Introduction to R

Reading and Writing Data

Rob Scharpf

rscharpf@jhu.edu

November 6, 2013

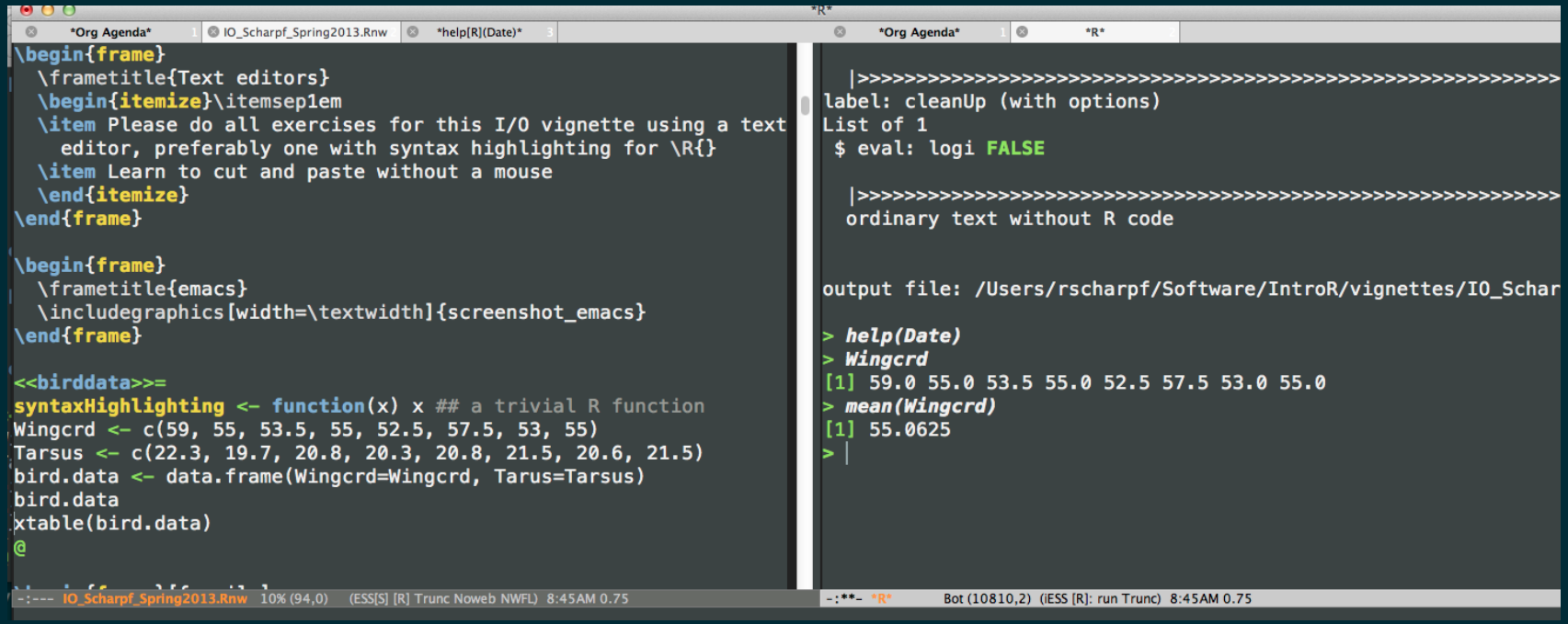
Course materials

http://tinyurl.com/Intro2R-Fall-2013

Text editors

- Use a text editor for all work in this course, preferably one with syntax highlighting for R
  - First step towards reproducibility
- Learn to move your cursor without a mouse
  - First step towards efficiency

emacs



- RStudio is probably the easiest

Organization

- Create a directory for each project with a descriptive name
  - avoid using adjectives like 'new' and 'old'
- Useful subdirectories:
  1. R: R functions
  2. vignettes: reproducible documents that weave text and R code
  3. data: Binary files with extension `.rda` or `.RData`
  4. inst/scripts: R scripts
  5. inst/extdata: external data

R utilities for files

- `dir.create`
- `file.exists`
- `getwd` and `setwd`

Example:

```
getwd()  ## get current working directory
mydir <- "IO_Nov6_2013"
dir.create(mydir)  ## create subdirectory InputOutput
setwd(mydir)
getwd()
list.files()  ## should be empty
```

Some data

```
print(xtable(bird.data))
```

	Wingcrd	Tarus
1	59.00	22.30
2	55.00	19.70
3	53.50	20.80
4	55.00	20.30
5	52.50	20.80
6	57.50	21.50
7	53.00	20.60
8	55.00	21.50

Getting data into R

```
Wingcrd <- c(59, 55, 53.5, 55, 52.5, 57.5, 53, 55)
Tarsus <- c(22.3, 19.7, 20.8, 20.3, 20.8, 21.5, 20.6,
    21.5)
bird.data <- data.frame(Wingcrd = Wingcrd, Tarsus = Tarsus)
bird.data
```

```
##   Wingcrd Tarsus
## 1    59.0   22.3
## 2    55.0   19.7
## 3    53.5   20.8
## 4    55.0   20.3
## 5    52.5   20.8
## 6    57.5   21.5
## 7    53.0   20.6
## 8    55.0   21.5
```

Reference R book

Importing data

Problems with this approach:

- Not practical / does not scale
- Typos

Exporting data

Exercise:

1. Use the R functions `apropos` or `help.search` to find functions that might be useful for *writing* data to a file.

```
apropos("write")
help("write")
`?`(write)
```

2. write as a tab-delimited file
3. write as a comma-delimited file
4. write as a comma-delimited file without rownames or header

apropos

```
apropos("write")
```

```
##  [1] "aspell_write_personal_dictionary_file"
##  [2] "RtangleWritedoc"
##  [3] "RweaveLatexWritedoc"
##  [4] "write"
##  [5] "write_bib"
##  [6] "write.csv"
##  [7] "write.csv2"
##  [8] "write.dcf"
##  [9] "write.ftable"
## [10] "write.socket"
## [11] "write.table"
## [12] "writeBin"
## [13] "writeChar"
## [14] "writeLines"
```

write help

`write` *Write Data to a File*

Description

The data (usually a matrix) `x` are written to file `file`. If `x` is a two-dimensional matrix you need to transpose it to get the columns in `file` the same as those in the internal representation.

Usage

```
write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE, sep = " ")
```

Arguments

`x`	the data to be written out, usually an atomic vector.
`file`	A connection, or a character string naming the file to write to. If `""`, print to the standard output connection. If it is `"|cmd"`, the output is piped to the command given by '`cmd`'.
`ncolumns`	the number of columns to write the data in.
`append`	if `TRUE` the data `x` are appended to the connection.
`sep`	a string used to separate columns. Using `sep = "\t"` gives tab delimited output; default is `" "`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`write` is a wrapper for `cat`, which gives further details on the format used.

`save` for writing any R objects, `write.table` for data frames, and `scan` for reading data.

Examples

```
# create a 2 by 5 matrix
x <- matrix(1:10, ncol = 5)

# the file data contains x, two rows, five cols
# 1 3 5 7 9 will form the first row
write(t(x))

# Writing to the "console" tab-delimited
# two rows, five cols but the first row is 1 2 3 4 5
write(x, "", sep = "\t")
unlink("data") # tidy up
```

write.table help

write.table *Data Output*

Description

write.table prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")

write.csv(...)
write.csv2(...)
```

Arguments

x	the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce x to a data frame.
file	either a character string naming a file or a connection open for writing. "" indicates output to the console.
append	logical. Only relevant if file is a character string. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
quote	a logical value (TRUE or FALSE) or a numeric vector. If TRUE, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If FALSE, nothing is quoted.
sep	the field separator string. Values within each row of x are separated by this string.
eol	the character(s) to print at the end of each line (row). For example, eol = "\r\n" will produce Windows' line endings on a Unix-alike OS, and eol = "\r" will produce files as expected by Excel:mac 2004.
na	the string to use for missing values in the data.
dec	the string to use for decimal points in numeric or complex columns: must be a single character.
row.names	either a logical value indicating whether the row names of x are to be written along with x, or a character vector of row names to be written.
col.names	either a logical value indicating whether the column names of x are to be written along with x, or a character vector of column names to be written. See the section on 'CSV files' for the meaning of col.names = NA.
qmethod	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of "escape" (default for write.table), in which case the quote character is escaped in C style by a backslash, or "double" (default for write.csv and write.csv2), in which case it is doubled. You can specify just the initial letter.

Exporting the data to a ' '-delimeted text file

```
write.table(bird.data)

## "Wingcrd" "Tarsus"
## "1" 59 22.3
## "2" 55 19.7
## "3" 53.5 20.8
## "4" 55 20.3
## "5" 52.5 20.8
## "6" 57.5 21.5
## "7" 53 20.6
## "8" 55 21.5

write.table(bird.data, file = "bird.txt")
```

Exporting the data to a ','-delimited text file

```
write.csv(bird.data)
```

```
## "","Wingcrd","Tarsus"
## "1",59,22.3
## "2",55,19.7
## "3",53.5,20.8
## "4",55,20.3
## "5",52.5,20.8
## "6",57.5,21.5
## "7",53,20.6
## "8",55,21.5
```

```
write.csv(bird.data, file = "bird.csv")
```

Importing data

Exercise:

1. Read the tab-delimited file `bird.txt` back into R
2. Read the comma-delimited file `bird.csv` back into R

read.table

IO_Scharpf_Spring2013.Rnw 2 | IO_Scharpf_Spring2013.log 3 | *help[R](read.table)* | *help[R](help)* 5 | *help[R](topic)* 6

```
read.table                package:utils                R Documentation

Data Input

Description:

     Reads a file in table format and creates a data frame from it,
     with cases corresponding to lines and variables to fields in the
     file.

Usage:

     read.table(file, header = FALSE, sep = "", quote = "\"'",
                dec = ".", row.names, col.names,
                as.is = !stringsAsFactors,
                na.strings = "NA", colClasses = NA, nrows = -1,
                skip = 0, check.names = TRUE, fill = !blank.lines.skip,
                strip.white = FALSE, blank.lines.skip = TRUE,
                comment.char = "#",
                allowEscapes = FALSE, flush = FALSE,
                stringsAsFactors = default.stringsAsFactors(),
                fileEncoding = "", encoding = "unknown", text)

     read.csv(file, header = TRUE, sep = ",", quote = "\"", dec = ".",
              fill = TRUE, comment.char = "", ...)

     read.csv2(file, header = TRUE, sep = ";", quote = "\"", dec = ",",
               fill = TRUE, comment.char = "", ...)

     read.delim(file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
                fill = TRUE, comment.char = "", ...)

     read.delim2(file, header = TRUE, sep = "\t", quote = "\"", dec = ","
                 fill = TRUE, comment.char = "", ...)
```

Importing data

I often read the first few lines of a file to make sure its what I want.

```
read.table("bird.txt")[1:3, ]  ## looks good

##   Wingcrd Tarsus
## 1    59.0   22.3
## 2    55.0   19.7
## 3    53.5   20.8

(dat <- read.table("bird.txt"))

##   Wingcrd Tarsus
## 1    59.0   22.3
## 2    55.0   19.7
## 3    53.5   20.8
## 4    55.0   20.3
## 5    52.5   20.8
## 6    57.5   21.5
## 7    53.0   20.6
## 8    55.0   21.5
```

Importing data

```
read.csv("bird.csv")[1:5, ]  ## Treats rownames as a column
```

```
##   X Wingcrd Tarsus
## 1 1    59.0   22.3
## 2 2    55.0   19.7
## 3 3    53.5   20.8
## 4 4    55.0   20.3
## 5 5    52.5   20.8
```

```
read.csv("bird.csv", row.names = 1)[1:5, ]  ## Better
```

```
##   Wingcrd Tarsus
## 1    59.0   22.3
## 2    55.0   19.7
## 3    53.5   20.8
## 4    55.0   20.3
## 5    52.5   20.8
```

Typically, one would assign the result of read.csv to an object

Big data

See the section "Memory usage" in the `read.table` help file. In particular, note that

- `read.table` requires a lot of memory
- `read.table` is meant for reading data.frames where the columns have different classes (e.g., numeric, dates, character strings, etc.)
- For matrices (all columns have the same class), use `scan`

Reading large data frames with `read.table`

- Read the first couple of lines to determine the classes of the columns. See the `nrows` argument to `read.table`.
- Specifying `colClasses` can reduce memory and speed up reading large data. Unwanted columns can be indicated with "NULL".

A more challenging `read.table` example

```
set.seed(1)  ## Set a seed for reproducibility
y <- rep(letters, length.out = 50000)
set.seed(1)
dates <- sample(c("02/27/2012", "01/14/2012", "02/28/2012",
    "02/01/2012", "10/31/2012"), length(y), replace = TRUE)
dat <- data.frame(x = rnorm(50000), y = y, date = dates,
    stringsAsFactors = FALSE)
nr <- nrow(dat)
index <- sample(seq_len(nr), 50)
dat[["date"]][index] <- "-999"
write.table(dat, file = "bigdata.txt", row.names = FALSE)
```

A more challenging `read.table` example

Exercise 4:

1. Read only columns 'x' and 'date' of `bigdata.txt`, using the `character` class for date.
2. coerce the date variable to class `Date`. See `?as.Date`
3. Suppose -999 was the code used for missing dates. Replace -999 with R's representation for missing data (see `?NA`)
4. How many observations were collected after January 30, 2012?
5. How many dates are Mondays, Tuesdays, ...?
6. plot x versus day of the week

Examine first few rows

```
header <- read.table("bigdata.txt", nrows = 3, header = TRUE)
str(header)

## 'data.frame': 3 obs. of  3 variables:
##  $ x   : num  0.197 -0.42 1.163
##  $ y   : Factor w/ 3 levels "a","b","c": 1 2 3
##  $ date: Factor w/ 2 levels "01/14/2012","02/28/2012": 1 1 2

## Use colClasses to read big tables.  Use class
## 'NULL' (in quotes) to skip a column
dat <- read.table("bigdata.txt", colClasses = c("numeric",
    "NULL", "character"), header = TRUE)
str(dat)

## 'data.frame': 50000 obs. of  2 variables:
##  $ x   : num  0.197 -0.42 1.163 -0.406 0.744 ...
##  $ date: chr  "01/14/2012" "01/14/2012" "02/28/2012" "10/31/2012" ...
```

Calculations on dates

How many observations do we have since January 30, 2012? Turns out we can do some calculations with dates as character strings (though this is not very reliable – may depend on how date is formatted):

```
sum(dat[["date"]] > "01/30/2012")

## [1] 39961
```

What if we wanted to know whether there was a day-of-the-week effect on our data 'x'?

Dates

R has a special class for dates. We begin by replacing instances of -999 with R's for missing data:

```
is.missing <- dat[["date"]] == "-999"
dat[["date"]][is.missing] <- NA
sum(is.na(dat[["date"]]))

## [1] 50
```

Searching R's help for 'date', we are referred to "Date" and "DateTimeClasses". Use `as.Date` to coerce our character string to an object of the class `Date`:

```
dat[["date"]] <- as.Date(dat[["date"]], "%m/%d/%Y")
```

The Date class

An advantage of using the Date class is that a number of methods for this class have been defined. For example, simple arithmetic operations:

```
mydate <- as.Date("01/30/2012", "%m/%d/%Y")
## a calculation on date
sum(dat[["date"]] > mydate, na.rm = TRUE)  ## or

## [1] 39961

table(dat[["date"]] > as.Date("01/30/2012", "%m/%d/%Y"))

##
## FALSE  TRUE
##  9989 39961
```

Day of the week

Looking at R's help for `Date`, we see that a method `weekdays` has been defined for objects of class `Date`.

```
weekdays(dat[["date"]])[1:10]

##  [1] "Saturday"  "Saturday"  "Tuesday"
##  [4] "Wednesday" "Saturday"  "Wednesday"
##  [7] "Wednesday" "Wednesday" "Wednesday"
## [10] "Monday"
```

1. how many observations do we have for each day?
2. how to plot our data 'x' against day of the week?
    - such exploratory data analyses are useful for identifying technical sources of variation (e.g., differences in reagents or lab personnel) that cause groups of samples to *look* different – known as batch effects

Day of the week

We can tabulate the number of observations on each day of the week and plot our data against day of the week in 2 lines of code:

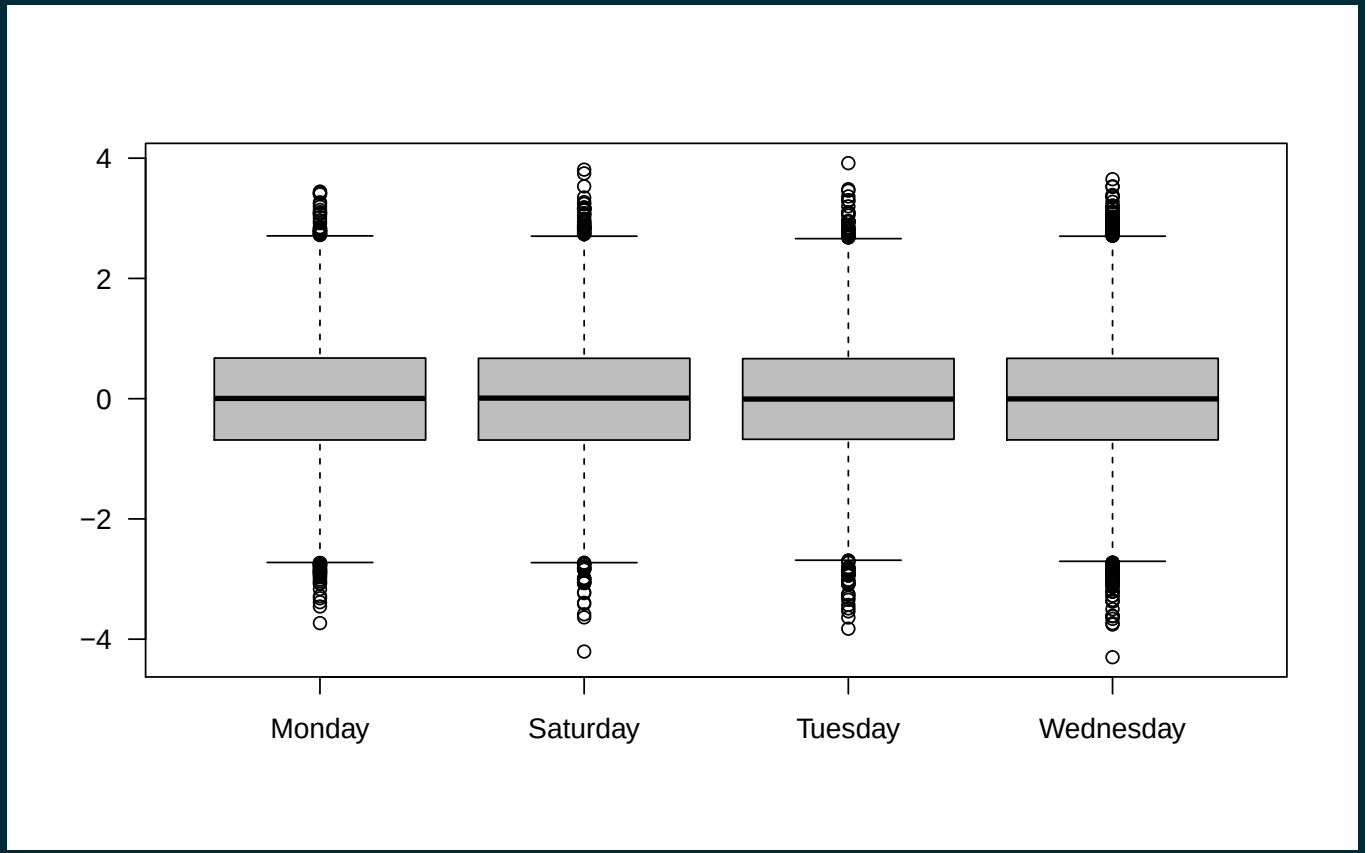
```
table(weekdays(dat[["date"]]))
```

```
##
##    Monday  Saturday   Tuesday Wednesday
##      9968      9989     10011     19982
```

Day of the week effect

```
par(las = 1)
boxplot(split(dat[["x"]], weekdays(dat[["date"]])),
    col = "gray")
```

Day of the week effect



There appears to be no difference in the distribution of 'x' by day of the week (as we would expect from our simulation)

Reordering the x-axis

In our previous graphic, Saturday appears after monday. How to arrange the boxplots in order (M, T, W, Sa)?

Reordering the x-axis

```
x.list <- split(dat[["x"]], weekdays(dat[["date"]]))
str(x.list)

## List of 4
##  $ Monday   : num [1:9968] 0.778 0.762 0.241 -1.423 0.848 ...
##  $ Saturday : num [1:9989] 0.197 -0.42 0.744 1.845 -0.153 ...
##  $ Tuesday  : num [1:10011] 1.1633 0.3057 -0.5475 0.1462 -0.0681 ...
##  $ Wednesday: num [1:19982] -0.4058 0.4766 0.5413 0.6106 0.0582 ...

## Approach 1:
x.list <- x.list[c("Monday", "Tuesday", "Wednesday",
    "Saturday")]
str(x.list)

## List of 4
##  $ Monday   : num [1:9968] 0.778 0.762 0.241 -1.423 0.848 ...
##  $ Tuesday  : num [1:10011] 1.1633 0.3057 -0.5475 0.1462 -0.0681 ...
##  $ Wednesday: num [1:19982] -0.4058 0.4766 0.5413 0.6106 0.0582 ...
##  $ Saturday : num [1:9989] 0.197 -0.42 0.744 1.845 -0.153 ...
```

Reordering the x-axis

Second approach: use `factor`

```
x.list <- split(dat[["x"]], factor(weekdays(dat[["date"]]),
    levels = c("Monday", "Tuesday", "Wednesday", "Saturday")))
str(x.list)

## List of 4
##  $ Monday   : num [1:9968] 0.778 0.762 0.241 -1.423 0.848 ...
##  $ Tuesday  : num [1:10011] 1.1633 0.3057 -0.5475 0.1462 -0.0681 ...
##  $ Wednesday: num [1:19982] -0.4058 0.4766 0.5413 0.6106 0.0582 ...
##  $ Saturday : num [1:9989] 0.197 -0.42 0.744 1.845 -0.153 ...

## boxplot(x.list, col='gray')
```

Large data example 2

```
bigmatrix <- replicate(100, rnorm(10000))
write.table(bigmatrix, file = "matrix.csv", sep = ",",
    row.names = FALSE, col.names = FALSE, quote = FALSE)
```

Remark: `write.matrix` would be much more efficient than `write.table`

Reading matrices

Exercise 5:

1. Read the file `matrix.csv` using `scan` and assign the result to `bigvector`
2. What are the dimensions of `bigvector`
3. Coerce `bigvector` to a matrix, say `bigmatrix2`, with the same dimensions as the simulated data (see `matrix`)
4. Use `all.equal` to see whether we have recovered the simulated data
5. Compare the `system.time` for reading the data with `scan` to the `system.time` for `read.csv`.

```r
## if(!exists('bigvector')){
bigvector <- scan("matrix.csv", sep = ",")
bigmatrix2 <- matrix(bigvector, 10000, 100, byrow = TRUE)
all.equal(bigmatrix, bigmatrix2)

## [1] TRUE

rm(bigmatrix2)
invisible(gc(verbose = FALSE))
system.time(scan("matrix.csv", sep = ","))

##    user  system elapsed
##   2.353   0.008   2.360

system.time(read.csv("matrix.csv"))

##    user  system elapsed
##   3.049   0.016   3.065

## }
```

Saving R objects

- It is often conventient to save a representation of an R object using the R function `save`. R objects should be saved with the file extension `.rda` or `.RData`.
- Unlike `write.table` where the data needs to be a simple matrix or data.frame, a binary of any R object in your workspace can be saved

```
save(object1, object2, object3, file = "somefile.rda")
```

- Use `load` to import a saved `.rda` object in your workspace

```
load("somefile.rda")
```

- See also `readRDS` and `saveRDS`

Exercise 6

1. Use the function `save` to save the `bigmatrix` object.
2. Remove the object `bigmatrix` from your workspace (see `rm`), and check that this object no longer exists (see `exists`).
3. Use the function `load` to bring the object back into your workspace.
4. Compare the size of the file `matrix.csv` to the size of the `.rda` file.
5. Compare the `system.time` for loading the `.rda` file to the `system.time` for reading the `.csv` file with `scan`

Saving R objects

Solution 6:

```
## if(!file.exists('bigmatrix.rda')){
save(bigmatrix, file = "bigmatrix.rda", compression_level = 9)
rm(bigmatrix)
isTRUE(!exists("bigmatrix"))

## [1] TRUE

load("bigmatrix.rda")
isTRUE(exists("bigmatrix"))

## [1] TRUE

system.time(load("bigmatrix.rda"))

##    user  system elapsed
##   0.055   0.002   0.057

system.time(scan("matrix.csv", sep = ","))

##    user  system elapsed
##   2.339   0.010   2.349
```

Files with headers

- Files that we wish to import in R often contain experimental meta-data in the header that is not part of the data
- Here, we use the `cat` to prepend experimental metadata to the first 10 rows of the matrix `bigmatrix`:

```
cat("Date: 10/31/2012\nExp. metadata\nblah blah blah\n",
    bigmatrix[1:10, ], file = "matrix_w_header.csv")
```

Files with headers

Exercise 7:

1. Read in the header of `matrix_w_header.csv` using `read.table`. Hint: Specify argument `sep` such that each row in the header is read as a single element (i.e., 3 rows, 1 column).

2. Use the function `readLines` to read in the header

3. Read in the data portion of `matrix_w_header.csv` using `scan` or `read.table`.

4. Compare the data portion to the first 10 rows of the `bigmatrix` using the function `all.equal`. Is this result expected? (Hint: see helpfile for `cat`)

Files with headers

```
tryCatch(header <- read.table("matrix_w_header.csv",
    nrows = 3), error = function(e) return("try again"))

## [1] "try again"

header <- read.table("matrix_w_header.csv", nrows = 3,
    sep = "\t")
header

##                  V1
## 1 Date: 10/31/2012
## 2    Exp. metadata
## 3   blah blah blah

x <- scan("matrix_w_header.csv", skip = nrow(header))
```

Importing Excel data

- easiest option is to export the excel data as a tab-delimited ascii file and import using `read.table`
- if you are stuck with a `.xls` file, the R package *xlsx* has utilities for reading specific rows and columns of an excel spreadsheet

Importing/Exporting files from other statistical software

- R can import data from other statistical software such as SPSS, Stata, and SAS.
- There are also utilities for writing data in an appropriate format for other statistical software
- See the R package *foreign*

For Friday

- Import a dataset into R that you will analyze as part of your class project
- Download and install the golubEsets package from Bioconductor
- Once installed, do

```
library(golubEsets)
help(package = "golubEsets")
```

  to find out what data is available in this package.
- For loading datasets provided with an R package, see `?data`

For Fun

Watch the demo for slidify: `http://slidify.org`

Questions

rscharpf@jhu.edu