Stat 240 - Lab 06

Dr. Lloyd T. Elliott

March 9, 2020

Web scraping refers to the process of parsing the HTML source code of a web page in order to extract, retrieve, or scrape some specific data. The goal of this lab is to show you different ways to read data files from the web into *R*. The most basic form of web scraping involves grabbing the entire web page and extracting the pieces needed. For basic web scraping tasks, the `readLines()` function from base *R* will usually suffice. We can also use `getURL` function from the R package *RCurl*. The key argument in the `readLines()` function is the connection which specifies the location of the file (here is just the URL of the web page).

Web Scraping

For a single web page, a basic procedure for web scraping in *R* is as follows:

1. View the source code; get familiar with the HTML tags surrounding the data you want
2. Read the web page source code into *R* using `readLines()`
3. Clean the data in *R*

Below is how we use the `readLines()` function and the page URL to import the web page into *R*. For this example, we'll use the SFU course outline webpage. The main advantage is that it provides a very simple query system. For example, `https://www.sfu.ca/outlines.html?2020/spring/stat/240/d100` returns a web page listing course outline information for our class.

```
course_url =
 "https://www.sfu.ca/outlines.html?2020/spring/stat/240/d100"
course_page = readLines(course_url)
```

```
length(course_page)

## [1] 449
```

The output vector will contain as many elements as number of lines in the read file. There 450ish elements in this vector, each of them representing a line in the HTML file. Let's have a look at an arbitrary set of 10 lines in the page: it is basically text information embedded inside HTML elements, with leading white spaces.

```
# arbitrary set of 10 lines of html
trimws(course_page[240:250])
```

```
##  [1] ""
##  [2] "<ul class=\"instructorBlock1\">"
##  [3] "<li class=\"instructor\">"
##  [4] "<h4>Instructor:</h4>"
##  [5] "<a href=\"http://www.sfu.ca/~lloyde/\">Lloyd Elliott</a>"
##  [6] "<br>"
##  [7] "<a href=\"mailto:lloyde@sfu.ca\">lloyde@sfu.ca</a><br>"
##  [8] ""
##  [9] "1 778 782-3376<br>"
## [10] ""
## [11] "Office: SC-K10550<br>"
```

HTML elements are written with a start tag, an end tag, and with the content in between: `<tagname>content</tagname>`.

- `<h1>, <h2>, ...`: Largest heading, second largest heading, etc.
- `<p>`: Paragraph elements
- `<ul>`: Unordered bulleted list
- `<ol>`: Ordered list
- `<li>`: Individual list item
- `<div>`: Division or section
- `<table>`: Table

It is through these tags that we can start to extract textual components of HTML web pages. Now we have to focus on what we are trying to extract. The first step is to find where it is. The page is structured into different sections by headings. Let's extract the elements of the largest heading, which should be surrounded by the HTML tag `<h1>`. We can locate this line using the `grep()` function

```
grep('<h3', course_page)
```

```
## [1] 214 216
```

Turn on `value = T` can help us to visually see the result. To extract the index for the line, we can just omit this argument.

```
trimws(grep('<h3', course_page, value = T))
```

```
## [1] "<h3 id=\"class-number\">Class Number: 3981</h3>"
## [2] "<h3 id=\"delivery-method\">Delivery Method: In Person</h3>"
```

```
heading_index = grep('<h3', course_page)
# note this gives us the same result as using grep(, value = T)
trimws(course_page[heading_index])

## [1] "<h3 id=\"class-number\">Class Number: 3981</h3>"
## [2] "<h3 id=\"delivery-method\">Delivery Method: In Person</h3>"
```

The advantage to extracting the location instead of the value is that we can extract information around that location.

```
heading = course_page[(heading_index[1]-1):(heading_index[2]+1)]
```

What we get are two strings with leading/trailing white spaces. The useful information is embedded in the HTML tags. We can recycle the regular expression from the previous lab to remove the white spaces and HTML tags. For the HTML tags, we can remove each of them separately, or we can remove all the `<...>` tags at once. The first way is somewhat inefficient, while the second way might cause trouble in some cases. My advice here is to try to remove all the `<...>` tags first. If the unexpected results appears, do some fixes. Excess whitespace can be removed with `trimws` and using `gsub` to replace the regular expression `\w+` with `' '` (*i.e.*, double spaces are replaced by single spaces).

Question 1a, (2 points): Write *R* code to download the course outline website for this year's offering of this course and extract all `h3` headings remove all of the HTML formatting and any excess white space (leading and trailing white space and also repeated whitespace characters) from all `h3` headings, and then print out those headings. Provide the *R* code. **Question 1b, (2 points):** Extract the course code from the text of the website `https://www.sfu.ca/outlines.html?2020/spring/stat/240/d100`, and provide the *R* code. Argue that the same code works on the pages for the outlines of other courses (or, modify that your code so that it does). **Question 1c, (6 points):** Write an *R* function called course. This function should take as an argument a string specifying the *URL* of a course outline, and return a list. The list should have two elements, one with name `course` and value given by the course code, and one with name `instructor` and value given by the name of the instructor of the course (with all extreneous whitespace removed). Demonstrate that this code works on a few URLs and provide the code and the demonstration.

Parsing HTML

Tables are pretty common in web pages as data sources. We begin by extracting a simple HTML table from a website. For the first example, we will use `http://www.imdb.com/chart`, which is a box-office summary of the top 10 movies along with their gross profits for the current weekend, and their total gross profit. We would like to make a data frame with that information. Of course we can use the method introduced in the previous section to read the page into *R*, and use the anchor to find the part of the data we want. By doing this, we will have to do extensive data cleaning to extract the information before wrapping it into a data frame. Luckily, *R* has nice packages that can help to

scrape tables from web pages. To read an HTML table we need to use the *R* package `rvest`, which is a convenient wrapper to parse an HTML page and retrieve the table elements.

```
library(rvest)

## Loading required package:  xml2

movie_url = "https://www.imdb.com/chart/boxoffice"

movie_table = read_html("https://www.imdb.com/chart/boxoffice")
length(html_nodes(movie_table, "table"))

## [1] 1

zz = html_table(html_nodes(movie_table, "table")[[1]])

zz[1,c(2,4)]

##    Title  Gross
## 1 Onward $40.0M
```

Question 2a, (2 points): Write an *R* function named `boxoffice`. This function should take no arguments and return a dataframe with columns *Name*, *BoxOffice*, and *PerWeek*, and 10 columns for each of the day's top movies on the website `https://www.imdb.com/chart/boxoffice` (*i.e.*, the function should scrape this website). The *PerWeek* column should contain the gross box office revenu (*i.e.*, the second column) divided by the number of weeks the movie's been running for. Provide the *R* code. **Question 2b, (3 points + 2 bonus points):** Modify the function you wrote in the first part of this question to add a column named `RT`. The values of this column should be the rating that the movie received on `https://www.rottentomatoes.com` (you may use the *tomatometer* or the *audience score*). Note that you will have to construct the URL for the movie in order to scrape it from the rotten tomatoes website. For example, the movie *Onward* is returned in the IMDB boxoffice page with title `Onward` (*i.e.*, with a capital O) and the corresponding rotten tomatoes website is `https://www.rottentomatoes.com/m/onward/` with a lower case o. Provide the modified *R* function. Your solution can be approximate: it need not work for all movie titles (for example, special characters or in the movie title or long movie titles may be challenges, but it's likely that you can get large coverage without handling it). Use NA (not available) to indicate ratings of movies that you can't match. Bonus points will be awarded for exceptionally large coverage.

Shiny Apps at SFU

SFU provides a service to host *R* shiny apps for students. Sign up to the `rcg-shiny-users` mailing list here: `https://maillist.sfu.ca`. This is required to let you use the service (next labs) and it takes a few days to get added after requesting the add.