CGD: Downloading Files

September 24, 2014

Downloading the files

First steps

- know what directory you are in
- use getwd() and setwd(), as necessary
- setwd("../") moves up one directory

1. Create a directory for the data to fall into, if such a directory doesn't already exist

if (!file.exists("directory name")){
dir.create("directory name")
}

2. Get the data. download.file() is the main way that we obtain internet data . *Always* include the time of download

download.file(url = , destfile = "./directory name/filename.ext", method = "curl")
dateDownloaded<-date()

The downloaded files can be viewed with list.files("./directory name")

3. The file is now stored locally. The most commonly used function to read it into R is read.table(). It is the most robust, and allows the most flexibility.
   However, because of its slowness, it is not the best method to read large tables into R.

eg: downloading data on Baltimore Fixed Speed Camera data and reading into R

```r
# Having checked that we are in the correct working directory

# The following function downloads the .csv file containing data for Balitmore's Fixed Speed Cameras

#Input: Link to the file, desired directory name, desired, file name, and file extension
#Output: The path to the required file

createAndDownload<-function(fileUrl, dir, filename, ext){
        # Step 1: create directory, if it is not already present
        dirName<-paste(dir, sep = "")
        if(!file.exists(dirName)){
                dir.create(path = dirName)
        }
        # Step 2: Get the data, unless this step has already been done
        dest<-paste("./", dirName,"/", filename, ext, sep = "")
        if(!file.exists(dest)){
                download.file(url = fileUrl, destfile = dest, method = "curl")
                datedownloaded<-date()
        }
        dest
}
fileUrl<-"https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD"
dest<-createAndDownload(fileUrl, dir="camera", filename="camera", ext = ".csv")
# Step 3: load the data
# because the variable names are included at the top of each file, header = T
camdat<-read.table(file = dest, header = T, sep = ",")
```

In the case that we are working with a .csv file, as here, read.csv() can be used. It sets sep =“,” and header =T by default

```
camdat2<-read.csv("./camera/camera.csv")
all.equal(camdat, camdat2)
```

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

```
rm(camdat2)
```

Troubleshooting

- set quote =“” to tell R to ignore quoted values
- na.strings = sets the character that represents the missing values

Reading in particular file formats

Excel

1. require the xlsx package
2. run createAndDownload() with the appropriate parameters
3. in step 3, use read.xlsx() instead of read.table() specifying which sheet the data is stored on with sheetIndex, and specify the header, if necessary

```
fileUrl<-"https://data.baltimorecity.gov/api/views/dz54-2aru/rows.xlsx?accessType=DOWNLOAD"
dest<-createAndDownload(fileUrl = fileUrl, dir = "camera", filename = "camera", ext = ".xlsx")
require("xlsx")
```

```
## Loading required package: xlsx
## Loading required package: rJava
## Loading required package: xlsxjars
```

```
camdat2<-read.xlsx(file = dest, sheetIndex = 1, header = T)
```

Nice things

- you can read specific rows and columns

```
colInd<-2:3; rowInd<-1:4
camdat2subset<-read.xlsx(file = dest, sheetIndex = 1, rowIndex = rowInd, colIndex = colInd, header = T)
camdat2subset
```

```
##   direction      street
## 1       N/B   Caton Ave
## 2       S/B   Caton Ave
## 3       E/B Wilkens Ave
```

- you can write an .xlsx file out with

```
write.xlsx(camdat2subset, file = "./camera/camera2.xlsx")
```

XML

- short for "extensible markup language"
- frequently used to store structured data
- widely used in internet applications
- extracting XML is the basis for most web scraping
- "XML" package does not support http**s** (secure http). Delete "s" of https manually or by using sub(https, http, fileURL)

Tags correspond to general labels

- start tags < section >
- end tags < \section >

Elements are specific examples of tags

eg: < Greeting > Hello <\ greeting >

Unlike the previous examples, we do **not** use download file. Instead use xmlTreeParse to parse out the XML file given in the URL

```
require(XML)
```

```
## Loading required package: XML
```

```
fileUrl<-"http://www.w3schools.com/xml/simple.xml"
doc<-xmlTreeParse(file = fileUrl, useInternalNodes = T)
doc
```

```
## <?xml version="1.0" encoding="UTF-8"?>
## <!-- Edited by XMLSpy -->
## <breakfast_menu>
##   <food>
##     <name>Belgian Waffles</name>
##     <price>$5.95</price>
##     <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
##     <calories>650</calories>
##   </food>
##   <food>
##     <name>Strawberry Belgian Waffles</name>
##     <price>$7.95</price>
##     <description>Light Belgian waffles covered with strawberries and whipped cream</description>
##     <calories>900</calories>
##   </food>
##   <food>
##     <name>Berry-Berry Belgian Waffles</name>
##     <price>$8.95</price>
##     <description>Light Belgian waffles covered with an assortment of fresh berries and whipped cream
##     <calories>900</calories>
##   </food>
##   <food>
##     <name>French Toast</name>
##     <price>$4.50</price>
##     <description>Thick slices made from our homemade sourdough bread</description>
##     <calories>600</calories>
##   </food>
##   <food>
##     <name>Homestyle Breakfast</name>
##     <price>$6.95</price>
##     <description>Two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>
##     <calories>950</calories>
##   </food>
## </breakfast_menu>
##
```

doc is parsed into nodes. The topmost node wraps the entire document. In this case it's < breakfast_menu>

```r
top<-xmlRoot(doc)
xmlName(top)
```

```
## [1] "breakfast_menu"
```

The elements in between the root node is given by names. Each item is wrapped within a "food" element

```r
#child nodes of this root
names(top)
```

```
##   food   food   food   food   food
## "food" "food" "food" "food" "food"
```

Accessing elements in an XML file is totally analogous to accessing elements of a list.

```r
#first element of the rootnode
top[[1]]
```

```
## <food>
##   <name>Belgian Waffles</name>
##   <price>$5.95</price>
##   <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
##   <calories>650</calories>
## </food>
```

```r
names(top[[1]])
```

```
##          name         price   description      calories
##        "name"       "price" "description"    "calories"
```

```r
#Suppose we want to extract the price of the first element
top[[1]][["price"]]
```

```
## <price>$5.95</price>
```

```r
# Extract the price  variable of all elements
xpathSApply(doc = top, "//price", xmlValue)
```

```
## [1] "$5.95" "$7.95" "$8.95" "$4.50" "$6.95"
```

```r
# similarly
xpathSApply(doc=top, "//name", xmlValue)
```

```
## [1] "Belgian Waffles"             "Strawberry Belgian Waffles"
## [3] "Berry-Berry Belgian Waffles" "French Toast"
## [5] "Homestyle Breakfast"
```

HTML

Right click here, and view the source. We want to drill into this source code and extract some information

Load the data with **html**TreeParse. Remember to delete "view source" from the head of the url

```r
fileUrl<-"http://espn.go.com/nfl/team/_/name/bal/baltimore-ravens"
doc<-htmlTreeParse(file = fileUrl, useInternalNodes = T)
str(doc)
```

```
## Classes 'HTMLInternalDocument', 'HTMLInternalDocument', 'XMLInternalDocument', 'XMLAbstractDocument'
```

Look for "list items" (li) with a particular class (in the example below, equal to score)

```r
xpathSApply(doc, "//li[@class ='score']", xmlValue)
```

```
## [1] "23-16" "26-6"  "23-21" "38-10"
```

```r
xpathSApply(doc, "//li[@class ='team-name']", xmlValue)
```

```
##  [1] "Cincinnati"   "Pittsburgh"   "Cleveland"    "Carolina"
##  [5] "Indianapolis" "Tampa Bay"    "Atlanta"      "Cincinnati"
##  [9] "Pittsburgh"   "Tennessee"    "New Orleans"  "San Diego"
## [13] "Miami"        "Jacksonville" "Houston"      "Cleveland"
```

JSON

JSON files are similar to XML files insofar as they are structured, and is very commonly used in Application Programming Interfaces. APIs are how you can access the data for companies like Twitter or facebook through URLs.
Click here to obtain the API for the github API containing data about the repos that the instructor's contributing to.
Reading data from a JSON file is similar to reading from an XML file

```r
require(jsonlite)
```

```
## Loading required package: jsonlite
##
## Attaching package: 'jsonlite'
##
## The following object is masked from 'package:utils':
##
##     View
```

```r
fileUrl<-"https://api.github.com/users/jtleek/repos"
jsonData<-fromJSON(fileUrl)
names(jsonData)
```

```
##  [1] "id"                "name"              "full_name"
##  [4] "owner"             "private"           "html_url"
##  [7] "description"       "fork"              "url"
## [10] "forks_url"         "keys_url"          "collaborators_url"
## [13] "teams_url"         "hooks_url"         "issue_events_url"
## [16] "events_url"        "assignees_url"     "branches_url"
## [19] "tags_url"          "blobs_url"         "git_tags_url"
## [22] "git_refs_url"      "trees_url"         "statuses_url"
## [25] "languages_url"     "stargazers_url"    "contributors_url"
## [28] "subscribers_url"   "subscription_url"  "commits_url"
## [31] "git_commits_url"   "comments_url"      "issue_comment_url"
## [34] "contents_url"      "compare_url"       "merges_url"
## [37] "archive_url"       "downloads_url"     "issues_url"
## [40] "pulls_url"         "milestones_url"    "notifications_url"
## [43] "labels_url"        "releases_url"      "created_at"
## [46] "updated_at"        "pushed_at"         "git_url"
## [49] "ssh_url"           "clone_url"         "svn_url"
## [52] "homepage"          "size"              "stargazers_count"
## [55] "watchers_count"    "language"          "has_issues"
## [58] "has_downloads"     "has_wiki"          "has_pages"
## [61] "forks_count"       "mirror_url"        "open_issues_count"
## [64] "forks"             "open_issues"       "watchers"
## [67] "default_branch"
```

jsonData is a *data frame* of *data frames*.

```r
class(jsonData)
```

```
## [1] "data.frame"
```

```r
# recall how to subset a data frame
c(head(jsonData[1]), head(jsonData["id"]))
```

```
## $id
## [1] 12441219 20234724  7751816  4772877 14204342 23840078
##
## $id
## [1] 12441219 20234724  7751816  4772877 14204342 23840078
```

```r
jsonData$id
```

```
##  [1] 12441219 20234724  7751816  4772877 14204342 23840078 11549405
##  [8] 14240696 11976743  8730097 14590772 12563551  6582536  6661008
## [15] 19133476 16584923  7745123 15639612 19133794 17446438 15723485
## [22] 11378145 17711648 20548045 16103392 23202748 12134722 13788992
## [29] 15532926 12931390
```

```r
# jsonData$owner is another dataframe
names(jsonData$owner)
```

```
##  [1] "login"               "id"                  "avatar_url"
##  [4] "gravatar_id"         "url"                 "html_url"
##  [7] "followers_url"       "following_url"       "gists_url"
## [10] "starred_url"         "subscriptions_url"   "organizations_url"
## [13] "repos_url"           "events_url"          "received_events_url"
## [16] "type"                "site_admin"
```

```r
#so we can drill further down, eg
jsonData$owner$login
```

```
##  [1] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
##  [8] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
## [15] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
## [22] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
## [29] "jtleek" "jtleek"
```

If we have to export to an API that requires API formatted data, use toJSON. To view the file use the **C**oncatenate **A**nd Prin**t** command, cat()

```r
myJson<-toJSON(iris, pretty = T)
# head(cat(myJson))
```

Quiz

Question 1 The American Community Survey distributes downloadable data about United States communities. Download the 2006 microdata survey about housing for the state of Idaho using download.file() from here: https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06hid.csv and load the data into R. The code book, describing the variable names is here: https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2FPUMSDataDict06.pdf How many properties are worth $1,000,000 or more?

Solution outline 1. run createAndDownload()
2. read the data into R using read.csv()
According to the code book, the variable "VAL" contains the property values, and those properties valued in excess of $1M are listed as 24.
3. locate and count up the number of entries under VAL that are 24

```
fileUrl<-"https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06hid.csv"
dest<-createAndDownload(fileUrl = fileUrl, dir = "microdatasurvey", filename =  "housingMicrodata", ext
data<-read.csv(dest)
milPlus<-length(which(data$VAL %in% 24))
paste("There are", milPlus, "properties worth in excess of 1M", sep = " ")
```

```
## [1] "There are 53 properties worth in excess of 1M"
```

Question 2 Use the data you loaded from Question 1. Consider the variable FES in the code book. Which of the "tidy data" principles does this variable violate?

The principles of tidy data are

1. Each variable should be in its own column
2. Each observation should be in its own row
3. There should be one table for every kind of variable (eg: all twitter data is in one table, tall fb data is in another)
4. If there are multiple tables, they should include a coulmn in the table that allow them to be linked

FES 1
Family type and employment status
b .N/A (GQ/vacant/not a family)
1 .Married-couple family: Husband and wife in LF
2 .Married-couple family: Husband in labor force, wife .not in LF
3 .Married-couple family: Husband not in LF, .wife in LF
4 .Married-couple family: Neither husband nor wife in .LF
5 .Other family: Male householder, no wife present, in .LF
6 .Other family: Male householder, no wife present, .not in LF
7 .Other family: Female householder, no husband .present, in LF
8 .Other family: Female householder, no husband .present, not in LF

Here, several variables are grouped together:

1. Married couple or Other family
2. Male or Female Householder
3. Wife or Husband present
4. In Labour Force, or Not In Labour Force
5. Spouse in Labour Force or Spouse Not In Labour Force

According to the Principles of Tidy Data, each of these vaibales should be listed in its own column

Question 3
Download the Excel spreadsheet on Natural Gas Aquisition Program here: https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2FDATA.gov_NGAP.xlsx Read rows 18-23 and columns 7-15 into R and assign the result to a variable called dat. What is the value of: sum(dat$Zip * dat$Ext,na.rm=T) (original data source: http://catalog.data.gov/dataset/natural-gas-acquisition-program)

Solution outline: 1. run createAndDownload()
2. assign rowInd:=18-23 and colInd:=7-15
3. read in those specific rows and colums using read.xlsx()
4. evaluate the given expression

```r
fileUrl<-"https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2FDATA.gov_NGAP.xlsx"
dest<-createAndDownload(fileUrl = fileUrl, dir = "NGA", filename = "NGAdata", ext = ".xlsx") #again, wh
rowInds<-18:23
colInds<-7:15
dat<-read.xlsx(file = dest, sheetIndex = 1, rowIndex = rowInds, colIndex = colInds, header = T)
sum(dat$Zip*dat$Ext,na.rm=T)
```

```
## [1] 36534720
```

Question 4
Read the XML data on Baltimore restaurants from here: https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Frestaurants.xml How many restaurants have zipcode 21231?

Solution Outline
1. Use xmlTreeParse to parse and read in the data
2. use xpathSApply() to abtain all zip codes
3. use which() to locate all zipcodes equal to 21231
4. use length to get the number of restaurants with the required zip code

```r
fileURL<-"https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Frestaurants.xml"
fileUrl<-sub(pattern = "s", replacement = "", x = fileURL)
doc<-xmlTreeParse(file = fileUrl, useInternalNodes = T)
zips<-xpathSApply(doc = doc, path = "//zipcode", fun = xmlValue)
keep<-which(zips %in% 21231)
length(keep)
```

```
## [1] 127
```

Question 5
The American Community Survey distributes downloadable data about United States communities. Download the 2006 microdata survey about housing for the state of Idaho using download.file() from here: https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06pid.csv using the fread() command load the data into an R object called DT.

Which of the following is the fastest way to calculate the average value of the variable
pwgtp15
broken down by sex using the data.table package?

1. rowMeans(DT)[DT$SEX==1]; rowMeans(DT)[DT$SEX==2]

2. mean(DT$pwgtp15, by = DT$SEX)

3. DT[,mean(pwgtp15),by=SEX]

4. tapply(DT$pwgtp15, DT$SEX,mean)

5. mean(DT[DT$SEX==1,]$pwgtp15); $mean(DT[DT$SEX==2,]$pwgtp15)

6. sapply(split(DT$pwgtp15, DT$SEX),mean)

Firstly, fread(), (Fast and friendly file finagler) is more-or-less just a faster version of read.table()

Aside: the data.table() package

```r
fileURL<-"https://d396qusza40orc.cloudfront.net/getdata%2Fdata%2Fss06pid.csv"
dest<-createAndDownload(fileUrl = fileURL, dir = "microdatasurvey", filename = "housingMicrodata", ext
```