Data analysis and visualization using R

R basics

Michiel Noback

november 2015

R basics

Contents

Part 1: The R console

The console awaits your commands

Console ~/

```
R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The console

- The console is one of your RStudio panels
- The prompt is the "greater than" symbol "$>$"
- The cursur waits there for your commands
- You can simply use it as a calculator

```
> 2 + 4
```

```
[1] 6
```

```
> 21 / 3
```

```
[1] 7
```

Part 2: First steps

R is all about vectors

- Repeat after me: **“everything in R lives inside vectors”**
- When you type ‘2 + 4’, R wil do this:
  - create a vector of length 1 with the value 2
  - create a vector of length 1 with the value 4
  - add the value of the second vector to ALL the values of vector one, and recycle vector 2 as many times as needed
- We’ll revisit this behavior later, after the basics of vector creation and conversion

Creating vectors

- The simplest way (there are many) to create a vector is to use the "Concatenate" function c()
- c() takes all its arguments and puts them behind each other in a vector

```
> c(2, 4, 3)
```

```
[1] 2 4 3
```

```
> c("a", "b", c("c", "d"))
```

```
[1] "a" "b" "c" "d"
```

```
> c(0.1, 0.01, 0.001)
```

```
[1] 0.100 0.010 0.001
```

Vectors can hold only one data type

- A vector can hold only one type of data
- R tries very hard to **coerce** all data to one type

```r
c(2, 4, "a")
```

```
[1] "2" "4" "a"
```

Ask for her type

```r
class(c(2, 4, "a"))
```

```
[1] "character"
```

```r
class(1:5)
```

```
[1] "integer"
```

```r
class(c(2, 4, 0.3))
```

```
[1] "numeric"
```

Ask for help

- Learning a language is also learning to find how to do things
  1. Use the help function `help("c")` or `?<function>`
  2. Google it "R function to concatenate vectors", "R package to read xml data"
  3. Use an expert site like Stackoverflow

Comments

- Everything on a line after a hash sign "#" will be ignored by R

```
## starting cool analysis
x <- c(T, F, T) # Creating a logical vector
y <- c(TRUE, FALSE, TRUE) # same
```

Assigning values

- The arrow symbol "<-" is used to assign values to variables
- You can also use "=" but you should do this only with function arguments

```
x <- c(1, 2, 3) # recommended
x
```

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

```
y = c(2, 3, 1) # legal but not recommended
y
```

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

Ending statements

- You can optionally end statements with a semicolon “;”
- Only when you have more statements on one line they are mandatory
- *Rule*: Have one statement per line and don’t use semicolons

```r
x <- c(1, 2, 3); x; x <- 42; x
```

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

```
[1] 42
```

Part 3: Vector fiddling

Vector arithmetic (1)

- Going back to the vector arithmetic
- Let's just look at some examples

```
x <- c(2, 4, 3, 5)
y <- c(6, 2)
x + y
```

```
[1] 8 6 9 7
```

```
x * 2
```

```
[1]  4  8  6 10
```

Vector arithmetic (2)

```
x <- c(2, 4, 3, 5)
z <- c(1, 2, 3)
x - z
```

```
Warning in x - z: longer object length is not a multiple of
```

```
[1] 1 2 0 4
```

- As you see, this also generates a warning that “longer object length is not a multiple of shorter object length”
- But R will proceed anyway, by recycling the shorter one!

The R basic data types

- These are the types all others are built from

class	type
integer	whole numbers
numeric	floating point numbers
character	textual data
factor	categorial data
logical	TRUE or FALSE

Creating vectors of specific type

- Often you want to be specific about what you create
- Use the class specific constructor **OR** the conversion methods
- constructor methods have the name of the type
- conversion methods have “as.” prepended to the name

Method 1: Constructor functions

```
integer(4)
```

```
[1] 0 0 0 0
```

```
character(4)
```

```
[1] "" "" "" ""
```

```
logical(4)
```

```
[1] FALSE FALSE FALSE FALSE
```

```
factor(4)
```

```
[1] 4
Levels: 4
```

Method 2: Conversion functions

Conversion methods have the name `as.XXX()` where XXX is the desired type

```
x <- c(1, 0, 2, 0, 2)
class(x)
```

```
[1] "numeric"
```

```
as.logical(x)
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

```
as.factor(x)
```

```
[1] 1 0 2 0 2
Levels: 0 1 2
```

Even R will stop trying at some point

- R will not coerce types that are non-coercable: you get an NA

```
x <- c(2, 3, "a")
y <- as.integer(x)
```

```
Warning: NAs introduced by coercion
```

```
class(y)
```

```
[1] "integer"
```

```
y
```

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

Method 3: Using colon to create a series

The colon *(:)* operator generates a series of integers

```
1 : 5
```

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

```
5 : 1
```

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

```
2 : 3.66
```

```
[1] 2 3
```

Method 4: Using the rep() function

```r
rep(1 : 3, times = 3)
```

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

```r
rep(1 : 3, each= 3)
```

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

```r
rep(1 : 3, times = 2, each = 3)
```

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

Method 5: Using the seq() function

```r
seq(from = 1, to = 3, by = .2)
```

```
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

```r
seq(1, 2, 0.2) # same
```

```
[1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```r
seq(1, 0, length.out = 5)
```

```
[1] 1.00 0.75 0.50 0.25 0.00
```

```r
seq(3, 0, by = -1)
```

```
[1] 3 2 1 0
```

Method 6: through vector operations

```
x <- rnorm(5); x
```

```
[1]  1.1581277  1.0012895 -0.2780903  0.6388799 -0.1657145
```

```
y <- rnorm(5); y
```

```
[1] -0.21411920 -0.51215132  1.90597925  2.02562949 -0.0341
```

```
z <- x < y ## which of x are smaller than corresponding inc
z ## you get a logical vector
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

Part 4: Advanced vector fiddling

Operators and vectors in practice

Suppose you have vectors a and b and you want to know which values in a are greater than in b and also smaller than 3

```
a <- c(2, 1, 3, 1, 5, 1)
b <- c(1, 2, 4, 2, 3, 0)
a > b & a < 3 ## returns a logical vector with test results
```

```
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE
```

```
6 - 2 : 5 ## think about this one!
```

```
[1] 4 3 2 1
```

Subsetting vectors

Often, you want to get to know things about values in a vector

- what is the highest value?
- which positions have negative values?
- what are the last 5 values?

There are several ways to do this

- by index
- by boolean test / logical vector
- using `which()`

Subsetting vectors by index

- The index is the position of a value in a vector
- R starts at 1 (unlike most other languages)

```r
x <- rep(c(1, 2, 3), 2); x
```

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

```r
x[4] ## fourth element
```

```
[1] 1
```

```r
x[3:5] ## elements 3 to 5
```

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

Subsetting vectors by index (cont.)

```
x <- c(1, 2, 3, 1, 2, 3)
x[c(1, 2, 5)] ## elements 1, 2 and 5
```

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

```
x[c(T, T, F, F, T, T)] ## select using booleans
```

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

```
x[x %% 2 == 0] ## all even elements using modulus
```

```
[1] 2 2
```

Subsetting vectors by index (cont.)

```
x <- c(1, 2, 3, 1, 2, 3)
x[(length(x) - 1) : length(x)] ## last 2 elements; note the
```

```
[1] 2 3
```

```
which(x >= 2) ## ask for the positions
```

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

```
which(x == max(x)) ## which positions have the maximum valu
```

```
[1] 3 6
```

```
x[x == max(x)]
```

```
[1] 3 3
```

Calculations with logical vectors

- Often, you want to know how many cases fit some condition
- Logical values have a numeric counterpart:
  - TRUE == 1
  - FALSE == 0
- Use `sum()` to use this feature

```
x <- c(2, 4, 2, 1, 5, 3, 6)
x > 3 ## which values are greater than 3 -- logical vector
```

```
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE
```

```
sum(x > 3) ## returns number of values greater than 3
```

```
[1] 3
```

Part 5: Plotting vectors

Basic plot types

- Looking at numbers is boring - people want to see pictures!
- There are a few basic plot types dealing with (combinations of) vectors:
  - scatter (or line-) plot
  - barplot
  - histogram
  - boxplot

Scatter and line plots

- Meet `plot()` - the workhorse of R plotting

```
time <- c(1, 2, 3, 4, 5, 6)
response <- c(0.09, 0.30, 0.41, 0.48, 0.72, 1.12)
plot(x = time, y = response)
```

Plot decoration

- By passing arguments to `plot()` you can modify or add many features of your plot
- Basic decoration includes
  - adjusting markers (`pch = 19`, `col = "blue"`)
  - adding connector lines (`type = "b"`)
  - adding axis labels and title (`xlab = "Time (hours)"`, `ylab = "Systemic response"`, `main = "Systemic response to agent X"`)
  - adjusting axis limits (`xlim = c(0, 8)`)

Plot decoration

```r
plot(x = time, y = response, pch = 19, type = "b", xlim = c
     xlab = "Time (hours)", ylab = "Systemic response",
     main = "Systemic response to agent X", col = "blue")
```

Barplots

- Barplots can be generated in two ways
  - By passing a factor to plot() - simple of level frequencies
  - By using barplot()

```
persons <- as.factor(sample(c("male", "female"), size = 10
plot(persons)
```

barplot()

- barplot() can be called with a vector of heights (frequencies) or a table() object

```
frequencies <- c(22, 54, 12, 29)
barplot(frequencies, names = c("one", "two", "three", "four
```

barplot() with a table object

```r
table(persons)
```

```
persons
female   male
    47     53
```

```r
barplot(table(persons))
```

Histograms

Histograms help you visualise the distribution of your data

```
male_weights <- c(rnorm(500, 80, 8)) ## create 500 random r
hist(male_weights)
```

Histograms (2)

Using the breaks argument, you can adjust the bin width. Always explore this option when creating histograms!

```
par(mfrow = c(1, 2)) # make 2 plots to sit side by side
hist(male_weights, breaks = 5, col = "gold", main = "Male w
hist(male_weights, breaks = 25, col = "green", main = "Male
```

Boxplots

This is the last of the basic plot types. A boxplot is a visual representation of the *5-number summary* of a variable.

```
persons <- rep(c("male", "female"), each = 100)
weights <- c(rnorm(100, 80, 6), rnorm(100, 75, 8))
#print 6-number summary (5-number + mean)
summary(weights[persons == "female"])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  62.07   70.20   74.14   76.27   82.03  100.90
```

Boxplots (2)

```
par(mfrow = c(1, 2)) # make 2 plots to sit side by side
# create boxplots of weights depending on sex
boxplot(weights ~ persons, ylab = "weight")
boxplot(weights ~ persons, notch = TRUE, col = c("yellow",
```

Part 6: Some ground rules

Plotting rules

Plots should always have these decorations

- A descriptive title
- Axis labels indicating measurement type and its units
- If multiple data series are plotted: a legend

For the sake of space saving, I will not always adhere to these rules in my presentations

Coding style rules (1)

- Follow Googles' `https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml`
- Names of variables start with a lower-case letter
- Words are separated using dots (or use camel case)
- Be descriptive with names
- Function names are verbs
- Write all code and comments in English
- Always use extensive comment in your code
- Preferentially use one statement per line

Coding style rules (2)

- Use spaces on both sides of ALL operators
- Use a space after a comma
- Indent code blocks -with {}- with 4 spaces

```
rnorm(5,mean=3) #bad
```

```
[1] 3.413103 2.756846 3.353251 3.425575 2.651585
```

```
rnorm(5, mean = 3) #good
```

```
[1] 3.796348 2.257082 2.700409 1.675996 4.654461
```

Source file template

```
# Name & Email address
# File description comment
# including purpose of program,
# inputs, and outputs

#### source() & library() statements ####

#### Function definitions ####

#### Excecuted code ####
```

Operators

- Math: `+ - * / ^`
- Logic:
  `& # and | # or ! # not`
- Comparison: `< <= > >= ==`

Operator precedence (1)

Some operators are evaluated before others. This is called operator precedence and it is best illustrated by an example

```
1 + 2 / 3
```

```
[1] 1.666667
```

```
(1 + 2) / 3
```

```
[1] 1
```

```
2 - 2 * 5 - 5
```

```
[1] -13
```

Operator precedence (2)

- When you have complex statements, you should be aware of operator precedence
- If you are not sure: use parentheses ()
- If you are still uncertain, look at this reference page

Wrap-up of the basics

- help on function: `help(function)`
- or `?function`
- autocomplete/suggestions in RStudio: `tab` key
- help (in Rstudio): F1
- installing a library that is not in the core packages: `install.packages("ggplot2"")`
- loading a library that is not in the core packages: `library(ggplot2)`
- remove variable(s): `rm(x, y, z, myData)`