - Are there any rate limits?
  - Can you share the data?

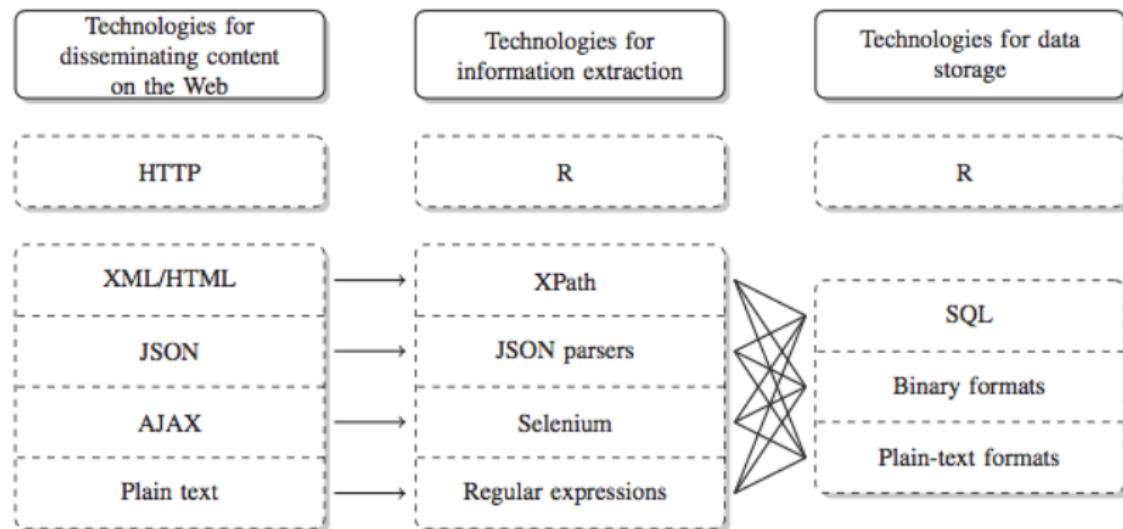
# Workflow

- (1) Learn about structure of website
- (2) Choose your strategy
- (3) Build prototype code: extract, prepare, validate
- (4) Generalize: functions, loops, debugging
- (5) Data cleaning

# Scenarios

- (1) Data in **table** format: rvest
- (2) Data in **unstructured** format: selectorGadget + rvest
- (3) Data behind **web forms**: selenium

# Technology and Packages

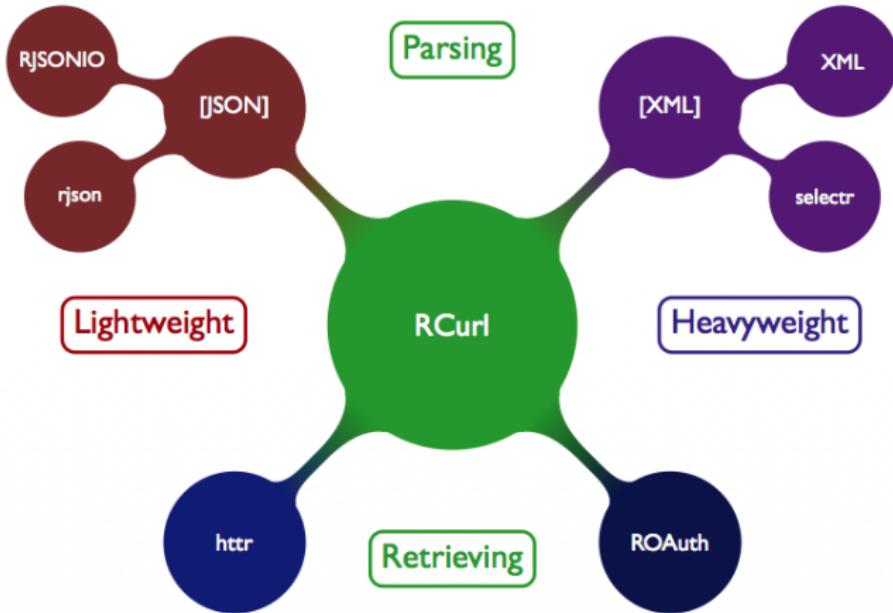


**Figure 1.4** Technologies for disseminating, extracting, and storing web data

# Technology and Packages

- ▶ Regular Expressions / String Handling
  - ▶ stringr, stringi
- ▶ HTML / XML / XPath / CSS Selectors
  - ▶ rvest, xml2, XML
- ▶ JSON
  - ▶ jsonlite, RJSONIO, rjson
- ▶ HTTP / HTTPS
  - ▶ httr, curl, Rcurl
- ▶ Javascript / Browser Automation
  - ▶ RSelenium
- ▶ URL
  - ▶ urltools

# Technology and Packages



# Reads

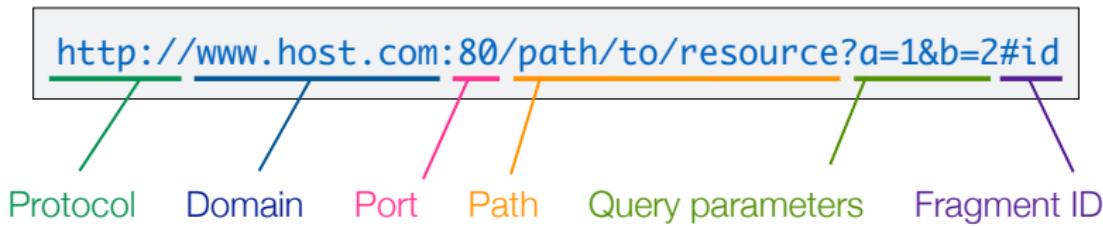
- ▶ **Basics on HTML, XML, JSON, HTTP, RegEx, XPath**
  - ▶ Munzert et al. (2014): *Automated Data Collection with R*. Wiley.  
<http://www.r-datacollection.com/>
- ▶ **curl / libcurl**
  - ▶ [http://curl.haxx.se/libcurl/c/curl\\_easy\\_setopt.html](http://curl.haxx.se/libcurl/c/curl_easy_setopt.html)
- ▶ **CSS Selectors**
  - ▶ W3Schools: [http://www.w3schools.com/cssref/css\\_selectors.asp](http://www.w3schools.com/cssref/css_selectors.asp)
- ▶ **Packages: httr, rvest, jsonlite, xml2, curl**
  - ▶ Readmes, demos and vignettes accompanying the packages
- ▶ **Packages: RCurl and XML**
  - ▶ Munzert et al. (2014): *Automated Data Collection with R*. Wiley.
  - ▶ Nolan and Temple-Lang (2013): *XML and Web Technologies for Data Science with R*. Springer

**HTTP**  
**H**yper**T**ext **T**ransfer **P**rotocol

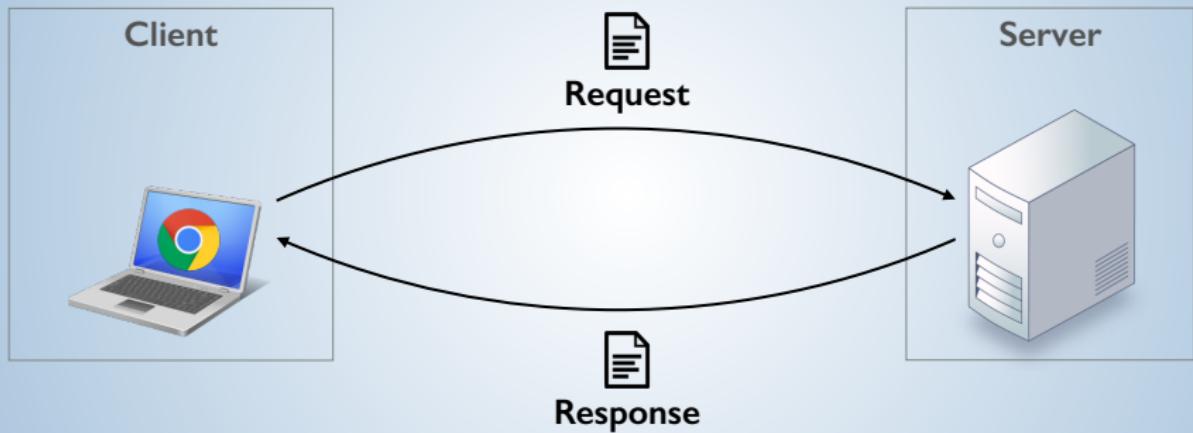
**URL**  
**U**niform **R**esource **L**ocator



# URL



# *HTTP*



# HTTP message structure

<Initial line different for request vs. response>

Header1: value1

Header2: value2

HeaderN: valueN

<Message body (optional). If contents are returned  
in a response, they will be contained in the body,  
perhaps as binary data>

Initial line

Optional headers

Blank Line

Optional body



# Sample Response

HTTP version

Status code

English explanation

HTTP/1.1 200 OK

Server: nginx

Date: Wed, 09 Nov 2016 14:14:47 GMT

Content-Type: application/json

<body with content as raw data>



Page not found – RStudio

https://www.rstudio.com/secret/stuff

Garrett

My Sites RStudio Customize 11 47 New WP Engine Quick Links Avada

Howdy, Garrett Grolemund  

R Studio rstudio::conf Products Resources Pricing About Us Blogs 

Oops, This Page Could Not Be Found!

404

Helpful Links

-  [rstudio::conf](#)
-  [Products](#)
-  [Resources](#)
-  [Pricing](#)
-  [About Us](#)
-  [Blogs](#)

Search Our Website

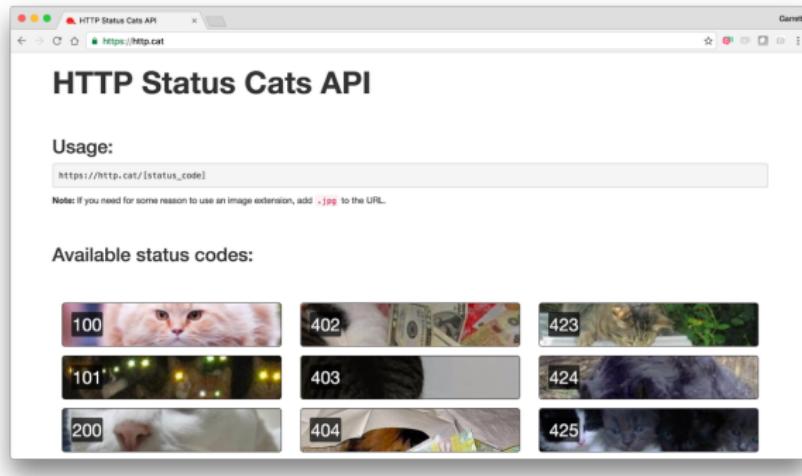
Can't find what you need? Take a moment and do a search below!

Search ... 



# Status codes

Cute dictionary at <http://http.cat>



A screenshot of a web browser window showing the "HTTP Status Cats API". The title bar says "HTTP Status Cats API". The URL bar shows "https://http.cat". The main content area has a heading "HTTP Status Cats API". Below it, a "Usage:" section shows a text input field containing "https://http.cat/[status\_code]". A note below the input field says "Note: If you need for some reason to use an image extension, add .jpg to the URL.". Another section titled "Available status codes:" shows a grid of nine cards, each containing a status code and a small cat image. The cards are arranged in three rows of three. The first row contains cards for 100, 402, and 423. The second row contains cards for 101, 403, and 424. The third row contains cards for 200, 404, and 425.

Status Code	Cat Image
100	White fluffy cat
402	Money bills
423	Two cats in a cage
101	Two glowing eyes
403	Black screen
424	Two cats in a dark space
200	Close-up of a cat's nose
404	Blank page
425	Two cats looking at the camera



# Status

Extract status with **\$status\_code** or **http\_status()**

```
r$status_code  
# 200  
http_status(r)  
# $category  
# [1] "Success"  
# $reason  
# [1] "OK"  
# $message  
# [1] "Success: (200) OK"
```



Program defensively with **`warn_for_status()`** and **`stop_for_status()`**

```
r2 <- r
r2$status_code <- 404
warn_for_status(r2)
# Warning message:
# Not Found (HTTP 404).

stop_for_status(r2)
# Error: Not Found (HTTP 404).
```



HTML, XML & CSS

# Dissemination I – HTML

Hypertext Markup Language (HTML): hidden standard behind every website.

HTML is text with marked-up structure, defined by tags:

# Dissemination I – HTML

Hypertext Markup Language (HTML): hidden standard behind every website.

HTML is text with marked-up structure, defined by tags:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

# Dissemination I – HTML

Hypertext Markup Language (HTML): hidden standard behind every website.

HTML is text with marked-up structure, defined by tags:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

What you see in your browser is an interpretation of the HTML document;

# Dissemination I – HTML

Hypertext Markup Language (HTML): hidden standard behind every website.

HTML is text with marked-up structure, defined by tags:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

What you see in your browser is an interpretation of the HTML document;

# Dissemination I – HTML

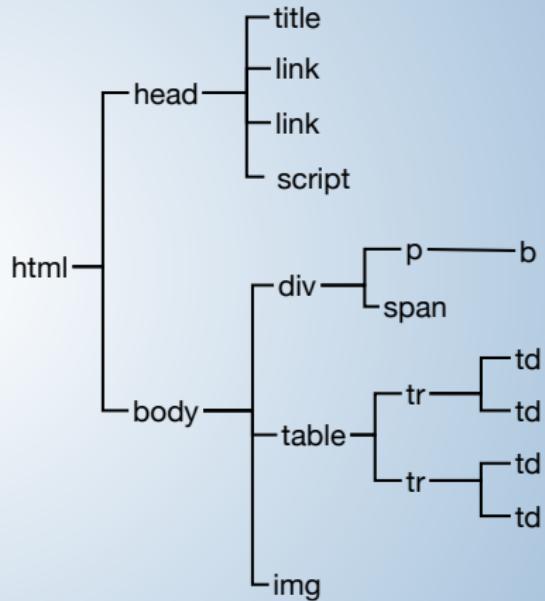
Some common tags:

- Document elements: <head>, <body>, <footer>...
- Document components: <title>, <h1>, <div>...
- Text style: <b>, <i>, <strong>...
- Hyperlinks/Anchor: <a>

# HTML (*Review*)



```
<html>
  <head>
    <title>Title</title>
    <link rel="icon" type="icon" href="http://a" />
    <link rel="icon" type="icon" href="http://b" />
    <script type="text/javascript">
      var ue_t0=window.ue_t0||+new Date();
    </script>
  </head>
  <body>
    <div>
      <p>Click <b>here</b> now.</p>
      <span>Frozen</span>
    </div>
    <table style="width:100%">
      <tr>
        <td>Kristen</td>
        <td>Bell</td>
      </tr>
      <tr>
        <td>Idina</td>
        <td>Menzel</td>
      </tr>
    </table>
    
  </body>
</html>
```



# *HTML (Review)*

Each element in the page is created by a tag.

```
<a href="http://github.com">GitHub</a>
```

tag name

attribute  
(name)

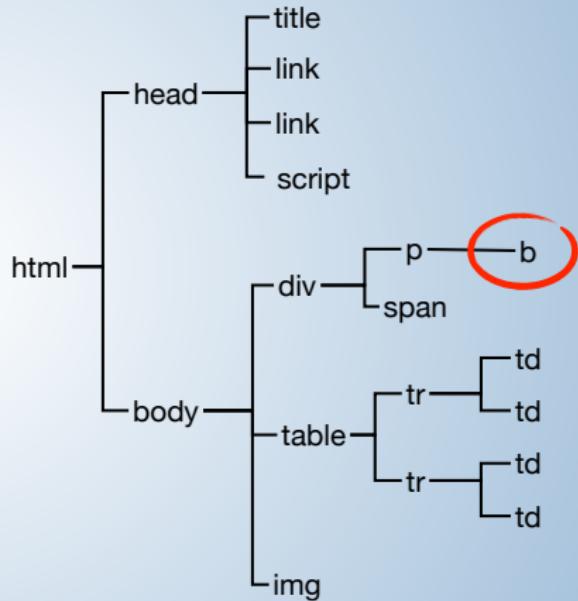
attribute  
(value)

content

# HTML (Review)



```
<html>
  <head>
    <title>Title</title>
    <link rel="icon" type="icon" href="http://a" />
    <link rel="icon" type="icon" href="http://b" />
    <script type="text/javascript">
      var ue_t0=window.ue_t0||+new Date();
    </script>
  </head>
  <body>
    <div>
      <p>Click <b>here</b> now.</p>
      <span>Frozen</span>
    </div>
    <table style="width:100%">
      <tr>
        <td>Kristen</td>
        <td>Bell</td>
      </tr>
      <tr>
        <td>Idina</td>
        <td>Menzel</td>
      </tr>
    </table>
    
  </body>
</html>
```



# Beyond HTML

- Cascading Style Sheets (CSS): describes formatting of HTML components (e.g. `<h1>`, `<div>`, ...), useful to scraping.

# Beyond HTML

- Cascading Style Sheets (CSS): describes formatting of HTML components (e.g. `<h1>`, `<div>`, ...), useful to scraping.



# Beyond HTML

- **Javascript**: adds functionalities to the website (e.g. change content/structure after website has been loaded).

# Extraction I – Parsing HTML code

First step in webscraping: read HTML code in R and **parse it**

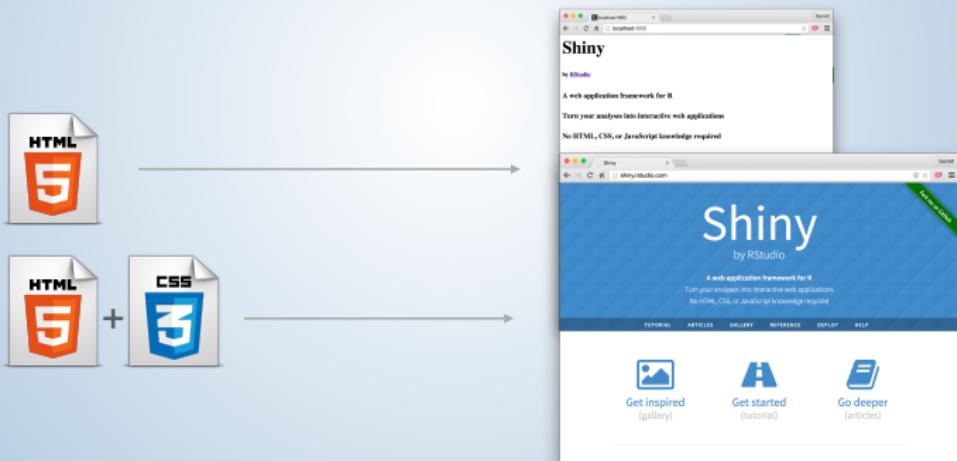
- Parsing = understanding structure
- How? `rvest` package in R:
  - `read_html`: parse HTML code into R
  - `html_text`: extract text from HTML code
  - `html_table`: extract tables in HTML code
  - `html_nodes`: extract components with CSS selector
  - `htmlAttrs`: extract attributes of nodes
- How to identify relevant CSS selectors? `selectorGadget` extension for Chrome and Firefox.

## Extraction II - CSS Selectors

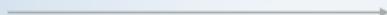
- Css selectors are widely used by frontend developers to associate css properties with their html elements: it styles the webpages;
- scrapers can use it to navigate in the structure of an html file.
- Xpath are far more powerful compared to CSS selectors because we can put a logic into a single XPath statement to precisely identify the right element on a web page;
- to reach elements via CSS selectors is conditional on the consistency of the webpage styling

# CSS (*Review*)

Cascading Style Sheets (CSS) are a framework for customizing the appearance of elements in a web page.



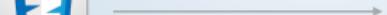
# CSS (*Review*)



The screenshot shows the Shiny website's landing page. At the top, there's a navigation bar with links for Home, Tutorials, Examples, Reference, Deploy, and Help. Below the navigation, the word "Shiny" is prominently displayed in large white letters against a dark background. A sub-header reads "A web application framework for R". A tagline says "Turn your analyses into interactive web applications". A note states "No HTML, CSS, or JavaScript knowledge required". There are several links: "Tutorial" (with sub-links for "Getting started", "Gallery", and "Reference"), "Reference" (with sub-links for "Data types", "UI components", and "Help"), "Get inspired" (with a "gallery" link), "Get started" (with a "tutorial" link), and "Go deeper" (with an "articles" link). The overall design is clean and professional.



+



This screenshot shows a simplified version of the Shiny website's homepage. The layout is more compact. The "Shiny" logo and tagline are at the top. Below them is a single "Get started" button with a "Tutorial" link. At the bottom, there are three main calls-to-action: "Get Inspired" (with a "gallery" link), "Get started" (with a "Tutorial" link), and "Go deeper" (with an "articles" link). The navigation bar at the very bottom includes links for TUTORIAL, ARTICLES, GALLERY, REFERENCE, DEPLOY, and HELP.

# CSS (Review)



```
span {  
    color: #ffffff;  
}  
  
.num {  
    color: #a8660d;  
}  
  
table.data {  
    width: auto;  
}  
  
#firstname {  
    background-color: yellow;  
}
```

← selector

← styling

← selector

← styling

← selector

← styling

← selector

← styling

# CSS (*Review*)

A CSS script describes an element by its tag, class, and/or ID.

```
<span class="bigname" id="shiny">Shiny</span>
```

tag name

class  
(optional)

id  
(optional)

# CSS (*Review*)

A CSS script describes an element by its tag, class, and/or ID.

```
<span class="bigname" id="shiny">Shiny</span>
```

span

CSS selector for **ALL** elements with:

- the **span tag**

# CSS (*Review*)

A CSS script describes an element by its tag, class, and/or ID.

```
<span class="bigname" id="shiny">Shiny</span>
```

```
.bigname
```

CSS selector for **ALL** elements with:

- the **bigname class**

# CSS (*Review*)

A CSS script describes an element by its tag, class, and/or ID.

```
<span class="bigname" id="shiny">Shiny</span>
```

```
span.bigname
```

CSS selector for **ALL** elements with:

- the **span tag**  
**AND**  
• the **bigname class**

# CSS (*Review*)

A CSS script describes an element by its tag, class, and/or ID.

```
<span class="bigname" id="shiny">Shiny</span>
```

```
#shiny
```

CSS selector for **ALL** elements with:

- the **shiny id**

# CSS (*Review*)

Prefix	Matches
none	tag
.	class
#	id

# selectorGadget

A GUI tool to identify CSS selector combinations

.fa-bolt

Clear (1)

Toggle Position

XPath

Help

X

.fa-bolt

Clear (1)

Toggle Position

XPath

Help

X

CSS selector to use

start over

move gadget

show XPath

help

close gadget

# Dissemination II – XML

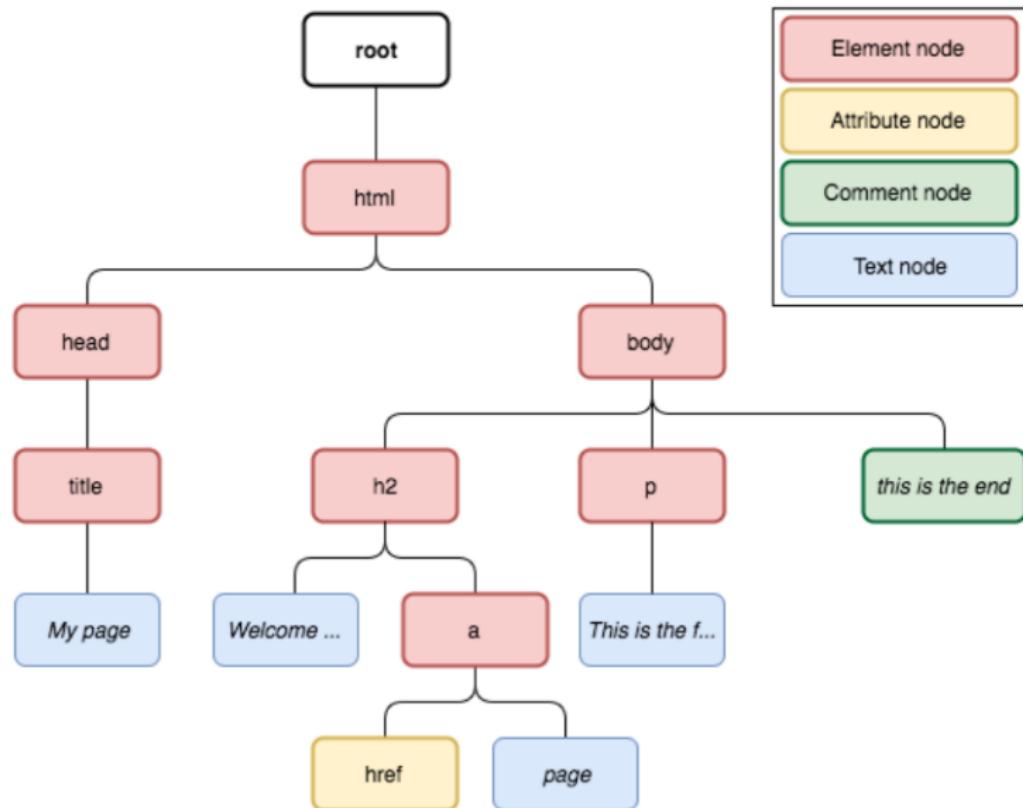
```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

- Short for Extensible Markup Language
- while HTML was designed to display data, XML was designed to **transport** and **store** data;
- XML separates data from HTML, simplifies it and stored in plain text format;
- From web scraping perspective, XML **tree structure** is its most important features;
- XML contain root element – called **parent** of all other elements
- Terms parent, child, sibling are used to describe relationships between elements;
- All elements can have text content and attributes;

## Extraction II - Xpath

- XPath (XML Path Language) is a syntax for defining parts of an XML document.
- XPath is a query language for identifying and selecting nodes or elements in an XML document using a tree like representation of the document.
- XPaths are one of the few ways in which you can select some content from a big blob of XML or HTML (properly structured HTML is similarly structured as an XML document) content. A Xpath tells you the location of an element, just like a catalog card does for books.
- Xpaths change when a website changes the way the HTML is structured. It is just like rearranging a library. Every time a library is rearranged the location of a book might change.
- in a well established scraping environment, the only things that often need changing are the selectors;

## Extraction II - Xpath



## Extraction II - Xpath

Expression	Meaning
/html	Selects the node named <code>html</code> , which is under the root.
/html/head	Selects the node named <code>head</code> , which is under the <code>html</code> node.
//title	Selects all the <code>title</code> nodes from the HTML tree.
//h2/a	Selects all <code>a</code> nodes which are directly under an <code>h2</code> node.

And here are some examples of node type tests:

Expression	Meaning
//comment()	Selects only comment nodes.
//node()	Selects any kind of node in the tree.
//text()	Selects only text nodes, such as "This is the first paragraph".
//*	Selects all nodes, except comment and text nodes.

Expression	Meaning
//a[@href="https://scrapy.org"]	Selects the <code>a</code> elements pointing to <code>https://scrapy.org</code> .
//a/@href	Selects the value of the <code>href</code> attribute from all the <code>a</code> elements in the document.
//li[@id]	Selects only the <code>li</code> elements which have an <code>id</code> attribute.

# Dissemination III - JSON/APIs

## Application Programming Interface

### Pro

- instant access to clean, structured data
- frees us from building manual scrapers
- forces us to understand the API/data architecture
- de facto automatic agreement of data owner
- Easy to process automatically; robustness of calls
- Can be directly embedded in your script

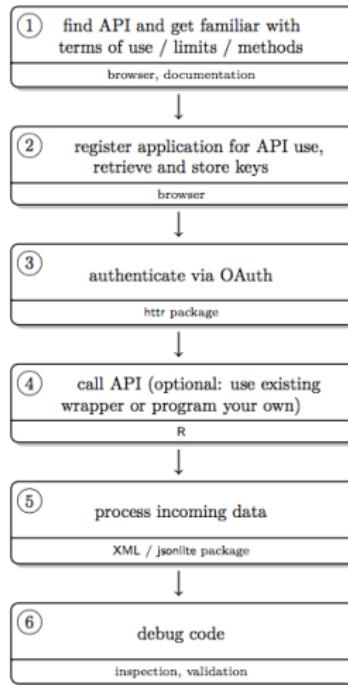
---

### Con

- Often limitations (requests per minute, sampling, . . . )
- You have to trust the provider that he delivers the right (free?) content
- Some APIs won't allow you to go back in time!

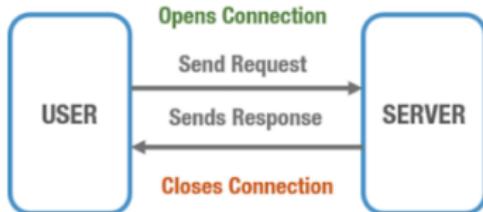
# Dissemination III - JSON/APIs

## Application Programming Interface steps...



# Dissemination III - JSON/APIs

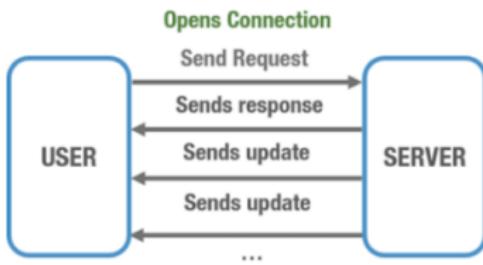
## REST API



- A REpresentational State Transfer API is a web service that follows a request-response pattern;
- Use makes individual request for info from the REST API and receive data in single response;
- After response is sent, connection closes only to be re-opened when another request is made by the user;
- Great for snapshots of the data;

# Dissemination III - JSON/APIs

## Streaming API



- **Streaming APIs** maintain persistent connection that continuously sends updated data to the user until terminated;;
- Great for constant flow of rapidly updating live data;
- The server repeatedly sends responses back with updated information;

# Dissemination III - JSON/APIs

## JSON: Javascript Object Notation

- a text format that facilitates structured data interchange between all programming languages.
- it's open, lightweight, and text-based data-interchange format
- Widely used in web APIs
- Becoming standard for online data format
- Great if we have nested data structure (items within feeds; personal data within authors within books; tweets within followers within users)
- Read into R with `jsonlite`, `rjson` and `rjsonio` package
- JSON files can carry huge amounts of data from the web; however, since R stores and processes all data in the memory, power of JSON is bounded by the limit of specific R machines

## Dissemination III - JSON/APIs

```
1 {'totalItems': 574, 'items': [{}'kind': 'books#volume', 'volumeInfo': {}  
2     'publisher': '"O'Reilly Media, Inc."', 'description': u'Get a  
3     comprehensive, in-depth introduction to the core Python language  
4     with this hands-on book. Based on author Mark Lutz\u2019s popular  
5     training course, this updated fifth edition will help you quickly  
6     write efficient, high-quality code with Python. It\u2019s an ideal  
7     way to begin, whether you\u2019re new to programming or a  
8     professional developer versed in other languages. Complete with  
9     quizzes, exercises, and helpful illustrations, this easy-to-follow,  
10    self-paced tutorial gets you started with both Python 2.7 and 3.3\u2014 the  
11    ...  
12    ...  
13    'kind': 'books#volumes'}  
14
```

# Dissemination III - JSON/APIs

## API Examples:

- [▶ Early Day Motions](#)
- [▶ API search](#)
- [▶ Search XML](#)
- [▶ EDM JSON](#)

## Extraction III - JSON parsers

- JSON files also come as a massive concatenation of characters (one huge string) with encoded strings or arrays inside it which cannot be easily consumed by other programming languages;
- the job of parser which takes this huge string and break it up into data structure comes in place in order to let other programming language work with it smoothly

## Extraction III - JSON parsers

Three R packages:

- RJSON: `fromJSON`, `toJSON`, `newJSONParser`
- RJSONIO: `fromJSON`, `toJSON`, `asJSVars`
- JSONLITE: `fromJSON()`, `toJSON()` to convert between JSON data and R objects. It can also interact with web APIs, building pipelines and streaming data between R and JSON.

# Dissemination IV - Asynchronous JavaScript and XML

## What is AJAX?

- HTML/HTTP are used for static display of content
- in order to display dynamic content, they lack
  1. mechanisms to detect user behavior in the browser (and not only on the server)
  2. a scripting engine that reacts on this behavior
  3. a mechanism for asynchronous queries
- Asynchronous JavaScript and XML is a set of technologies that serve these purposes
- massively used in modern webpage design and architecture
- makes classical screen scraping more difficult

Example:  Integrated Census Microdata (GB 1851-1911)

# Extraction IV - Selenium

## The problem reconsidered

- dynamic data requests are not stored in the static HTML page
- therefore, we cannot access them with classical methods and packages (`httr`, `XML`, `download.file()`, etc.)

## The solution

- initiate and control a web browser session with R
- let the browser do the JavaScript interpretation work and the manipulations in the live DOM tree
- access information from the web browser session

# Extraction IV - Selenium

## What's Selenium?

- <http://www.seleniumhq.org>
- free software environment for automated web application testing
- several modules for different tasks; most important for our purposes: Selenium WebDriver
- Selenium WebDiver starts a server instance (as proxy) and passes commands (posed in R in our case) to the browser
- automated browsing via scripts

# Extraction IV - Selenium

## Software requirements

- Java, <https://www.java.com/de/download/>
- Selenium server, <http://selenium-release.storage.googleapis.com/3.12/selenium-server-standalone-3.12.0.jar> or via RSelenium and checkForServer()
- Firefox browser, <https://www.mozilla.org/en-US/firefox/new/>
- **RSelenium** package

# Regular Expressions

matches **string**  
posix the characters one  
text language used literal strings  
for many abc syntax world  
many languages perl  
hello set unicode match  
can print example  
**character** pattern

# Regex - when NOT to use it



4424



You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach between this world and the dread realm of corrupt entities (like SGML entities, but more corrupt) a mere glimpse of the world of regex parsers for HTML will instantly transport a programmer's consciousness into a world of ceaseless screaming, he comes, the pestilent slithy regex-infection will devour your HTML parser, application and existence for all time like Visual Basic only worse he comes he comes do not fight he comes, his unholy radiancé destroying all enlightenment, HTML tags leaking from your eyes like liquid pain, the song of regular expression parsing will extinguish the voices of mortal man from the sphere I can see it can you see if it is beautiful the final snuffing of the lies of Man ALL IS LOST ALL IS LOST the pony he comes he comes he comes the anchor permeates all MY FACE

# String Manipulation & Basic Functions

## Manipulation of strings

Function	Description
<code>nchar()</code>	number of characters
<code>tolower()</code>	convert to lower case
<code>toupper()</code>	convert to upper case
<code>casefold()</code>	case folding
<code>chartr()</code>	character translation
<code>abbreviate()</code>	abbreviation
<code>substring()</code>	substrings of a character vector
<code>substr()</code>	substrings of a character vector

# String Manipulation & Basic Functions

Function	Description	Similar to
<code>str_c()</code>	string concatenation	<code>paste()</code>
<code>str_length()</code>	number of characters	<code>nchar()</code>
<code>str_sub()</code>	extracts substrings	<code>substring()</code>
<code>str_dup()</code>	duplicates characters	<i>none</i>
<code>str_trim()</code>	removes leading and trailing whitespace	<i>none</i>
<code>str_pad()</code>	pads a string	<i>none</i>
<code>str_wrap()</code>	wraps a string paragraph	<code>strwrap()</code>
<code>str_trim()</code>	trims a string	<i>none</i>

# String Manipulation & Basic Functions

## Complementary Matching Functions

Function	Purpose	Characteristic
<code>regmatches()</code>	extract or replace matches	use with data from <code>regexpr()</code> , <code>gregexpr()</code> or <code>regexec()</code>
<code>match()</code>	value matching	finding positions of (first) matches
<code>pmatch()</code>	partial string matching	finding positions
<code>charmatch()</code>	similar to <code>pmatch()</code>	finding positions

# String Manipulation & Basic Functions

## Accessory Functions

Function	Description
<code>apropos()</code>	find objects by (partial) name
<code>browseEnv()</code>	browse objects in environment
<code>glob2rx()</code>	change wildcard or globbing pattern into Regular Expression
<code>help.search()</code>	search the help system
<code>list.files()</code>	list the files in a directory/folder

# Regular expressions

- Regular expressions form a meta-language
- Regular expressions can be thought of as a combination of literals and *metacharacters*
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters
- In two words: **pattern matching**

# Regex metacharacters & functions in R

## Metacharacters and how to escape them in R

Metacharacter	Literal meaning	Escape in R
.	the period or dot	\\.
\$	the dollar sign	\\\$
*	the asterisk or star	\\*
+	the plus sign	\\+
?	the question mark	\\?
	the vertical bar or pipe symbol	\\
\	the backslash	\\\\
^	the caret	\\^
[	the opening square bracket	\\[
]	the closing square bracket	\\]
{	the opening curly bracket	\\{
}	the closing curly bracket	\\}
(	the opening round bracket	\\(
)	the closing round bracket	\\)

# Regex metacharacters & functions in R

## Anchor Sequences in R

Anchor	Description
\d	match a digit character
\D	match a non-digit character
\s	match a space character
\S	match a non-space character
\w	match a word character
\W	match a non-word character
\b	match a word boundary
\B	match a non-(word boundary)
\h	match a horizontal space
\H	match a non-horizontal space
\v	match a vertical space
\V	match a non-vertical space

# Regex metacharacters & functions in R

## Some (Regex) Character Classes

Anchor	Description
[aeiou]	match any one lower case vowel
[AEIOU]	match any one upper case vowel
[0123456789]	match any digit
[0-9]	match any digit (same as previous class)
[a-z]	match any lower case ASCII letter
[A-Z]	match any upper case ASCII letter
[a-zA-Z0-9]	match any of the above classes
[^aeiou]	match anything other than a lowercase vowel
[^0-9]	match anything other than a digit

# Regex metacharacters & functions in R

## POSIX Character Classes in R

Class	Description
[[[:lower:]]]	Lower-case letters
[[[:upper:]]]	Upper-case letters
[[[:alpha:]]]	Alphabetic characters ([[[:lower:]]] and [[[:upper:]]])
[[[:digit:]]]	Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[[[:alnum:]]]	Alphanumeric characters ([[[:alpha:]]] and [[[:digit:]]])
[[[:blank:]]]	Blank characters: space and tab
[[[:cntrl:]]]	Control characters
[[[:punct:]]]	Punctuation characters: ! " # % & ' ( ) * + , - . / : ;
[[[:space:]]]	Space characters: tab, newline, vertical tab, form feed, carriage return, and space
[[[:xdigit:]]]	Hexadecimal digits: 0-9 A B C D E F a b c d e f
[[[:print:]]]	Printable characters ([[[:alpha:]]], [[[:punct:]]] and space)
[[[:graph:]]]	Graphical characters ([[[:alpha:]]] and [[[:punct:]]])

# Regex metacharacters & functions in R

## Quantifiers in R

Quantifier	Description
?	The preceding item is optional and will be matched at most once
*	The preceding item will be matched zero or more times
+	The preceding item will be matched one or more times
{n}	The preceding item is matched exactly n times
{n,}	The preceding item is matched n or more times
{n,m}	The preceding item is matched at least n times, but not more than m times

# Regex metacharacters & functions in R

## Regular Expression Functions in R

Function	Purpose	Characteristic
<code>grep()</code>	finding regex matches	which elements are matched (index or value)
<code>grepl()</code>	finding regex matches	which elements are matched (TRUE & FALSE)
<code>regexpr()</code>	finding regex matches	positions of the first match
<code>gregexpr()</code>	finding regex matches	positions of all matches
<code>regexec()</code>	finding regex matches	hybrid of <code>regexpr()</code> and <code>gregexpr()</code>
<code>sub()</code>	replacing regex matches	only first match is replaced
<code>gsub()</code>	replacing regex matches	all matches are replaced
<code>strsplit()</code>	splitting regex matches	split vector according to matches

# Regex metacharacters & functions in R

## Regex functions in stringr

Function	Description
<code>str_detect()</code>	Detect the presence or absence of a pattern in a string
<code>str_extract()</code>	Extract <b>first</b> piece of a string that matches a pattern
<code>str_extract_all()</code>	Extract <b>all</b> pieces of a string that match a pattern
<code>str_match()</code>	Extract <b>first</b> matched group from a string
<code>str_match_all()</code>	Extract <b>all</b> matched groups from a string
<code>str_locate()</code>	Locate the position of the <b>first</b> occurrence of a pattern in a string
<code>str_locate_all()</code>	Locate the position of <b>all</b> occurrences of a pattern in a string
<code>str_replace()</code>	Replace <b>first</b> occurrence of a matched pattern in a string
<code>str_replace_all()</code>	Replace <b>all</b> occurrences of a matched pattern in a string
<code>str_split()</code>	Split up a string into a variable number of pieces
<code>str_split_fixed()</code>	Split up a string into a fixed number of pieces

## Literals

Simplest pattern consists only of literals. The literal “nuclear” would match to the following lines:

Ooh. I just learned that to keep myself alive after a nuclear blast! All I have to do is milk some rats then drink the milk. Aweosme. :}

Laozi says nuclear weapons are mas macho

Chaos in a country that has nuclear weapons -- not good.

my nephew is trying to teach me nuclear physics, or possibly just trying to show me how smart he is so I'll be proud of him [which I am].

lol if you ever say "nuclear" people immediately think DEATH by radiation LOL

## Literals

The literal “Obama” would match to the following lines

Politics r dum. Not 2 long ago Clinton was sayin Obama  
was crap n now she sez vote 4 him n unite? WTF?  
Screw em both + Mcain. Go Ron Paul!

Clinton conceeds to Obama but will her followers listen??

Are we sure Chelsea didn't vote for Obama?

thinking ... Michelle Obama is terrific!

jetlag..no sleep...early mornig to starbux..Ms. Obama  
was moving

# Regular Expressions

- Simplest pattern consists only of literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want sentences that end in the word “Clinton”, or “clinton” or “clinto”?

# Regular Expressions

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives (“war” or “peace”)

Metacharacters to the rescue!

# Metacharacters

Some metacharacters represent the start of a line

`^i think`

will match the lines

i think we all rule for participating

i think i have been outed

i think this will be quite fun actually

i think i need to go to work

i think i first saw zombo in 1999.

# Metacharacters

\$ represents the end of a line

morning\$

will match the lines

well they had something this morning  
then had to catch a tram home in the morning  
dog obedience school in the morning  
and yes happy birthday i forgot to say it earlier this morning  
I walked in the rain this morning  
good morning

## Character Classes with []

We can list a set of characters we will accept at a given point in the match

[Bb] [Uu] [Ss] [Hh]

will match the lines

The democrats are playing, "Name the worst thing about Bush!"  
I smelked the desert creosote bush, brownies, BBQ chicken  
BBQ and bushwalking at Molonglo Gorge  
Bush TOLD you that North Korea is part of the Axis of Evil  
I'm listening to Bush - Hurricane (Album Version)

## Character Classes with []

`^ [Ii] am`

will match

i am so angry at my boyfriend i can't even bear to  
look at him

i am boycotting the apple store

I am twittering from iPhone

I am a very vengeful person when you ruin my sweetheart.

I am so over this. I need food. Mmmm bacon...

## Character Classes with []

Similarly, you can specify a range of letters [a-z] or [a-zA-Z]; notice that the order doesn't matter

`^ [0-9] [a-zA-Z]`

will match the lines

7th inning stretch

2nd half soon to begin. OSU did just win something

3am - cant sleep - too hot still.. :(

5ft 7 sent from heaven

1st sign of starvagation

## Character Classes with []

When used at the beginning of a character class, the “^” is also a metacharacter and indicates matching characters NOT in the indicated class

[^?.]\$\n

will match the lines that do not end in “?” or “.”

i like basketballs

6 and 9

don't worry... we all die anyway!

Not in Baghdad

helicopter under water? hmmm

# More Metacharacters

“.” is used to refer to any character. So

9.11

will match the lines

its stupid the post 9-11 rules

if any 1 of us did 9/11 we would have been caught in days.

NetBios: scanning ip 203.169.114.66

Front Door 9:11:46 AM

Sings: 0118999881999119725...3 !

## More Metacharacters: |

This does not mean “pipe” in the context of regular expressions; instead it translates to “or”; we can use it to combine two expressions, the subexpressions being called alternatives

flood|fire

will match the lines

is firewire like usb on none macs?

the global flood makes sense within the context of the bible

yeah ive had the fire on tonight

... and the floods, hurricanes, killer heatwaves, rednecks, g

## More Metacharacters: |

We can include any number of alternatives...

flood|earthquake|hurricane|coldfire

will match the lines

Not a whole lot of hurricanes in the Arctic.

We do have earthquakes nearly every day somewhere in our State

hurricanes swirl in the other direction

coldfire is STRAIGHT!

'cause we keep getting earthquakes

## More Metacharacters: |

The alternatives can be real expressions and not just literals

`^ [Gg]ood | [Bb]ad`

will match the lines

good to hear some good knews from someone here

Good afternoon fellow american infidels!

good on you-what do you drive?

Katie... guess they had bad experiences...

my middle name is trouble, Miss Bad News

## More Metacharacters: ( and )

Subexpressions are often contained in parentheses to constrain the alternatives

`^([Gg]ood| [Bb]ad)`

will match the lines

bad habit

bad coordination today

good, because there is nothing worse than a man in kinky under-

Badcop, its because people want to use drugs

Good Monday Holiday

Good riddance to Limey

## More Metacharacters: ?

The question mark indicates that the indicated expression is optional

[Gg]eorge( [Ww]\.)? [Bb]ush

will match the lines

i bet i can spell better than you and george bush combined  
BBC reported that President George W. Bush claimed God told hi  
a bird in the hand is worth two george bushes

## One thing to note...

In the following

```
[Gg]eorge( [Ww]\. )? [Bb]ush
```

we wanted to match a “.” as a literal period; to do that, we had to “escape” the metacharacter, preceding it with a backslash In general, we have to do this for any metacharacter we want to include in our match

## More metacharacters: \* and +

The \* and + signs are metacharacters used to indicate repetition; \* means “any number, including none, of the item” and + means “at least one of the item”

\(.\*\)

will match the lines

anyone wanna chat? (24, m, germany)

hello, 20.m here... ( east area + drives + webcam )

(he means older men)

()

## More metacharacters: \* and +

The \* and + signs are metacharacters used to indicate repetition; \* means “any number, including none, of the item” and + means “at least one of the item”

[0-9]+ (.\*) [0-9]+

will match the lines

working as MP here 720 MP battallion, 42nd birgade  
so say 2 or 3 years at collegae and 4 at uni makes us 23 when  
it went down on several occasions for like, 3 or 4 \*days\*  
Mmm its time 4 me 2 go 2 bed

## More metacharacters: { and }

{ and } are referred to as interval quantifiers; they let us specify the minimum and maximum number of matches of an expression

[Bb]ush( +[^ ]+ +){1,5} debate

will match the lines

Bush has historically won all major debates he's done.  
in my view, Bush doesn't need these debates..

bush doesn't need the debates? maybe you are right

That's what Bush supporters are doing about the debate.

Felix, I don't disagree that Bush was poorly prepared for the  
indeed, but still, Bush should have taken the debate more seri

Keep repeating that Bush smirked and scowled during the debate

## More metacharacters: { and }

- $\{m,n\}$  means at least m but not more than n matches
- $\{m\}$  means exactly m matches
- $\{m,\}$  means at least m matches

## More metacharacters: ( and ) revisited

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a “|”, but also can be used to “remember” text matched by the subexpression enclosed
- We refer to the matched text with \1, \2, etc.

## More metacharacters: ( and ) revisited

So the expression

```
+([a-zA-Z]+) +\1 +
```

will match the lines

time for bed, night night twitter!

blah blah blah blah

my tattoo is so so itchy today

i was standing all all alone against the world outside...

hi anybody anybody at home

estudiando css css css css.... que desastritooooo

## More metacharacters: \* revisited

The \* is “greedy” so it always matches the *longest* possible string that satisfies the regular expression. So

`^s(.*)s`

matches

sitting at starbucks

setting up mysql and rails

studying stuff for the exams

spaghetti with marshmallows

stop fighting with crackers

sore shoulders, stupid ergonomics

## More metacharacters: ( and ) revisited

The greediness of \* can be turned off with the ?, as in

```
^s(.*)?s$
```

# Summary

- Regular expressions are used in many different languages; not unique to R.
- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from “unfriendly” sources (not all data comes as a CSV file)

# Regular Expression Functions

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector; either return the indices into the character vector that match, the strings that happen to match, or a TRUE/FALSE vector indicating which elements match
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices of the string where the match begins and the length of the match
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`, `rematches`: Easier to explain through demonstration.

## grep

Here is an excerpt of the Baltimore City homicides dataset obtained from  
<http://data.baltimoresun.com/homicides/>

```
> homicides <- readLines("homicides.txt")
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl>
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore
21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
> homicides[1000]
[1] "39.33626300000, -76.55553990000, icon_homicide_shooting,
```

How can I find the records for all the victims of shootings (as opposed to other causes)?

## grep

```
> length(grep("iconHomicideShooting", homicides))
[1] 228
> length(grep("iconHomicideShooting|icon_homicide_shooting", homicides))
[1] 1003
> length(grep("Cause: shooting", homicides))
[1] 228
> length(grep("Cause: [Ss]hooting", homicides))
[1] 1003
> length(grep("[Ss]hooting", homicides))
[1] 1005
```

## grep

```
> i <- grep("[cC]ause: [Ss]hootings", homicides)
> j <- grep("[Ss]hootings", homicides)
> str(i)
int [1:1003] 1 2 6 7 8 9 10 11 12 13 ...
> str(j)
int [1:1005] 1 2 6 7 8 9 10 11 12 13 ...
> setdiff(i, j)
integer(0)
> setdiff(j, i)
[1] 318 859
```

# grep

```
> homicides[859]
[1] "39.33743900000, -76.66316500000, icon_homicide_bluntforce
'p914', '<dl><dt><a href=\"http://essentials.baltimoresun.com/
micro_sun/homicides/victim/914/steven-harris\">Steven Harris</
</dt><dd class=\"address\">4200 Pimlico Road<br />Baltimore, M
</dd><dd>Race: Black<br />Gender: male<br />Age: 38 years old<
<dd>Found on July 29, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Blunt Force</dd><dd class=\"popup-note\"><p>Harris
found dead July 22 and ruled a shooting victim; an autopsy
subsequently showed that he had not been shot,...</dd></dl>'"
```

grep

By default, grep returns the indices into the character vector where the regex pattern matches.

```
> grep("New", state.name)
[1] 29 30 31 32
```

Setting value = TRUE returns the actual elements of the character vector that match.

```
> grep("New", state.name, value = TRUE)
[1] "New Hampshire" "New Jersey"     "New Mexico"      "New York"
```

`grepl` returns a logical vector indicating which element matches.

## Some limitations of grep

- The `grep` function tells you which strings in a character vector match a certain pattern but it doesn't tell you exactly where the match occurs or what the match is (for a more complicated regex).
- The `regexpr` function gives you the index into each string where the match begins and the length of the match for that string.
- `regexpr` only gives you the first match of the string (reading left to right). `gregexpr` will give you all of the matches in a given string.

## regexp

How can we find the date of the homicide?

```
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl>
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore
MD 21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
```

Can we just 'grep' on "Found" ?

## regexp

The word 'found' may be found elsewhere in the entry.

```
> homicides[954]
[1] "39.30677400000, -76.59891100000, icon_homicide_shooting,
'<dl><dd class=\"address\">1400 N Caroline St<br />Baltimore,
<dd>Race: Black<br />Gender: male<br />Age: 29 years old</dd>
<dd>Found on March 3, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Shooting</dd><dd class=\"popup-note\"><p>Wheeler\\
was found on the grounds of Dr. Bernard Harris Sr. School</p></dd></dl>'"
```

## regexpr

Let's use the pattern

```
<dd>[F|f]ound(.*)</dd>
```

What does this look for?

```
> regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
[1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
[1] 93 86 89 90 89 84 85 84 88 84
attr(,"useBytes")
[1] TRUE
> substr(homicides[1], 177, 177 + 93 - 1)
[1] "<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd>"
```

## regexpr

The previous pattern was too greedy and matched too much of the string.  
We need to use the ? metacharacter to make the regex “lazy”.

```
> regexpr("<dd>[F|f]ound(.*)?</dd>", homicides[1:10])
[1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
[1] 33 33 33 33 33 33 33 33 33 33
attr(,"useBytes")
[1] TRUE

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"
```

## regmatches

One handy function is `regmatches` which extracts the matches in the strings for you without you having to use `substr`.

```
> r <- regexpr("<dd>[F|f]ound(.*)?</dd>", homicides[1:5])
> regmatches(homicides[1:5], r)
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January
[5] "<dd>Found on January 5, 2007</dd>"
```

## sub/gsub

Sometimes we need to clean things up or modify strings by matching a pattern and replacing it with something else. For example, how can we extract the data from this string?

```
> x <- substr(homicides[1], 177, 177 + 33 - 1)
> x
[1] "<dd>Found on January 1, 2007</dd>"
```

We want to strip out the stuff surrounding the “January 1, 2007” piece.

```
> sub("<dd>[F|f]ound on |</dd>", "", x)
```

```
[1] "January 1, 2007</dd>"
```

```
> gsub("<dd>[F|f]ound on |</dd>", "", x)
```

```
[1] "January 1, 2007"
```

## sub/gsub

sub/gsub can take vector arguments

```
> r <- regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:5])
> m <- regmatches(homicides[1:5], r)
> m
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January
[5] "<dd>Found on January 5, 2007</dd>"
> d <- gsub("<dd>[F|f]ound on |</dd>", "", m)
[1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "Jan
[5] "January 5, 2007"
> as.Date(d, "%B %d, %Y")
[1] "2007-01-01" "2007-01-02" "2007-01-02" "2007-01-03" "2007-
```

## regexec

The `regexec` function works like `regexpr` except it gives you the indices for parenthesized sub-expressions.

```
> regexec("<dd>[F|f]ound on (.*)</dd>", homicides[1])
[[1]]
```

```
[1] 177 190
```

```
attr(),"match.length")
[1] 33 15
```

```
> regexec("<dd>[F|f]ound on .*?</dd>", homicides[1])
[[1]]
```

```
[1] 177
```

```
attr(),"match.length")
[1] 33
```

## regexec

Now we can extract the string in the parenthesized sub-expression.

```
> regexec("<dd>[F|f]ound on (.*)</dd>", homicides[1])
[[1]]
[1] 177 190
attr(,"match.length")
[1] 33 15

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"

> substr(homicides[1], 190, 190 + 15 - 1)
[1] "January 1, 2007"
```

## regexec

Even easier with the `regmatches` function.

```
> r <- regexec("<dd>[F|f]ound on (.*)</dd>", homicides[1:2])
> regmatches(homicides[1:2], r)
[[1]]
[1] "<dd>Found on January 1, 2007</dd>" "January 1, 2007"

[[2]]
[1] "<dd>Found on January 2, 2007</dd>" "January 2, 2007"
```

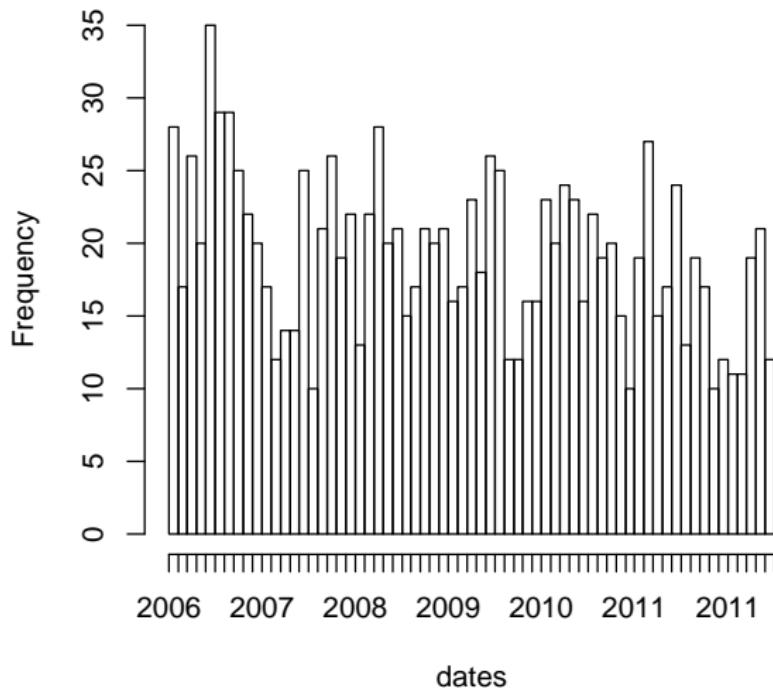
## regexec

```
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl>
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltim
21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
```

Let's make a plot of monthly homicide counts

```
> r <- regexec("<dd>[F|f]ound on (.*)</dd>", homicides)
> m <- regmatches(homicides, r)
> dates <- sapply(m, function(x) x[2])
> dates <- as.Date(dates, "%B %d, %Y")
> hist(dates, "month", freq = TRUE)
```

## Histogram of dates



# Summary

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices where the match begins; useful in conjunction with `regmatches`
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`, `rematches`: Gives you indices of parenthesized sub-expressions.

# LAB EXERCISE

You will work with data from [www.ipaidabribe.com](http://www.ipaidabribe.com). This is a crowdsourced attempt to measure corruption in India. The idea is simple: Every time you pay a bribe, you upload the amount, details of the transaction, and location. This data thus provides an alternative to official government statistics of corruption.

From their website:

*I Paid a Bribe is a site that focuses on crowdsourced reports of corruption and bribery from India and all over the world.*

*The site lays emphasis on the various kinds of Retail Corruption. This is kind of corruption that confronts ordinary citizens in their daily lives when they're not able to avail of services they are legitimately entitled to from the government – getting a driver's license, a birth certificate, registering a purchase of property.*

Their most recent report can be found [here](#). You shall scrape the latest 200 reports from the website and include the following information in a data frame:

- title
- amount
- name of department
- transaction detail
- number of views
- city
- date