Unit 3: Data input/output and webscraping

August 29, 2017

References:

- Adler
- Nolan and Temple Lang, XML and Web Technologies for Data Sciences with R.
- Chambers
- R intro manual on CRAN (R-intro).
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies.
- R Data Import/Export manual on CRAN (R-data).
- SCF tutorial on "Working with large datasets in SQL, R, and Python", available from http://statistics.berkeley

1 Data storage and formats (outside R)

At this point, we're going to turn to getting data, reading data in, writing data out to disk, and webscraping. We'll focus on doing these manipulations in R, but the concepts and tools involved are common to other languages, so familarity with these in R should allow you to pick up other tools more easily. The main downside to working with datasets in R (true for Python as well) is that the entire dataset resides in memory, so R is not so good for dealing with very large datasets. More on alternatives in a bit. Another common frustration is controlling how the variables are interpreted (numeric, character, factor) when reading data into a data frame.

R has the capability to read in a wide variety of file formats. Let's get a feel for some of the common ones.

1. Flat text files (ASCII files): data are often provided as simple text files. Often one has one record or observation per row and each column or field is a different variable or type of information about the record. Such files can either have a fixed number of characters in each field (fixed width format) or a special character (a delimiter) that separates the fields in each row. Common delimiters are tabs, commas, one or more spaces, and the pipe (|). Common file extensions are *.txt* and *.csv*. Metadata (information about the data) are often stored in a separate file. I like CSV files but if you have files where the data contain commas, other delimiters can be good. Text can be put in quotes in CSV files. This is difficult to deal with in bash, but *read.table()* in R handles this situation.

   - One occasionally tricky difficulty is as follows. If you have a text file created in Windows, the line endings are coded differently than in UNIX (a newline (the ASCII character *\n*) and a carriage return (the ASCII character *\r*) in Windows vs. only a newline in UNIX). There are UNIX utilities (*fromdos* in Ubuntu, including the SCF Linux machines and *dos2unix* in other Linux distributions) that can do the necessary conversion. If you see *^M* at the end of the lines in a file, that's the tool you need. Alternatively, if you open a UNIX file in Windows, it may treat all the lines as a single line. You can fix this with *todos* or *unix2dos*.
     As a side note, Macs have line endings as in UNIX, but before Mac OS X, lines ended only in a carriage return. There is a UNIX utility call *mac2unix* that can convert such text files.

2. In some contexts, such as textual data and bioinformatics data, the data may in a text file with one piece of information per row, but without meaningful columns/fields.

3. In scientific contexts, netCDF (*.nc*) (and the related HDF5) are popular format for gridded data that allows for highly-efficient storage and contains the metadata within the file. The basic structure of a netCDF file is that each variable is an array with multiple dimensions (e.g., latitude, longitude, and time), and one can also extract the values of and metadata about each dimension. The *ncdf4* package in R nicely handles working with netCDF files. These are examples of a binary format, which is not (easily) human readable but can be more space-efficient and faster to work with (because they can allow random access into the data rather than requiring sequential reading).

4. Data may also be in text files in formats designed for data interchange between various languages, in particular XML or JSON. These formats are "self-describing"; namely the metadata is part of the file. The *XML* and *jsonlite* packages are useful for reading and writing from these formats.

5. You may be scraping information on the web, so dealing with text files in various formats, including HTML. The *XML* package is also useful for reading HTML.

6. Data may already be in a database or in the data storage of another statistical package (*Stata*, *SAS*, *SPSS*, etc.). The *foreign* package in R has excellent capabilities for importing Stata (*read.dta()*), SPSS (*read.spss()*), and SAS (*read.ssd()* and, for XPORT files, *read.xport()*), among others.

7. For Excel, there are capabilities to read an Excel file (see the *readxl* and *XLConnect* package among others), but you can also just go into Excel and export as a CSV file or the like and then read that into R. In general, it's best not to pass around data files as Excel or other spreadsheet format files because (1) Excel is proprietary, so someone may not have Excel and the format is subject to change, (2) Excel imposes limits on the number of rows, (3) one can easily manipulate text files such as CSV using UNIX tools, but this is not possible with an Excel file, (4) Excel files often have more than one sheet, graphs, macros, etc., so they're not a data storage format per se.

8. R can easily interact with databases (SQLite, PostgreSQL, MySQL, Oracle, etc.), querying the database using SQL and returning results to R. More in the big data unit and in the large datasets tutorial mentioned above.

2 Reading data from text files into R

2.1 Core R functions

read.table() is probably the most commonly-used function for reading in data. It reads in delimited files (*read.csv()* and *read.delim()* are special cases of *read.table()*). The key arguments are the delimiter (the *sep* argument) and whether the file contains a header, a line with the variable names. We can use *read.fwf()* to read from a fixed width text file into a data frame.

The most difficult part of reading in such files can be dealing with how R determines the classes of the fields that are read in. There are a number of arguments to *read.table()* and *read.fwf()* that allow the user to control the classes. One difficulty is that character and numeric fields are sometimes read in as factors. Basically *read.table()* tries to read fields in as numeric and if it finds non-numeric and non-NA values, it reads in as a factor. This can be annoying.

Let's work through a couple examples. Before we do that, let's look at the arguments to *read.table()*. Note that *sep=''* separates on any amount of white space. In the code chunk below, I've told *knitr* not to print the output to the PDF; we'll see the full output in class during the demo.

```
getwd()  # a common error is not knowing what directory R is looking at
setwd('../data')
dat <- read.table('RTADataSub.csv', sep = ',', head = TRUE)
sapply(dat, class)
levels(dat[ ,2])
dat2 <- read.table('RTADataSub.csv', sep = ',', head = TRUE,
   na.strings = c("NA", "x"), stringsAsFactors = FALSE)
unique(dat2[ ,2])
## hmmm, what happened to the blank values this time?
which(dat[ ,2] == "")
dat2[which(dat[, 2] == "")[1], ] # deconstruct it!

# using 'colClasses'
sequ <- read.table('hivSequ.csv', sep = ',', header = TRUE,
  colClasses = c('integer','integer','character',
    'character','numeric','integer'))
## let's make sure the coercion worked - sometimes R is obstinant
sapply(sequ, class)
## that made use of the fact that a data frame is a list
```

Note that you can avoid reading in one or more columns by specifying *NULL* as the column class for those columns to be omitted. Also, specifying the *colClasses* argument explicitly should make for faster file reading. Finally, setting `stringsAsFactors=FALSE` is standard practice. You can set that by default to apply generally in your *.Rprofile* using `options(stringsAsFactors = FALSE)`. Or use *readr::read_csv()* as discussed below.

If possible, it's a good idea to look through the input file in the shell or in an editor before reading into R to catch such issues in advance. Using *less* on *RTADataSub.csv* would have revealed these various issues, but note that *RTADataSub.csv* is a 1000-line subset of a much larger file of data available from the kaggle.com website. So more sophisticated use of UNIX utilities as we saw in Unit 2 is often useful before trying to read something into R.

The basic function *scan()* simply reads everything in, ignoring lines, which works well and very quickly if you are reading in a numeric vector or matrix. *scan()* is also useful if your file is free format - i.e., if it's not one line per observation, but just all the data one value after another; in this case you can use *scan()* to read it in and then format the resulting character or numeric vector as a matrix with as many columns as fields in the dataset. Remember that the default is to fill the matrix by column.

If the file is not nicely arranged by field (e.g., if it has ragged lines), we'll need to do some more work. *readLines()* will read in each line into a separate character vector, after which we can process the lines using text manipulation. Here's an example from some US meteorological data where I know from metadata (not provided here) that the 4-11th values are an identifier, the 17-20th are the year, the 22-23rd the month, etc.

```
dat <- readLines('../data/precip.txt')
id <- as.factor(substring(dat, 4, 11) )
year <- substring(dat, 18, 21)
year[1:5]

## [1] "2010" "2010" "2010" "2010" "2010"

class(year)

## [1] "character"

year <- as.integer(substring(dat, 18, 21))
month <- as.integer(substring(dat, 22, 23))
nvalues <- as.integer(substring(dat, 28, 30))
```

Note that for *precip.txt*, reading in using *read.fwf()* would be a good strategy.

R allows you to read in not just from a file but from a more general construct called a *connection*. Here are some examples of connections:

```
dat <- readLines(pipe("ls -al"))
dat <- read.table(pipe("unzip dat.zip"))
dat <- read.csv(gzfile("dat.csv.gz"))
dat <- readLines("http://www.stat.berkeley.edu/~paciorek/index.html")
```

In some cases, you might need to create the connection using *url()* or using the *curl()* function from the *curl* package. Though for the example here, simply passing the URL to *readLines()* does work. (In general, *curl::curl()* provides some nice features for reading off the internet.)

```
wikip1 <- readLines("https://wikipedia.org")
wikip2 <- readLines(url("https://wikipedia.org"))
library(curl)
wikip3 <- readLines(curl("https://wikipedia.org"))
```

If a file is large, we may want to read it in in chunks (of lines), do some computations to reduce the size of things, and iterate. *read.table()*, *read.fwf()* and *readLines()* all have the arguments that let you read in a fixed number of lines. To read-on-the-fly in blocks, we need to first establish the connection and then read from it sequentially.

```
con <- file("../data/precip.txt", "r")
## "r" for 'read' - you can also open files for writing with "w"
## (or "a" for appending)
class(con)
blockSize <- 1000 # obviously this would be large in any real application
nLines <- 300000
for(i in 1:ceiling(nLines / blockSize)){
    lines <- readLines(con, n = blockSize)
    # manipulate the lines and store the key stuff
}
close(con)
```

Here's an example of using *curl()* to do this for a file on the web.

```
URL <- "https://www.stat.berkeley.edu/share/paciorek/2008.csv.gz"
con <- gzcon(curl(URL, open = "r"))
## url() in place of curl() works too
for(i in 1:8) {
        print(i)
        print(system.time(tmp <- readLines(con, n = 100000)))
        print(tmp[1])
}

## [1] 1
##    user  system elapsed
##   0.736   0.008   0.745
## [1] "Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTi
## [1] 2
##    user  system elapsed
##   0.624   0.004   0.631
## [1] "2008,1,29,2,1938,1935,2308,2257,XE,7676,N11176,150,142,104,11,3,SLC
## [1] 3
```

```
##    user  system elapsed
##   0.544   0.000   0.543
## [1] "2008,1,20,7,1540,1525,1651,1637,OO,5703,N227SW,71,72,58,14,15,SBA,S
## [1] 4
##    user  system elapsed
##   0.532   0.000   0.536
## [1] "2008,1,2,3,1313,1250,1443,1425,WN,440,N461WN,150,155,138,18,23,MCO,
## [1] 5
##    user  system elapsed
##   0.532   0.004   0.538
## [1] "2008,1,24,4,1026,1015,1116,1110,MQ,3926,N653AE,50,55,38,6,11,MLI,OR
## [1] 6
##    user  system elapsed
##   0.544   0.000   0.544
## [1] "2008,1,4,5,1129,1125,1352,1350,AA,1145,N438AA,203,205,187,2,4,ORD,S
## [1] 7
##    user  system elapsed
##   0.532   0.008   0.542
## [1] "2008,1,10,4,716,720,1025,1024,DL,1590,N991DL,129,124,107,1,-4,AUS,A
## [1] 8
##    user  system elapsed
##   0.548   0.000   0.552
## [1] "2008,2,15,5,2127,2132,2254,2312,XE,7663,N33182,87,100,71,-18,-5,SLC

close(con)
```

More details on sequential (on-line) processing of large files can be found in the tutorial on large datasets mentioned in the reference list above.

One cool trick that can come in handy is to create a *text connection*. This lets you 'read' from an R character vector as if it were a text file and could be handy for processing text. For example, you could then use *read.fwf()* applied to *con*.

```
dat <- readLines('../data/precip.txt')
con <- textConnection(dat[1], "r")
read.fwf(con, c(3,8,4,2,4,2))

##    V1       V2   V3 V4   V5 V6
```

```
## 1 DLY 1000807 PRCP HI 2010  2
```

We can create connections for writing output too. Just make sure to open the connection first.

2.2 File paths

A few notes on file paths, related to ideas of reproducibility.

1. In general, you don't want to hard-code absolute paths into your code files because those absolute paths won't be available on the machines of anyone you share the code with. Instead, use paths relative to the directory the code file is in, or relative to a baseline directory for the project, e.g.:

```
dat <- read.csv('../data/cpds.csv')
```

2. Be careful with the directory separator in Windows files: you can either do *"C:\\mydir\\file.txt"* or *"C:/mydir/file.txt"*, but not *"C:\mydir\file.txt"*, and note the next comment about avoiding use of '\\' for portability.

3. Using UNIX style directory separators will work in Windows, Mac or Linux, but using Windows style separators is not portable across operating systems.

```
## good: will work on Windows
dat <- read.csv('../data/cpds.csv')
## bad: won't work on Mac or Linux
dat <- read.csv('..\\data\\cpds.csv')
```

4. Even better, use *file.path()* so that paths are constructed specifically for the operating system the user is using:

```
## good: operating-system independent
dat <- read.csv(file.path('..', 'data', 'cpds.csv'))
```

2.3 The *readr* package

readr is intended to deal with some of the shortcomings of the base R functions, such as defaulting to `stringsAsFactors=FALSE`, leaving column names unmodified, and recognizing dates/times. It reads data in much more quickly than the base R equivalents. See this blog post. Some of the readr functions that are analogs to the comparably-named base R functions are *read_csv()*, *read_fwf()*, *read_lines()*, and *read_table()*.

Let's try out *read_csv()* on the airline dataset used in the R bootcamp.

```
library(readr)

##
## Attaching package: 'readr'
## The following object is masked from 'package:curl':
##
##     parse_date

## I'm violating the rule about absolute paths here!!
## (airline.csv is big enough that I don't want to put it in the
##     course repository)
setwd('~/staff/workshops/r-bootcamp-2017/data')
system.time(dat <- read.csv('airline.csv', stringsAsFactors = FALSE))

##     user  system elapsed
##    5.312   0.252   5.563

system.time(dat2 <- read_csv('airline.csv'))

## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   UniqueCarrier = col_character(),
##   TailNum = col_character(),
##   Origin = col_character(),
##   Dest = col_character(),
##   CancellationCode = col_character()
## )
## See spec(...)  for full column specifications.

##     user  system elapsed
##    1.044   0.032   1.077
```

2.4 Reading data quickly

In addition to the tips above, there are a number of packages that allow one to read large data files quickly, in particular *data.table*, *ff*, and *bigmemory*. In general, these provide the ability to load datasets into R without having them in memory, but rather stored in clever ways on disk that allow for fast access. Metadata is stored in R. More on this in the unit on big data and in the tutorial on large datasets mentioned in the reference list above.

3 Webscraping and working with XML and JSON

The new (well, as of 2015) book *XML and Web Technologies for Data Sciences with R* by Deb Nolan (UCB Stats faculty) and Duncan Temple Lang (UCB Stats PhD alumnus and UC Davis Stats faculty) provides extensive information about getting and processing data off of the web, including interacting with web services such as REST and SOAP and programmatically handling authentication.

Here are some UNIX command-line tools to help in webscraping and working with files in formats such as JSON, XML, and HTML: http://jeroenjanssens.com/2013/09/19/seven-command-line-tools-for-data-science.html.

We'll cover a few basic examples in this section, but HTML and XML formatting and navigating the structure of such pages is beyond the scope of what we can cover in detail. The key thing is to know that the tools exist so that you can learn how to use them if faced with such formats.

3.1 Reading HTML

Let's see a brief example of reading in HTML tables. One lesson here is not to write a lot of your own code to do something that someone else has probably already written a package for. Unfortunately, there are some issues with dealing with https-based websites that we need to work around, rather than directly using *readHTMLTable()* as can be done with http-based websites. So we need to use *url()* to get the HTML via https and then use the XML package functionality for parsing the HTML.

```
library(XML)

## Loading required package:  methods

library(curl)
URL <- "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_
html <- readLines(URL)
```

```r
## alternative
## library(RCurl); html <- getURLContent(URL)
tbls <- readHTMLTable(html)
sapply(tbls, nrow)

## NULL NULL
##  243   12

pop <- readHTMLTable(html, which = 1)
head(pop)

##   Rank Country\n(or dependent territory)    Population
## 1    1                    China[Note 2] 1,385,110,000
## 2    2                            India 1,320,530,000
## 3    3            United States[Note 3]   325,669,000
## 4    4                        Indonesia   261,890,900
## 5    5                         Pakistan   208,727,000
## 6    6                           Brazil   207,936,000
##              Date % of world\npopulation
## 1 August 29, 2017                  18.4%
## 2 August 29, 2017                  17.5%
## 3 August 29, 2017                  4.33%
## 4    July 1, 2017                  3.48%
## 5 August 29, 2017                  2.77%
## 6 August 29, 2017                  2.76%
##                        Source
## 1  Official population clock
## 2  Official population clock
## 3  Official population clock
## 4 Official annual projection
## 5  Official population clock
## 6  Official population clock
```

readHTMLTable() works by using *htmlParse()* and then looking for <*table*> tags. In the example above, there were multiple tables, so we need to either specify or (after reading all of them) extract the one of interest. There is a related function, *readHTMLList()*.

It's often useful to be able to extract the hyperlinks in an HTML document. In this example, I'm not sure why the *relative* argument (see `help(getHTMLLinks)`) doesn't seem to work in terms of giving back absolute paths.

```
URL <- "http://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year"
html <- readLines(URL)
links <- getHTMLLinks(html)
head(links, n = 10)

##  [1] "?C=N;O=D"                "?C=M;O=A"
##  [3] "?C=S;O=A"                "?C=D;O=A"
##  [5] "/pub/data/ghcn/daily/" "1763.csv.gz"
##  [7] "1764.csv.gz"             "1765.csv.gz"
##  [9] "1766.csv.gz"             "1767.csv.gz"

links <- getHTMLLinks(html, baseURL = URL, relative = FALSE)
head(links, n = 10)

##  [1] "?C=N;O=D"                "?C=M;O=A"
##  [3] "?C=S;O=A"                "?C=D;O=A"
##  [5] "/pub/data/ghcn/daily/" "1763.csv.gz"
##  [7] "1764.csv.gz"             "1765.csv.gz"
##  [9] "1766.csv.gz"             "1767.csv.gz"
```

More generally, we may want to read an HTML document and parse it into its components (i.e., the HTML elements). Here we use the *XPath* language in the second argument to *getNodeSet()*. XPath can also be used for navigating through XML documents.

```
tutorials <- htmlParse("http://statistics.berkeley.edu/computing/training/tu
listOfANodes <- getNodeSet(tutorials, "//a[@href]")
head(listOfANodes)

## [[1]]
## <a href="#navigation" class="element-invisible element-focusable">Jump to
##
## [[2]]
## <a href="/" title="Home" rel="home" id="logo">
##   <img src="http://statistics.berkeley.edu/sites/all/themes/stats/logo.pn
## </a>
```

```
##
## [[3]]
## <a href="http://berkeley.edu">University of California, Berkeley</a>
##
## [[4]]
## <a href="/" title="Home" rel="home">
##   <span class="sitename-text">Department of Statistics</span>
## </a>
##
## [[5]]
## <a href="/cas">Log in</a>
##
## [[6]]
## <a href="http://github.com/berkeley-scf/tutorial-unix-basics">materials o
```

```
sapply(listOfANodes, xmlGetAttr, "href")[1:10]
```

```
##  [1] "#navigation"
##  [2] "/"
##  [3] "http://berkeley.edu"
##  [4] "/"
##  [5] "/cas"
##  [6] "http://github.com/berkeley-scf/tutorial-unix-basics"
##  [7] "http://youtu.be/pAY6E0FdWUo"
##  [8] "http://github.com/berkeley-scf/tutorial-latex-intro"
##  [9] "http://youtu.be/8khoelwmMwo"
## [10] "http://github.com/berkeley-scf/tutorial-dynamic-docs"
```

```
sapply(listOfANodes, xmlValue)[1:10]
```

```
##  [1] "Jump to navigation"
##  [2] ""
##  [3] "University of California, Berkeley"
##  [4] "Department of Statistics"
##  [5] "Log in"
##  [6] "materials on Github"
##  [7] "screencast"
```

```
##  [8] "materials on Github"
##  [9] "screencast"
## [10] "materials on Github"
```

The XPath syntax above in `getNodeSet()` says to find all of the nodes (i.e., elements) that are named ‘a’ and have attribute *href*.

Here’s another example of extracting specific components of information from a webpage. We can explore the underlying HTML source in advance of writing our code by looking at the page source (e.g., in Firefox see `Developer -> Page Source` and in Chrome `More tools -> View Source`)

```
doc <- htmlParse(readLines("https://www.nytimes.com"))
storyDivs <- getNodeSet(doc, "//h2[@class = 'story-heading']")
sapply(storyDivs, xmlValue)[1:5]

## [1] "Waters Still Rising as Death Toll From Storm Reaches 30"
## [2] "â\u0080\u0098We Want to Do It Better,â\u0080\u0099 Trump Says"
## [3] "A Guide: The Storm So Far"
## [4] "Share Your Hurricane Photos and Videos With Us "
## [5] "Where to Donate to Storm Victims (and How to Avoid Scams) 4:48 PM ET
```

3.2 XML

XML is a markup language used to store data in self-describing (no metadata needed) format, often with a hierarchical structure. It consists of sets of elements (also known as nodes because they generally occur in a hierarchical structure and therefore have parents, children, etc.) with tags that identify/name the elements, with some similarity to HTML. Some examples of the use of XML include serving as the underlying format for Microsoft Office and Google Docs documents and for the KML language used for spatial information in Google Earth.

Here’s a brief example. The book with id attribute *bk101* is an element; the author of the book is also an element that is a child element of the book. The id attribute allows us to uniquely identify the element.

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
```

```
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications with XML.</desc
    </book>
    <book id="bk102">
        <author>Ralls, Kim</author>
        <title>Midnight Rain</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies, an evil sorc
    </book>
</catalog>
```

We can read XML documents into R using *xmlToList()* or *xmlToDataFrame()*. Here's an example of working with lending data from the Kiva lending non-profit. You can see the XML format in a browser at http://api.kivaws.org/v1/loans/newest.xml.

```
doc <- xmlParse("http://api.kivaws.org/v1/loans/newest.xml")
data <- xmlToList(doc, addAttributes = FALSE)
names(data)

## [1] "paging" "loans"

length(data$loans)

## [1] 20

data$loans[[2]][c('name', 'activity', 'sector', 'location', 'loan_amount')]

## $name
## [1] "Ria"
##
## $activity
## [1] "Personal Housing Expenses"
##
```

```
## $sector
## [1] "Housing"
##
## $location
## $location$country_code
## [1] "ID"
##
## $location$country
## [1] "Indonesia"
##
## $location$town
## [1] "Lebak"
##
## $location$geo
## $location$geo$level
## [1] "town"
##
## $location$geo$pairs
## [1] "-5 120"
##
## $location$geo$type
## [1] "point"
##
##
##
## $loan_amount
## [1] "375"
```

```
## let's try to get the loan data into a data frame
loansNode <- xmlRoot(doc)[["loans"]]
length(xmlChildren(loansNode))
```

```
## [1] 20
```

```
loans <- xmlToDataFrame(xmlChildren(loansNode))
dim(loans)
```

```
## [1] 20 20
```

```r
head(loans)
```

```
##        id          name description      status
## 1 1360952      Rukhsana          en fundraising
## 2 1360991           Ria          en fundraising
## 3 1360992 Pakngao Group          en fundraising
## 4 1361019   Sania Latif          en fundraising
## 5 1361027     Magdalena          en fundraising
## 6 1360938          Rona          en fundraising
##   funded_amount basket_amount    image
## 1             0             0 26123741
## 2             0             0 26140281
## 3             0             0 26135711
## 4             0             0 26134441
## 5             0             0 24510031
## 6             0             0 26139491
##                    activity       sector
## 1                 Tailoring     Services
## 2 Personal Housing Expenses      Housing
## 3           Home Appliances Personal Use
## 4                 Tailoring     Services
## 5             General Store       Retail
## 6 Personal Housing Expenses      Housing
##                                        themes
## 1                              Rural Exclusion
## 2                         Water and Sanitation
## 3 GreenWater and SanitationEarth Day Campaign
## 4                              Rural Exclusion
## 5                                         <NA>
## 6                         Water and Sanitation
##
## 1                                  to buy needed stitching inputs such a
## 2   to build a clean water source facility at their home to improve acce
## 3                         to purchase TerraClear water filters so they ca
## 4                                           to acquire raw materials like
## 5                                 to buy sacks of rice, grocery items, an
```

```
## 6 to buy materials like GI sheets, lumber, tiles, paint, a toilet bowl, r
##                                         location
## 1 PKPakistanGujranwalatown32.15 74.183333point
## 2              IDIndonesiaLebaktown-5 120point
## 3                 LALao PDRLaostown18 105point
## 4 PKPakistanGujranwalatown32.15 74.183333point
## 5    PHPhilippinesBinan, Lagunatown13 122point
## 6    PHPhilippinesGetafe, Boholtown13 122point
##   partner_id          posted_date
## 1        455 2017-08-30T00:50:03Z
## 2        406 2017-08-30T00:50:03Z
## 3        393 2017-08-30T00:50:03Z
## 4        455 2017-08-30T00:50:03Z
## 5        144 2017-08-30T00:50:03Z
## 6        125 2017-08-30T00:50:02Z
##   planned_expiration_date loan_amount borrower_count
## 1    2017-09-29T00:50:03Z         300              1
## 2    2017-09-29T00:50:03Z         375              1
## 3    2017-09-29T00:50:03Z        3800             34
## 4    2017-09-29T00:50:03Z         300              1
## 5    2017-09-29T00:50:02Z         400              1
## 6    2017-09-29T00:50:02Z         250              1
##   lender_count bonus_credit_eligibility tags
## 1            0                        0
## 2            0                        0
## 3            0                        0
## 4            0                        0
## 5            0                        1
## 6            0                        1
```

```
## suppose we only want the country locations of the loans
countries <- sapply(xmlChildren(loansNode), function(node)
   xmlValue(node[['location']][['country']]))
countries[1:10]
```

```
##          loan          loan          loan          loan
##    "Pakistan"   "Indonesia"     "Lao PDR"    "Pakistan"
```

```
##          loan          loan          loan          loan
## "Philippines" "Philippines" "Philippines"    "Pakistan"
##          loan          loan
##    "Pakistan"    "Pakistan"

## this fails because node is not a standard list:
countries <- sapply(xmlChildren(loansNode), function(node)
   xmlValue(node$location$country))

## Error in node$location:  object of type 'externalptr' is not subsettable
```

XML documents have a tree structure with information at nodes. As above with HTML, one can use the *XPath* language for navigating the tree and finding and extracting information from the node(s) of interest.

xml2 is a new package from RStudio for reading XML and HTML.

3.3 Reading JSON

JSON files are structured as “attribute-value” pairs (aka “key-value” pairs), often with a hierarchical structure. Here’s a brief example:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
```

```
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

A set of key-value pairs is a named array and is placed inside braces (squiggly brackets). Note the nestedness of arrays within arrays (e.g., address within the overarching person array and the use of square brackets for unnamed arrays (i.e., vectors of information), as well as the use of different types: character strings, numbers, null, and (not shown) boolean/logical values. JSON and XML can be used in similar ways, but JSON is less verbose than XML.

We can read JSON into R using *fromJSON()* in the *jsonlite* package. Let's play again with the Kiva data. The same data that we had worked with in XML format is also available in JSON format: http://api.kivaws.org/v1/loans/newest.json.

```
library(jsonlite)
data <- fromJSON("http://api.kivaws.org/v1/loans/newest.json")
names(data)

## [1] "paging" "loans"

class(data$loans) # nice!

## [1] "data.frame"

head(data$loans)

##        id          name languages      status
## 1 1360952      Rukhsana        en fundraising
## 2 1360991           Ria        en fundraising
## 3 1360992 Pakngao Group        en fundraising
## 4 1361019   Sania Latif        en fundraising
## 5 1361027     Magdalena        en fundraising
## 6 1360938          Rona        en fundraising
##   funded_amount basket_amount image.id
```

```
## 1             0             0  2612374
## 2             0             0  2614028
## 3             0             0  2613571
## 4             0             0  2613444
## 5             0             0  2451003
## 6             0             0  2613949
##   image.template_id                  activity
## 1                 1                 Tailoring
## 2                 1 Personal Housing Expenses
## 3                 1           Home Appliances
## 4                 1                 Tailoring
## 5                 1             General Store
## 6                 1 Personal Housing Expenses
##         sector
## 1     Services
## 2      Housing
## 3 Personal Use
## 4     Services
## 5       Retail
## 6      Housing
##                                                  themes
## 1                                       Rural Exclusion
## 2                                  Water and Sanitation
## 3 Green, Water and Sanitation, Earth Day Campaign
## 4                                       Rural Exclusion
## 5                                                  NULL
## 6                                  Water and Sanitation
##
## 1                                       to buy needed stitching inputs such as
## 2   to build a clean water source facility at their home to improve acces
## 3                              to purchase TerraClear water filters so they ca
## 4                                                to acquire raw materials like
## 5                                     to buy sacks of rice, grocery items, and
## 6 to buy materials like GI sheets, lumber, tiles, paint, a toilet bowl, r
##   location.country_code location.country location.town
## 1                    PK         Pakistan    Gujranwala
```

```
## 2                    ID         Indonesia          Lebak
## 3                    LA           Lao PDR           Laos
## 4                    PK          Pakistan     Gujranwala
## 5                    PH       Philippines Binan, Laguna
## 6                    PH       Philippines Getafe, Bohol
##   location.geo.level location.geo.pairs
## 1               town    32.15 74.183333
## 2               town             -5 120
## 3               town             18 105
## 4               town    32.15 74.183333
## 5               town             13 122
## 6               town             13 122
##   location.geo.type partner_id          posted_date
## 1             point        455 2017-08-30T00:50:03Z
## 2             point        406 2017-08-30T00:50:03Z
## 3             point        393 2017-08-30T00:50:03Z
## 4             point        455 2017-08-30T00:50:03Z
## 5             point        144 2017-08-30T00:50:03Z
## 6             point        125 2017-08-30T00:50:02Z
##   planned_expiration_date loan_amount borrower_count
## 1    2017-09-29T00:50:03Z         300              1
## 2    2017-09-29T00:50:03Z         375              1
## 3    2017-09-29T00:50:03Z        3800             34
## 4    2017-09-29T00:50:03Z         300              1
## 5    2017-09-29T00:50:02Z         400              1
## 6    2017-09-29T00:50:02Z         250              1
##   lender_count bonus_credit_eligibility tags
## 1            0                    FALSE NULL
## 2            0                    FALSE NULL
## 3            0                    FALSE NULL
## 4            0                    FALSE NULL
## 5            0                     TRUE NULL
## 6            0                     TRUE NULL
```

One disadvantage of JSON is that it is not set up to deal with missing values, infinity, etc.