≡

Web Scraping

Disclaimer

As web scraping involves pulling data directly off a website, its replicable success depends entirely on the webpage in question not changing! The last modification to this document was made March 21, 2017, at which point all these scripts worked. Please contact me (mailto:jerrick@umich.edu) if any of the pages change and break the code.

Introduction

Web scraping is a general term for any sort of procedure that retrieves data stored on the web. This can be a simple as downloading a csv file that's hosted online (E.g. `read.csv("www.website.com/data.csv")`). However, we'll be discussing slightly more interesting versions, where the data is embedded in a website and we want to extract it. Typically this data will already be a table or list, though in theory we could work with more complicated structures.

The real trick of web scraping is that there is no silver bullet. Any web scraping procedure will work only with the page it was designed for[1]. These notes will cover several general approaches which can be used, but if you are scraping your own data, you'll need to start from scratch, figure out which method is appropriate, and implement it (often from scratch).

Manual scraping

The most basic form of scraping (basic in terms of required tools, not simplicity) involves grabbing the entire webpage and extracting the pieces needed. There are two components - obtaining the text of the webpage and extracting only the relevant information. The first piece is very straightforward, but the second can, in some situations, be extremely time-consuming. Modern web design is much more opaque than classic design, not to mention often very poorly done, such that even if the data is clearly laid out on screen, it may not be as clearly laid in the HTML.

Copy and Pasting

No, this isn't a joke. If the data is in a clean table, and small enough that you can hightlight it all, copying and pasting often produces a text document that can be read in easily. For example, Wikipedia tables are easy to approach this way. Consider Comparison of file systems (https://en.wikipedia.org/wiki/Comparison_of_file_systems). If you highlight the General Information table, paste it in a plaintext editor[2], you could load it in R with

```
read.csv("comparison_of_file_systems.txt", sep = "\t")
```

Note that the format is what's known as tsv (tab separated values), which is similar to csv (comma separated values). Recall that "\t" is the escape character for tab.

If the data is not clearly presented (or the webpage is "modern"), or if the data is at all large, this method won't work.

Obtaining the webpage

There are various ways to get the HTML creating a website. The easiest is with the `readLines` function. This function is similar to `read.csv` in that it takes a source of text, but instead of trying to force the results into a table, it stores each line of the file as a separate string, in a character vector.

For this example, we'll use hockey-reference.com's list of NHL career leaders in goals, http://www.hockey-reference.com/leaders/goals_career.html (http://www.hockey-reference.com/leaders/goals_career.html).

```
page <- readLines("http://www.hockey-reference.com/leaders/goals_career.html
head(page)
```

```
## [1] "<!DOCTYPE html>"
## [3] "<head>"
## [5] "    <link rel=\"dns-prefetch\" href=\"https://d2p3bygnnzw9w3.cloudfr
```

Extracting the relevant information

There are *many* ways to do this extraction. This is only one. At this point this is merely an exercise in data cleaning, so whatever tools you use to accomplish this are fine.

If you start looking at `page`, you'll notice a lot of the beginning information is general web stuff - javascript, web SEO tools, the site header itself. We can drop all of that. The top scoring player is Wayne Gretzky. Let's find which entry he's listed at.

```
library(stringr)
str_which(page, "[Gg]retzky")
```

```
## [1]   39   45  504 1267 1518 1627
```

I search for both "gretzky" and "Gretzky" because occasionally capitalization is due to CSS formatting instead of actual saved capitals. In this case it doesn't matter, but I include it as best practices.

We see that Greztky was found in 6 places. Look at the entries around each and see if we've found the right one.

```
page[37:41]
```

```
## [1] "    <meta itemprop=\"name\"          content=\"Hockey Reference\">"
## [2] "    <meta itemprop=\"alternateName\" content=\"HkRef\">"
## [3] "    <meta name=\"Description\" content=\"1. Wayne Gretzky (894), 2.
## [4] "<meta property=\"fb:app_id\"     content=\"212945115395487\">"
## [5] "    <meta property=\"og:url\"          content=\"http://www.hockey-r
```

```
# page[43:47] Not run to save space, since I know it's in the next one...
page[504:510]
```

```
## [1] "</tr></thead><tbody><tr><td class=\"right\">1.</td><td><a href=\"/pl
## [4] "<tr><td class=\"right\">2.</td><td><a href=\"/players/h/howego01.htm
## [7] "<tr><td class=\"right\">3.</td><td><strong><a href=\"/players/j/jagr
```

Let's do the same thing with the last entry, Keith Primeau:

```
str_which(page, "[Pp]rimeau")
```

```
## [1] 1251
```

```
page[1251:1253]
```

```
## [1] "<tr><td class=\"right\">&nbsp</td><td><a href=\"/players/p/primeke02
```

Now we can see that all the data we need is between rows 504 and 1252.

```
page <- page[504:1252]
```

Let's look at the data in more detail.

```
page[1:9]
```

```
## [1] "</tr></thead><tbody><tr><td class=\"right\">1.</td><td><a href=\"/pl
## [4] "<tr><td class=\"right\">2.</td><td><a href=\"/players/h/howego01.htm
## [7] "<tr><td class=\"right\">3.</td><td><strong><a href=\"/players/j/jagr
```

We note a few things.

- Each entry in the table is three strings.
- The third string (`</tr>`) in each entry is uselss.
- The first string in each entry contains the ranking (useless) and the player's name.
- The second string in each entry contains the years played and the total goals.
- Aside from the data itself (and the very first entry), each string across entries is identical.

So this suggests a very natural strategy:

1. Drop the useless third string
2. Link the first and second strings
3. Remove the common text
4. Split into the proper three variables.

Drop the useless third string

Here's two easy ways to do this.

```
page2 <- page[seq_len(length(page)) %% 3 > 0]
# WRONG - see below
#page3 <- str_subset(page, "[^</tr>$]")
#identical(page2, page3)
#page <- page2
```

The first approach creates a logical vector which is TRUE whenever the entry modulo 3 is non-zero, then extracts just those rows.

~~The second approach uses a regular expression which does *not* match `</tr>` at the end of a line. Note:~~

- ~~The `$` to check the end of a line is necessary because the first string of each entry starts with `</tr>` as well.~~
- ~~None of `<`, `/` or `>` are special characters and don't need to be escaped, but it wouldn't hurt to do it to be safe - `"[^\\<\\/tr\\>$]"` would work as well (but don't escape `^` which means "not" or `$` which means end of line, as both should be treated as their special characters instead of strings).~~

Update: The 2nd version as above won't work. Regular expression's "can't"[3] do inverse matching, that is, matching NOT a string. It *can* match not a character (e.g. `"[^a]"` matches any string containing at least one non-"a" character).

The correct 2nd version is to use `str_detect` and `!`:

```
page3 <- page[!str_detect(page, "^</tr>$")]
identical(page2, page3)
```

```
## [1] TRUE
```

```
page <- page2
```

End Update

Separate the first and second strings

We have alternating rows, so cast it as a `matrix`, filling it in by row.

```
pagedata <- matrix(page, ncol = 2, byrow = TRUE)
head(pagedata)
```

```
##      [,1]
## [1,] "</tr></thead><tbody><tr><td class=\"right\">1.</td><td><a href=\"/p
## [2,] "<tr><td class=\"right\">2.</td><td><a href=\"/players/h/howego01.ht
## [3,] "<tr><td class=\"right\">3.</td><td><strong><a href=\"/players/j/jag
## [4,] "<tr><td class=\"right\">4.</td><td><a href=\"/players/h/hullbr01.ht
## [5,] "<tr><td class=\"right\">5.</td><td><a href=\"/players/d/dionnma01.h
## [6,] "<tr><td class=\"right\">6.</td><td><a href=\"/players/e/esposph01.h
```

Remove the common text

Let's consider the first column, which contains the players name.

```
pagedata[1:3, 1]
```

```
## [1] "</tr></thead><tbody><tr><td class=\"right\">1.</td><td><a href=\"/pl
```

We see that the first row has some extraneous tags (it's not uncommon for the first and/or last rows to have a bit extra, make sure to check this). Regardless, the players name is always between `html\">` and `</a>`. We can use `str_replace` to replace those bits with blanks.

```
pagedata[, 1] <- str_replace(pagedata[, 1], "^[:print:]+html\">", "")
pagedata[, 1] <- str_replace(pagedata[, 1], "</a>[:print:]+$", "")
head(pagedata[, 1])
```

```
## [1] "Wayne Gretzky" "Gordie Howe"   "Jaromir Jagr"  "Brett Hull"    "Marc
```

Using `^` and `$` to mark the beginning and end of the lines here isn't neccesary, but it provides a bit of protection against accidentally matching in the middle of a string.

Alternatively, we could have done this by removing all the `<...>` tags:

```
pagedata2 <- matrix(page, ncol = 2, byrow = TRUE)
tmp <- str_replace_all(pagedata2[, 1], "<[^<>]+>", "")
# head(str_replace_all(tmp, "\\*|[0-9].", "")) # WRONG, see update
```

Update: A student pointed out that this actually isn't identical. There are a few ties, such as 22 and 23. When this occurs, the page displays 22. for the first player and a blank for the second. If you look at the code, it's not actually a blank, but `&nbsp`, which is a Non-Breaking SPace.

Also, there's a mistake, as I want to match multiple numbers, and I need to escape the `.` (a `.` matches any character, including non-printable [such as `\t` for tab] so is a more inclusive version of `[:print:]`). A fixed version of the line is

```
tmp <- str_replace_all(tmp, "\\*|[0-9]+\\.|&nbsp", "")
identical(tmp, pagedata[, 1])
```

```
## [1] TRUE
```

End Update

The first line matches any `<...>` tags. By default regexp is "greedy", so it would have matched the entire string! By ensuring that there are some non-`<>` characters between them, we restrict to the actual `<...>` tags. The second removes the extra information, the rank (listed as 1.) and the * which represents Hall of Fame members.

The second column is much the same, except with the added complication that we want to extract two pieces of information from it.

```
pagedata[, 2] <- str_replace(pagedata[, 2], "^<td>", "")
pagedata[, 2] <- str_replace(pagedata[, 2], "</td>[:print:]+\">", ",")
pagedata[, 2] <- str_replace(pagedata[, 2], "</td>$", "")
```

I replace the middle piece with a comma to make my next step a bit easier, …

Split into the proper three variables

… so I can use `str_split` directly.

```
year_goals <- str_split(pagedata[, 2], ",")
goals <- data.frame(player = pagedata[, 1],
                    years = sapply(year_goals, "[", 1),
                    goals = as.numeric(sapply(year_goals, "[", 2)),
                    stringsAsFactors = FALSE)
str(goals)
```

```
## 'data.frame':    250 obs. of  3 variables:
##  $ player: chr  "Wayne Gretzky" "Gordie Howe" "Jaromir Jagr" "Brett Hull"
##  $ years : chr  "1979-99" "1946-80" "1990-17" "1986-06" ...
##  $ goals : num  894 801 763 741 731 717 708 694 692 690 ...
```

```
head(goals)
```

```
##          player   years goals
## 1 Wayne Gretzky 1979-99   894
## 2   Gordie Howe 1946-80   801
## 3  Jaromir Jagr 1990-17   763
## 4    Brett Hull 1986-06   741
## 5 Marcel Dionne 1971-89   731
## 6 Phil Esposito 1963-81   717
```

A second example

Presented with minimal comments, extracting the National Occupation Employment and Wage Estimates table from https://www.bls.gov/oes/current/oes_nat.htm (https://www.bls.gov/oes/current/oes_nat.htm).

```
page <- readLines("https://www.bls.gov/oes/current/oes_nat.htm")

# Since there's more columns than before, lets extract and save the variable
str_which(page, "[Oo]ccupation [Cc]ode")
## [1] 745
str_which(page, "[Mm]ean [Ww]age [RSE|rse]")
## [1] 754
colnames <- page[745:754]
str_which(page, "All Occupations")
## [1] 717 772

# First is links to subtables, so first row is 772
page[768:790]
##  [1] "</thead>"
##  [5] "  <td class=\"bold\" id=\"00-0000\">All Occupations</td>"
##  [9] "  <td>1000.000</td>"
## [13] "  <td>0.1%</td>"
## [17] "  <td class=\"bold\" id=\"11-0000\"><a HREF=\"oes110000.htm\">Manag
## [21] "  <td>50.306</td>"

# Going to start with 770 so each data point as the same entries (capturing
# the "<tr>" and its following uninforative row)
str_which(page, "[Mm]aterial [Mm]oving [Ww]orkers, [Aa]ll [Oo]ther")
## [1] 13828
page[13828:13837]
##  [1] "<td><a HREF=\"oes537199.htm\">Material Moving Workers, All Other</a
##  [6] "  <td>$14.60</td>"
page <- page[770:13837]

# Drop "<tr>" and "</tr>" lines.
page <- str_subset(page, "[^<[/]?tr>$]")

# Get rid of all <td> tags
page <- str_replace_all(str_trim(page), "<[/]?td[^>]*>", "")

# 10 columns of data
mat <- matrix(page, ncol = 10, byrow = TRUE)

# Column 7 and 8 sometimes have (4) or (5), which correspond to special
# cases, which I'll remove in this context.
mat[, 7:8] <- str_replace(mat[, 7:8], "^[:print:]+href[:print:]+$", "NA")
# NA's make them easier to deal with later.

# Column 2 is sometimes a link
mat[, 2] <- str_replace_all(mat[, 2], "<[/]?a[^>]*>", "")

# Convert to data.frame, then to numeric.
saldata <- data.frame(mat, stringsAsFactors = FALSE)
```

```
saldata$X4 <- as.numeric(str_replace_all(saldata$X4, ",", ""))
saldata$X5 <- as.numeric(str_replace(saldata$X5, "%", ""))/100
saldata$X6 <- as.numeric(saldata$X6)
saldata$X7 <- as.numeric(str_replace_all(saldata$X7, "\\$", ""))
## Warning: NAs introduced by coercion
saldata$X8 <- as.numeric(str_replace_all(saldata$X8, "\\$", ""))
## Warning: NAs introduced by coercion
saldata$X9 <- as.numeric(str_replace_all(saldata$X9, "[\\$|,]", ""))
## Warning: NAs introduced by coercion
saldata$X10 <- as.numeric(str_replace(saldata$X10, "%", ""))/100

# Get the names
colnames <- str_replace_all(colnames, "<[/]?th>", "")
colnames[2] <- str_replace(colnames[2], " \\([:print:]+\\)", "")
names(saldata) <- colnames
str(saldata)
## 'data.frame':    1089 obs. of  10 variables:
##  $ Occupation code         : chr  "00-0000" "11-0000" "11-1000" "11-1011
##  $ Occupation title        : chr  "All Occupations" "Management Occupati
##  $ Level                   : chr  "total" "major" "minor" "detail" ...
##  $ Employment              : num  1.38e+08 6.94e+06 2.44e+06 2.39e+05 2.
##  $ Employment RSE          : num  0.001 0.002 0.002 0.007 0.003 0.013 0.
##  $ Employment per 1,000 jobs: num  1000 50.31 17.69 1.73 15.56 ...
##  $ Median hourly wage      : num  17.4 47.4 48.5 84.2 47 ...
##  $ Mean hourly wage        : num  23.2 55.3 59.7 89.3 57.4 ...
##  $ Annual mean wage        : num  48320 115020 124210 185850 119460 ...
##  $ Mean wage RSE           : num  0.001 0.001 0.002 0.004 0.002 0.012 0.
head(saldata)
##   Occupation code               Occupation title  Level Employment Emplo
## 1         00-0000                All Occupations  total  137896660
## 2         11-0000         Management Occupations  major    6936990
## 3         11-1000                 Top Executives  minor    2439900
## 4         11-1011               Chief Executives detail     238940
## 5         11-1021 General and Operations Managers detail    2145140
## 6         11-1031                    Legislators detail      55820
```

Using rvest

Now that we've gone over all that, let's replicate the results in just a few lines of code. The package rvest (https://cran.r-project.org/web/packages/rvest/index.html) by Hadley Wickham automates a lot of this. Note though that a) it is not infallible so you still might need to do the above, and b) it will not do the cleaning we did above.

```
library(rvest)
```

If this works, the basic structure of how it will work is

1. Grab the website using `read_html`.
2. Find the correct label for the table you want by either inspecting the source code or using `html_nodes(..., "table")`.
3. Extract the table you want using `html_nodes(..., "table name")[#]`.
4. Convert to a data frame using `html_table`. (This returns a list, so you'll want `html_table(...)[[1]]`.)

Hockey (again)

Let's replicate the hockey results.

```
page <- read_html("http://www.hockey-reference.com/leaders/goals_career.html
```

From looking at the page, we expect to see two tables.

```
html_nodes(page, "table")
```

```
## {xml_nodeset (2)}
## [1] <table class="suppress_glossary suppress_csv sortable stats_table" id
## [2] <table class="suppress_glossary suppress_csv sortable stats_table" id
```

Thankfully, this is a simple page, and we see only two tables. The `id` argument shows us that we have the NHL and WHA tables, so we can extract either.

```
nhl <- html_table(html_nodes(page, "table")[1])[[1]]
str(nhl)
## 'data.frame':    250 obs. of  4 variables:
##  $ Rank  : chr  "1." "2." "3." "4." ...
##  $ Player: chr  "Wayne Gretzky*" "Gordie Howe*" "Jaromir Jagr" "Brett Hul
##  $ Years : chr  "1979-99" "1946-80" "1990-17" "1986-06" ...
##  $ G     : int  894 801 763 741 731 717 708 694 692 690 ...
head(nhl)
##   Rank         Player   Years   G
## 1   1. Wayne Gretzky* 1979-99 894
## 2   2.   Gordie Howe* 1946-80 801
## 3   3.   Jaromir Jagr 1990-17 763
## 4   4.    Brett Hull* 1986-06 741
## 5   5. Marcel Dionne* 1971-89 731
## 6   6. Phil Esposito* 1963-81 717
```

If desired, we can clean up from here.

Salary data (again)

For the salary data, a similar workflow suffices.

```
page <- read_html("https://www.bls.gov/oes/current/oes_nat.htm")
html_nodes(page, "table")
```

```
## {xml_nodeset (2)}
## [1] <table id="main-content-table"><tr>\n<td id="secondary-nav-td">\r\n\t
## [2] <table class="display sortable_datatable fixed-headers">\n<thead>\n<t
```

There are only two tables listed here. We could examine each and pick the right one, or look at the page source and figure out whether "main-content-table" or "display sortable_datatable fixed-headers" is correct. From either, the second is what we want.

```
salary <- html_table(html_nodes(page, "table")[2])[[1]]
str(salary)
## 'data.frame':    1090 obs. of  10 variables:
##  $ Occupation code                                                        :
##  $ Occupation title (click on the occupation title to view its profile):
##  $ Level                                                                  :
##  $ Employment                                                             :
##  $ Employment RSE                                                         :
##  $ Employment per 1,000 jobs                                              :
##  $ Median hourly wage                                                     :
##  $ Mean hourly wage                                                       :
##  $ Annual mean wage                                                       :
##  $ Mean wage RSE                                                          :
head(salary)
##   Occupation code Occupation title (click on the occupation title to view
## 1
## 2         00-0000                                                       Al
## 3         11-0000                                                Managemen
## 4         11-1000                                                        T
## 5         11-1011                                                      Chi
## 6         11-1021                                           General and Operat
```

Note that this is looking a lot less clean than the hockey data. There's a blank row in the front (usually we can pass `header = TRUE` to `html_table`, but it doesn't work here because of the `NA` in one of the columns), and we need to convert a lot from character to numeric.

Failure of rvest

rvest is not perfect. Here's an example of a page that fails. The Goodreads site strives to be like the IMDB for books. Here's a page that lists the most recent history books added to the site: https://www.goodreads.com/genres/new_releases/history (https://www.goodreads.com/genres/new_releases/history). Notice that the list is stored as images, not text. However, if we examine the html closely, we do see the book information is stored as text, which appears if you mouse-over a book.

rvest fails us here[4]:

```
books <- read_html("https://www.goodreads.com/genres/new_releases/history")
html_nodes(books, "table")
```

```
## {xml_nodeset (0)}
```

However, a manual scrape works fine:

```
page <- readLines("https://www.goodreads.com/genres/new_releases/history")
```

```
## Warning in readLines("https://www.goodreads.com/genres/new_releases/histo
```

The warning here doens't bother us, as the data we need is stored in the middle of the html, not the last line.

Looking at the HTML, we see that each mouseover is inside a <script>. Each script includes the words "readable bookTitle" and that there are 52 total books.

```
length(str_which(page, "readable bookTitle"))
```

```
## [1] 54
```

Looking at the mouseover, we can extract the title, author, rating, number of ratings. The year is irrelevant here because these are all new books, and the description we'll skip.

```
books <- str_subset(page, "readable bookTitle")
books <- str_replace(books, "^[:print:]+readable bookTitle[^>]+>", "")
books <- str_replace(books, "<[:print:]+authorName[^>]+>", "@@@")
```

I use "@@@" as the split between title and author because titles may have commas in them. You can use anything here that you're sure won't be in a title.

It tricky to clean up around the rating, so lets instead extact it.

```
ratings <- as.numeric(str_extract(books, "[0-9]\\.[0-9]{2}"))
numratings <- str_extract(books, "[0-9]+ ratings")
numratings <- as.numeric(str_replace(numratings, " ratings", ""))
```

```
books <- str_replace(books, "<[:print:]+$", "")
bookdata <- data.frame(title = sapply(str_split(books, "@@@"), "[", 1),
                       author = sapply(str_split(books, "@@@"), "[", 2),
                       rating = ratings,
                       numrating = numratings,
                       stringsAsFactors = FALSE)
str(bookdata)
```

```
## 'data.frame':    54 obs. of  4 variables:
##  $ title    : chr  "Age of Anger: A History of the Present" "Cannibalism:
##  $ author   : chr  "Pankaj Mishra" "Bill Schutt" "Anders Rydell" "Damion
##  $ rating   : num  3.75 4.1 3.93 3.85 3.82 4.28 3.64 4.1 4.14 3.89 ...
##  $ numrating: num  145 229 98 67 179 133 69 94 146 54 ...
```

Data Spread Over Multiple pages

Often data will not be limited to a single page, but spread across many. This could be a separate page per year/date or just a limit on the number of results listed per page.

As an example, let's try and collect the salary data for all employees of Tennessee public schools: https://www.tbr.edu/hr/salaries (https://www.tbr.edu/hr/salaries).

We see the first 50 individuals listed here with no evidence of how to list more individuals on the single page. However, if we go to the second page of results (the pages are listed below the data), we see the URL changes to https://www.tbr.edu/hr/salaries?firstname=&lastname=&department=&jobtitle=&institution=&page=1 (https://www.tbr.edu/hr/salaries?firstname=&lastname=&department=&jobtitle=&institution=&page=1). This is common in webpage design. Each of those arguments (`firstname=` and `lastname=` etc) could be non-blank (try it: restrict the results by using `lastname=smith`).

The most interesting argument for our purposes is `page=1`. Changing that to `page=0` gives us the first page of results[5], and clicking to the the "last" results gives `page=323`. We can therefore use rvest and scrape all these pages simultaneously.

```
tmp <- lapply(0:323, function(i) {
  url <- str_c("https://www.tbr.edu/hr/salaries?firstname=&lastname=&departm
  page <- read_html(url)
  html_table(html_nodes(page, "table"))[[1]]
})
tnsal <- do.call("rbind", tmp)
```

```
str(tnsal)
## 'data.frame':    16151 obs. of  7 variables:
##  $ Institution: chr  "Middle TN State University" "Walters State Comm Col
##  $ Last Name  : chr  "Aaron" "Aarons" "Abadie" "Abbott" ...
##  $ First Name : chr  "Joshua" "Andrew" "Cynthia" "Joyce" ...
##  $ Job Title  : chr  "Associate Professor" "Associate Professor" "Associa
##  $ Department : chr  "Management" "Industrial Technology" "Business and L
##  $ Salary     : chr  "$100,350" "$59,583" "$47,868" "$21,684" ...
##  $ FTE        : num  1 1 1 0.8 1 1 1 1 1 1 ...
tail(tnsal, 1)
##        Institution Last Name First Name                      Job Title
## 16151 TCAT Dickson   Zwingle    Cynthia Administrative Support Assoc 6 St
```

Note that this can be quite slow if the website you are loading is slow (even an impercetible sluggishness on the site will blow up here; each page takes just short of 800ms to load, over 324 pages this becomes over 4 minutes).

APIs and JSON

API

An API (Application Programming Interface) is a set methods to access data which is not publically available as a complete data set. For example, if you wanted to get access to Google search results, you'd use the Google JSON/Atom Custom Search API (https://developers.google.com/custom-search/json-api/v1/overview) or if you wanted to get a list of Yelp businesses, you'd use the Yelp Fusion API (https://www.yelp.com/developers). These are most commonly used by programmers (e.g. if you use any third-party apps to connect to resources such as Facebook, Twitter, etc; the developers of those apps use APIs to access your information) but are also useful to extract data.

A lot of APIs are private - either they're only available to people who are authorized to use them, or they're used internally by development teams. However, a good number are public, but may require purchasing access or at least registering to get an API key. There's a useful list of public APIs (https://github.com/toddmotto/public-apis) maintained by Todd Motto. A lot of the truely open APIs are done by fans and volunteers as opposed to the official company released ones.

We've technically already seen a (basic and informal) API. The Tennessee salary url, "https://www.tbr.edu/hr/salaries?firstname=&lastname=&department=&jobtitle=&institution=&page=1" is an example. We could create an R function to generate that URL and extract just the data we need:

```
scrapeTN <- function(firstname = "",
                     lastname = "",
                     department = "",
                     jobtitle = "",
                     institution = "",
                     page = 1) {
  url <- str_c("https://www.tbr.edu/hr/salaries?",
               "firstname=", firstname,
               "&lastname=", lastname,
               "&department=", department,
               "&jobtitle=", jobtitle,
               "&institution=", institution,
               "&page=", page)
  page <- read_html(url)
  html_table(html_nodes(page, "table"))[[1]]
}
scrapeTN(lastname = "Horton")
```

```
##                  Institution  Last Name First Name                  Jo
## 1  East Tennessee State Univ Fox-Horton      Julie          Assistant Pr
## 2      University of Memphis     Horton      Cathy            Cares Coun
## 3      University of Memphis     Horton    Stephen          Research Sc
## 4      University of Memphis     Horton      Tracy   Asst Dir Empl Rel An
## 5      University of Memphis     Horton    Whitney    Undergraduate Adms 
## 6 Jackson State Comm College     Horton      Sonya Clerk, Learning Resour
## 7      TN Technological Univ     Horton      Laura                  Coor
## 8 Tennessee State University     Horton     Dianne            Records As
```

Rate Limits

Note that a lot of API's have rate limits - the number of requests you can send in a particular window of time (e.g. 100 requests per hour). Before spending a lot of time scraping, make sure you know the rate limit and structure your requests appropriately. You may need to space your scraping out over a couple of days.

JSON

A lot of formal APIs return data in a format known as JSON (JavaScript Object Notation). It is very similar to `list`s in R. An example modified from Wikipedia:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "children": [],
  "spouse": null
}
```

We don't have to parse this ourselves, thanks to the jsonlite (https://cran.r-project.org/package=jsonlite) package by Jeroen Ooms and others. We will obtain the JSON data from an API query and use the `fromJSON` function.

Example - Magic: The Gathering Cards

Let's look at a complete example. The examples are limited to those with completely open APIs, so the sources are somewhat limited.

Magic The Gathering is a "trading card game" that's been around since the 1990's. It involves two players crafting a deck of cards, from which a hand is drawn, and on each turn a player can play creatures or spells to attempt to reduce the life total of their opponent. There are over 15,000 different cards printed.

The Magic: The Gathering Developers site http://magicthegathering.io/ (http://magicthegathering.io/) provides an unofficial API to the cards. The general URL is "https://api.magicthegathering.io/v1/cards (https://api.magicthegathering.io/v1/cards)" and it follows with query parameters such as "power=gt5" for cards with attack power greater than 5. The full list of query parameters is available in the documentation (https://docs.magicthegathering.io/).

Building an appropriate query can be done exactly as before. The only change is we pass the results through `fromJSON` rather than `read_html`. Here we obtain a list of all Green[6] creatures with mana cost 1[7] and are rare[8] and banned[9].

```
url <- str_c("http://api.magicthegathering.io/v1/cards?",
             "legality=banned&",
             "type=Creature&",
             "rarity=rare&",
             "colors=green&",
             "cmc=1")
library(jsonlite)
mtgbanned <- fromJSON(url)[[1]]
dim(mtgbanned)
## [1] 56 32
mtgbanned[1,1:10]
##                name manaCost cmc colors colorIdentity            type
## 1 Birds of Paradise      {G}   1  Green             G Creature — Bird Cre
```

If you read the documentation, you will see that the rate limit is 5000 requests per hour which is plenty high, but each request is limited to 100 cards returned. Similar to the Tennessee salary example, to obtain them all you'd need to iterate over query parameter `page=` until the returned entry was empty:

```
url <- str_c(url, "&page=1000")
fromJSON(url)[[1]]
```

```
## list()
```

Example - Currency conversion

The site fixer.io (http://fixer.io/) provides an API to obtain daily currency conversion rates. The URL differs depending on the date desired; current results are "http://api.fixer.io/latest (http://api.fixer.io/latest)" whereas historical data is "http://api.fixer.io/2005-04-20 (http://api.fixer.io/2005-04-20)". Query parameters include `base=` to get the comparison (default is "EUR".)

```
f <- fromJSON("http://api.fixer.io/2013-10-20?base=USD")
f[[1]]
## [1] "USD"
f[[2]]
## [1] "2013-10-18"
head(f[[3]])
## $AUD
## [1] 1.0357
##
## $BGN
## [1] 1.4293
##
## $BRL
## [1] 2.152
##
## $CAD
## [1] 1.0296
##
## $CHF
## [1] 0.902
##
## $CNY
## [1] 6.097
```

---

1. Almost Surely in my experience ↩
2. **NOT** Word ↩
3. It does via something called "negative lookahead" which is far more complicated than we need for this context, so we'll just say it's true that it can't. ↩
4. It may be possible that there is a complicated secondary way to use rvest, specifically `html_text`. But at that point, rvest provides no benefit over doing it manually since we're still cleaning up everything ourselves. ↩
5. Like a lot of computer programming, this is using zero-based numbering (https://en.wikipedia.org/wiki/Zero-based_numbering). ↩
6. There are five "colors of magic", red, green, blue, black and white. ↩

7. Each card takes “mana” to cost.↩
8. In terms of how common the card is when opening packs.↩
9. Not legal for tournament play - generally overpowered cards.↩