SML 201 – Week 2

John D. Storey

Spring 2016

Contents

Getting Started in R

Summary from Week 1

Last week we learned about R:

- calculations
- getting help
- atomic classes
- assigning values to variables
- factors
- vectors, matrices, lists
- some basic functions

Missing Values

In data analysis and model fitting, we often have missing values. `NA` represents missing values and `NaN` means "not a number", which is a special type of missing value.

```
> m <- matrix(nrow=3, ncol=3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

```
[3,]   NA   NA   NA
> 0/1
[1] 0
> 1/0
[1] Inf
> 0/0
[1] NaN
```

NULL

NULL is a special type of reserved value in R.

```
> x <- vector(mode="list", length=3)
> x
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL
```

Coercion

We saw earlier that when we mixed classes in a vector they were all coerced to be of type character:

```
> c("a", 1:3, TRUE, FALSE)
[1] "a"     "1"     "2"     "3"     "TRUE"  "FALSE"
```

You can directly apply coercion with functions `as.numeric()`, `as.character()`, `as.logical()`, etc.

This doesn't always work out well:

```
> x <- 1:3
> as.character(x)
[1] "1" "2" "3"
>
> y <- c("a", "b", "c")
> as.numeric(y)
Warning: NAs introduced by coercion
[1] NA NA NA
```

Lists (review)

Lists allow you to hold different classes of objects in one variable.

```
> x <- list(1:3, "a", c(TRUE, FALSE))
> x
[[1]]
[1] 1 2 3

[[2]]
[1] "a"

[[3]]
[1]  TRUE FALSE
>
> # access any element of the list
> x[[2]]
[1] "a"
> x[[3]][2]
[1] FALSE
```

Lists with Names (review)

The elements of a list can be given names.

```
> x <- list(counting=1:3, char="a", logic=c(TRUE, FALSE))
> x
$counting
[1] 1 2 3

$char
[1] "a"

$logic
[1]  TRUE FALSE
>
> # access any element of the list
> x$char
[1] "a"
> x$logic[2]
[1] FALSE
```

Data Frames

The data frame is one of the most important objects in R. Data sets very often come in tabular form of mixed classes, and data frames are constructed exactly for this.

Data frames are lists where each element has the same length.

Data Frames

```
> df <- data.frame(counting=1:3, char=c("a", "b", "c"),
+                  logic=c(TRUE, FALSE, TRUE))
> df
  counting char logic
1        1    a  TRUE
2        2    b FALSE
3        3    c  TRUE
>
> nrow(df)
[1] 3
> ncol(df)
[1] 3
```

Data Frames

```
> dim(df)
[1] 3 3
>
> names(df)
[1] "counting" "char"     "logic"
>
> attributes(df)
$names
[1] "counting" "char"     "logic"

$row.names
[1] 1 2 3

$class
[1] "data.frame"
```

Attributes

Attributes give information (or meta-data) about R objects. The previous slide shows `attributes(df)`, the attributes of the data frame `df`.

```
> x <- 1:3
> attributes(x) # no attributes for a standard vector
NULL
>
> m <- matrix(1:6, nrow=2, ncol=3)
> attributes(m)
$dim
[1] 2 3
```

Attributes (cont'd)

```
> paint <- factor(c("red", "white", "blue", "blue", "red",
+                  "red"))
> attributes(paint)
$levels
[1] "blue"  "red"   "white"

$class
[1] "factor"
```

Names

Names can be assigned to columns and rows of vectors, matrices, and data frames. This makes your code easier to write and read.

```
> names(x) <- c("Princeton", "Rutgers", "Penn")
> x
Princeton   Rutgers      Penn
        1         2         3
>
> colnames(m) <- c("NJ", "NY", "PA")
> rownames(m) <- c("East", "West")
> m
     NJ NY PA
East  1  3  5
West  2  4  6
> colnames(m)
[1] "NJ" "NY" "PA"
```

Accessing Names

Displaying or assigning names to these three types of objects does not have consistent syntax.

Object	Column Names	Row Names
vector	`names()`	N/A
data frame	`names()`	`row.names()`
data frame	`colnames()`	`rownames()`
matrix	`colnames()`	`rownames()`

Reproducibility

Definition and Motivation

- Reproducibility involves *being able to recalculate the exact numbers in a data analysis using the code and raw data provided by the analyst.*
- Reproducibility is often difficult to achieve and has slowed down the discovery of important data analytic errors.
- Reproducibility should not be confused with "correctness" of a data analysis. A data analysis can be fully reproducible and recreate all numbers in an analysis and still be misleading or incorrect.

Taken from *Elements of Data Analytic Style*

Reproducible vs. Replicable

Reproducible research is often used these days to indicate the ability to recalculate the exact numbers in a data analysis

Replicable research results often refers to the ability to independently carry out a study (thereby collecting new data) and coming to equivalent conclusions as the original study

These two terms are often confused, so it is important to clearly state the definition

Steps to a Reproducible Analysis

1. Use a data analysis script – e.g., R Markdown (discussed next section!) or iPython Notebooks

2. Record versions of software and paramaters – e.g., use `sessionInfo()` in R as in `project_1.Rmd`
3. Organize your data analysis
4. Use version control – e.g., GitHub
5. Set a random number generator seed – e.g., use `set.seed()` in R
6. Have someone else run your analysis

Organizing Your Data Analysis

- Data
  - raw data
  - processed data (sometimes multiple stages for very large data sets)
- Figures
  - Exploratory figures
  - Final figures

Organizing Your Data Analysis (cont'd)

- R code
  - Raw or unused scripts
  - Data processing scripts
  - Analysis scripts
- Text
  - README files explaining what all the components are
  - Final data analysis products like presentations/writeups

Common Mistakes

- Failing to use a script for your analysis
- Not recording software and package version numbers or other settings used
- Not sharing your data and code
- Using reproducibility as a social weapon

R Markdown

R + Markdown + knitr

R Markdown was developed by the RStudio team to allow one to write reproducible research documents using Markdown and `knitr`. This is contained in the `rmarkdown` package, but can easily be carried out in RStudio.

Markdown was originally developed as a very simply text-to-html conversion tool. With Pandoc, Markdown is a very simply text-to-`X` conversion tool where `X` can be many different formats: html, LaTeX, PDF, Word, etc.

R Markdown Files

R Markdown documents begin with a metadata section, the YAML header, that can include information on the title, author, and date as well as options for customizing output.

```
---
title: "SML 201 -- Project 1"
author: "Your Name"
date: February 8, 2016
output:
  pdf_document:
    toc: true
    toc_depth: 2
    keep_tex: true
geometry: right=2.5in
---
```

Many options are available. See http://rmarkdown.rstudio.com for full documentation.

Markdown

Headers:

```
# Header 1
## Header 2
### Header 3
```

Emphasis:

```
*italic* **bold**
_italic_ __bold__
```

Tables:

```
First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

Markdown (cont'd)

Unordered list:

```
- Item 1
- Item 2
    - Item 2a
    - Item 2b
```

Ordered list:

```
1. Item 1
2. Item 2
3. Item 3
    - Item 3a
    - Item 3b
```

Markdown (cont'd)

Links:

```
http://example.com

[linked phrase](http://example.com)
```

Blockquotes:

```
Florence Nightingale once said:

> For the sick it is important
> to have the best.
```

Markdown (cont'd)

Plain code blocks:

````
```
This text is displayed verbatim with no formatting.
```
````

Inline Code:

```
We use the `print()` function to print the contents
of a variable in R.
```

Additional documentation and examples can be found here and here.

knitr

The `knitr` R package allows one to execute R code within a document, and to display the code itself and its output (if desired). This is particularly easy to do in the R Markdown setting. For example...

Placing the following text in an R Markdown file

```
The sum of 2 and 2 is `r 2+2`.
```

produces in the output file

The sum of 2 and 2 is 4.

knitr Chunks

Chunks of R code separated from the text. In R Markdown:

````
```{r}
x <- 2
x + 1
print(x)
```
````

Output in file:

```
> x <- 2
> x + 1
[1] 3
> print(x)
[1] 2
```

Chunk Option: `echo`

In R Markdown:

````
```{r, echo=FALSE}
x <- 2
x + 1
print(x)
```
````

Output in file:

```
[1] 3
[1] 2
```

Chunk Option: `results`

In R Markdown:

````
```{r, results="hide"}
x <- 2
x + 1
print(x)
```
````

Output in file:

```
> x <- 2
> x + 1
> print(x)
```

Chunk Option: `include`

In R Markdown:

````
```{r, include=FALSE}
x <- 2
x + 1
print(x)
```
````

Output in file:

(nothing)

Chunk Option: `eval`

In R Markdown:

````
```{r, eval=FALSE}
x <- 2
x + 1
print(x)
```
````

Output in file:

```
> x <- 2
> x + 1
> print(x)
```

Chunk Names

Naming your chunks can be useful for identifying them in your file and during the execution, and also to denote dependencies among chunks.

````
```{r my_first_chunk}
x <- 2
x + 1
print(x)
```
````

knitr Option: `cache`

Sometimes you don't want to run chunks over and over, especially for large calculations. You can "cache" them.

````
```{r chunk1, cache=TRUE, include=FALSE}
x <- 2
```
````

````
```{r chunk2, cache=TRUE, dependson="chunk1"}
y <- 3
z <- x + y
```
````

This creates a directory called `cache` in your working directory that stores the objects created or modified in these chunks. When `chunk1` is modified, it is re-run. Since `chunk2` depends on `chunk1`, it will also be re-run.

knitr Options: figures

You can add chunk options regarding the placement and size of figures. Examples include:

- `fig.width`
- `fig.height`
- `fig.align`

Changing Default Chunk Settings

If you will be using the same options on most chunks, you can set default options for the entire document. Run something like this at the beginning of your document with your desired chunk options.

````
```{r my_opts, cache=FALSE, echo=FALSE}
library("knitr")
opts_chunk$set(fig.align="center", fig.height=4, fig.width=6)
```
````

Documentation and Examples

- http://yihui.name/knitr/
- http://kbroman.org/knitr_knutshell/pages/Rmarkdown.html
- https://github.com/SML201/lectures

Control Structures

Rationale

- Control structures in R allow you to control the flow of execution of a series of R expressions
- They allow you to put some logic into your R code, rather than just always executing the same R code every time
- Control structures also allow you to respond to inputs or to features of the data and execute different R expressions accordingly

Paraphrased from *R Programming for Data Science*

Common Control Structures

- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `while`: execute a loop while a condition is true
- `repeat`: execute an infinite loop (must break out of it to stop)
- `break`: break the execution of a loop
- `next`: skip an interation of a loop

From *R Programming for Data Science*

Some Boolean Logic

R has built-in functions that produce `TRUE` or `FALSE` such as `is.vector` or `is.na`. You can also do the following:

- `x == y` : does x equal y?
- `x > y` : is x greater than y? (also `<` less than)
- `x >= y` : is x greater than or equal to y?
- `x && y` : are both x and y true?
- `x || y` : is either x or y true?
- `!is.vector(x)` : this is `TRUE` if x is not a vector

`if`

Idea:

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

Example:

```
> x <- c(1,2,3)
> if(is.numeric(x)) {
+   x+2
+ }
[1] 3 4 5
```

if-else

Idea:

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

Example:

```
> x <- c("a", "b", "c")
> if(is.numeric(x)) {
+   print(x+2)
+ } else {
+   class(x)
+ }
[1] "character"
```

for Loops

Example:

```
> for(i in 1:10) {
+   print(i)
+ }
[1] 1
[1] 2
```

```
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

`for` Loops (cont'd)

Examples:

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+   print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
>
> for(i in seq_along(x)) {
+   print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested `for` Loops

Example:

```
> m <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
>
> for(i in seq_len(nrow(m))) {
+   for(j in seq_len(ncol(m))) {
+     print(m[i,j])
+   }
+ }
[1] 1
```

```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```

`while`

Example:

```
> x <- 1:10
> idx <- 1
>
> while(x[idx] < 4) {
+   print(x[idx])
+   idx <- idx + 1
+ }
[1] 1
[1] 2
[1] 3
>
> idx
[1] 4
```

Repeats the loop until while the condition is `TRUE`.

`repeat`

Example:

```
> x <- 1:10
> idx <- 1
>
> repeat {
+   print(x[idx])
+   idx <- idx + 1
+   if(idx >= 4) {
+     break
+   }
+ }
[1] 1
[1] 2
[1] 3
```

```
>
> idx
[1] 4
```

Repeats the loop until `break` is executed.

`break` and `next`

`break` ends the loop. `next` skips the rest of the current loop iteration.

Example:

```
> x <- 1:1000
> for(idx in 1:1000) {
+   # %% calculates division remainder
+   if((x[idx] %% 2) > 0) {
+     next
+   } else if(x[idx] > 10) { # an else-if!!
+     break
+   } else {
+     print(x[idx])
+   }
+ }
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

Vectorized Operations

Calculations on Vectors

R is usually smart about doing calculations with vectors. Examples:

```
>
> x <- 1:3
> y <- 4:6
>
> 2*x     # same as c(2*x[1], 2*x[2], 2*x[3])
[1] 2 4 6
> x + 1   # same as c(x[1]+1, x[2]+1, x[3]+1)
[1] 2 3 4
```

```
> x + y   # same as c(x[1]+y[1], x[2]+y[2], x[3]+y[3])
[1] 5 7 9
> x*y     # same as c(x[1]*y[1], x[2]*y[2], x[3]*y[3])
[1]  4 10 18
```

A Caveat

If two vectors are of different lengths, R tries to find a solution for you (and doesn't always tell you).

```
> x <- 1:5
> y <- 1:2
> x+y
Warning in x + y: longer object length is not a multiple of
shorter object length
[1] 2 4 4 6 6
```

What happened here?

Vectorized Matrix Operations

Operations on matrices are also vectorized. Example:

```
> x <- matrix(1:4, nrow=2, ncol=2, byrow=TRUE)
> y <- matrix(1:4, nrow=2, ncol=2)
>
> x+y
     [,1] [,2]
[1,]    2    5
[2,]    5    8
>
> x*y
     [,1] [,2]
[1,]    1    6
[2,]    6   16
```

Mixing Vectors and Matrices

What happens when we do calculations involving a vector and a matrix? Example:

```
> x <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
> z <- 1:2
>
> x + z
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    6    7    8
>
> x * z
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    8   10   12
```

Mixing Vectors and Matrices

Another example:

```
> x <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
> z <- 1:3
>
> x + z
     [,1] [,2] [,3]
[1,]    2    5    5
[2,]    6    6    9
>
> x * z
     [,1] [,2] [,3]
[1,]    1    6    6
[2,]    8    5   18
```

What happened this time?

Vectorized Boolean Logic

We saw `&&` and `||` applied to pairs of logical values. We can also vectorize these operations.

```
> a <- c(TRUE, TRUE, FALSE)
> b <- c(FALSE, TRUE, FALSE)
>
> a | b
[1]  TRUE  TRUE FALSE
> a & b
[1] FALSE  TRUE FALSE
```

Extras

License

https://github.com/SML201/lectures/blob/master/LICENSE.md

Source Code

https://github.com/SML201/lectures/tree/master/week2

Session Information

```
> sessionInfo()
R version 3.2.3 (2015-12-10)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.11.3 (El Capitan)

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
[1] knitr_1.12.3    devtools_1.10.0

loaded via a namespace (and not attached):
 [1] magrittr_1.5    formatR_1.2.1   tools_3.2.3
 [4] htmltools_0.3   yaml_2.1.13     memoise_1.0.0
 [7] stringi_1.0-1   rmarkdown_0.9.2 highr_0.5.1
[10] stringr_1.0.0   digest_0.6.9    evaluate_0.8
```