

Reading and Writing to/from R

August 30, 2015

References:

- Adler
- Chambers
- [R intro manual](#) on CRAN (R-intro).
- Nolan and Temple Lang, XML and Web Technologies for Data Sciences with R.
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies.
- [R Data Import/Export manual](#) on CRAN (R-data).

1 Data storage and formats (outside R)

At this point, we're going to turn to reading data into R and manipulating text, including regular expressions. We'll focus on doing these manipulations in R, but the concepts involved in reading in data, database manipulations, and regular expressions are common to other languages, so familiarity with these in R should allow you to pick up other tools more easily. The main downside to working with datasets in R 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) is 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 was difficult to deal with in the shell in PS1, but `read.table()` in R handles this situation.

- (a) 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()*), *SAS* (*read.ssd()*) and, for *XPORT* files, *read.xport()*, and *dbf* (a common database format) (*read.dbf()*), 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.

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('RTADDataSub.csv', sep = ',', head = TRUE)
sapply(dat, class)
levels(dat[,2])
```

```

dat2 <- read.table('RTADDataSub.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!
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.

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 *RTADDataSub.csv* would have revealed these various issues, but note that *RTADDataSub.csv* is a 1000-line subset of a much larger file of data available from the kaggle.com website.

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.

```

a <-5
print(a)

## [1] 5

```

```

dat <- readLines('../data/precip.txt')
id <- as.factor(substring(dat, 4, 11) )
year <- substring(dat, 17, 20)
year[1:5]

## [1] "I201" "I201" "I201" "I201" "I201"

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")

```

The *curl()* function in the *curl* package provides some nice features for reading off the internet, including being able to use https.

```

library(curl)
# equivalent to readLines(url("https://cran.r-project.org")):
cran <- readLines("https://cran.r-project.org")
cran <- readLines(curl("https://cran.r-project.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.

```

library(jsonlite)

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

URL <- "http://www.stat.berkeley.edu/share/paciorek/2008.csv.gz"
con <- gzcon(curl(URL, open = "r"))
for(i in 1:10) {
  print(i)
  print(system.time(tmp <- readLines(con, n = 100000)))
  print(tmp[1])
}

## [1] 1
##      user  system elapsed
## 0.675    0.008    0.682
## [1] "Year,Month,DayOfMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime"
## [1] 2

```

```

##      user      system elapsed
##    0.604    0.004    0.608
## [1] "2008,1,29,2,1938,1935,2308,2257,XE,7676,N11176,150,142,104,11,3,SLC,
## [1] 3
##      user      system elapsed
##    0.524    0.000    0.524
## [1] "2008,1,20,7,1540,1525,1651,1637,OO,5703,N227SW,71,72,58,14,15,SBA,S
## [1] 4
##      user      system elapsed
##    0.506    0.003    0.510
## [1] "2008,1,2,3,1313,1250,1443,1425,WN,440,N461WN,150,155,138,18,23,MCO,S
## [1] 5
##      user      system elapsed
##    0.512    0.000    0.511
## [1] "2008,1,24,4,1026,1015,1116,1110,MQ,3926,N653AE,50,55,38,6,11,MLI,ORI
## [1] 6
##      user      system elapsed
##    0.522    0.004    0.527
## [1] "2008,1,4,5,1129,1125,1352,1350,AA,1145,N438AA,203,205,187,2,4,ORD,S
## [1] 7
##      user      system elapsed
##    0.514    0.001    0.515
## [1] "2008,1,10,4,716,720,1025,1024,DL,1590,N991DL,129,124,107,1,-4,AUS,A
## [1] 8
##      user      system elapsed
##    0.517    0.000    0.516
## [1] "2008,2,15,5,2127,2132,2254,2312,XE,7663,N33182,87,100,71,-18,-5,SLC,
## [1] 9
##      user      system elapsed
##    0.530    0.000    0.529
## [1] "2008,2,23,6,1056,1103,1218,1225,OO,3761,N448SW,82,82,58,-7,-7,SLC,B
## [1] 10
##      user      system elapsed
##    0.518    0.000    0.519
## [1] "2008,2,22,5,818,823,925,942,DL,1140,N967DL,127,139,105,-17,-5,ATL,M

```

```
close(con)
```

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

### 2.2 The readr package
```

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

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”.

2.2 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

setwd('~/.staff/workshops/r-bootcamp-2015/data')
system.time(dat <- read_csv('airline.csv', stringsAsFactors = FALSE))
```



```
##      user  system elapsed
##    5.604   0.116   5.720

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

##      user  system elapsed
##    1.270   0.023   1.294

#####
# 3: Webscraping and working with XML and JSON
#####

### 3.1 Reading HTML and XML
```

2.3 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.

3 Webscraping and working with XML and JSON

The new book *XML and Web Technologies for Data Sciences with R* by Deb Nolan (UCB Stats faculty) and Duncan Temple Lang (UCB Stats PhD alumnus) provides extensive information about getting and processing data off of the web, including interacting with web services such as SOAP and REST 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 examples of this sort of thing briefly here, but going into the details of 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 these tools exist so that you can learn how to use them if faced with such formats.

3.1 Reading HTML and XML

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 that we need to work around, rather than directly using `readHTMLTable` as can be done with http. So we need to use `curl()` to get the HTML via https and then use the XML package functionality for parsing the HTML.

```
library(XML)
library(curl)
URL <- "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_
html <- readLines(curl(URL))
# alternative
# library(RCurl); html <- getURLContent(URL)
tbls <- readHTMLTable(html)

## Warning: closing unused connection 6 (https://en.wikipedia.org/wiki/List

sapply(tbls, nrow)

## $`NULL`
## [1] 257
##
## $`NULL`
## NULL
##
## $`NULL`
## NULL
##
## $`NULL`
## [1] 18
##
## $`NULL`
## [1] 16

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

```
##      Rank Country (or dependent territory)      Population
## 1      1                China[Note 2] 1,371,790,000
## 2      2                India 1,276,370,000
## 3      3          United States[Note 3]   321,719,000
## 4      4                Indonesia   255,461,700
## 5      5                Brazil   204,825,000
## 6      6                Pakistan   190,768,000
##
##      Date % of world\npopulation
## 1 August 30, 2015      18.9%
## 2 August 30, 2015      17.6%
## 3 August 30, 2015      4.43%
## 4      July 1, 2015      3.52%
## 5 August 30, 2015      2.82%
## 6 August 30, 2015      2.63%
##
##      Source
## 1 Official population clock
## 2 Official population clock
## 3 Official population clock
## 4      Official 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 doesn't seem to work in terms of giving back absolute paths.

```
links <- getHTMLLinks("http://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year")
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. Here we use the *XPath* language in the second argument to `getNodeSet`. XPath can also be used for navigating through XML documents.

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

## [[1]]
## <a href="#navigation" class="element-invisible element-focusable">Jump to
##
## [[2]]
## <a href="/" title="Home" rel="home" id="logo">
##   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")[1:20]

## [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"
## [11] "http://github.com/berkeley-scf/tutorial-git-basics"
## [12] "http://github.com/berkeley-scf/tutorial-using-bash"
## [13] "http://github.com/berkeley-scf/tutorial-string-processing"
## [14] "http://github.com/berkeley-scf/tutorial-make-workflows"
## [15] "http://youtu.be/-Cp3jBBHQBE"
## [16] "/computing"
## [17] "/computing/mission"
## [18] "/computing/accounts"
## [19] "/computing/help"
## [20] "/computing/software"
```

```
sapply(listOfANodes, xmlValue)[1:20]
```

```
## [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"
## [11] "materials on Github"
## [12] "materials on Github"
## [13] "materials on Github"
## [14] "materials on Github"
## [15] "screencast"
## [16] "SCF"
## [17] "Mission"
## [18] "Accounts"
## [19] "Help"
```

```
## [20] "Software"
```

The XPath syntax above in `getNodeSet()` says to find all of the nodes 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)

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

## [1] "Fed, Challenged on Left and Right, Faces a Choice on Rates"
## [2] "Behind Tianjin\nBlasts, Corruption\nand Flouted Rules"
## [3] "Islamic State\nDamages Temple of\nBaal in Palmyra"
## [4] "Banking Ties Could\nHurt Biden in Race\nWith Populist Tone"
## [5] "Wall St. Policy Poses a Challenge for Presidential Candidates 9:02 1
```

XML is a markup language used to store data in self-describing (no metadata needed) way, often with a hierarchical structure. It consists of sets of elements (also known as nodes) 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.</des
  </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 sorcerer,
  </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]]

## $id
## [1] "939631"
##
## $name
## [1] "Juana Francisca"
##
## $description
## $description$languages
## $description$languages[[1]]
## [1] "es"
##
## $description$languages[[2]]

```

```

## [1] "en"
##
## attr(,"type")
## [1] "list"
##
##
## $status
## [1] "fundraising"
##
## $funded_amount
## [1] "0"
##
## $basket_amount
## [1] "0"
##
## $image
## $image$id
## [1] "1966626"
##
## $image$template_id
## [1] "1"
##
##
## $activity
## [1] "Retail"
##
## $sector
## [1] "Retail"
##
## $use
## [1] "to purchase shoes, underwear, and a variety of clothing for her clients"
##
## $location
## $location$country_code
## [1] "NI"
##

```



```
## $location$country
## [1] "Nicaragua"
##
## $location$town
## [1] "Masaya"
##
## $location$geo
## $location$geo$level
## [1] "town"
##
## $location$geo$pairs
## [1] "11.966667 -86.1"
##
## $location$geo$type
## [1] "point"
##
##
##
## $partner_id
## [1] "120"
##
## $posted_date
## [1] "2015-08-31T02:50:02Z"
##
## $planned_expiration_date
## [1] "2015-09-30T02:50:02Z"
##
## $loan_amount
## [1] "350"
##
## $borrower_count
## [1] "1"
##
## $lender_count
## [1] "0"
##
```

```
## $bonus_credit_eligibility
## [1] "1"
##
## $tags
## NULL

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

	id	name	description	status
## 1	938576	Firuz	ruen	fundraising
## 2	939631	Juana Francisca	esen	fundraising
## 3	939630	Enzo	esen	fundraising
## 4	939593	Jose Lino	esen	fundraising
## 5	939594	Anonymous	esen	fundraising
## 6	938578	Rayhona	ruen	fundraising

	funded_amount	basket_amount	image
## 1	0	0	19649641
## 2	0	0	19666261
## 3	0	0	19666271
## 4	0	0	19665861
## 5	0	0	19665871
## 6	0	0	19649621

	activity	sector
## 1	Cattle	Agriculture
## 2	Retail	Retail
## 3	Higher education costs	Education
## 4	Manufacturing	Manufacturing
## 5	Clothing Sales	Clothing
## 6	Higher education costs	Education


```
##
## 1 to obtain cat
## 2 to purchase shoes, underwear, and a variety of clothing for her clients
## 3 To pay for university
## 4 to buy cement, rebar, and
```

```

## 5 to purchase more wholesale cloth
## 6 to pay for her son's annual tuition
## location
## 1 TJTajikistanShahrituztown39 71point
## 2 NINicaraguaMasayatown11.966667 -86.1point
## 3 PYParaguayCaaguazútown-22.993333 -57.996389point
## 4 SVE1 Salvadorcountry13.833333 -88.916667point
## 5 SVE1 Salvadorcountry13.833333 -88.916667point
## 6 TJTajikistanKhurosontown39 71point
## partner_id posted_date
## 1 63 2015-08-31T02:50:02Z
## 2 120 2015-08-31T02:50:02Z
## 3 58 2015-08-31T02:30:02Z
## 4 81 2015-08-31T02:10:03Z
## 5 81 2015-08-31T02:10:03Z
## 6 63 2015-08-31T02:10:02Z
## planned_expiration_date loan_amount borrower_count
## 1 2015-09-30T02:50:02Z 1200 1
## 2 2015-09-30T02:50:02Z 350 1
## 3 2015-09-30T02:30:02Z 575 1
## 4 2015-09-30T02:10:03Z 500 1
## 5 2015-09-30T02:10:03Z 1600 1
## 6 2015-09-30T02:10:02Z 1050 1
## lender_count bonus_credit_eligibility tags
## 1 0 1
## 2 0 1
## 3 0 1
## 4 0 1
## 5 0 1
## 6 0 1 volunteer_pick
## themes
## 1 <NA>
## 2 <NA>
## 3 Higher Education
## 4 <NA>
## 5 Vulnerable GroupsYouth

```

```
## 6          Higher Education

# suppose we only want the country locations of the loans
countries <- sapply(xmlChildren(loansNode), function(node)
  xmlValue(node[['location']][['country']]))
countries <- sapply(xmlChildren(loansNode), function(node)
  xmlValue(node$location$country)) # node is not a standard list...

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

### 3.2 Reading JSON
```

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. Finally *rvest* is a package that helps with scraping information from HTML pages.

3.2 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", nullValue = NA)

## Error in simplify(obj, simplifyVector = simplifyVector, simplifyDataFrame
= simplifyDataFrame, : unused argument (nullValue = NA)

names(data)

## [1] "paging" "loans"

class(data$loans) # nice!

## [1] "list"

head(data$loans)

## [[1]]
## [[1]]$id
## [1] "938576"
##

```

```

## [[1]]$name
## [1] "Firuz"
##
## [[1]]$description
## [[1]]$description$languages
## [[1]]$description$languages[[1]]
## [1] "ru"
##
## [[1]]$description$languages[[2]]
## [1] "en"
##
## attr("type")
## [1] "list"
##
##
## [[1]]$status
## [1] "fundraising"
##
## [[1]]$funded_amount
## [1] "0"
##
## [[1]]$basket_amount
## [1] "0"
##
## [[1]]$image
## [[1]]$image$id
## [1] "1964964"
##
## [[1]]$image$template_id
## [1] "1"
##
##
## [[1]]$activity
## [1] "Cattle"
##
## [[1]]$sector

```

```

## [1] "Agriculture"
##
## [[1]]$use
## [1] "to obtain cattle."
##
## [[1]]$location
## [[1]]$location$country_code
## [1] "TJ"
##
## [[1]]$location$country
## [1] "Tajikistan"
##
## [[1]]$location$town
## [1] "Shahrituz"
##
## [[1]]$location$geo
## [[1]]$location$geo$level
## [1] "town"
##
## [[1]]$location$geo$pairs
## [1] "39 71"
##
## [[1]]$location$geo$type
## [1] "point"
##
##
##
## [[1]]$partner_id
## [1] "63"
##
## [[1]]$posted_date
## [1] "2015-08-31T02:50:02Z"
##
## [[1]]$planned_expiration_date
## [1] "2015-09-30T02:50:02Z"
##

```

```

## [[1]]$loan_amount
## [1] "1200"
##
## [[1]]$borrower_count
## [1] "1"
##
## [[1]]$lender_count
## [1] "0"
##
## [[1]]$bonus_credit_eligibility
## [1] "1"
##
## [[1]]$tags
## NULL
##
##
## [[2]]
## [[2]]$id
## [1] "939631"
##
## [[2]]$name
## [1] "Juana Francisca"
##
## [[2]]$description
## [[2]]$description$languages
## [[2]]$description$languages[[1]]
## [1] "es"
##
## [[2]]$description$languages[[2]]
## [1] "en"
##
## attr(,"type")
## [1] "list"
##
##
## [[2]]$status

```



```

## [1] "fundraising"
##
## [[2]]$funded_amount
## [1] "0"
##
## [[2]]$basket_amount
## [1] "0"
##
## [[2]]$image
## [[2]]$image$id
## [1] "1966626"
##
## [[2]]$image$template_id
## [1] "1"
##
##
## [[2]]$activity
## [1] "Retail"
##
## [[2]]$sector
## [1] "Retail"
##
## [[2]]$use
## [1] "to purchase shoes, underwear, and a variety of clothing for her clients"
##
## [[2]]$location
## [[2]]$location$country_code
## [1] "NI"
##
## [[2]]$location$country
## [1] "Nicaragua"
##
## [[2]]$location$town
## [1] "Masaya"
##
## [[2]]$location$geo

```

```

## [[2]]$location$geo$level
## [1] "town"
##
## [[2]]$location$geo$pairs
## [1] "11.966667 -86.1"
##
## [[2]]$location$geo$type
## [1] "point"
##
##
##
## [[2]]$partner_id
## [1] "120"
##
## [[2]]$posted_date
## [1] "2015-08-31T02:50:02Z"
##
## [[2]]$planned_expiration_date
## [1] "2015-09-30T02:50:02Z"
##
## [[2]]$loan_amount
## [1] "350"
##
## [[2]]$borrower_count
## [1] "1"
##
## [[2]]$lender_count
## [1] "0"
##
## [[2]]$bonus_credit_eligibility
## [1] "1"
##
## [[2]]$tags
## NULL
##
##

```

```

## [[3]]
## [[3]]$id
## [1] "939630"
##
## [[3]]$name
## [1] "Enzo"
##
## [[3]]$description
## [[3]]$description$languages
## [[3]]$description$languages[[1]]
## [1] "es"
##
## [[3]]$description$languages[[2]]
## [1] "en"
##
## attr("type")
## [1] "list"
##
##
## [[3]]$status
## [1] "fundraising"
##
## [[3]]$funded_amount
## [1] "0"
##
## [[3]]$basket_amount
## [1] "0"
##
## [[3]]$image
## [[3]]$image$id
## [1] "1966627"
##
## [[3]]$image$template_id
## [1] "1"
##
##

```

```

## [[3]]$activity
## [1] "Higher education costs"
##
## [[3]]$sector
## [1] "Education"
##
## [[3]]$themes
## [[3]]$themes[[1]]
## [1] "Higher Education"
##
## attr("type")
## [1] "list"
##
## [[3]]$use
## [1] "To pay for university fees."
##
## [[3]]$location
## [[3]]$location$country_code
## [1] "PY"
##
## [[3]]$location$country
## [1] "Paraguay"
##
## [[3]]$location$town
## [1] "Caaguazú"
##
## [[3]]$location$geo
## [[3]]$location$geo$level
## [1] "town"
##
## [[3]]$location$geo$pairs
## [1] "-22.993333 -57.996389"
##
## [[3]]$location$geo$type
## [1] "point"
##

```

```

##
##
## [[3]]$partner_id
## [1] "58"
##
## [[3]]$posted_date
## [1] "2015-08-31T02:30:02Z"
##
## [[3]]$planned_expiration_date
## [1] "2015-09-30T02:30:02Z"
##
## [[3]]$loan_amount
## [1] "575"
##
## [[3]]$borrower_count
## [1] "1"
##
## [[3]]$lender_count
## [1] "0"
##
## [[3]]$bonus_credit_eligibility
## [1] "1"
##
## [[3]]$tags
## NULL
##
##
## [[4]]
## [[4]]$id
## [1] "939593"
##
## [[4]]$name
## [1] "Jose Lino"
##
## [[4]]$description
## [[4]]$description$languages

```

```

## [[4]]$description$languages[[1]]
## [1] "es"
##
## [[4]]$description$languages[[2]]
## [1] "en"
##
## attr(,"type")
## [1] "list"
##
##
## [[4]]$status
## [1] "fundraising"
##
## [[4]]$funded_amount
## [1] "0"
##
## [[4]]$basket_amount
## [1] "0"
##
## [[4]]$image
## [[4]]$image$id
## [1] "1966586"
##
## [[4]]$image$template_id
## [1] "1"
##
##
## [[4]]$activity
## [1] "Manufacturing"
##
## [[4]]$sector
## [1] "Manufacturing"
##
## [[4]]$use
## [1] "to buy cement, rebar, and sand"
##

```

```
## [[4]]$location
## [[4]]$location$country_code
## [1] "SV"
##
## [[4]]$location$country
## [1] "El Salvador"
##
## [[4]]$location$geo
## [[4]]$location$geo$level
## [1] "country"
##
## [[4]]$location$geo$pairs
## [1] "13.833333 -88.916667"
##
## [[4]]$location$geo$type
## [1] "point"
##
##
##
## [[4]]$partner_id
## [1] "81"
##
## [[4]]$posted_date
## [1] "2015-08-31T02:10:03Z"
##
## [[4]]$planned_expiration_date
## [1] "2015-09-30T02:10:03Z"
##
## [[4]]$loan_amount
## [1] "500"
##
## [[4]]$borrower_count
## [1] "1"
##
## [[4]]$lender_count
## [1] "0"
```

```

##
## [[4]]$bonus_credit_eligibility
## [1] "1"
##
## [[4]]$tags
## NULL
##
##
## [[5]]
## [[5]]$id
## [1] "939594"
##
## [[5]]$name
## [1] "Anonymous"
##
## [[5]]$description
## [[5]]$description$languages
## [[5]]$description$languages[[1]]
## [1] "es"
##
## [[5]]$description$languages[[2]]
## [1] "en"
##
## attr("type")
## [1] "list"
##
##
## [[5]]$status
## [1] "fundraising"
##
## [[5]]$funded_amount
## [1] "0"
##
## [[5]]$basket_amount
## [1] "0"
##

```



```

## [[5]]$image
## [[5]]$image$id
## [1] "1966587"
##
## [[5]]$image$template_id
## [1] "1"
##
##
## [[5]]$activity
## [1] "Clothing Sales"
##
## [[5]]$sector
## [1] "Clothing"
##
## [[5]]$themes
## [[5]]$themes[[1]]
## [1] "Vulnerable Groups"
##
## [[5]]$themes[[2]]
## [1] "Youth"
##
## attr("type")
## [1] "list"
##
## [[5]]$use
## [1] "to purchase more wholesale clothing."
##
## [[5]]$location
## [[5]]$location$country_code
## [1] "SV"
##
## [[5]]$location$country
## [1] "El Salvador"
##
## [[5]]$location$geo
## [[5]]$location$geo$level

```

```

## [1] "country"
##
## [[5]]$location$geo$pairs
## [1] "13.833333 -88.916667"
##
## [[5]]$location$geo$type
## [1] "point"
##
##
##
## [[5]]$partner_id
## [1] "81"
##
## [[5]]$posted_date
## [1] "2015-08-31T02:10:03Z"
##
## [[5]]$planned_expiration_date
## [1] "2015-09-30T02:10:03Z"
##
## [[5]]$loan_amount
## [1] "1600"
##
## [[5]]$borrower_count
## [1] "1"
##
## [[5]]$lender_count
## [1] "0"
##
## [[5]]$bonus_credit_eligibility
## [1] "1"
##
## [[5]]$tags
## NULL
##
##
## [[6]]

```

```
## [[6]]$id
## [1] "938578"
##
## [[6]]$name
## [1] "Rayhona"
##
## [[6]]$description
## [[6]]$description$languages
## [[6]]$description$languages[[1]]
## [1] "ru"
##
## [[6]]$description$languages[[2]]
## [1] "en"
##
## attr("type")
## [1] "list"
##
##
## [[6]]$status
## [1] "fundraising"
##
## [[6]]$funded_amount
## [1] "0"
##
## [[6]]$basket_amount
## [1] "0"
##
## [[6]]$image
## [[6]]$image$id
## [1] "1964962"
##
## [[6]]$image$template_id
## [1] "1"
##
##
## [[6]]$activity
```

```

## [1] "Higher education costs"
##
## [[6]]$sector
## [1] "Education"
##
## [[6]]$themes
## [[6]]$themes[[1]]
## [1] "Higher Education"
##
## attr("type")
## [1] "list"
##
## [[6]]$use
## [1] "to pay for her son's annual tuition."
##
## [[6]]$location
## [[6]]$location$country_code
## [1] "TJ"
##
## [[6]]$location$country
## [1] "Tajikistan"
##
## [[6]]$location$town
## [1] "Khuroson"
##
## [[6]]$location$geo
## [[6]]$location$geo$level
## [1] "town"
##
## [[6]]$location$geo$pairs
## [1] "39 71"
##
## [[6]]$location$geo$type
## [1] "point"
##
##

```

```
##
## [[6]]$partner_id
## [1] "63"
##
## [[6]]$posted_date
## [1] "2015-08-31T02:10:02Z"
##
## [[6]]$planned_expiration_date
## [1] "2015-09-30T02:10:02Z"
##
## [[6]]$loan_amount
## [1] "1050"
##
## [[6]]$borrower_count
## [1] "1"
##
## [[6]]$lender_count
## [1] "0"
##
## [[6]]$bonus_credit_eligibility
## [1] "1"
##
## [[6]]$tags
## [[6]]$tags[[1]]
## [[6]]$tags[[1]]$name
## [1] "volunteer_pick"
##
##
## attr(,"type")
## [1] "list"

### 3.3 Using web APIs to get data

### 3.3.1 HTTP requests
```

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

3.3 Using web APIs to get data

Here we'll see briefly some examples of making requests over the Web to get data. We'll see simple http requests, as well as use of RESTful and SOAP APIs. The package *RCurl* is the main package useful for a wide variety of such functionality. Note that all of the functionality I describe below is also possible within bash using either *wget* or *curl*.

We've already seen some basic downloading of html from webpages, which uses the HTTP request GET.

3.3.1 HTTP requests

First as was noted as a sidenote earlier, we can use *RCurl* to access secure http pages (i.e., pages that use https). This may not be possible or may be more difficult when using base R functionality.

Here `getURLContent` makes a GET request.

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

## Loading required package: bitops

html <- getURLContent(URL)
tbls <- readHTMLTable(html)
```

Sometime specific information can be downloaded simply by constructing a static URL. Suppose we want to get data off of Yahoo Finance. By going to <http://finance.yahoo.com/q/hp?s=DATA+Historical+Pr> we can enter a date range and a company and see the HTML output, which we could download and extract the data from using tools seen in this Unit. But often we'll see a download link (or we can see the HTTP request by using features of the browser (e.g., Ctrl-Shift-q in Firefox or More Tools -> Developer Tools -> Network in Chrome) and if we look at the URL associated with that link, we can see it looks like this: <http://real-chart.finance.yahoo.com/table.csv?s=AAPL&d=7&e=30>. So we could more easily download the data using that URL, which we can fairly easily construct using string processing in bash, R, or Python (more in Unit 4). A more sophisticated way to do the download is to pass the request in a structured way with named input parameters. This request is easier to construct programmatically.

```
txt <- getForm("http://ichart.finance.yahoo.com/table.csv",
s = "AAPL", a = 2, b = 27, c = 2014, d = 7, e = 30, f = 2015, g = "d", ignor
aapl <- read.csv(textConnection(txt))
head(aapl)
```

##		Date	Open	High	Low	Close	Volume
## 1		2015-08-28	112.17	113.31	111.54	113.29	52896400
## 2		2015-08-27	112.23	113.24	110.02	112.92	83265100
## 3		2015-08-26	107.09	109.89	105.05	109.69	96226300
## 4		2015-08-25	111.11	111.11	103.50	103.74	102240200
## 5		2015-08-24	94.87	103.45	92.00	103.12	161454200
## 6		2015-08-21	110.43	111.90	105.65	105.76	126289200
##		Adj.Close					
## 1		113.29					
## 2		112.92					
## 3		109.69					
## 4		103.74					
## 5		103.12					
## 6		105.76					

In some cases we may need to send a lot of information as part of the URL in a GET request. If it gets to be too long (e.g., more than 2048 characters) many web servers will reject the request. Instead we may need to use http POST. A typical request would have syntax like this, supposing that the inputs were named *start-year* and *end-year*.

```
URL <- "http://somewhere.com"
txt <- postForm(URL, "start-year" = "1995", "end-year" = "2005", style = "p
result <- readHTMLTable(txt, header = TRUE)

### 3.3.2 REST- and SOAP-based web services
```

RCurl can handle other kinds of HTTP requests such as PUT and DELETE. Finally, some websites use cookies to keep track of users and you may need to download a cookie in the first interaction with the HTTP server and then send that cookie with later interactions. More details are available in the Nolan and Temple Lang book.

3.3.2 REST- and SOAP-based web services

While webscraping with requests such as just described can work well, it was a bit convoluted. We basically needed to deconstruct the queries a browser makes and then mimic that behavior, in some cases having to parse HTML output to get at data. If the webpage changes even a little bit, our carefully constructed query syntax may fail. An alternative is to use a web service specifically designed to serve data or allow other interactions via an Applications Programming Interface

(API). Both REST and SOAP use HTTP requests; we'll focus on REST as it is more common and simpler.

When using REST, we access *resources*, which might be a Facebook account or a database of stock quotes. The resource may return information in the form of an HTML file or JSON, CSV or something else. REST generally uses XML or JSON as the format for the request and what is returned.

Let's see an example of accessing climate model output data from the World Bank. The API is documented here: <http://data.worldbank.org/developers/climate-data-api>. Following that documentation we can download monthly average precipitation data for 1980-1999 for the US (ISO3 code 'USA').

```
times <- c(1980, 1999)
countryCode <- 'USA'
baseURL <- "http://climatedataapi.worldbank.org/climateweb/rest/v1/country"
type <- "mavg"
var <- "pr"
data <- read.csv(paste(baseURL, type, var, times[1], times[2], paste0(countryCode, ".csv")))
head(data)
```

		GCM	var	from_year	to_year	Jan	Feb
## 1	bccr_bcm2_0	pr	1980	1999	54.08543	50.10967	
## 2	cccma_cgcm3_1	pr	1980	1999	62.24072	57.24048	
## 3	cnrm_cm3	pr	1980	1999	64.41341	57.52318	
## 4	csiro_mk3_5	pr	1980	1999	69.54382	64.10854	
## 5	gfdl_cm2_0	pr	1980	1999	67.70540	66.69405	
## 6	gfdl_cm2_1	pr	1980	1999	69.01939	61.67754	
##	Mar	Apr	May	Jun	Jul	Aug	
## 1	59.51437	65.55661	70.43494	68.00115	74.48428	74.43676	
## 2	60.46462	59.70685	65.48142	67.01422	68.07160	65.01432	
## 3	70.80297	80.30452	91.53901	92.17371	97.61402	90.73969	
## 4	66.03074	57.06771	65.87576	62.80174	61.71441	64.88963	
## 5	73.53323	75.44286	87.80408	85.86737	74.33900	67.00282	
## 6	67.24041	66.84488	70.03789	71.76724	69.27572	65.72554	
##	Sep	Oct	Nov	Dec			
## 1	76.55365	69.60231	65.11055	57.89742			
## 2	66.89814	64.13913	71.38599	70.08153			
## 3	86.94436	81.82628	68.19158	66.61349			


```
## 4 68.92179 73.01946 71.35768 72.82133
## 5 70.37530 72.64690 78.07028 70.55305
## 6 70.24607 67.78789 74.31133 73.19746

#####
# 4: Output from R
#####

### 4.2 Formatting output
```

The Nolan and Temple Lang book provides a number of examples of different ways of authenticating with web services that control access to the service.

Finally, some web services allow us to pass information to the service in addition to just getting data or information. E.g., you can programmatically interact with your Facebook, Dropbox, and Google Drive accounts using REST based on HTTP POST, PUT, and DELETE. Authentication is of course important in these contexts and some times you would first authenticate with your login and password and receive a token. This token would then be used in subsequent interactions in the same session.

4 Output from R

4.1 Writing output to files

Functions for text output are generally analogous to those for input. *write.table()*, *write.csv()*, and *writeLines()* are analogs of *read.table()*, *read.csv()*, and *readLines()*. *write_csv()* is the *readr* version of *write.csv*. *write()* can be used to write a matrix to a file, specifying the number of columns desired. *cat()* can be used when you want fine control of the format of what is written out and allows for outputting to a connection (e.g., a file).

toJSON() in the *jsonlite* package will output R objects as JSON. One use of JSON as output from R would be to *serialize* the information in an R object such that it could be read into another program.

And of course you can always save to an R data file using *save.image()* (to save all the objects in the workspace or *save()* to save only some objects. Happily this is platform-independent so can be used to transfer R objects between different OS.

4.2 Formatting output

`cat()` is a good choice for printing a message to the screen, often better than `print()`, which is an object-oriented method. You generally won't have control over how the output of a `print()` statement is actually printed.

```
val <- 1.5
cat('My value is ', val, '.\n', sep = ' ')

## My value is 1.5.

print(paste('My value is ', val, '.', sep = ' '))

## [1] "My value is 1.5."
```

We can do more to control formatting with `cat()`:

```
# input
x <- 7
n <- 5
display powers
cat("Powers of", x, "\n")
cat("exponent  result\n\n")
result <- 1
for (i in 1:n) {
  result <- result * x
  cat(format(i, width = 8), format(result, width = 10), "\n", sep = " ")
}
x <- 7
n <- 5
# display powers
cat("Powers of", x, "\n")
cat("exponent  result\n\n")
result <- 1
for (i in 1:n) {
  result <- result * x
  cat(i, '\t', result, '\n', sep = ' ')
}
```

One thing to be aware of when writing out numerical data is how many digits are included. For example, the default with *write()* and *cat()* is the number of digits displayed to the screen, controlled by *options()* *\$digits*. (to change this, do *options(digits = 5)* or specify as an argument to *write()* or *cat()*) If you want finer control, use *sprintf()*, e.g. to print out temperatures as reals (“f”=floating points) with four decimal places and nine total character positions, followed by a C for Celsius:

```
temps <- c(12.5, 37.234324, 1342434324.79997234, 2.3456e-6, 1e10)
sprintf("%9.4f C", temps)

## [1] " 12.5000 C"      " 37.2343 C"
## [3] "1342434324.8000 C" " 0.0000 C"
## [5] "10000000000.0000 C"

city <- "Boston"
sprintf("The temperature in %s was %.4f C.", city, temps[1])

## [1] "The temperature in Boston was 12.5000 C."

sprintf("The temperature in %s was %9.4f C.", city, temps[1])

## [1] "The temperature in Boston was 12.5000 C."

#####
# 5: File and string encodings
#####
```

5 File and string encodings

Text (either in the form of a file with regular language in it or a data file with fields of character strings) will often contain characters that are not part of the [limited ASCII set of characters](<http://en.wikipedia.org/wiki/ASCII>), which has $2^7 = 128$ characters and control codes; basically what you see on a standard US keyboard. So for non-ASCII files you may need to deal with the text encoding (the mapping of individual characters (including tabs, returns, etc.) to a set of numeric codes). There are a variety of different encodings for text files, with different ones common on different operating systems. UTF-8 is an encoding for the Unicode characters that include more than 110,000 characters from 100 different alphabets/scripts. It’s widely used on the web. Latin-1

encodes a small subset of Unicode and contains the characters used in many European languages (e.g., letters with accents).

The UNIX utility *file*, e.g. `file tmp.txt` can help provide some information. *read.table()* in R takes arguments *fileEncoding* and *encoding* that address this issue. The UNIX utility *iconv* and the R function *iconv()* can help with conversions.

In US installations of R, the default encoding is UTF-8; note that various types of information are interpreted in US English with the encoding UTF-8:

```
Sys.getlocale()
```

```
## [1] "LC_CTYPE=en_US.UTF-8;LC_NUMERIC=C;LC_TIME=en_US.UTF-8;LC_COLLATE=en_
```

With strings already in R, you can convert between encodings with *iconv()*:

```
text <- "_Melhore sua seguran\xe7a_"
textUTF8 <- iconv(text, from = "latin1", to = "UTF-8")
Encoding(textUTF8)

## [1] "UTF-8"

textUTF8

## [1] "_Melhore sua segurança_"

iconv(text, from = "latin1", to = "ASCII", sub = "??")

## [1] "_Melhore sua seguran???a_"
```

You can also mark a string with an encoding, so R knows how to display it correctly:

```
x <- "fa\xe7ile"
Encoding(x) <- "latin1"
x

## [1] "façile"

## playing around...
x <- "\xa1 \xa2 \xa3 \xf1 \xf2"
Encoding(x) <- "latin1"
x

## [1] "¡ ¢ £ ñ ò"
```

An R error message with "multi-byte string" in the message often indicates an encoding issue. In particular errors often arise when trying to do string manipulations in R on character vectors for which the encoding is not properly set. Here's an example with some Internet logging data that we used last year in 243 in a problem set and which caused some problems.

```
load('../data/IPs.RData') # loads in an object named 'text'
tmp <- substring(text, 1, 15)

## Error in substring(text, 1, 15):  invalid multibyte string at '<bf>a7lw8

## the issue occurs with the 6402th element (found by trial and error):
tmp <- substring(text[1:6401], 1, 15)
tmp <- substring(text[1:6402], 1, 15)

## Error in substring(text[1:6402], 1, 15):  invalid multibyte string
at '<bf>a7lw8'

text[6402] # note the Latin-1 character

## [1] "from 5#c\xbfa7lw8lz2nX,%@ [128.32.244.179] by ncpc-email with ESMTP

## Interesting:
table(Encoding(text))

##
## unknown
##      6936

## Option 1
Encoding(text) <- "latin1"
tmp <- substring(text, 1, 15)

## Option 2
load('../data/IPs.RData') # loads in an object named 'text'
tmp <- substring(text, 1, 15)

## Error in substring(text, 1, 15):  invalid multibyte string at '<bf>a7lw8

text <- iconv(text, from = "latin1", to = "UTF-8")
tmp <- substring(text, 1, 15)
```