Lecture 8: Getting Data

36-350

22 September 2014

In Previous Episodes

- Seen functions to load data in passing
- Learned about string manipulation and regexp

Agenda

- Getting data into and out of the system when it's already in R format
- Import and export when the data is already very structured and machine-readable
- Dealing with less structured data
- Web scraping

Reading Data from R

- You can load and save R objects
  - R has its own format for this, which is shared across operating systems
  - It's an open, documented format if you really want to pry into it
- `save(thing, file="name")` saves `thing` in a file called `name` (conventional extension: `rda` or `Rda`)
- `load("name")` loads the object or objects stored in the file called `name`, *with their old names*

```r
gmp <- read.table("http://www.stat.cmu.edu/~cshalizi/statco
gmp$pop <- round(gmp$gmp/gmp$pcgmp)
save(gmp,file="gmp.Rda")
rm(gmp)
exists("gmp")
```

```
## [1] FALSE
```

```r
not_gmp <- load(file="gmp.Rda")
colnames(gmp)
```

```
## [1] "MSA"    "gmp"    "pcgmp" "pop"
```

```r
not_gmp
```

```
## [1] "gmp"
```

- We can load or save more than one object at once; this is how RStudio will load your whole workspace when you're starting, and offer to save it when you're done
- Many packages come with saved data objects; there's the convenience function `data()` to load them

```
data(cats,package="MASS")
summary(cats)
```

```
##   Sex         Bwt             Hwt       
##   F:47    Min.   :2.00   Min.   : 6.30  
##   M:97    1st Qu.:2.30   1st Qu.: 8.95  
##           Median :2.70   Median :10.10  
##           Mean   :2.72   Mean   :10.63  
##           3rd Qu.:3.02   3rd Qu.:12.12  
##           Max.   :3.90   Max.   :20.50  
```

Note: `data()` returns the name of the loaded data file!

Non-R Data Tables

- Tables full of data, just not in the R file format
- Main function: `read.table()`
  - Presumes space-separated fields, one line per row
  - Main argument is the file name or URL
  - Returns a dataframe
  - Lots of options for things like field separator, column names, forcing or guessing column types, skipping lines at the start of the file...
- `read.csv()` is a short-cut to set the options for reading comma-separated value (CSV) files
  - Spreadsheets will usually read and write CSV

Writing Dataframes

- Counterpart functions `write.table()`, `write.csv()` write a dataframe into a file
- Drawback: takes a lot more disk space than what you get from `load` or `save`
- Advantage: can communicate with other programs, or even edit manually

Less Friendly Data Formats

- The `foreign` package on CRAN has tools for reading data files from lots of non-R statistical software
- Spreadsheets are special

Spreadsheets Considered Harmful

- Spreadsheets look like they should be dataframes
- Real spreadsheets are full of ugly irregularities
  - Values or formulas?
  - Headers, footers, side-comments, notes
  - Columns change meaning half-way down
  - Whole separate programming languages apparently intended to mostly to spread malware
- Ought-to-be-notorious source of errors in both industry (1, 2) and science (e.g., Reinhart and Rogoff)

Spreadsheets, If You Have To

- Save the spreadsheet as a CSV; `read.csv()`
- Save the spreadsheet as a CSV; edit in a text editor; `read.csv()`
- Use `read.xls()` from the `gdata` package
- Tries very hard to work like `read.csv()`, can take a URL or filename
- Can skip down to the first line that matches some pattern, select different sheets, etc.
- You may still need to do a lot of tidying up after

```r
require(gdata, quietly=TRUE)
```

```
## gdata: read.xls support for 'XLS' (Excel 97-2004) files
##
## gdata: read.xls support for 'XLSX' (Excel 2007+) files 
##
## Attaching package: 'gdata'
##
## The following object is masked from 'package:stats':
##
##     nobs
##
## The following object is masked from 'package:utils':
##
##     object.size
```

```r
setwd("~/Downloads/")
gmp_2008_2013 <- read.xls("gdp_metro0914.xls",pattern="U.S
head(gmp_2008_2013)
```

```
##        U.S..metropolitan.areas X13.269.057 X12.994.636 X1
## 1                  Abilene, TX       5,725       5,239
## 2                    Akron, OH      28,663      27,761
## 3                   Albany, GA       4,795       4,957
## 4                   Albany, OR       3,235       3,064
## 5 Albany-Schenectady-Troy, NY       40,365      42,454
## 6              Albuquerque, NM      37,359      38,110
##   X13.953.082 X14.606.938 X15.079.920 .......
## 1       5,761       6,143       6,452     252
## 2      29,425      31,012      31,485      80
## 3       4,938       5,122       5,307     290
## 4       3,170       3,294       3,375     363
## 5      43,663      45,330      46,537      58
## 6      39,967      41,301      41,970      64
```

Semi-Structured Files, Odd Formats

- Files with metadata (e.g., earthquake catalog)
- Non-tabular arrangement
- Generally, write function to read in one (or a few) lines and split it into some nicer format
  - Generally involves a lot of regexps
  - Functions are easier to get right than code blocks in loops

In Praise of Capture Groups

- Parentheses don't just group for quantifiers; they also create *capture groups*, which the regexp engine remembers
- Can be referred to later (\1, \2, etc.)
- Can also be used to simplify getting stuff out
- Examples in the handout on regexps, but let's reinforce the point

Scraping the Rich

- Remember that the lines giving net worth looked like

```
<td class="worth">$72 B</td>
```

or

```
<td class="worth">$5,3 B</td>
```

One regexp which catches this:

```
richhtml <- readLines("http://www.stat.cmu.edu/~cshalizi/st
worth_pattern <- "\\$[0-9,]+ B"
worth_lines <- grep(worth_pattern, richhtml)
length(worth_lines)
```

```
## [1] 100
```

(that last to check we have the right number of matches)

Just using this gives us strings, including the markers we used to pin down where the information was:

```
worth_matches <- regexpr(worth_pattern, richhtml)
worths <- regmatches(richhtml, worth_matches)
head(worths)
```

```
## [1] "$72 B"   "$58,5 B" "$41 B"   "$36 B"   "$36 B"   "$
```

Now we'd need to get rid of the anchoring $ and B; we could use `substr`, but...

Adding a capture group doesn't change what we match:

```
worth_capture <- worth_pattern <- "\\$([0-9,]+) B"
capture_lines <- grep(worth_capture, richhtml)
identical(worth_lines, capture_lines)
```

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

but it *does* have an advantage

Using regexec

```r
worth_matches <- regmatches(richhtml[capture_lines],
  regexec(worth_capture, richhtml[capture_lines]))
worth_matches[1:2]
```

```
## [[1]]
## [1] "$72 B" "72"
##
## [[2]]
## [1] "$58,5 B" "58,5"
```

List with 1 element per matching line, giving the whole match and then each paranethesized matching sub-expression

Functions make the remaining manipulation easier:

```
second_element <- function(x) { return(x[2]) }
worth_strings <- sapply(worth_matches, second_element)
comma_to_dot <- function(x) {
  return(gsub(pattern=",",replacement=".",x))
}
worths <- as.numeric(sapply(worth_strings, comma_to_dot))
head(worths)
```

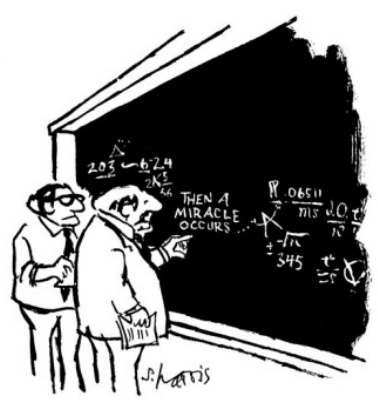
```
## [1] 72.0 58.5 41.0 36.0 36.0 35.4
```

Exercise: Write *one* function which takes a single line, gets the capture group, and converts it to a number

Web Scraping

1. Take a webpage designed for humans to read
2. Have the computer extract the information we actually want
3. Iterate as appropriate

Take in unstructured pages, return rigidly formatted data



"I think you should be more explicit here in step two."

CN
COLLECTION

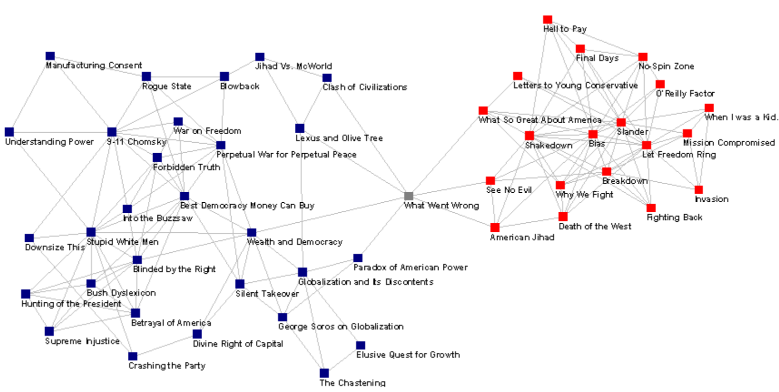
Being More Explicit in Step 2

- The information we want is *somewhere* in the page, possibly in the HTML
- There are usually markers surrounding it, probably in the HTML
- We now know how to pick apart HTML using regular expressions

- Figure out *exactly* what we want from the page
- Understand how the information is organized on the page
  - What does a human use to find it?
  - Where do those cues appear in the HTML source?
- Write a function to automate information extraction
  - Generally, this means regexps
  - Parenthesized capture groups are helpful
  - The function may need to iterate
  - You may need more than one function
- Once you've got it working for one page, iterate over relevant pages

Example: Book Networks

Famous example from Vladis Krebs



- Two books are linked if they're bought together at Amazon
- Amazon gives this information away (to try to drive sales)
- How would we replicate this?

[http://www.amazon.com/dp/0387747303/]

- Do we want “frequently bought together”, or “customers who bought this also bought that”? Or even “what else do customers buy after viewing this”?
  - Let’s say “customers who bought this also bought that”
- Now look carefully at the HTML
  - There are over 14,000 lines in the HTML file for this page; you’ll need a text editor
  - Fortunately most of it’s irrelevant

```
<div class="shoveler" id="purchaseShvl">
    <div class="shoveler-heading">
        <h2>Customers Who Bought This Item Also Bought</h2>
    </div>

<div class="shoveler-pagination" style="display:none">

<span> </span>
<span>
Page <span class="page-number"></span>  of  <span class="nu
<span class="start-over"><span class="a-text-separator"></s
</span>
</div>

    <div class="shoveler-button-wrapper" id="purchaseButto
        <a class="back-button" href="#Back" style="display:
        <div class="shoveler-content">
            <ul tabindex="-1">
```

Here's the first of the also-bought books:

```
<li>
  <div class="new-faceout p13nimp"  id="purchase_038798140
<a href="/ggplot2-Elegant-Graphics-Data-Analysis/dp/0387981
                       <img src="http://ecx.images-amazon.c
                    </div>
                    <span title="ggplot2: Elegant Graphics

    <div class="byline">
        <span class="carat">&#8250</span>
```

We *could* extract the ISBN from this, and then go on to the next book, and so forth…

```
<div id="purchaseSimsData" class="sims-data"
style="display:none" data-baseAsin="0387747303"
data-deviceType="desktop" data-featureId="pd_sim" data-isAU
data-wdg="book_display_on_website"
data-widgetName="purchase">0387981403,0596809158,1593273843
0387938362,144931208X,0387790535,0387886974,0470973927,0387
1439810184,1461413648,1461471370,1782162143,1441998896,1429
1612903436,1441996494,1461468485,1617291560,1439831769,0321
1119962846,0521762936,1446200469,1449358659,1935182390,0123
0387759352,1461476178,0387773169,0387922970,0073523321,1412
1612900275,1449339735,052168689X,0387781706,1584884509,0387
1441915753,1466572841,1107422221,111844714X,0716762196,0133
0963488406,1466586966,0470463635,1493909827,1420079336,0321
158488424X,1441926127,1466570229,1590475348,1430266406,0071
111866146X,1441977864,1782160604,1449340377,1449309038,0963
1461406846,0073014664,1449370780,144197864X,3642201911,0534
158488651X,1449357105,1118208781,1420099604,1107057132,1449
1449361323,0470890819,0387245448,0521518148,0521169828,1584
0387781889,0387759581,0387717617,0123748569,188652923X,0155
```

In this case there's a big block which gives us the ISBNs of *all* the also-bought books

Strategy:

- Load the page as text
- Search for the regexp which begins this block, contains at least one ISBN, and then ends
- Extract the sequence of ISBNs as a string, split on comma
- Record in a dataframe that *Data Manipulation*'s ISBN is also bought with each of those ISBNs
- *Snowball sampling*: Go to the webpage of each of those books and repeat
  - Stop when we get tired...
  - Or when Amazon gets annoyed with us

More considerations on web-scraping

- You should really look at the site's `robots.txt` file and respect it
- See [`https://github.com/hadley/rvest`] for a prototype of a package to automate a lot of the work of scraping webpages

Summary

- Loading and saving R objects is very easy
- Reading and writing dataframes is pretty easy
- Extracting data from unstructured sources is about using regexps appropriately
  - Maybe not *easy*, but at least *feasible*