# Reading and Writing to/from R

September 7, 2014

References:

- Adler
- 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).

## 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 familarity 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 the form of XML or HTML files. The XML package is useful for reading and writing from these formats.

5. 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.

6. For Excel, there are capabilities to read an Excel file (see the *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

*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!
```

```
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, secifying 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 *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.

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, 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")
```

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 <- 3e+05
for (i in 1:ceiling(nLines/blockSize)) {
    lines <- readLines(con, n = blockSize)
    # manipulate the lines and store the key stuff
}
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
```

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"*. [I think; I haven't checked this, so a Windows user should correct me if I'm wrong.]

### Reading HTML

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.

```
library(XML)

## Loading required package:  methods

URL <- "http://en.wikipedia.org/wiki/Brad_Pitt_filmography"
pitt <- readHTMLTable(URL, stringsAsFactors = FALSE)
pittFilm <- pitt[[2]]
```

### 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 Output from R

## 3.1 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()* 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).

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.

## 3.2 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 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, 1342434324.79997, 2.3456e-06,
    1e+10)
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."
```

# 4 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:  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:  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:  invalid multibyte string at '<bf>a7lw8'
```

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