# Webscraping and Parallel Processing

*Greg Ridgeway (gridge@upenn.edu)*
*Ruth Moyer (moyruth@upenn.edu)*

*February 13, 2019*

## Introduction

At the end of our discussion about regular expressions, we introduced the concept of web scraping. Not all online data is in a tidy, downloadable format such as a .csv or .RData file. Yet, patterns in the underlying HTML code and regular expressions together provide a valuable way to "scrape" data off of a webpage. Here, we're going to work through an example of webscraping. We're going to get data on ticket sales of every movie, for every day going back to 2010.

As a preliminary matter, some R packages, such as `rvest`, can help with web scraping. Eventually you may wish to explore webscraping packages. For now, we are going to work with basic fundamentals so that you have the most flexibility to extract data from most websites.

First, you will need to make sure that you can access the underlying HTML code for the webpage that you want to scrape. If you're using Firefox, you can simply right click on a webpage and then click "View Page Source." If you're using Microsoft Edge, you can right click on the webpage, click "View Source" and then look at the "Debugger" tab. In Safari select "Preferences" under the "Safari" tab, select the "Advanced" tab, check "Show Develop menu", and then whenever viewing a page you can right click and select "show page source".

Have a look at the webpage http://www.the-numbers.com/box-office-chart/daily/2018/07/04. This page contains information about the movies that were shown in theaters on July 4, 2018 and the amount of money (in dollars) that each of those movies grossed that day.

Have a look at the HTML code by looking at the page source for this page using the methods described above. The first 10 lines should look something like this:

```
<!DOCTYPE html>
<html>
<head>
<!-- Global site tag (gtag.js) - Google Analytics -->
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-1343128-1"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
  gtag('config', 'UA-1343128-1');
```

This is all HTML code to set up the page. If you scroll down a few hundred lines, you will find code that looks like this:

```
<TR><TH>&nbsp;</TH><TH>&nbsp;</TH><TH>Movie</TH><TH>Distributor</TH><TH>Gross</TH><TH>Chang
<tr>
<td  class="data">1</td>
```

```
<td  class="data">(1)</td>
<td><b><a href="/movie/Jurassic-World-Fallen-Kingdom-(2018)#tab=box-office">Jurassic World:
<td><a href="/market/distributor/Universal">Universal</a></td>
<td class="data">$11,501,395</td>
<td class="data chart_down">-3%</td>
<td  class="data">4,485</td>
<td class="data chart_grey">$2,564</td>
<td class="data">&nbsp;&nbsp;$297,672,320</td>
<td class="data">13</td>
</tr>
<tr bgcolor=#ffeeff>
<td  class="data">2</td>
<td  class="data">(2)</td>
<td><b><a href="/movie/Incredibles-2#tab=box-office">Incredibles 2</a></b></td>
```

Here you can see the data! You can see the movie name, ticket sales, number of theaters, and more. It's all wrapped in a lot of HTML code to make it look pretty on a web page, but for our purposes we just want to pull those numbers out.

`scan()` is a basic R function for reading in text, from the keyboard, from files, from the web, … however data might arrive. Giving `scan()` a URL causes `scan()` to pull down the HTML code for that page and return it to you. Let's try one page of movie data.

```
a <- scan("http://www.the-numbers.com/box-office-chart/daily/2018/07/04",
          what="", sep="\n")
# examine the first few lines
a[1:5]
```

```
[1] "<!DOCTYPE html>"
[2] "<html>"
[3] "<head>"
[4] "<!-- Global site tag (gtag.js) - Google Analytics -->"
[5] "<script async src=\"https://www.googletagmanager.com/gtag/js?id=UA-1343128-1\"></script>"
```

`what=""` tells `scan()` to expect plain text and `sep="\n"` tells `scan()` to separate each element when it reaches a line feed character, signaling the end of a line.

Some websites are more complex or use different text encoding. For example, using `scan()` on https://stores.org/stores-top-retailers-2017/ produces unintelligible text. The `GET()` function from the `httr` package can sometimes resolve this.

```
library(httr)
```

```
Warning: package 'httr' was built under R version 3.5.2
```

```
resp <- GET("https://stores.org/stores-top-retailers-2017/")
a1 <- content(resp, as="text")
a1 <- strsplit(a1,"\n")[[1]]
cat(a1[1:10], sep="\n")
```

```
<!DOCTYPE html>
```

```
<!--[if IE 8]> <html class="ie ie8" lang="en-US" prefix="og: http://ogp.me/ns#"> <![endif]-->
<!--[if IE 9]> <html class="ie ie9" lang="en-US" prefix="og: http://ogp.me/ns#"> <![endif]-->
<!--[if gt IE 9]><!--> <html lang="en-US" prefix="og: http://ogp.me/ns#"> <!--<![endif]-->

<head>


<meta charset="UTF-8" />
```

Also, some Mac users will encounter snags with both of these methods and receive "403 Forbidden" errors while their Mac colleague right next to them on the same network will not. I have not figured out why this happens, but have found that making R masquerade as different browser sometimes works.

```
resp <- GET("https://stores.org/stores-top-retailers-2017/",
            user_agent("Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/537.13+ (KHTM
a1 <- content(resp, as="text")
a1 <- strsplit(a1,"\n")[[1]]
```

## Scraping one page

Now that we have stored in the variable a the HTML code for one day's movie data in R, let's apply some regular expressions to extract the data. The HTML code includes a lot of lines that do not involve data that interests us. There is code for making the page look nice and code for presenting advertisements. Let's start by finding the lines that have the movie names in them.

Going back to the HTML code, I noticed that both the line with *Jurassic Park* and *Incredible 2* have the sequence of characters "#tab=box-office". By finding a pattern of characters that always precedes the text that interests us, we can use it to grep the lines we want. Let's find every line that has "#tab=box-office" in it.

```
i <- grep("#tab=box-office", a)
i
```

```
 [1] 220 232 244 256 268 280 292 304 316 328 340 352 364 376 388 400 412
[18] 424 436 448 460 472 484 496 508 520 532 544 556 568 580 592 604 616
[35] 628 640 652 664 676 688 700 712 724 736 748 760 772 784
```

These are the line numbers that, if the pattern holds, contain our movie titles. Note that the source code that you might see in your browser may be a little different from the line numbers you see here. Even if you run this code on a different day, you might get different line numbers because some of the code, code for advertisements in particular, can frequently change.

Let's see what these lines of HTML code look like.

```
a[i]
```

```
 [1] "<td><b><a href=\"/movie/Jurassic-World-Fallen-Kingdom-(2018)#tab=box-office\">Jurassic Wor
 [2] "<td><b><a href=\"/movie/Incredibles-2#tab=box-office\">Incredibles 2</a></b></td>"
 [3] "<td><b><a href=\"/movie/First-Purge-The#tab=box-office\">The First Purge</a></b></td>"
 [4] "<td><b><a href=\"/movie/Sicario-Day-of-the-Soldado#tab=box-office\">Sicario: Day of the So
 [5] "<td><b><a href=\"/movie/Uncle-Drew#tab=box-office\">Uncle Drew</a></b></td>"
 [6] "<td><b><a href=\"/movie/Oceans-8-(2018)#tab=box-office\">Oceanâ\200\231s 8</a></b></td>"
 [7] "<td><b><a href=\"/movie/Tag-(2018)#tab=box-office\">Tag</a></b></td>"
 [8] "<td><b><a href=\"/movie/Wont-You-Be-My-Neighbor-(Documentary)-(2018)#tab=box-office\">Wonâ
 [9] "<td><b><a href=\"/movie/Deadpool-2#tab=box-office\">Deadpool 2</a></b></td>"
[10] "<td><b><a href=\"/movie/Solo-A-Star-Wars-Story#tab=box-office\">Solo: A Star Wars Story</a
[11] "<td><b><a href=\"/movie/Sanju-(India)#tab=box-office\">Sanju</a></b></td>"
[12] "<td><b><a href=\"/movie/Avengers-Infinity-War#tab=box-office\">Avengers: Infinity War</a><
[13] "<td><b><a href=\"/movie/Hereditary-(2018)#tab=box-office\">Hereditary</a></b></td>"
[14] "<td><b><a href=\"/movie/SuperFly-(Remake)#tab=box-office\">Superfly</a></b></td>"
[15] "<td><b><a href=\"/movie/Book-Club-(2018)#tab=box-office\">Book Club</a></b></td>"
[16] "<td><b><a href=\"/movie/Adrift-(2018)#tab=box-office\">Adrift</a></b></td>"
[17] "<td><b><a href=\"/movie/American-Animals-(2018)#tab=box-office\">American Animals</a></b><
[18] "<td><b><a href=\"/movie/Gotti-(2018)#tab=box-office\">Gotti</a></b></td>"
[19] "<td><b><a href=\"/movie/Leave-No-Trace-(2018)#tab=box-office\">Leave No Trace</a></b></td>
[20] "<td><b><a href=\"/movie/Three-Identical-Strangers-(2018)#tab=box-office\">Three Identical
[21] "<td><b><a href=\"/movie/First-Reformed#tab=box-office\">First Reformed</a></b></td>"
[22] "<td><b><a href=\"/movie/Rampage-(2018)#tab=box-office\">Rampage</a></b></td>"
[23] "<td><b><a href=\"/movie/Overboard-(Remake)-(2018)#tab=box-office\">Overboard</a></b></td>"
[24] "<td><b><a href=\"/movie/Catcher-Was-A-Spy-The-(2018)#tab=box-office\">The Catcher Was A Sp
[25] "<td><b><a href=\"/movie/Quiet-Place-A-(2018)#tab=box-office\">A Quiet Place</a></b></td>"
[26] "<td><b><a href=\"/movie/Breaking-In-(2018)#tab=box-office\">Breaking In</a></b></td>"
[27] "<td><b><a href=\"/movie/Upgrade#tab=box-office\">Upgrade</a></b></td>"
[28] "<td><b><a href=\"/movie/Show-Dogs-(2018)#tab=box-office\">Show Dogs</a></b></td>"
[29] "<td><b><a href=\"/movie/Ready-Player-One#tab=box-office\">Ready Player One</a></b></td>"
[30] "<td><b><a href=\"/movie/Pandas-(Documentary)#tab=box-office\">Pandas</a></b></td>"
[31] "<td><b><a href=\"/movie/2001-A-Space-Odyssey#tab=box-office\">2001: A Space Odyssey (1968&
[32] "<td><b><a href=\"/movie/Life-of-the-Party#tab=box-office\">Life of the Party</a></b></td>"
[33] "<td><b><a href=\"/movie/Black-Panther#tab=box-office\">Black Panther</a></b></td>"
[34] "<td><b><a href=\"/movie/Isle-of-Dogs-(2018)#tab=box-office\">Isle of Dogs</a></b></td>"
[35] "<td><b><a href=\"/movie/I-Can-Only-Imagine#tab=box-office\">I Can Only Imagine</a></b></td
[36] "<td><b><a href=\"/movie/Wrinkle-in-Time-A-(2018)#tab=box-office\">A Wrinkle in Time</a></b
[37] "<td><b><a href=\"/movie/Uchiage-Hanabi-Shita-Kara-Miru-ka-Yoko-Kara-Miru-ka-(Japan)#tab=bo
[38] "<td><b><a href=\"/movie/Disobedience-(2018)#tab=box-office\">Disobedience</a></b></td>"
[39] "<td><b><a href=\"/movie/Eating-Animals-(Documentary)#tab=box-office\">Eating Animals</a></
[40] "<td><b><a href=\"/movie/Hotel-Artemis-(UK)#tab=box-office\">Hotel Artemis</a></b></td>"
[41] "<td><b><a href=\"/movie/Death-of-Stalin-The-(UK)#tab=box-office\">The Death of Stalin</a><
[42] "<td><b><a href=\"/movie/Miracle-Season-The#tab=box-office\">The Miracle Season</a></b></td
[43] "<td><b><a href=\"/movie/Beast-(2017-United-Kingdom)#tab=box-office\">Beast</a></b></td>"
[44] "<td><b><a href=\"/movie/On-Chesil-Beach#tab=box-office\">On Chesil Beach</a></b></td>"
[45] "<td><b><a href=\"/movie/Beau-Soleil-Interieur-Un-(France)#tab=box-office\">Let The Sunshin
[46] "<td><b><a href=\"/movie/Super-Troopers-2#tab=box-office\">Super Troopers 2</a></b></td>"
[47] "<td><b><a href=\"/movie/Summer-of-67#tab=box-office\">Summer of 67</a></b></td>"
[48] "<td><b><a href=\"/movie/Chappaquiddick#tab=box-office\">Chappaquiddick</a></b></td>"
```

Double checking and indeed the first line here is *Jurassic Park: Fallen Kingdom* and the last line is *Chappaquiddick*. This matches what is on the web page. We now are quite close to having a list of movies that played in theaters on July 4, 2018. However, as you can see, we have a lot of excess symbols and HTML code to eliminate before we can have a neat list of movie names.

HTML tags are always have the form `<some code here>`. Therefore, any text between a less than and greater than symbol we should remove. Here's a regular expression that will look for a < followed by a bunch of characters that are not > followed by the HTML tag ending >... and `gsub()` will delete them.

```
gsub("<[^>]*>", "", a[i])
```

```
 [1] "Jurassic World: Fallen Kingdom"
 [2] "Incredibles 2"
 [3] "The First Purge"
 [4] "Sicario: Day of the Soldado"
 [5] "Uncle Drew"
 [6] "Oceanâ\200\231s 8"
 [7] "Tag"
 [8] "Wonâ\200\231t You Be My Neighbor?"
 [9] "Deadpool 2"
[10] "Solo: A Star Wars Story"
[11] "Sanju"
[12] "Avengers: Infinity War"
[13] "Hereditary"
[14] "Superfly"
[15] "Book Club"
[16] "Adrift"
[17] "American Animals"
[18] "Gotti"
[19] "Leave No Trace"
[20] "Three Identical Strangers"
[21] "First Reformed"
[22] "Rampage"
[23] "Overboard"
[24] "The Catcher Was A Spy"
[25] "A Quiet Place"
[26] "Breaking In"
[27] "Upgrade"
[28] "Show Dogs"
[29] "Ready Player One"
[30] "Pandas"
[31] "2001: A Space Odyssey (1968&hellip;"
[32] "Life of the Party"
[33] "Black Panther"
[34] "Isle of Dogs"
[35] "I Can Only Imagine"
[36] "A Wrinkle in Time"
[37] "Fireworks"
[38] "Disobedience"
[39] "Eating Animals"
```

```
[40] "Hotel Artemis"
[41] "The Death of Stalin"
[42] "The Miracle Season"
[43] "Beast"
[44] "On Chesil Beach"
[45] "Let The Sunshine In"
[46] "Super Troopers 2"
[47] "Summer of 67"
[48] "Chappaquiddick"
```

Perfect! Now we just have movie names. You'll see some movie names have strange symbols, like `&hellip;`. That's the HTML code for horizontal ellipses or "…". These make the text look prettier on a webpage, but you might need to do more work with `gsub()` if it is important that these movie names look right. Some movies, like *Ocean's 8* and *Won't You Be My Neighbor*, have strange text in place of the apostrophe. This is because `scan()` has made some assumptions about the character set used for reading the text on the page… and it turns out that it is not quite right for some special symbols like smart quotes. If we add the parameter `fileEncoding="UTF-8"`, then R will know to interpret the text it is reading using UTF8, a more complete set of characters that includes newer symbols like the euro sign, nicer looking characters like smart quotes, and many non-Latin alphabets.

Let's put these movie names in a data frame, `data0`. This data frame currently has only one column.

```
data0 <- data.frame(movie=gsub("<[^>]*>", "", a[i]))
```

Now we also want to get the daily gross for each movie. Let's take another look at the HTML code for *Jurassic Park*.

```
a[i[1]:(i[1]+8)]
```

```
[1] "<td><b><a href=\"/movie/Jurassic-World-Fallen-Kingdom-(2018)#tab=box-office\">Jurassic Worl
[2] "<td><a href=\"/market/distributor/Universal\">Universal</a></td>"
[3] "<td class=\"data\">$11,501,395</td>"
[4] "<td class=\"data chart_down\">-3%</td>"
[5] "<td  class=\"data\">4,485</td>"
[6] "<td class=\"data chart_grey\">$2,564</td>"
[7] "<td class=\"data\">&nbsp;&nbsp;$297,672,320</td>"
[8] "<td class=\"data\">13</td>"
[9] "</tr>"
```

Note that the movie gross is two lines after the movie name. It turns out that this is consistent for all movies. Since `i` has the line numbers for the movie names, then `i+2` must be the line numbers containing the daily gross.

```
a[i+2]
```

```
 [1] "<td class=\"data\">$11,501,395</td>"
 [2] "<td class=\"data\">$9,646,015</td>"
 [3] "<td class=\"data\">$9,305,875</td>"
 [4] "<td class=\"data\">$2,577,639</td>"
 [5] "<td class=\"data\">$2,177,946</td>"
 [6] "<td class=\"data\">$2,093,164</td>"
 [7] "<td class=\"data\">$1,201,586</td>"
 [8] "<td class=\"data\">$769,090</td>"
 [9] "<td class=\"data\">$681,243</td>"
[10] "<td class=\"data\">$666,040</td>"
[11] "<td class=\"data\">$621,784</td>"
[12] "<td class=\"data\">$385,514</td>"
[13] "<td class=\"data\">$344,958</td>"
[14] "<td class=\"data\">$246,830</td>"
[15] "<td class=\"data\">$189,837</td>"
[16] "<td class=\"data\">$119,985</td>"
[17] "<td class=\"data\">$89,845</td>"
[18] "<td class=\"data\">$81,243</td>"
[19] "<td class=\"data\">$63,671</td>"
[20] "<td class=\"data\">$53,080</td>"
[21] "<td class=\"data\">$45,167</td>"
[22] "<td class=\"data\">$44,162</td>"
[23] "<td class=\"data\">$40,114</td>"
[24] "<td class=\"data\">$38,837</td>"
[25] "<td class=\"data\">$38,785</td>"
[26] "<td class=\"data\">$21,450</td>"
[27] "<td class=\"data\">$20,925</td>"
[28] "<td class=\"data\">$18,505</td>"
[29] "<td class=\"data\">$16,706</td>"
[30] "<td class=\"data\">$14,875</td>"
[31] "<td class=\"data\">$12,404</td>"
[32] "<td class=\"data\">$11,183</td>"
[33] "<td class=\"data\">$8,150</td>"
[34] "<td class=\"data\">$7,577</td>"
[35] "<td class=\"data\">$6,462</td>"
[36] "<td class=\"data\">$4,798</td>"
[37] "<td class=\"data\">$4,238</td>"
[38] "<td class=\"data\">$4,183</td>"
[39] "<td class=\"data\">$3,711</td>"
[40] "<td class=\"data\">$2,253</td>"
[41] "<td class=\"data\">$1,373</td>"
[42] "<td class=\"data\">$1,370</td>"
[43] "<td class=\"data\">$1,332</td>"
[44] "<td class=\"data\">$1,305</td>"
[45] "<td class=\"data\">$1,098</td>"
[46] "<td class=\"data\">$718</td>"
[47] "<td class=\"data\">$450</td>"
[48] "<td class=\"data\">$445</td>"
```

Again we need to strip out the HTML tags. We will also remove the dollar signs and commas so that R will recognize it as a number. We'll add this to `data0` also.

```
data0$gross <- as.numeric(gsub("<[^>]*>|[$,]", "", a[i+2]))
```

Take a look at the webpage and compare it to the dataset you've now created. All the values should now match.

```r
head(data0)
tail(data0)
```

```
                           movie    gross
1 Jurassic World: Fallen Kingdom 11501395
2                  Incredibles 2  9646015
3                The First Purge  9305875
4    Sicario: Day of the Soldado  2577639
5                     Uncle Drew  2177946
6                    Oceanâ\200\231s 8  2093164
                 movie gross
43               Beast  1332
44     On Chesil Beach  1305
45 Let The Sunshine In  1098
46    Super Troopers 2   718
47        Summer of 67   450
48      Chappaquiddick   445
```

## Scraping Multiple Pages

We've now successfully scraped data for one day. This is usually the hardest part. But if we have R code that can correctly scrape one day's worth of data *and* the website is consistent across days, then it is simple to adapt our code to work for *all* days. So let's get all movie data from January 1, 2010 through December 31, 2018. That means we're going to be web scraping over 3,200 pages of data.

First note that the URL for July 4, 2018 was

```
http://www.the-numbers.com/box-office-chart/daily/2018/07/04
```

We can extract data from any other date by using the same URL, but changing the ending to match the date that we want. Importantly, note that the 07 and the 04 in the URL must have the leading 0 for the URL to return the correct page.

To start, let's make a list of all the dates that we intend to scrape.

```r
library(lubridate)
```

```

Attaching package: 'lubridate'

The following object is masked from 'package:base':

    date
```

```r
# create a sequence of all days to scrape
dates2scrape <- seq(ymd("2010-01-01"), ymd("2018-12-31"), by="days")
```

Now `dates2scrape` contains a collection of all the dates with movie data that we wish to scrape.

```r
dates2scrape[1:5]
# gsub() can change the - to / to match the appearance of the numbers.com URL
gsub("-", "/", dates2scrape[1:5])
```

```
[1] "2010-01-01" "2010-01-02" "2010-01-03" "2010-01-04" "2010-01-05"
[1] "2010/01/01" "2010/01/02" "2010/01/03" "2010/01/04" "2010/01/05"
```

Our plan is to construct a for loop within which we will construct a URL from `dates2scrape`, pull down the HTML code from that URL, scrape the movie data into a data frame, and then combine the each day's data frame into one data frame will all of the movie data. First we create a list that will contain each day's data frame.

```r
results <- vector("list", length(dates2scrape))
```

On iteration `i` of our for loop we will store that day's movie data frame in `results[[i]]`. The following for loop can take several minutes to run and its speed will depend on your network connection and how responsive the web site is. Before running the entire for loop, it may be a good idea to temporarily set the dates to a short period of time (e.g., a month or two) just to verify that your code is functioning properly. Once you've concluded that the code is doing what you want it to do, you can set the dates so that the for loop runs for the entire analysis period.

```r
timeStart <- Sys.time() # record the starting time
for(iDate in 1:length(dates2scrape))
{
   # uncomment the next line to display progress
   #    useful to know how much is done/left to go
   # print(dates2scrape[iDate])

   # construct URL
   urlText <- paste0("http://www.the-numbers.com/box-office-chart/daily/",
                     gsub("-", "/", dates2scrape[iDate]))

   # read in the HTML code... now using UTF8
   a <- scan(urlText, what="", sep="\n", fileEncoding="UTF-8")

   # find movies
   i <- grep("#tab=box-office", a)

   # get movie names and gross
   data0 <- data.frame(movie=gsub("<[^>]*>", "", a[i]),
                       gross=as.numeric(gsub("<[^>]*>|[$,]","",a[i+2])))

   # add date into the dataset
   data0$date  <- dates2scrape[iDate]

   results[[iDate]] <- data0
}
# calculate how long it took
timeEnd <- Sys.time()
```

```
timeEnd-timeStart
```

```
Time difference of 17.73764 mins
```

Let's look at the first 3 lines of the first and last 3 days.

```
lapply(head(results,n=3), head, n=3)
lapply(tail(results,n=3), head, n=3)
```

```
[[1]]
                                movie    gross       date
1                              Avatar 25274008 2010-01-01
2                     Sherlock Holmes 14889882 2010-01-01
3 Alvin and the Chipmunks: Th&hellip; 12998264 2010-01-01

[[2]]
                                movie    gross       date
1                              Avatar 25835551 2010-01-02
2                     Sherlock Holmes 14373564 2010-01-02
3 Alvin and the Chipmunks: Th&hellip; 14373273 2010-01-02

[[3]]
                                movie    gross       date
1                              Avatar 17381129 2010-01-03
2 Alvin and the Chipmunks: Th&hellip;  7818116 2010-01-03
3                     Sherlock Holmes  7349035 2010-01-03

[[1]]
                 movie    gross       date
1              Aquaman 18632907 2018-12-29
2 Mary Poppins Returns  9541501 2018-12-29
3            Bumblebee  7502014 2018-12-29

[[2]]
                 movie    gross       date
1              Aquaman 16440551 2018-12-30
2 Mary Poppins Returns  8224343 2018-12-30
3            Bumblebee  6603471 2018-12-30

[[3]]
                 movie    gross       date
1              Aquaman 10011638 2018-12-31
2 Mary Poppins Returns  6666586 2018-12-31
3            Bumblebee  4241953 2018-12-31
```

Looks like we got them all. Now let's combine them into one big data frame. This use of `do.call()` is a short hand way of saying `rbind(results[[1]], results[[2]], ...)`.

```
movieData <- do.call(rbind, results)
```

```
# check that the number of rows and dates seem reasonable
nrow(movieData)
range(movieData$date)
```

```
[1] 144520
[1] "2010-01-01" "2018-12-31"
```

If you ran that for loop to gather nearly a decade's worth of data, most likely you walked away from your computer to do something more interesting than watch it print out dates. In these situations, I like to send myself a message when it is complete. The `mailR` package is a convenient way to send yourself an email. If you fill it in with your email, username, and password, the following code will send you an email when the script reaches this point.

```
library(mailR)
send.mail(from = "", #replace with your email address
          to = c(""), #replace with email addresses to send to
          subject = "Movies",
          body = "R has finished downloading all the movie data",
          smtp = list(host.name="smtp.gmail.com",
                      port     =465,
                      user.name="", # add your username
                      passwd   ="", # and password
                      ssl      =TRUE),
          authenticate = TRUE,
          send = TRUE)
```

Note that the password here is in plain text so do not try this on a public computer. R also saves your history so even if it's not on the screen it might be saved somewhere else on the computer.

## Parallel Computing

Since 1965 Moore's Law has predicted the power of computation over time. Moore's Law predicted the doubling of transistors about every two years. Moore's prediction has held true for decades. However, to get that speed the transistors were made smaller and smaller. Moore's Law cannot continue indefinitely. The diameter of a silicon atom is 0.2nm. Transistors today contain less than 70 atoms and some transistor dimensions are between 10nm and 40nm. Since 2012, computing power has not changed greatly signaling that we might be getting close to the end of Moore's Law, at least with silicon-based computing. What has changed is the widespread use of multiprocessor and multicore processors. Rather than having a single processor, a typical laptop might have an 8 or 16 core processor (meaning they have 8 or 16 processors that share some resources like high speed memory).

R can guess how many cores your computer has on hand.

```
library(doParallel)
```

```
Loading required package: foreach

Loading required package: iterators
Loading required package: parallel
```

```r
detectCores()
```

```
[1] 8
```

Having access to multiple cores allows you to write scripts that send different tasks to different processors to work on simultaneously. While one processor is busy scraping the data for January 1st, the second can get to work on January 2nd, and another can work on January 3rd. All the processors will be fighting over the one connection you have to the internet, but they can `grep()` and `gsub()` at the same time other processors are working on other dates.

To write a script to work in parallel, you will need the `doParallel` and `foreach` packages. Let's first test whether parallelization actually speed things up. We've made two foreach loops below. In both of them each iteration of the loop does not really do anything except pause for 2 seconds. The first loop, which does not use parallelization, includes 10 iterations and so should take 20 seconds to run. The second foreach loop looks the same, except right before the foreach loop we have told R to make use of two of the computer's processors rather than the default of one processor. This should cause one processor to sleep for 2 seconds 5 times and the other processor to sleep for 2 seconds 5 times. and should take about 10 seconds.

```r
library(foreach)

# should take 10*2=20 seconds
system.time( # time how long this takes
  foreach(i=1:10) %do% # run not in parallel
  {
     Sys.sleep(2)  # wait for 2 seconds
     return(i)
  }
)
```

```
   user  system elapsed 
   0.00    0.00   20.56 
```

```r
# set up R to use 2 processors
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)

# with two processors should take about 10 seconds
system.time( # time how long this takes
  foreach(i=1:10) %dopar% # run in parallel
  {
    Sys.sleep(2)  # wait for 2 seconds
    return(i)
  }
)
```

```
   user  system elapsed 
   0.00    0.00   10.29 
```

```r
stopCluster(cl)
```

Sure enough, the parallel implementation was able to complete 20 seconds worth of sleeping in only 10 seconds. To set up code to run in parallel, the key steps are to set up the cluster of processors using `makeCluster()` and to tell parallel `foreach()` to use that cluster of processors with `registerDoParallel()`. Note that the key difference between the two `foreach()` statements is that the first `foreach()` is followed by a `%do%` while the second is followed by a `%dopar%`. When `foreach()` sees the `%dopar%` it will check what was setup in the `registerDoParallel()` call and spread the computation among those processors.

Note that the `foreach()` differs a little bit in its syntax compared with our previous use of for loops. While for loops have the syntax `for(i in 1:10)` the syntax for `foreach()` looks like `foreach(i=1:10)` and is followed by a `%do%` or a `%dopar%`. Lastly, note that the final step inside the `{ }` following a `foreach()` is a `return()` statement. `foreach()` will take the returned values of each of the iterations and assemble them into a single list by default. In the following `foreach()` we've added `.combine=rbind` to the `foreach()` so that the final results will be stacked into one data frame, avoiding the need for a separate `do.call()` like we used previously.

With all this in mind, let's web scrape the movie data using multiple processors:

```r
cl <- makeCluster(8)
registerDoParallel(cl)

timeStart <- Sys.time() # record the starting time
movieData <-
   foreach(iDate=1:length(dates2scrape),
           .combine=rbind) %dopar%
{
   urlText <- paste0("http://www.the-numbers.com/box-office-chart/daily/",
                     gsub("-", "/", dates2scrape[iDate]))

   a <- scan(urlText, what="", sep="\n", fileEncoding="UTF-8")
   i <- grep("#tab=box-office", a)

   data0 <- data.frame(movie=gsub("<[^>]*>", "", a[i]),
                       gross=as.numeric(gsub("<[^>]*>|[$,]","",a[i+2])))

   data0$date  <- dates2scrape[iDate]

  return(data0)
}

# change HTML codes to something prettier
movieData$movie <- gsub("&hellip;", "...", movieData$movie)

# calculate how long it took
timeEnd <- Sys.time()
timeEnd-timeStart
```

```
stopCluster(cl)
```

```
Time difference of 2.195714 mins
```

This code makes use of 8 processors. Unlike our 2 second sleep example, this script may or may not run 8 times faster. This is mostly due to the fact each processor still needs to wait its turn in order to pull down its webpage from the internet. Still you should observe the parallel version finishing much sooner than the first version. In just a few lines of code and a few minutes of waiting, you now have almost a decade worth of movie data.

Parallelization introduces two complications. The first is that if anything goes wrong in this script, then the whole `foreach()` fails. For example, let's say that after scraping movie data from 2000-2016 you briefly lose your internet connection. If this happens, then `scan()` fails and the whole `foreach()` ends with an error, tossing all of your already complete computation. To avoid this you need to either be sure you have a solid internet connection, or wrap the call to `scan()` in a `try()` and a `while` loop that is smart enough to wait a few seconds and try the scan again rather than fail completely. A second, and minor issue, is that you cannot print to the screen from inside a parallelized `foreach()`. So there's no printing the progress of the computation to the screen. This can leave you wondering if your script is still working. You can set an `outfile` parameter when creating your cluster of processors, like `makecluster(8, outfile="log.txt")`. Any output that R prints will be redirected to this log.txt file. Remember that all the processors will be at different steps and they will all be dumping output to log.txt. It can be difficult to determine which process is responsible for which line of output, but at least you will know that your script is progressing.

It's probably wise at this point to save `movieData` so that you won't have to rerun this in the future.

```
save(movieData, file="movieData.RData", compress=TRUE)
```

## Fun With Movie Data

You can use the dataset to answer questions such as "which movie yielded the largest gross?"

```
movieData[which.max(movieData$gross),]
```

```
                                movie     gross       date
92116 Star Wars Ep. VII: The Forc... 119119282 2015-12-18
```

Which ten movies had the largest total gross during the period this dataset covers?

```
a <- aggregate(gross~movie, data=movieData, sum)
a[order(-a$gross)[1:10],]
```

```
                               movie     gross
1890 Star Wars Ep. VII: The Forc... 935642689
319                    Black Panther 700059566
222           Avengers: Infinity War 678815482
853  Harry Potter and the Deathl... 676012289
1077                  Jurassic World 652198010
1891 Star Wars Ep. VIII: The Las... 620181382
2208 The Hunger Games: Mockingja... 618859787
```

```
1998                   The Avengers 618839615
974                   Incredibles 2 608581744
2448 The Twilight Saga: Breaking... 573611870
```

Which days of the week yielded the largest total gross?

```
aggregate(gross~wday(date, label=TRUE), data=movieData, sum)
```

```
  wday(date, label = TRUE)       gross
1                      Sun 18341970220
2                      Mon  7754234000
3                      Tue  7779629491
4                      Wed  6676148038
5                      Thu  6361523338
6                      Fri 22188960970
7                      Sat 26626155478
```

Now that you have movie data and in a previous section you assembled Chicago crime data, combine the two datasets so that you can answer the question "what happens to crime when big movies come out?"