Data Mania Inc.

Session Ten - Case Studies

Web Scraping In R

By Hellen Gakuruh
1/5/2017

Session Goal

The main goal of this session is to apply lessons learnt in level one through a series of case studies. We will begin with one case study and subsequent case studies will be added from suggestions made.

Prerequisite

To appreciate this session, you must have gone through all the other sessions (1-9) in level one.

Case Study One:- Importing online data

Task

Using pure base R (no packages), extract 800m and marathon data on recently held (summer) Olympics games and prepare it for data analysis (create data frames).

There are a number of excellent R packages suitable for this task, however, it is best to use packages when we have some basic idea of how it works.

What we shall cover

In addition to reviewing what we have covered in session one, we shall also learn how to:

- Import web data using base R `readLines()`
- Read and understand web data (basic HTML)
- Extract web data (using CSS selectors)

Background and Data

Our first case study is all about extracting online data and making it available for analysis. Basic reason for having this as our first case study is because the web is full of (underutilized) data and most often you will find it necessary to use some of these data.

To make this interesting, we will look at the recent (2016) Summer Olympics games.

As we venture into this case study, some of the topics we will review are creating data objects (data frame) and converting data into a date-time object.

Olympics has quite a number of events, from `Archery` all the way to `Wrestling`. For this exercise, our interest will be `Athletics` and more specifically 800m and Marathon events.

We will scrape our data from Wikipedia; `https://en.wikipedia.org/wiki/800_meteres` and `https://en.wikipedia.org/wiki/Marathon_at_the_Olympics`. From these two pages, we will look for and extract tables giving Olympics medal standing for each event as of 2016 dis-aggregated by gender.

Web scraping

The simplest part of web scraping (data extraction) is actual data importation. This is because in addition to base R's `readLines()` functions, there are contributed packages (from CRAN, and other repositories) with functions capable of achieving this task.

To learn more about what is available in terms of contributed packages, read `https://cran.r-project.org/web/view/WebTechnologies.html`. However, as mentioned before, for this session we will try as much as possible to use only base R functions as we want to get a good understanding of the concepts right from the beginning. This will not only help us understand how contributed packages like *XML* and *rvest* work, but will also ensure we can still extract and traverse through web data when we do not have access to these packages.

With that said, our first task is to download/scrap/import our data.

```
url1 <- "https://en.wikipedia.org/wiki/800_metres"
url2 <- "https://en.wikipedia.org/wiki/Marathon_at_the_Olympics"
wiki800mDOM <- readLines(url1)
wikiMarathonDOM <- readLines(url2)
```

As you have seen, downloading data is rather easy. Now let's find out what data structure is created when we import web data.

```
str(wiki800mDOM)
##  chr [1:2546] "<!DOCTYPE html>" ...
str(wikiMarathonDOM)
##  chr [1:1391] "<!DOCTYPE html>" ...
```

We have character vectors, now let's take a close look at these character.

```
head(wiki800mDOM)
## [1] "<!DOCTYPE html>"
## [2] "<html class=\"client-nojs\" lang=\"en\" dir=\"ltr\">"
## [3] "<head>"
## [4] "<meta charset=\"UTF-8\"/>"
## [5] "<title>800 metres - Wikipedia</title>"
## [6] "<script>document.documentElement.className =
document.documentElement.className.replace( /(^|\\s)client-nojs(\\s|$)/,
\"$1client-js$2\" );</script>"
tail(wiki800mDOM)
## [1] "\t\t\t\t\t\t\t\t\t\t</ul>"
## [2] "\t\t\t\t\t\t<div style=\"clear:both\"></div>"
## [3] "\t\t</div>"
## [4]
"\t\t<script>(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({\"wgB
ackendResponseTime\":1522,\"wgHostname\":\"mw1263\"});});</script>"
## [5] "\t</body>"
## [6] "</html>"
head(wikiMarathonDOM)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html class=\"client-nojs\" lang=\"en\" dir=\"ltr\">"
## [3] "<head>"
## [4] "<meta charset=\"UTF-8\"/>"
## [5] "<title>Marathons at the Olympics - Wikipedia</title>"
## [6] "<script>document.documentElement.className =
document.documentElement.className.replace( /(^|\\s)client-nojs(\\s|$)/,
\"$1client-js$2\" );</script>"
tail(wikiMarathonDOM)
## [1] "\t\t\t\t\t\t\t\t\t\t</ul>"
## [2] "\t\t\t\t\t\t\t<div style=\"clear:both\"></div>"
## [3] "\t\t</div>"
## [4]
"\t\t<script>(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({\"wgB
ackendResponseTime\":182,\"wgHostname\":\"mw1237\"});});</script>"
## [5] "\t</body>"
## [6] "</html>"
```

This is not exactly what we expected. What we wanted to see are headers, paragraphs and tables at the very least. So what happened, did R corrupt our data?

To answer this question, lets take a good look at our wiki pages and determine what content we want. From our 800m webpage, we are interested in the two tables (men and women) following the header "Olympic medalist" and from our marathon page we what the men and women tables under the title "Medal Summary".

Now, right click on either of the pages and scroll to the bottom of the pop-up window, click on *view page source* (or `ctrl+u` from Windows). This will open another window with URL starting with "view-source:", this window can be refereed to as a `source` page. Content of this page looks a lot like R's scraped data, which means R did not corrupted our data, it only shipped in data in a format that we did not expect.

With that, our next logical question is; "Given this format, how can we locate and extract our data (tables)?". To answer this question we need to know what type of data R has scraped. Let's go back to "view-source:" web page and look at line 1. We can see document type is declared as "html" which stands for "hypertext markup language". *Markup* is information and/or instruction added to web content mostly for structure and display purposes. Basically this is what will distinguish a header and a paragraph, usual content and content meant to be part of a table, as well as layout of different content on the webpage. This markup comes in the form of tags; the angle brackets and text () you see on the source page.

So somewhere withing all we see on the source page is our data and we must now figure out how to extract it in a form ready for analysis or reporting. In order for us to do this we need to know a little bit of HTML (just enough to traverse and extract what we need).

Based on this, in our next section we get to understand HTML from a data extraction point of view or the basic concepts (as opposed to a web developers point of view which requires an in-depth understanding).

Understanding HTML

There are three widely used web development languages, these are `XML` (Extensible Markup Language), `HTML` (Hypertext Markup Language) and `XHTM` (Extensible Hypertext Markup Language). Out of these three, HTML is the most widely used web language.

As you can tell from the names of all of these three languages, they are all markup languages, which means they use things called **tags** to give information about a web content (think of markup as a language to a web browser, that is, markup tells a web browser the type of content you have and how to handle it). The core distinguishing feature of these three markup languages are the names given to their tags; for HTML, they are clearly defined, but for XML, they are not defined. XHTML has both defined a undefined tags names, it is more like a cross breed of XML and HTML (All this will become clear in our next section).

Since we are most likely to come across HTML than the other two languages and since our "Wikipedia webpage" is written in HTML, we shall discuss and explore core concepts of HTML but with an understanding that these concepts can easily be extended to XML and XHTML.

Basic HTML

There are two things to grasp as far as HTML for data extraction/scraping is concerned. These are elements and their relationships. If you understand these two concepts, then you can easily traverse through imported data.

Let's take look at each of these concepts in turn, but take note, HTML code can be what I call `general HTML` (omissions allowed by HTML specifications) or strict HTML (parsed HTML). Where necessary, we shall highlight difference between the two.

HTML Elements

HTML elements are single components that build a web page. These single components (elements) are built with `tags`, `attributes` and `text`.

Tags

Tags are `markup` which enclose content with information about the content; information like type of content. There are two types of tags, `opening` and `closing`. Both tags begin with a left angle bracket (<) and end with a right (>) angle bracket. For opening tags, in between the angle brackets are tag name and optional additional information known as `attributes`. For closing brackets, in between the angle brackets is a backslash (`/`) and a tag name (presence of a `/` indicates a closing bracket).

In HTML, tag names are predefined, for example, header begin with letter `h` and end with a number between 1 and 6 which indicate header level. Paragraphs have a `p` tag name, tables have `table` tag name, table rows have a `tr` tag name, and images have a `img` tag name. Check w3schools to see all HTML defined tag names.

Here are examples of opening and closing tags. Take note, HTML comments are added with `/*Comment*/`.

```
Opening tags
=============
<h1>      /*This is a level one header*/
<p>       /*This indicates content is a paragraph*/
<table>   /*This begins a table*/
<a>       /*This is used to add links both internal (from the document) and
external urls*/

Closing tags
============
</h1>     /*This closes a level one header*/
</p>      /*This marks end of a paragraph*/
</table>  /*This ends a table section*/
</a>      /*This ends a given link*/
```

Attributes

Attributes are additional information added to opening tags. This additional information could be an identifier (mostly used for styling or display purposes), or description of content type (for example, `lang` which specifies language used).

Attributes usually come in `name=value` pair, for example `lang="en">`; here "lang" is an attribute name (short for "language") "en" is its value meaning English.

```
Attributes added to opening tags
================================
<h1 class="center">                /*Attribute "class" is used by multiple
html elements to provide uniform styling or identification, in this case all
elements with class center can be center aligned*/
<table id="firstTable">           /*Attribute "id" is a unique html object
identifier, here it maps location of first table*/
<a href="https://myWebPage.org"> /*Attribute "href" creates a hypelink to an
external url*/
```

Both tags and attributes are the markup component of an element, they are not visible on a web page although they are used to change it's appearance or make them dynamic/interactive.

Text

Text is the visible part of a web page; this what you see on web page.

Before we look at examples of HTML elements, let's discuss types of elements as each type will have different markup and text.

Types of HTML element

There are five types of elements (https://www.w3.org/TR/html5/syntax.html#elements-0), two of which are relevant for a web scraper, these are **normal elements** and **void elements**.

Normal elements are composed of markup and text.Markup is not invisible on a web page, it is composed of **tags** and **attributes** which provide details of an element.

```
Normal Elements
===============
<h1>Level 1 header</h1>
<p>A paragraph</p>
<div>A division/section</div>
```

Examples of normal elements from 800m wikipedia web page

```
wiki800mDOM[c(5, 58, 84)]
## [1] "<title>800 metres - Wikipedia</title>"
## [2] "<th scope=\"row\">World</th>"
## [3] "<h2>Contents</h2>"
```

Void elements (also called self-closing elements) are elements which have no text or closing tag, they are basically opening tags. To distinguish these tags from `normal` elements, `strict HTML` includes a forward slash right before closing angle bracket.

```
Void/Self-Closing
================
<br/>       /*Used to add line breaks*/
<meta/>    /*Used to add meta data*/
<link/>   /*Used to add links to url's*/
<image/> /*Used to add images*/
```

Examples of void/self-closing elements from 800m wikipedia web page

```
wiki800mDOM[c(4, 27)]
## [1] "<meta charset=\"UTF-8\"/>"
## [2] "<link rel=\"dns-prefetch\" href=\"//meta.wikimedia.org\" />"
```

There are normal elements called `Empty elements` which are important to know. This is because empty elements contain no text.

```
wiki800mDOM[81]
## [1] "<p></p>"
```

Ideally, all elements that are not defined by W3C Recommendation as void/self-closing elements should have a closing tag (even empty elements). However, the same recommendations allow some elements to make some omission one being closing tags. From a data extraction point of view, omitting closing tags would cause a problem as it would be difficult to map-out beginning and ending of an element.

For void/self-closing elements, the recommendations allows omission of forward slash / right before end angle bracket (>). These omissions are not good when extracting data as the forward slash inform us we are dealing with void/self-closing elements.

Nesting Tags

Tags can embed other tags as it's content, this is refereed to as **nesting**.

```
Nesting Elements
================
<p><dfn id ="myWebPage>My web page is a blog on matters R</dfn></p>
<table>
   <thead><th>Column Head 1</th><th>Column Head 2</th></thead>
   <tbody><td>Cell Input 1</td><td>Cell Input 2</td></tbody>
</table>

wiki800mDOM[194]
## [1] "<td><b>1:56.58</b></td>"
```

Extracted HTML Document

Given the allowance made by HTML syntax specifications, it is quite possible for a HTML script to be parse and loaded on a web page while it contains numerous omissions. It is also possible to parse and load error prone HTML even though specification clearly prohibit certain code. For example, one can easily write text without any markup and it will still be loaded on a web page, for example a script with only "Hello welcome to my webpage".

Basically, HTML is a very relax and user-friendly language. There is nothing like an error in HTML, this is because W3C HTML specification provides corrections for all errors. These corrections are implemented by `HTML parsers` as they are configured to construct syntactically correct HTML code before parsing. All browsers have a HTML parser and follow these specifications when parsing HTML scripts, that is why web pages opened in one browser looks the same in all browser. For example, our wiki pages should look the same on `chome`, `mozilla`, `operamini` or any other browser. However, there a few subtle difference which might not be of concern to us as data extractors.

When a HTML script is parsed, it becomes a `HTML document`. HTML documents are well formed as they use the strict HTML syntax which requires all elements that are not void elements to be closed and void elements to include a forward slash before it's ending angle bracket. From these HTML documents, browsers use `Application Programming Interfaces (API)` called `Document Object Model (DOM)` to create `document objects` which are basically used to structure HTML elements. HTML document objects show how each element is related to another element. It is from these relationship that we will be able to map-out exact location of elements of interest.

Unfortunately, when we read in web data using "readLine()", what we receive is the HTML script used to load the page rather than the parsed HTML document. Hence, it is quite possible to extract data that does not follow strict HTML syntax. Creating our our own HTML parser might be out of scope of this section, there are also contributed packages which have HTML parser. Fortunate for us, Wikipedia pages are `well-formed` as far as closing and self-closing tags are concerned so we can safely proceed.

Having discussed HTML elements, we can now discuss how elements relate to one another.

Relationships among Elements

From our preceding discourse, we know `HTML documents` are `parsed HTML scripts` which are used to generate a `HTML document object`. We also mentioned that HTML document objects show relationship between elements, the question now is: *how can we describe these relationships?*

Relationships among elements is described by how elements are nested. Beginning from the `root` element which is `<html>`, all other elements can be described as either `child`, `sibling`, `descendant` or `ancenstors` (`html` being the `parent`). This description can also be used to describe any other element which becomes a `parent` element. This type of relationship is often refereed to as `familial relationship`.

For example, take the following document object;

```
<html>
   <head>
      <title>Sample web page</title>
   </head>
   <body>
      <h2>Welcome to my webpage</h2>
   </body>
</html>
```

Here we have our root element which is `html`, this element nests four other elements, that is, `head`, `title`, `body`, and `h2`. In terms of familial relationship, we call `html` the parent element while `head` and `body` are it's children, `title` and `h2` are `html's` descendants (but children of head and body respectively).

These relationships can be displayed in a tree-like structure generated by a Document Object Model (DOM). As web scrapers, our interest with DOM are the relationships, we can leave all the other technical bits on DOM implementation.

Now that we know what HTML elements are and how we can refer to them using their relationships, let's look at a basic HTML document just to get a hang of how they look like.

HTML Document's

A basic HTML document is composed of `declaration`, `elements` and `comments`. Declaration is the first line of code in a HTML document, it is used to indicate the html version used (there are currently six versions with HTML5 being the latest). This declaration is referred to as "Document Type Declaration" (DTD). Declarations are encased in `<! >` for example `<!DOCTYPE html>` for HTML5. HTML comments like R comments provide guiding information, in HTML, they begin with `<!--` and end in `-->` for instance "".

Like all other programming scripts, indentations are used to show structure, in HTML indentation is used to show nested elements. For example, in the following HTML snippet, indentation is used to show `li` elements are nested elements of `ol`.

```
<ol>
  <li>Item one</li>
  <li>Item two</li>
  <li>Item three</li>
</ol>
```

Using `familial relationships`, elements on the same line are siblings and the immediate opening tag is their parent.

Strict HTML required all HTML documents to have a `<head>` and `<body>` tag as children of root element `html`. head element is meant to give information about the web document, this element along side it's nested tags are not visible on a web page, visible section of a web page begins from body element.

Before HTML5 came along, it was quite common to find a lot of web documents separating different sections with

tag (div means division), this tag would embed other div tags. This divisions were mostly used for layout purposes. Newer web pages using HTML5 would use self describing tags for different section of a web page like

tag for a section, tag for a navigation panel, and

tag for a side section.

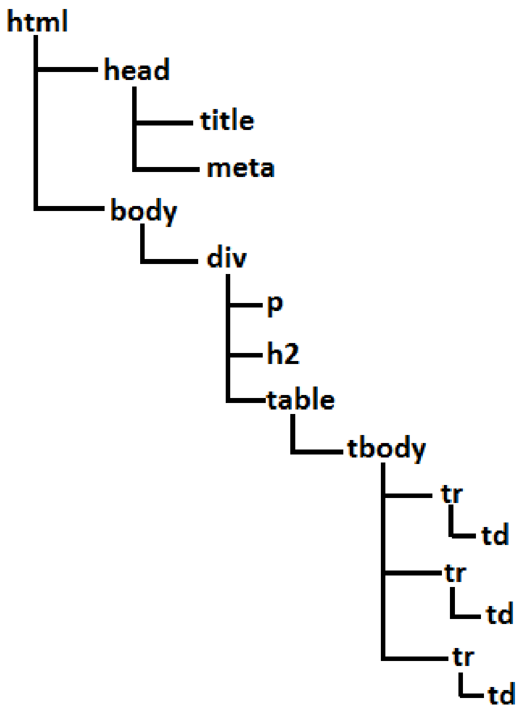
With this brief description of basic HTML web page, take a look at basic HTML document.

HTML code for our sample webpage:

```
<!DOCTYPE html>

<html lang="en-uk">

   <head>  <!--This is not visible on web page-->
      <title>Sample Webpage</title>
      <meta name="author" content="Hellen Gakuruh">
   </head>

   <body>  <!--This begins the visible part of webpage-->
      <div style="margin:0 25%; width:50%; height:520px; border:2px solid
pink; background:#FFB6C1">
         <p style="text-align:center; font-size:450%; font-weight:bold;
color:blue">R</p>
         <h2 style="text-align:center">An Introduction to R</h2>
         <table style="margin-left:50%">
            <tbody>
               <tr><td style="text-align:center; text-decoration:underline;
color:blue">Level 1</td></tr>
```

```
                <tr><td style="text-align:center; text-decoration:underline;
color:blue">Level 2</td></tr>
                <tr><td style="text-align:center; text-decoration:underline;
color:blue">Level 3</td></tr>
            <tbody>
         </table>
      <div>
   </body>

</html>
```

From above HTML script, the first line is declaration of HTML version used, next line is the root element `html`. `html` has two children `head` and `body`, they are therefore siblings. `head` has two children, `title`, and `meta`; these two are siblings and descendants of `html`. `body` has one child, also called an only child, this is the `div` element. `div` has three three children; `p`, `h2` and `table`. The first two children `p` and `h2` have no children, the last child `table` has one child `tbody`. `tbody` has three children `tr` each with one child `td`. These relationships can be shown using a tree like structure as shown below.



Sample of a HTML Tree

This sample web page is quite basic and small, it has only 24 line of code, hence it is easy to locate and extract any part of the web page. However, if you have a larger web page like our two wiki pages (2243 and 1359 lines of code), then it is important to have a technique for extracting the data. One such technique is using CSS (cascading styling sheets) selectors.

Extracting data using CSS selectors

Cascading Styling Sheets (CSS) are code written to add style to web documents. Style can be text alignment, color, background and border features, width and height, or margins. These styles basically describe how HTML elements should be displayed.

CSS selectors are `patterns` used to select HTML elements. Web developers use CSS selectors to style web pages, for us as web scrapers, we use CSS selectors to locate data to extract.

There are two issues we need to grasp as far as CSS selectors for data extraction is concerned. These are `simple selectors` and `combinators`.

Simple Selectors

Simple selectors are search patterns composed of `type`, `universal`, `class`, `id`, `attributes` and `pseudo-class` selectors.

Type Selectors

Type is basically a HTML element name like `p` for paragraphs. Using type as a CSS selector means all elements with given type would be selected, hence for `p`, all paragraphs would be selected. Using our sample web page, if we wanted all content on the table, we can simple use `td` as our CSS selector thereby extracting all elements with a `td` element name.

Universal Selectors

Universal selector denoted with an asterix `*` is used to select all elements or all elements of a specific kind. When `*` is used alone, it selects all elements, for our sample web page this would be all elements from `html`. If it is combined with other simple selectors like type, then it matches all elements of that type. For example, `* p` means all `p` elements. Note, `* p` and `p` are essentially the same. This selector is particularly useful for web developers as it can be used to style an entire web page rather than individual element/sections.

Class and Id Selectors

Class and id are widely used `attributes` selectors. As you may recall, attributes are used to add additional information about an element. A class attribute is used to earmark certain elements for which a common styling will be applied. This means a class attribute can be applied to multiple elements. For example in our wiki800mDOM object, class "plainrowheader" has been used in six table elements (line number 672, 877, 1003, 1117, 1232, 1359).

```
wiki800mDOM[c(672, 877, 1003, 1117, 1232, 1359)]
## [1] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
## [2] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
## [3] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
```

```
## [4] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
## [5] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
## [6] "<table class=\"wikitable plainrowheaders\" border=\"1\" style=\"font-
size: 100%\">"
```

Take note, an element can have more than one class like the table elements above. Classes are separated with a space like class="wikitable plainrowheaders". In terms of data extraction, classes are ideal for extracting multiple elements.

Id's are unique element identifiers. An id name can only be used once in a HTML document, hence these are perhaps the most ideal single element data extractor.

Classes are denoted with a dot `.` while an id is denoted by a hash `#`. To extract all table with class `plainrowheader` we use `.plainrowheaders`. To extract an element with id of "Women_2" we use `#Women_2`.

Attribute Selectors

Other attributes can be matched using attribute selectors. These selectors match elements with a given attribute name, value or value-pair. There are seven way in which an attribute can be specified for possible match, these are `[attr]`, `[attr=value]`, `[attr~=value]`, `[attr|=value]`, `[attr^=value]`, `[attr$=value]`, and `[attr*=value]`. Note that in all of these selectors, square brackets are used to denote it is an attribute selector.

`attr` means attribute name, when specified alone ([attr]), it matches all elements with the given name, this includes classes and id's as they are also attributes. For example we can match all elements with a colspan (meaning column span) attribute with selector `[colspan]`. When a value is included, then matching is more specific for example selector `[colspan=2]` would only match if attribute colspan has a value of 2. When `value` has a number of words separated by a space like attribute `style`, then a particular value from these list of words can be matched using selector `[attr~=value]`. `[attr|=value]` is used to match elements with attribute value pair "attr=value" or attr with a value beginning with given value followed immediately by a hyphen "-" for example `lang=en` or `lang=en-uk`. If only beginning part of an attributes value is known, then `[attr^=value]` can be used to match all elements with given attribute whose value begin with given value. If end of an attributes value is known, then `$` is used instead of `^`. When partial value of an attribute is known and not necessarily at the beginning or end of a value, then `*` is used instead of `^` or `$`. Essentially, any attribute (name-value pair after element name) can be matched with these attribute selectors.

Pseudo-class

Pseudo-class selectors are CSS selectors which match elements at certain state or position. Since our interest is matching specific elements, then we will only discuss pseudo-class selectors that match elements at certain position, these are commonly called `structural pseudo-classes`. Structural pseudo-classes include; `root`, `nth-child`, `nth-last-child`, `nth-of-type`, `nth-last-of-type`, `first-child`, `last-child`, `first-of-type`, `last-of-`

`type`, `only-child`, `only-of-type`, and `empty`. Pseudo-classes are denoted by full colon `:` for example `:root` and `:empty`. To increase specificity (getting exact match), other simple selectors can be added before a pseudo-class selectors like `p:first-child`.

`root` matches root element, in HTML, this is usually `html` element. All pseudo-classes beginning with `nth` match elements at a certain numerical position. These numerical position are stated as a keyword, an exact number or algebraically in the form `an+b`. Keywords can be either `even` or `odd`, for example we can select all even children in a document with `:nth-child(even)` or all odd tables with `table:nth-of-type(odd)`. Exact numbers match exact positions like second table in a document would have this `table:nth-of-type(2)` selector. For algebra expressions, `a` and `b` are integers with `a` being an incrementer, and `b` the stating position. For example, expression `2n+3`, means every second element beginning with the third element. Integers `a` and `b` can be positive or negative, for example `-2n+10` which selects the 10th, 8th, 6th, 4th, 3th, and 2nd element. Using expressions is a great way to target uniquely positioned elements, you can learn more and practice writing expressions from these sites.

Nth-child and nth-of-type can sometimes result in the same element(s), however, nth-child is more specific. Read this article to get a more in-depth distinction between the two.

Examples of Simple Selectors

Selector	What will it select	Expected output from wiki800mDOM
Table	All tables	7, 44, 76, 129, 198, 203, 437, 440, 667, 672, 875, 877, 1000, 1003, 1115, 1117, 1229, 1232, 1357, 1359, 1484, 1490, 1494, 1783, 1787, 2080, 2083, 2095, 2103, 2143, 2257, 2267, 2299, 2329, 2366, 2377, 2508
* meta	All meta elements	4, 13, 15, 16, 27, 2516
.plainrowheader	All elements with that class	672, 877, 1003, 1117, 1232, 1359
#mw-page-base	An element with that id	29
[rel]	All elements with attribute rel	11, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 2087, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2434, 2500, 2524
table.wikitable.plainrowheader	All table elements with both classes	672, 877, 1003, 1117, 1232, 1359

p:nth-of-type(2)	Second `p` element	6
p:nth-child(2)	All `p` elements that are second children of their parent	
table.plainrowheader:nth-of-type(2n)	All even positioned table element with class plainrowheader	877, 1117, 1359

Combinators

Combinators are used to describe relationship between two or more simple selector. There are four types of combinators denoted by `space`, `greater than sign (>)`, `plus (+)` and `tilde (~)`.

A space means simple selector on right side is a **descendant** of the simple sector on the left hand side.

Greater than sign is used to indicate simple selector on right side is a **direct child** of simple selector on the left hand side.

A plus indicates the simple selector on the right hand side is an **adjacent sibling** to the simple selector on the left hand side.

A tilde indicates the right hand selector is a **general sibling** of the simple selector on the left hand side.

Some combinators Examples

Selector: `h2 a` Meaning: Selects any `a` element nested in a `h2` element

```
<h2>
   <span>
      <a href="address">http://someaddress.html</a>
   </span>
</h2>
```

Selector: `h2.firstheader + table` Meaning: Selects a `table` element adjacent to a `h2` element with class "firstheader"

```
<h2 class="firstheader">First Heading</h2>
<table>
```

Selector: `table > tr` Meaning: Selects any `tr` that is directly nested in a table element

```
<table>
   <tr>First tr</tr>      <!-- This will be selected -->
</table>
<table>
```

```
   <thead>
      <tr>Second tr</tr>  <!-- This will not be selected -->
   </thead>
</table>
```

Selector: `h2 ~ p` Meaning: Selects all `p` elements that are sibling of a `h2` element. `p` and `h2` element are on the same hierarchical level.

```
<div>
   <h2>Some header</h2>
   <div>
      <p>First paragraph</p>    <!-- This will not be selected -->
   </div>
   <p>Second paragraph</p>      <!-- This will be selected -->
</div>
```

CSS Selectors for Our Data

From our 800 metres wikipedia web page, we want to extract the two table for men and women below heading "Olympic medalists". From our marathon Wikipedia page, we want tables titled men and women under the header Medal summary. Our challenge is to craft a CSS selectors to extract these table (totaling to 4).

The trick with writing CSS selectors is identify what is common among the elements we want and what is unique to an individual element; things like classes and id are quite handy.

For us to identify these things we need to look at our source document. We can easily print our our two object "wiki800mDOM" and "wikiMarathonDOM", but it will be quite challenging to read the whole script. Best solution is to view the online source document which can be displayed by pressing `ctrl+U` (windows and Linux) if on chrome, if you are using another browser, read this wikihow.

Recall only element nested in `body` element are visible on a web page, therefore for 800 metres, line number 1 up to line number 28 is not visible on the web page, the visible part begins from line number 29. From this point we want to know where our tables are located in order for use to write it's CSS selector. We can go one-by-one matching each element with it's web content, like matching `h1` element on line number 37 with the first header "800 metres". Alternatively, since we know what we seek will be a table element, we can find all table elements by pressing `ctrl+f` on our source document and imputing word `<table`. Since we know our tables fall below header "Olympic medals", we want to click next until we get to a table below a "men" header and preceded by "Olympics medals" header. This should be table 5 (from a total of 17 tables) at line number 672, before this table are two headers `h3` with "men" as it's text and `h2` with "Olympics medal" as it's text. We can move to the next table (6) as we also want to extract "women's" table. Table 6 is at line number 877 right after `h3` with "Women" as it's text.

So now we know where our table are on our source code, question is how best can we identify them using CSS selectors. Note, it is not easy to know which elements are sibling of these table element, or even their parent, so it would not be possible right now to use CSS selectors have "child" keyword, hence we will use something with "type" keyword. With this in mind, we can comfortably use the following CSS selectors to identify our two tables.

```
table:nth-of-type(5)
table:nth-of-type(6)
```

From our marathon web page, our interest are male and female tables showing their medal standing. These are tables 2 at line number 131 and table 5 at line number 660. Using our earlier logic, we can use the following CSS selectors.

```
table:nth-of-type(2)
table:nth-of-type(6)
```

So far we have relied on visual inspection and `ctrl+f` to locate our elements of interest, however, better CSS selectors can be written "inspecting HTML elements" on a web page.

Right click on either of our two web pages, then click on inspect, your window should split into two, one showing the web page and another with HTML element (and a few other things). Now hover on the HTML elements and observe what is highlighted on the web page. You can hover until table of interest is highlighted, when this is done, you should be able so see description of it's CSS selector. In our case proposed CSS selector is `table.wikitable.plainrowheaders`. If you hover all the tables of interest you will see they all have the same CSS selector, this is good if you want to extract all tables at the same time, but if we want to extract one table at time, then our earlier CSS selectors (with pseudo-classes) are preferable.

Another way to identify correct CSS selector is by using a point-and-click applications like SelectorGadget.

Now that we have our CSS selectors, we can use them to extract our tables. We can do this using contributed packages like "rvest" but our agreement was to use only base R functions. To this end, we are going to use functions I created to extract web data. In the future, these functions will form a tutorial package meant to teach web scraping using R. Currently (Jan 2017), these functions are not complete and may not work with combinators and pseudo-classes other than type, however they are adequate for this session. * Keep checking my GitHub page for progress on this tutorial package.

With our CSS selectors, we will use `simpleSelector` function to create data frames of all our tables.

```r
source("webScrapingFunctions.R")
```

800 metres male olympiads

```r
men800m <- simpleSelectors(css = "table:nth-of-type(5)", doc = wiki800mDOM,
constructTable = TRUE)
str(men800m)
## 'data.frame':    28 obs. of  4 variables:
##  $ Games : chr  "1896 Athens; <span" "1900 Paris; <span" "1904 St. Louis;
```

```
<span" "1908 London; <span" ...
##  $ Gold  : chr  " Edwin Flack (AUS)" " Alfred Tysoe (GBR)" " James
Lightbody (USA)" " Mel Sheppard (USA)" ...
##  $ Silver: chr  " Nándor Dáni (HUN)" " John Cregan (USA)" " Howard
Valentine (USA)" " Emilio Lunghi (ITA)" ...
##  $ Bronze: chr  " Dimitrios Golemis (GRE)" " David Hall (USA)" " Emil
Breitkreutz (USA)" " Hanns Braun (GER)" ...
head(men800m)
##                    Games                    Gold                  Silver
## 1    1896 Athens; <span      Edwin Flack (AUS)       Nándor Dáni (HUN)
## 2     1900 Paris; <span     Alfred Tysoe (GBR)       John Cregan (USA)
## 3 1904 St. Louis; <span  James Lightbody (USA)  Howard Valentine (USA)
## 4    1908 London; <span     Mel Sheppard (USA)     Emilio Lunghi (ITA)
## 5 1912 Stockholm; <span     Ted Meredith (USA)      Mel Sheppard (USA)
## 6   1920 Antwerp; <span      Albert Hill (GBR)          Earl Eby (USA)
##                     Bronze
## 1  Dimitrios Golemis (GRE)
## 2         David Hall (USA)
## 3   Emil Breitkreutz (USA)
## 4        Hanns Braun (GER)
## 5      Ira Davenport (USA)
## 6         Bevil Rudd (RSA)
```

800 metres female olympiads

```
female800m   <- simpleSelectors(css = "table:nth-of-type(6)", doc =
wiki800mDOM, constructTable = TRUE)
str(female800m)
## 'data.frame':    17 obs. of  4 variables:
##  $ Games : chr  "1928 Amsterdam; <span" "1932<U+0096>1956""| __truncated__
"1960 Rome; <span" "1964 Tokyo; <span" ...
##  $ Gold  : chr  " Lina Radke (GER)" "not included in the Olympic program"
" Lyudmila Shevtsova (URS)" " Ann Packer (GBR)" ...
##  $ Silver: chr  " Kinuye Hitomi (JPN)" NA " Brenda Jones (AUS)" "
Maryvonne Dupureur (FRA)" ...
##  $ Bronze: chr  " Inga Gentzel (SWE)" NA " Ursula Donath (EUA)" " Marise
Chamberlain (NZL)" ...
head(female800m)
##                      Games                                Gold
## 1   1928 Amsterdam; <span                     Lina Radke (GER)
## 2            1932<U+0096>1956 not included in the Olympic program
## 3        1960 Rome; <span             Lyudmila Shevtsova (URS)
## 4       1964 Tokyo; <span                     Ann Packer (GBR)
## 5 1968 Mexico City; <span               Madeline Manning (USA)
## 6      1972 Munich; <span               Hildegard Falck (FRG)
##                       Silver                   Bronze
## 1       Kinuye Hitomi (JPN)        Inga Gentzel (SWE)
## 2                      <NA>                      <NA>
## 3        Brenda Jones (AUS)       Ursula Donath (EUA)
## 4  Maryvonne Dupureur (FRA)   Marise Chamberlain (NZL)
## 5         Ilona Silai (ROU)          Mia Gommers (NED)
## 6      Nijole Sabaite (URS)  Gunhild Hoffmeister (GDR)
```

Male marathon olympiads

```
menMarathon <- simpleSelectors(css = "table:nth-of-type(2)", doc =
wikiMarathonDOM, constructTable = TRUE)
str(menMarathon)
## 'data.frame':    28 obs. of  4 variables:
##  $ Games : chr  "1896 Athens; <span" "1900 Paris; <span" "1904 St. Louis;
<span" "1908 London; <span" ...
##  $ Gold  : chr  " Spiridon Louis (GRE)" " Michel Théato (FRA)[10]" "
Thomas Hicks (USA)" " Johnny Hayes (USA)" ...
##  $ Silver: chr  " Charilaos Vasilakos (GRE)" " Émile Champion (FRA)" "
Albert Corey (USA)[11]" " Charles Hefferon (RSA)" ...
##  $ Bronze: chr  " Gyula Kellner (HUN)" " Ernst Fast (SWE)" " Arthur Newton
(USA)" " Joseph Forshaw (USA)" ...
head(menMarathon)
##                  Games                    Gold
## 1    1896 Athens; <span      Spiridon Louis (GRE)
## 2     1900 Paris; <span   Michel Théato (FRA)[10]
## 3 1904 St. Louis; <span        Thomas Hicks (USA)
## 4    1908 London; <span        Johnny Hayes (USA)
## 5 1912 Stockholm; <span        Ken McArthur (RSA)
## 6   1920 Antwerp; <span  Hannes Kolehmainen (FIN)
##                      Silver                  Bronze
## 1  Charilaos Vasilakos (GRE)    Gyula Kellner (HUN)
## 2       Émile Champion (FRA)       Ernst Fast (SWE)
## 3     Albert Corey (USA)[11]    Arthur Newton (USA)
## 4     Charles Hefferon (RSA)   Joseph Forshaw (USA)
## 5    Christian Gitsham (RSA)  Gaston Strobino (USA)
## 6         Jüri Lossmann (EST)     Valerio Arri (ITA)
```

Female Marathon Olympiads

```
femaleMarathon <- simpleSelectors(css = "table:nth-of-type(5)", doc =
wikiMarathonDOM, constructTable = TRUE)
str(femaleMarathon)
## 'data.frame':    9 obs. of  4 variables:
##  $ Games : chr  "1984 Los Angeles; <span" "1988 Seoul; <span" "1992
Barcelona; <span" "1996 Atlanta; <span" ...
##  $ Gold  : chr  " Joan Benoit (USA)" " Rosa Mota (POR)" " Valentina
Yegorova (EUN)" " Fatuma Roba (ETH)" ...
##  $ Silver: chr  " Grete Waitz (NOR)" " Lisa Martin (AUS)" " Yuko Arimori
(JPN)" " Valentina Yegorova (RUS)" ...
##  $ Bronze: chr  " Rosa Mota (POR)" " Katrin Dörre (GDR)" " Lorraine Moller
(NZL)" " Yuko Arimori (JPN)" ...
head(femaleMarathon)
##                     Games                  Gold
## 1 1984 Los Angeles; <span         Joan Benoit (USA)
## 2       1988 Seoul; <span           Rosa Mota (POR)
## 3   1992 Barcelona; <span  Valentina Yegorova (EUN)
## 4     1996 Atlanta; <span        Fatuma Roba (ETH)
## 5      2000 Sydney; <span     Naoko Takahashi (JPN)
## 6      2004 Athens; <span      Mizuki Noguchi (JPN)
##                       Silver                   Bronze
## 1          Grete Waitz (NOR)          Rosa Mota (POR)
## 2          Lisa Martin (AUS)       Katrin Dörre (GDR)
## 3         Yuko Arimori (JPN)    Lorraine Moller (NZL)
## 4   Valentina Yegorova (RUS)       Yuko Arimori (JPN)
## 5   Lidia <U+0218>imon (ROU)  Joyce Chepchumba (KEN)
## 6    Catherine Ndereba (KEN)       Deena Kastor (USA)
```

With that, we have good data sets ready for analysis; mission accomplished.

References

Online web sites

1. Introduction to HTML: http://www.w3schools.com/html/html_intro.asp
2. HTML specifications: https://www.w3.org/TR/html5/
3. Introduction to CSS: http://www.w3schools.com/css/css_intro.asp
4. Rvest, a web scraping package: https://cran.r-project.org/web/packages/rvest/index.html

More on Nth Pseudo-class expressions

- How nth-child works by CSS tricks: https://css-tricks.com/how-nth-child-works/
- Understanding nth-child pseudo-class expressions: https://www.sitepoint.com/web-foundations/understanding-nth-child-pseudo-class-expressions/

Exercise

1. Practice scraping other web pages
2. Think of other case studies we can use to practice lessons learnt in level 1