



# Introduction to R: Doing things the R way

Aron C. Eklund

Center for Biological Sequence Analysis  
Technical University of Denmark

October 7, 2008



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



# Outline

## Indexing

### Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Indexing: selecting a subset of an object

### Simple vector (1 dimension)

- `x <- rivers` – a numeric vector of length 141
- `x[14]` – the fourteenth element of `x`
- `x[1:3]` – the first 3 elements of `x`



## Indexing in multiple dimensions

### A matrix or data.frame (or 2-dim. array)

- `x <- swiss` – a  $47 \times 6$  data.frame
- `x[1:3, ]` – the first 3 rows of `x`
- `x[, 1:2]` – the first 2 columns of `x`
- `x[1:3, 1:2]` – the  $3 \times 2$  sub-matrix `x`

### Hint

- Index a matrix (or DF) by `x[rows, columns]`
- Do `dim(x)` if you forget the order.



## Indexing pitfalls

### A data.frame

- `x <- swiss` – a  $47 \times 6$  data.frame
- `x[1:3]` – the first 3 **columns** of `x`
- `x$Catholic` – the 5th column of `x`

### A matrix

- `xm <- as.matrix(swiss)` – a  $47 \times 6$  matrix
- `xm[1:3]` – the first 3 **items** of the **first column** of `xm`
- `xm$Catholic` – **error**



## Indexing goes both ways

Indexing can be used to *select* and to *modify*

- `y <- x[1:3]` – select
- `x[1:3] <- y` – modify

...but what if `x[1:3]` and `y` are different lengths?

Try these:

- `rivers[2:9] <- 1`
- `rivers[2:9] <- 1:2`
- `rivers[2:10] <- 1:2`
- `rivers[1:3] <- 3:10`



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Indexing becomes even more complicated

There are *four* ways of indexing:

1. Positive integers
2. Negative integers
3. Logical
4. Character



## Indexing: methods 1 and 2

How we can subset the vector `x <- letters[1:5]`

### Method 1 = positive integers

- `x[3]` - the third element of `x`
- `x[1:3]` - the first three elements of `x`
- `x[c(2, 4)]` - the second and fourth elements of `x`

### Method 2 = negative integers

- `x[-3]` - all except the third element of `x`
- `x[c(-1, -3)]` - all except the first and third

### Method 1 $\frac{1}{2}$ = mixed positive and negative integers

- `x[c(-1, -3, 5)]` - **error**



## Indexing: method 3 (logical)

How we can subset the vector `x <- letters[1:5]`

### Method 3 = a logical vector

- `x[c(F, F, T, F, F)]` - the third element of `x`
- `x[c(T, T, T, F, F)]` - the first three elements
- `x[c(T, F)]` - `hmmmm ...`

### A note on T and F

- `TRUE` and `FALSE` are reserved words
- `T` and `F` are variables, predefined as `TRUE` and `FALSE`
- $\therefore$  Entering `T <- FALSE` would change the meaning of the statements above!



## Indexing: method 4 (character)

How we can subset the vector `x <- letters[1:5]`

- We cannot use a character vector, because this `x` has no names!

How we can subset the vector `x2 <- islands`

- `x2["Asia"]` - the third element of `x2`
- `x2[c("Africa", "Asia")]` - the first and third elements



## Detour: names in R objects

Where do these names come from?

Try these:

- `names(x)`
- `names(x) <- c("Al", "Bob", "Cyd", "Dave", "Ed")`
- `names(x2)`
- `names(x2) <- c("Al", "Bob", "Cyd", "Dave", "Ed")`
- `x3 <- c(Al = 47, Bob = 12, Cyd = 3, Ed = 21)`



## Detour: names in R objects, part 2

What about more complicated objects?

Try these:

- `colnames(sleep)`
- `rownames(sleep)`
- `dimnames(sleep)`
- `dimnames(sleep) <- list(letters[1:20], c("speed", "wheels"))`



## Summary so far

There are *four* ways of indexing:

1. Positive integers → **address list**
2. Negative integers → **address list to avoid**
3. Logical → **checklist**
4. Character → **names**

But there are really only two types of logic at work here:

1. Logical → Boolean logic
2. The other three → Set logic



# Outline

## Indexing

Overview

Four ways of indexing

**Logic, sets, and conversion**

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Conversion between indexing methods

Start with: `i.log <- islands > 8000`

`which` : logical index  $\rightarrow$  integer index

- `i.log <- islands > 8000`
- `i.int <- which(i.log)`

`names` : logical or integer  $\rightarrow$  character index

- `i.char <- names(islands)[i.log]`
- `i.char2 <- names(islands)[i.int]`

`%in%` : character  $\rightarrow$  logical index

- `i.log2 <- names(islands) %in% i.char`



## Logical operators and functions

### Element-wise operators

- `&` (and), `|` (or), `!` (not), `xor` (exclusive or)
- `big <- islands > 8000`
- `small <- islands < 20`
- `medium <- !big & !small`

### Functions returning a single logical value

- `any` – are any values TRUE?
- `all` – are all values TRUE?
- `all(big)`
- `all(big | small | medium)`
- `any(big & small)`



## Set operations

### Functions

- `union(x, y)`, `intersect(x, y)`,
- `setdiff(x, y)` – what is in `x` but not `y`

### Example in indexing

- `big <- names(islands)[islands > 8000]`
- `small <- names(islands)[islands < 20]`
- `medium <- setdiff(names(islands), union(big, small))`

### Related functions

- `unique(x)` – remove duplicated items
- `setequal(x, y)` – are two sets equivalent?



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

**Exercise**

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Indexing Exercises

- Does the order of the indexing vector matter?  
`letters[1:3]` vs. `letters[3:1]`
- What if elements of the indexing vector are repeated?  
`islands[c(2, 2, 2, 2, 3)]`
- Can we mix indexing types in multi-dimensional indexing?  
`iris[1:10, "Sepal.Length"]`
- Modifying a subset of a matrix meeting some criteria:  

```
myMat <- matrix(runif(100), nrow = 10)
myIndex <- (myMat < 0.3)
myMat2 <- myMat
myMat2[myIndex] <- 0.3
```



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## The for loop.

It works like this ...

- ```
j <- rep(NA, 10)
for(i in 1:10) {
  j[i] <- sqrt(i)
}
```

...but we tend to avoid them in R.

- Yes, `for` loops are hard to avoid in most programming languages.
- There is nothing wrong with `for` loops.
- If you find yourself writing a `for` loop, you may be missing an easier solution.



## Avoiding `for` loops, the easy way

### Most functions work directly on vectors

- `sqrt(1:10)` – square root
- `log2(rivers)` – log base 2

### Some functions operate on components of a matrix

- `colMeans(swiss)` – mean of each column
- `rowMeans(swiss)` – mean of each row

### Some functions operate on entire matrices

- `t(swiss)` – transpose a matrix
- `svd(swiss)` – singular value decomposition



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

**The "apply" functions**

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Avoiding `for` loops, the only slightly less easy way

### The `apply` family of functions

- `apply` – Apply a Functions Over Array Margins
- `sapply` – Apply a Function over a List or Vector
- `lapply` – Apply a Function over a List or Vector
- `tapply` – Apply a Function Over a "Ragged" Array
- `mapply` – Multivariate version of `sapply`
- `rapply` – A recursive version of `lapply`
- `eapply` – Apply a Function over values in an environment



## The apply family: apply

apply – apply a function over array margins

- usage: `apply(X, MARGIN, FUN, ...)`
- *MARGIN* → 1 for rows, 2 for columns, ...

apply returns a vector if *FUN* returns a scalar

- `apply(swiss, 2, mean)`  
– same as `colMeans(swiss)`
- `apply(swiss, 1, var)`  
– returns the variance of each row

apply returns a matrix if *FUN* returns a vector

- `apply(swiss, 2, sort)` – sort each column independently  
(usually a bad idea)



## The apply family: apply, part 2

What if *FUN* returns vectors of different lengths, for each row?

- `over10 <- function(x) x[x > 10]`
- `swiss.10 <- apply(swiss, 2, over10)`



## The apply family: lapply

`lapply` – apply a function over each element of a vector or list

- **usage:** `lapply(X, FUN, ...)`
- `lapply` **always returns a list.**

`lapply` **always returns a list**

- `islandsDia.l <- lapply(islands, sqrt)` – **same as**  
`sqrt(islands)`
- `misc <- list(trees = trees$Height, women =  
women$height, chicks = chickwts$weight)`
- `misc.mean.l <- lapply(misc, mean)`
- `misc.summ.l <- lapply(misc, summary)`



## The apply family: `sapply`

`sapply` – apply a function over a vector or list

- **usage:** `sapply(X, FUN, ...)`
- `sapply` is the same as `lapply` but tries to return a vector or matrix, when possible.

`sapply` returns a vector if *FUN* returns a scalar

- `islandsDia.s <- sapply(islands, sqrt)` – same as `sqrt(islands)`
- `misc.mean.s <- sapply(misc, mean)`

`sapply` returns a matrix if *FUN* returns a vector

- `misc.summ.s <- sapply(misc, summary)`
- ... but only if each vector is the same size



## Comparison of `apply`, `lapply`, `sapply`

### `apply`

- input: matrix, data frame, array
- output: vector, matrix, or (if necessary) a list

### `lapply`

- input: list or vector
- output: *always* a list

### `sapply`

- input: list or vector
- output: vector, matrix, or (if necessary) a list



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

**Exercise**

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Apply Exercise

1. Read the following expression data into R, calling the resulting object "ovc.df":

```
http://www.cbs.dtu.dk/r-intro/ovc.csv
```

The rows are probe sets; the columns are tumor samples.

2. Convert this object into a numeric matrix called "ovc".  
Hint: If the first column of `ovc.df` is not numeric, try adding the parameter `row.names = 1` in step 1.
3. Calculate the mean (across all samples) of each probeset.
4. Calculate the standard deviation (across all samples) of each probeset.  
Hint: use the function `sd`
5. Make a scatter plot of the standard deviations vs. the means.



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Examples of user-defined functions

### Take the logarithm, and then add 223

- ```
logPlus223 <- function(x) {
  step1 <- log(x)
  step2 <- step1 + 223
  return(step2) }
```

### Return all elements of an object greater than 10

- ```
over10 <- function(x) x[x > 10]
```



## Examples of user-defined functions, part 2

Plot the mean vs. the standard deviation of the rows of an object

- ```
plotMSD <- function(x, ...) {
  m <- rowMeans(x)
  s <- apply(x, 1, sd)
  plot(m, s, ...)
  invisible(data.frame(mean = m, sd = s))
}
```

### Try it

- `plotMSD(ovc)`
- `ovc.ms <- plotMSD(ovc)`
- `plotMSD(ovc, main = 'the OVC data')`



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

**Details on how they work**

Exercises

## Miscellaneous notes



# Functions

## What is returned?

1. Whatever is inside `return`.
2. Or, the last expression evaluated.

## Lexical scope

- Variables defined (or changed) in a function do not persist after the function has finished.
- If you want them to persist use `«-`.

## The `...` parameter

- All "extra" parameters are passed through.
- `...` but only if you explicitly include the `...`



## Functions 2

You can overwrite an existing function by defining a function with the same name.

- `sqrt <- function(x) { x + 749 }`

### Functions are R objects

- `ls()` – lists all object, including functions

### Functions can be passed to functions

- `lapply(misc, mean)`



## Functions: how did they do that?

### Reasons to look at an existing function.

- To understand how it works.
- To modify it to your own liking.
- To get ideas about R programming, in general.

Often, it is easy: just type the name of the function (without parentheses).

- `sd`
- `rowMeans`

Sometimes it is hidden, but you can use `getAnywhere`.

- `print.xtabs`
- `getAnywhere('print.xtabs')`



# Outline

## Indexing

Overview

Four ways of indexing

Logic, sets, and conversion

Exercise

## Loops

"for" loops

The "apply" functions

Exercise

## Writing Functions

Examples

Details on how they work

Exercises

## Miscellaneous notes



## Functions Exercise

1. Write a function that concatenates a vector with itself. Test it.
2. Write a function that returns the first letter of each item in a character vector. Test it on the names of `islands`.



# Debugging

- `traceback`
- `browser`
- `recover`
- `options(error = recover)`
- `debug`

```
○○○○○  
○○○○○○○○  
○○○○  
○○
```

```
○○○  
○○○○○○○  
○○
```

```
○○○  
○○○○  
○○
```

## To Interact with other programs

- system
- pipe



# Scripting R

- R `[options] < infile > outfile`
- **options**: `-save`, `-nosave`, `-vanilla`
- R `-help` for other options



# Time for a break