Day 3: Introduction to R scripting

Jerry Davison and Martin Morgan

Contents

1 What is a script? **1**

2 Motivation: scripts are valuable because they're: **2**

3 Writing your own functions **2**

4 R programming functions **2**
4.1 apply . . . 3
4.2 lapply . . . 3
4.3 sapply . . . 4
4.4 if . . . 5
4.5 for . . . 6

5 Good practices **7**

6 An example script **8**

7 Advanced **8**

8 Bioconductor and CRAN packages **8**

9 Resources **9**

1 What is a script?

A *script* is a text file containing R commands on separate lines; commands will be sequentially executed from the top of the file downward, for example:

```
x = 2 + 2
cat(x, '\n')
x = 7
```

will print the number 4. To execute a script *myScript.R* that you've written, in an R session enter the command `source('myScript.R')`. Scripts' greatest value may be that they assist you in generating REPRODUCIBLE RESULTS.

When you execute a script it produces the same result each time it is run – **when** its starting conditions such as input files remain the same. This characteristic can bring welcome relief if your project or complex analysis has to be recapitulated with a small change a year after its completion.

2 Motivation: scripts are valuable because they're:

- *reproducible* – share the script and its results with colleagues !
- *re-usable* – analyze additional input files with one command !!
- *extensible* – never again start related analyses from scratch !!!
- *reliable* – no danger of typing x/y when you meant x*y !!!!

3 Writing your own functions

So far we've used mainly functions that are natively available in R, like mean() and plot(). A powerful capability R and other programming languages provide is creating – that is, writing – functions yourself. For example:

```
matrixMaker <- function(m, n) {
  x = matrix(floor(runif(m*n, min=0, max=10)), nrow=m, ncol=n)
  colnames(x) = LETTERS[1:n]
  rownames(x) = 1:m
  return(x)
}
```

```
 matrixMaker(4,3) # 4 rows, 3 columns
```

```
  A B C
1 4 5 9
2 8 8 4
3 9 5 6
4 0 4 5
```

This function does several things in succession: (1) it creates a **mXn** matrix of integers uniformly distributed from 0 to 9, (2) assigns row and column names to the matrix, and (3) returns the matrix as the value of the function. Functions have a specific syntax, with zero to many arguments, a body, and a return value. In general, variables created inside a function body **are not known** outside of the function. We recommend functions as they:

- perform identical operations to different data
- guide you to thinking in structured modules
- can help you simplify a complex algorithm

Exercise: write a function that takes as its input arguments two numbers, multiplies them together, then finds the square root of their product, and returns a list with elements 1) the two numbers as a vector, 2) their product and 3) its square root. Name the elements of that list with the third, first and 20th letters of the alphabet. Execute the function twice, the first time with input values 3 and 18, the second time with values 42 and negative 9.

4 R programming functions

R provides additional resources for scripting. We'll cover a few in detail here – see Table 1 for others you may find useful. Let's first create a matrix we'll use as a test subject:

```
 set.seed(11)
 mx = matrixMaker(10,5)
 mx
```

```
   A B C D E
1  2 1 2 5 2
2  0 4 6 3 2
3  5 9 3 4 4
```

```
4  0 8 3 2 6
5  0 7 0 8 3
6  9 5 4 6 8
7  0 4 3 2 6
8  2 3 0 1 0
9  8 1 1 2 5
10 1 4 3 0 8
```

4.1 apply

Use 'apply' to operate on rows or columns individually

```
apply(mx, 1, sum) # sum values for each 'sample' 1 through 10
```

```
 1  2  3  4  5  6  7  8  9 10
12 15 25 19 18 32 15  6 17 16
```

```
apply(mx, 2, sum) # sum values for each 'experiment' A through E
```

```
 A  B  C  D  E
27 46 25 33 44
```

```
### Create an on-the-fly function for a more complex evaluation
apply(mx, 2, function(x) length(which(x!=0))) # count column non-zero values
```

```
 A  B  C  D  E
 6 10  8  9  9
```

4.2 lapply

Similar to 'apply' but operates on lists: first letś generate a list

```
mxl = list(A=mx[,'A'], B=mx[,'B'], C=mx[,'C'], D=mx[,'D'], E=mx[,'E'])
mxl
```

```
$A
 1  2  3  4  5  6  7  8  9 10
 2  0  5  0  0  9  0  2  8  1

$B
 1  2  3  4  5  6  7  8  9 10
 1  4  9  8  7  5  4  3  1  4

$C
 1  2  3  4  5  6  7  8  9 10
 2  6  3  3  0  4  3  0  1  3

$D
 1  2  3  4  5  6  7  8  9 10
 5  3  4  2  8  6  2  1  2  0

$E
 1  2  3  4  5  6  7  8  9 10
 2  2  4  6  3  8  6  0  5  8
```

```
### use 'lapply' on a 'L'-ist
lapply(mxl, sum) # total score for each exp A through E
```

```
$A
[1] 27

$B
[1] 46

$C
[1] 25

$D
[1] 33

$E
[1] 44
```

```
 lapply(mxl, function(x) length(which(x!=0))) # num. samples with value != 0
```

```
$A
[1] 6

$B
[1] 10

$C
[1] 8

$D
[1] 9

$E
[1] 9
```

```
 lapply(seq(along=mxl), function(k) sum(mxl[[k]]))
```

```
[[1]]
[1] 27

[[2]]
[1] 46

[[3]]
[1] 25

[[4]]
[1] 33

[[5]]
[1] 44
```

```
 ### Particular utility of a list -- collect objects of different classes
 mxl2 = lapply(seq(ncol(mx)), function(k) list(mx[,k], paste('Sample', LETTERS[k])))
 names(mxl2) = tolower(colnames(mx))
```

4.3 sapply

The class of result differs between 'lapply' and 'sapply'

```
 lapply(mxl, sum)
```

```
$A
[1] 27

$B
[1] 46

$C
[1] 25

$D
[1] 33

$E
[1] 44
```

```
 sapply(mxl, sum) # Operate on list
```

```
 A  B  C  D  E
27 46 25 33 44
```

```
 sapply(mxl, sum, simplify=FALSE)
```

```
$A
[1] 27

$B
[1] 46

$C
[1] 25

$D
[1] 33

$E
[1] 44
```

```
 sqrt(sapply(mxl, sum)) # Continue with additional operations
```

```
       A        B        C        D        E
5.196152 6.782330 5.000000 5.744563 6.633250
```

4.4 if

The 'if' statement assesses whether a statement is true, and acts based on that:

```
 if(any(mx==0)) cat('Some values are zero!\n')
```

```
Some values are zero!
```

```
 if(all(mx!=0)) cat('<NO> values are zero!\n')
 if (any(mx==0)) cat('Some values are zero!\n') else cat('<NO> values are zero!\n')
```

```
Some values are zero!
```

```
 if (any(mx==0)) {
   msg = 'Some values are zero!'
   cat(msg, '\n')
 } else {
   msg = '<NO> values are zero!'
   cat(msg, '\n')
 }
```

```
Some values are zero!
```

```
 ### Scope of variables assigned within 'if' includes the parent environment:
 msg
```

```
[1] "Some values are zero!"
```

4.5 for

The scope of variables assigned within 'for' also includes the parent environment:

```
 for (k in seq(ncol(mx))) {
   u = unique(mx[,k])
   len = length(u)
 }
 ### But!
 u
```

```
[1] 2 4 6 3 8 0 5
```

```
 len
```

```
[1] 7
```

```
 ### Be careful :)
 u = vector('list', ncol(mx))
 len = vector('integer', ncol(mx))
 #
 for (k in seq(ncol(mx))) {
   u[[k]] = unique(mx[,k])
   len[k] = length(u[[k]])
 }
 names(u) = names(len) = colnames(mx)
 u
```

```
$A
[1] 2 0 5 9 8 1

$B
[1] 1 4 9 8 7 5 3

$C
[1] 2 6 3 0 4 1

$D
[1] 5 3 4 2 8 6 1 0

$E
[1] 2 4 6 3 8 0 5
```

```
 len

A B C D E
6 7 6 8 7
```

Table 1: A selection of *R* functions

`dir`, `read.table` **(and friends),** `scan` List files in a directory, read spreadsheet-like data into *R*, efficiently read homogeneous data (e.g., a file of numeric values) to be represented as a matrix.

`c`, `factor`, `data.frame`, `matrix` Create a vector, factor, data frame or matrix.

`summary`, `table`, `xtabs` Summarize, create a table of the number of times elements occur in a vector, cross-tabulate two or more variables.

`t.test`, `aov`, `lm`, `anova`, `chisq.test` Basic comparison of two (`t.test`) groups, or several groups via analysis of variance / linear models (`aov` output is probably more familiar to biologists), or compare simpler with more complicated models (`anova`); χ^2 tests.

`dist`, `hclust` Cluster data.

`plot` Plot data.

`ls`, `str`, `library`, `search` List objects in the current (or specified) workspace, or peak at the structure of an object; add a library to or describe the search path of attached packages.

`lapply`, `sapply`, `mapply`, `aggregate` Apply a function to each element of a list (`lapply`, `sapply`), to elements of several lists (`mapply`), or to elements of a list partitioned by one or more factors (`aggregate`).

`with` Conveniently access columns of a data frame or other element without having to repeat the name of the data frame.

`match`, `%in%` Report the index or existence of elements from one vector that match another.

`split`, `cut`, `unlist` Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor, unlist (concatenate) list elements.

`strsplit`, `grep`, `sub` Operate on character vectors, splitting it into distinct fields, searching for the occurrence of a patterns using regular expressions (see `?regex`, or substituting a string for a regular expression).

`install.packages` Install a package from an on-line repository into your *R*.

`traceback`, `debug`, `browser` Report the sequence of functions under evaluation at the time of the error; enter a debugger when a particular function or statement is invoked.

5 Good practices

To increase their value to you, work towards building scripts that are

- *readable* – organize related commands into blocks separated by vertical space. Use comments freely – text on any line after a '#' is ignored by R. Indent commands in entering nested levels of control and reverse the indentation after exiting them.
- *modular* – put functions you write in separate files and `source()` them in scripts that use the functions.
- *maintainable* – variable and file names are meaningful, projects are in separate directories: create a set of directories in a **project** directory.

For example you may create a folder or directory **RNAseq** and under that the following folders or directories:

Subdirectory/Folder	Can contain
data	Data files created during the analysis (.rda for example)
extdata	Data files available at the start of the analysis (.txt for example)
R	R script files that only contain R functions you or others wrote
script	R script files with analyses that may call functions in the R directory
doc	Documents and similar files

6 An example script

```
# filename: little_script.R
#
# Example script illustrating good practices
#
# jdavison & mtmorgan
#
# -> source('little_script.R', echo=TRUE, max=Inf)
#

### Scatter plot with title, "toFile" and ... passed as arguments
plotIt <- function(x, y, main='', toFile=FALSE, ...) {
  if(toFile) pdf(file='myPlot.pdf')
  plot(x,y, type='o', main=main, ...)
  abline(h=0, lty=3,col='blue')
  abline(v=0, lty=2, col='red')
  if(toFile) invisible(dev.off())
}

### Use defaults
x = 0:20/pi
y = sin(x)
plotIt(x,y)

### Specify title
x = seq(0,10,0.1)
y = x^2*exp(-x)
plotIt(x,y, main='Exponential function')

### Write to file, specify additional plot options
zed = seq(0,20,0.1)/pi
plotIt(sin(2*zed), cos(3*zed), main='Lissajous figure a:b = 2:3',
       toFile=TRUE, pch=22, lwd=3, col='orange')
```

7 Advanced

If a function you write or use produces an error message when you run it, or not what you expect, use the debugger to help you understand what is happening inside the function:

```
debug(plotIt)
x = -5:5
y = log(abs(x))
plotIt(x, y)
```

8 Bioconductor and CRAN packages

There are many packages available to address both general needs, like the limma package, and more specific ones like edgeR. Bioconductor and CRAN packages can be downloaded and added to your R environment with the following commands:

```
source('http://bioconductor.org/biocLite.R')
biocLite('edgeR') # For example
library(edgeR)
```

9 Resources

- The Art of R Programming, Matloff
- The Wikipedia R entry
- workflow ideas
- Much more documentation is available at CRAN – be sure to see the *Manuals*, *FAQS* and *Contributed* links.

```
options(prompt='> ', continue='+ ')
```