R

A rapid overview for the brave

Will Pearse

March 15, 2013

The Three Golden Rules

- R is not statistics
- R does not make mistakes, we do
- Relax!

Contents

Vectors

- Length (e.g., 0, 3, 10)
- Type (logical, integer, double, complex, character, raw)

```
> nerds <- c("Will", "David")
> nerds

[1] "Will"  "David"

> typeof(nerds)

[1] "character"

> is.character(nerds)

[1] TRUE

> is.numeric(nerds)

[1] FALSE
```

Arrays and Matrices

- Length (the number of elements)
- Each element has a type

Their elements do not have to be of the same length or type

Technically a vector—distinct from earlier atomic vectors

```
> lab <- list(postdocs = c("Will",
+     "Matt"), funding = 1e+07)
> lab

$postdocs
[1] "Will" "Matt"

$funding
[1] 1e+07

> lab$postdocs

[1] "Will" "Matt"

> lab[[1]]

[1] "Will" "Matt"

> lab[1:2]

$postdocs
[1] "Will" "Matt"

$funding
[1] 1e+07
```

Arrays and Matrices

- A single type
- Lengths in a number of dimensions
- Matrices are a special case of arrays with only two dimensions
- Vectors are kind of one-dimensional arrays

```
> mat <- matrix(1:4, nrow = 2)
> mat

     [,1] [,2]
[1,]    1    3
[2,]    2    4

> arr <- array(1:8, dim = c(2, 2,
+     2))
> arr

, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Compound Objects

Factors are vectors with *levels*:

```
> myfac <- factor(c(5, 3, 5, 1))
> myfac

[1] 5 3 5 1
Levels: 1 3 5

> levels(myfac)

[1] "1" "3" "5"

> as.numeric(myfac)

[1] 3 2 3 1
```

Data frames are lists where all elements are of the same length

Subsetting—Basics

```
> num <- 1:10
> lett <- letters[1:10]
> num > 5

 [1] FALSE FALSE FALSE FALSE
 [5] FALSE  TRUE  TRUE  TRUE
 [9]  TRUE  TRUE

> lett[num > 5]

[1] "f" "g" "h" "i" "j"
```

Why learn `subset` when you can do it yourself?

Subsetting—Data Frames

```
> data <- data.frame(y = 1:10, lett = letters[1:10])
> data[data$lett == "a", ]

  y lett
1 1    a

> data[data$lett != "a", ]

    y lett
2   2    b
3   3    c
4   4    d
5   5    e
6   6    f
7   7    g
8   8    h
9   9    i
10 10    j
```

In a healthy relationship, you have a *row* before a *cuddle*—not the other way round!

Subsetting—Combinations

Test multiple things with AND and OR

```
> TRUE & FALSE
[1] FALSE
> TRUE | FALSE
[1] TRUE
```

XOR selects things that are only one or the other

```
> xor(TRUE, TRUE)
[1] FALSE
> xor(TRUE, FALSE)
[1] TRUE
```

Functions — Overview

- Neatly structure code
- Create their own (lexical) scope
- Take input values
- Return values

```
> square <- function(number) {
+     output <- number^2
+     return(output)
+ }
> square(4)

[1] 16
```

Functions — Scope

- There are two vectors called output, in two different scopes
  - The global scope (funky)
  - The function squares scope'
- When the function returns, everything in its scope is destroyed

```
> output <- "funky"
> square <- function(number) {
+     output <- number^2
+     return(output)
+ }
> square(4)

[1] 16

> output

[1] "funky"
```

Functions — input

- R functions are call-by-value
- They never see the actual variable they were called with; only its value

```
> input <- 4
> square <- function(number) {
+     output <- number^2
+     number <- "funky"
+     return(output)
+ }
> square(input)

[1] 16

> input

[1] 4
```

Functions — Return

- R functions return values into a variable, or print them if they have nothing to return into
- Functions (e.g., boxplot) can invisibly return

```
> square <- function(number) {
+     output <- number^2
+     return(output)
+ }
> answer <- square(4)
> answer

[1] 16

> square <- function(number) {
+     output <- number^2
+     invisible(output)
+ }
> square(4)
```

If...else

- If X, do Y, else do Z
- Lazy evaluation means bugs can hide in code that isnt evaluated when you test it
- The commonly ignored case statement is useful too

```
> if (TRUE == TRUE) {
+     print("duh...")
+ } else {
+     lazy
+ }

[1] "duh..."

> if (TRUE == FALSE) {
+     lazy
+ } else {
+     print("nuh-uh...")
+ }

[1] "nuh-uh..."
```

Loops — Overview

- Take an input vector
- For each element in that input vector, execute the block
- We are iterating over the vector
- Commonly ignored friend while

```
> some.letters <- letters[1:5]
> for(each in some.letters){
+ print(each)
+ }

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

> for(i in seq(along=some.letters
+ print(some.letters[i])

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

Loops — Control Flow

- You dont have to finish a loop
- Skip to the next iteration
- ...or break out of the loop altogether

```
> for(i in seq(5)){
+ if(i == 2) next
+ print(i)
+ }

[1] 1
[1] 3
[1] 4
[1] 5

> for(i in seq(5)){
+ if(i == 2) break
+ print(i)
+ }

[1] 1
```

Loops — Stop Using Them!

- Each iteration, R reads your block for the first time lazy evaluation
- Other functions use C and FORTAN code, that is optimised for loops, to do them quicker

```
> data <- data.frame(y=
+  rnorm(200000), x=rnorm(200000))
> result <- numeric(nrow(data))
> system.time(
+  for(i in seq(nrow(data)))
+  result[i] <- data$x[i] +
+   data$y[i])

   user  system elapsed
  1.221   0.000   1.221

> result <- numeric(nrow(data))
> system.time(
+  result <- apply(data, 1, sum))

   user  system elapsed
  0.782   0.005   0.787
```

Loops — table

```
> data <- data.frame(y=rnorm(100),
+  fac1=sample(letters[1:4], 100, replace=TRUE),
+  fac2=sample(1:4, 100, replace=TRUE))
> with(data, table(fac1, fac2))

    fac2
fac1  1  2  3  4
   a  6  7  5  1
   b  5  1  8  9
   c  6  5  6 11
   d  8  3 10  9
```

Loops — apply

- For data.frames
- 1 go along rows (first dimension)
- 2 go along columns (second dimension)

```
> data <- data.frame(y=rnorm(10),
+  x=rnorm(10))
> apply(data, 1, sum)

 [1]  0.5025  0.7472 -1.3941
 [4] -0.4333 -0.5547  0.3906
 [7]  1.8426 -0.6460 -0.4689
[10]  1.1112

> apply(data, 2, sum)

      y       x
 1.6670 -0.5699
```

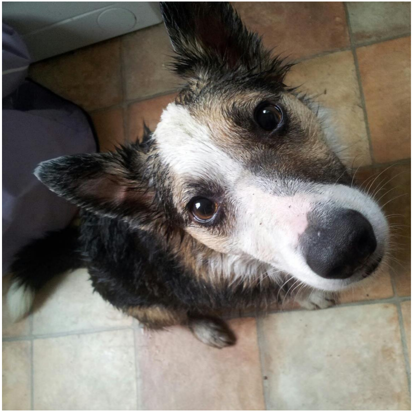
Loops — apply and Friends

- `tapply` vectors
- `lapply` lists
- `sapply` simplify the output from `lapply`

- I describe all this on the EEB-R list!
- `http://tiny.cc/eeb-r`

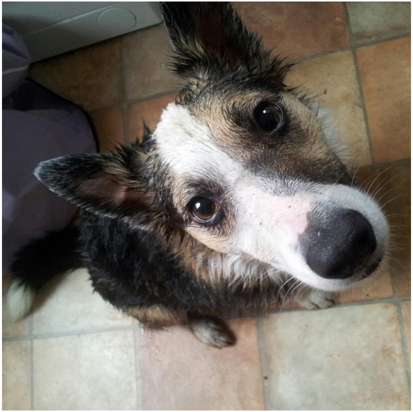
Classes — introduction

- Grouping things into *classes* helps us understand them
- Dexter is a `dog`
- Dogs have properties
  - weight
  - breed
- Dogs do things
  - bark
  - chase balls



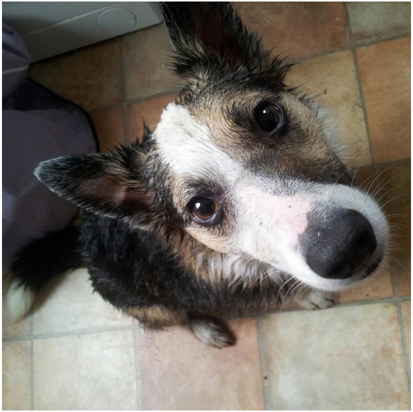
Classes — introduction

- Dexter is an *instance* of class `dog`
- The Dexter instance has *slots* (giving it *internal state*)
  - weight
  - breed
- Dexter has *class methods* (functions)
  - bark
  - chase balls



Classes — introduction

- The `dog` class inherits methods and slots from the class `mammal`
  - lung capacity
  - walk
- *Encapsulating* our code by using classes makes our code
  - easier to read
  - easier to generalise



Classes — S3

- Use the `class` attribute to change class
- R will look for `METHOD.CLASS`
- No explicit definition of what an S3 class should be!
- S4 classes are similar in concept, but different in implementation

```
> dexter <- list(weight=30,
+  breed="mongrel/collie")
> class(dexter) <- "dog"
> print.dog <- function(x)
+  cat(paste("Breed:", x$breed))
> print(dexter)

Breed: mongrel/collie
```

Classes — S3 Inheritance

- Can have more than one class
- `plotting` a `glm` uses `plot.lm`
- Incredibly powerful and saves time

```
> dexter <- list(weight=30,
+  breed="mongrel/collie")
> class(dexter) <-
+  c("dog", "mammal")
> print.dog <- function(x)
+  cat(paste("Breed:", x$breed))
> print.mammal <- function(x)
+  cat("I'm not needed")
> summary.mammal <- function(x)
+  cat("I am!")
> print(dexter)

Breed: mongrel/collie

> summary(dexter)

I am!
```

Top Tips—Don't Attach

Attaching

- Copies your `data.frame`
- Alters your search path
- Masks important things
- Unlinks your columns
- Makes programming a nightmare
- ...is the biggest single cause of beginner R problems

```
> length(search())

[1] 9

> data<-data.frame(x=1:10,y=1:10)
> attach(data)
> length(search())

[1] 10

> y <- 1:12
> length(y)

[1] 12

> rm(y)
> length(y)

[1] 10
```

Top Tips—Don't Attach

Instead:

- Use the full name
- Use `with`
- Use `data` arguments (sometimes)

```
> data<-data.frame(x=1:10,y=1:10)
> model <- lm(data$y ~ data$x)
> model <- with(data, lm(y ~ x))
> model <- lm(y ~ x, data=data)
```

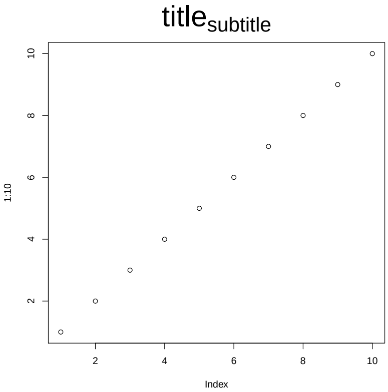
Top Tips—Don't use =

- It looks like "=="
- It isn't guaranteed to do the same as $<-$
- It is evaluated in a different scope

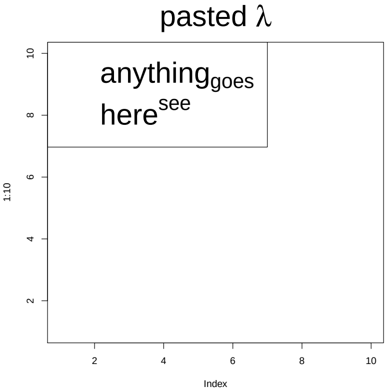
```
> test <- function(x) return(x)
> x <- 3
> test(x = 5)

[1] 5

> x

[1] 3

> test(x <- 5)
> x

[1] 5
```

Top Tips—Use Plotmath

```
> plot(1:10, main = expression(title[subtitle]),
+      cex.main = 3)
```




```
> plot(1:10, main = expression(paste("pasted ",
+      lambda)), cex.main = 3, type = "n")
> legend("topleft", c(expression(anything[goes]),
+      expression(here^see)), cex = 3)
```



Top Tips—Use Debuggers

`browser` brings up an R prompt even in the middle of a function
`fix` shows you a function, or a `data.frame` as if it were an Excel table

`trace(read.csv, edit=TRUE)` will allow you to edit a temporary version of a function—you can even insert `browser()`
`untrace(read.csv)` will remove your changes

Top Tips—Use knitr

Top Tips—Write Code Neatly

Encapsulate!

Top Tips—Recursion

You'll understand recursion when you understand recursion. Rarely used, because it's often inefficient, but can be *incredibly powerful*.

```
> factorial <- function(x)
+  if(x == 1) return(x) else
+   return(x * factorial(x-1))
> factorial(5)

## [1] 120
```

```
> factorial <- function(x)
+  if(x == 1) return(x) else
+   return(x * Recall(x-1))
> factorial(5)

## [1] 120
```

Acknowledgements

Thank you for listening! Ask lots of questions please!