R language: A quick tutorial

Denis Puthier

December 4, 2008

Laboratoire INSERM TAGC/ERM206, Parc Scientifique de Luminy case 928,13288 MARSEILLE cedex 09, FRANCE.

Contents

1 Basic aspects of the language. 2

2 Syntax for calling Functions. 3

3 Modes. Functions for creating vectors. 5
3.1 Modes . . . 5
3.2 Functions and operators for creating vectors. . . . 5

4 Vector manipulation. 7
4.1 Indexing vectors . . . 7
4.2 Replacing parts of a vector . . . 8
4.3 Vectorization . . . 9

5 Objects of class: `factor`, `Matrix`, `data.frame` and `list`. 9
5.1 factor . . . 9
5.2 Matrix . . . 10
5.3 data.frame . . . 11
5.4 list . . . 11
5.5 The "Apply" family of functions . . . 12
5.6 The `tapply` function . . . 12
5.7 Graphics with R . . . 13
5.8 Graphics . . . 13
5.9 A simple example using two colour microarray data processed with basic R functions. 13
5.10 S4 objects in R . . . 14

6 Bioconductor 17
6.1 Example: Affy Data normalization . . . 18

7 Further readings 18

This tutorial is just a brief tour of the language capabilities and is intented to give some clues to begin with the R programming language. For a more detailled overview see "R for beginners" (E. Paradis, http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf)

1 Basic aspects of the language.

R is an object-oriented programming language. You can easily create basic objects of class `vector`, `matrix`, `data.frame`, `list`, `factor`,...

A vector x that contains one value.

```
> x <- 15
```

You can see the content of x by simply calling it.

```
> x
```

```
[1] 15
```

Alternatively, you can use the "=" operator. However "<-" is most generally prefered.

```
> x = 22
> x
```

```
[1] 22
```

You can add comment by starting a line with a hash (#). In this case the code at the right of the hash won't be interpreted.

```
> #x <-  57
> x
```

```
[1] 22
```

```
> x
```

```
[1] 22
```

Instructions can be separated by semi-colons (;) or new-line.

```
> x <- 12; y <- 13
> x;y
```

```
[1] 12
```

```
[1] 13
```

Once values are assigned to an object, R will store this object in the memory (RAM). Previously created objects can be **lis**ted using the `ls` function.

```
> ls()

[1] "x" "y"
```

Object can be deleted using the `rm` (**r**e**m**ove) function.

```
> rm(x)
> rm(y)
> ls()

character(0)
```

2 Syntax for calling Functions.

In the above section we have created vectors containing numeric data. We have also used functions (`ls` and `rm`). We can use numerous functions to perform specific tasks. When calling a function, we will use this generic syntax:

```
 - NameOfTheFunction(arg1= a, arg2= b,...)
```

arg1 et arg2 (...) : arguments of the function.a and b : The objects that will be passed to the function.

To access the documentation of a given function, use the `help` function (or the question mark). The documentation gives you an overview of the function (usage, arguments name and class, returned values and examples). For instance to get information about the `substr` function (used to extract part of a character string) use one of the following instructions:

```
> help(substr) # or ?substr

substr                package:base                R Documentation

Substrings of a Character Vector

Description:

     Extract or replace substrings in a character vector.

Usage:

     substr(x, start, stop)
     substring(text, first, last = 1000000)
     substr(x, start, stop) <- value
     substring(text, first, last = 1000000) <- value

Arguments:

 x, text: a character vector.
```

start, first: integer. The first element to be replaced.

stop, last: integer. The last element to be replaced.

   value: a character vector, recycled if necessary.

Details:

     'substring' is compatible with S, with 'first' and 'last' instead
     of 'start' and 'stop'. For vector arguments, it expands the
     arguments cyclically to the length of the longest _provided_ none
     are of zero length.

     When extracting, if 'start' is larger than the string length then
     '""' is returned.

     For the extraction functions, 'x' or 'text' will be converted to a
     character vector by 'as.character' if it is not already one.

     For the replacement functions, if 'start' is larger than the
     string length then no replacement is done. If the portion to be
     replaced is longer than the replacement string, then only the
     portion the length of the string is replaced.

     If any argument is an 'NA' element, the corresponding element of
     the answer is 'NA'.

Value:

     For 'substr', a character vector of the same length and with the
     same attributes as 'x' (after possible coercion).

     For 'substring', a character vector of length the longest of the
     arguments. This will have names taken from 'x' (if it has any
     after coercion, repeated as needed), and other attributes copied
     from 'x' if it is the longest of the arguments).

Note:

     The S4 version of 'substring<-' ignores 'last'; this version does
     not.

     These functions are often used with 'nchar' to truncate a display.
      That does not really work (you want to limit the width, not the
     number of characters, so it would be better to use 'strtrim'), but
     at least make sure you use 'nchar(type="c")'.

References:

     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole. ('substring'.)

See Also:

     'strsplit', 'paste', 'nchar'.

Examples:

```
substr("abcdef",2,4)
substring("abcdef",1:6,1:6)
## strsplit is more efficient ...

substr(rep("abcdef",4),1:4,4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
substr(x, 2, 5)
substring(x, 2, 4:6)

substring(x, 2) <- c("..", "+++")
x
```

When calling a function, the name of the arguments can be omitted if they are placed as expected. For instance if one wants to extract character 2 to 4 in the string "microarray":

```
> substr("microarray", 2, 4)

[1] "icr"
```

If the arguments are not in the expected order their names are mandatory (note that, for convenience, they can be abbreviated but the abbreviation used should be unambiguous):

```
> substr(2,4,"microarray")                    # arguments are misplaced
[1] NA
Warning message:
In substr(2, 4, "microarray") :
  NAs introduits lors de la conversion automatique

> substr(start=2,stop=4,x="microarray") # works

[1] "icr"

> substr(st=2,st=4,x="microarray")            #ambiguous
Erreur dans substr(st = 2, st = 4, x = "microarray") :
  argument formel "start" correspondant Ã  plusieurs arguments fournis
```

3 Modes. Functions for creating vectors.

3.1 Modes

A vector can contain several variables of the same mode. Most frequently, the mode will be one of: "numeric", "character" or "logical".

3.2 Functions and operators for creating vectors.

The function `c` The function `c` is used to **c**ombine values into a vector.

```
> mic <- c("Agilent", "Affy")       # A character vector
> mic

[1] "Agilent" "Affy"

> is(mic)

[1] "character"           "vector"              "data.frameRowLabels"

> num <- c(1,2,3)                   # A numeric vector
> num
```

```
[1] 1 2 3

> is(num)

[1] "numeric" "vector"

> bool <- c(T,F,T)                  # A logical vector
> bool

[1]  TRUE FALSE  TRUE

> is(bool)

[1] "logical" "vector"

> num > 1.5                          # Also returns a logical vector

[1] FALSE  TRUE  TRUE
```

The operator ":" This operator generates a sequence from 'from' to 'to' in steps of '1' . For numeric arguments 'from:to' is equivalent to 'seq(from, to)'.

```
> 1:3

[1] 1 2 3

> seq(3, 1)

[1] 3 2 1

> 3:1                    # Values in decreasing order

[1] 3 2 1
```

Functions `rep`, `seq` The `rep` function **rep**eats a value as many times as requested.

```
> rep(3,5)

[1] 3 3 3 3 3
```

The `seq` function is used to create numeric **seq**uence.

```
> seq(0,10,by=2)

[1]  0  2  4  6  8 10

> seq(0,10,length.out=3)

[1]  0  5 10
```

Functions to generate random number the `rnorm` function is used to generate normally distributed values with mean equal to 'mean' (default 0) and standard deviation equal to 'sd' (default 1).

```
> x <- rnorm(1000,mean=2,sd=2)
> hist(x)
```

4 Vector manipulation.

4.1 Indexing vectors

Extraction or replacement of parts of a vector can be performed using the "[" operator (which is equivalent to the `subset` function). Numeric vectors, logical vectors or names are used to indicate which positions in the vector are to be extracted or replaced.

```
> set.seed(1)
> x <- round(rnorm(10),2)
> x

 [1] -0.63  0.18 -0.84  1.60  0.33 -0.82  0.49  0.74  0.58 -0.31

> x[2]

[1] 0.18

> x[1:3]

[1] -0.63  0.18 -0.84

> x[c(2,6)]

[1]  0.18 -0.82

> which(x>0)       # returns the positions containing positive values

[1] 2 4 5 7 8 9

> x[which(x>0)]    # returns the requested positive values (using a vector of integers)

[1] 0.18 1.60 0.33 0.49 0.74 0.58

> x > 0            # returns TRUE if a positions contains a positive value

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

> x[x > 0]

[1] 0.18 1.60 0.33 0.49 0.74 0.58
```

```
> nm <- paste("n",1:10, sep="")
> nm

 [1] "n1"  "n2"  "n3"  "n4"  "n5"  "n6"  "n7"  "n8"  "n9"  "n10"

> names(x) <- nm
> x["n10"]          # Indexing with the names of the element

  n10
-0.31
```

4.2 Replacing parts of a vector

Simply use the "<-" operators. Note that in R, missing values are defined as "NA" (Not Attributed).

```
> x[1:2] <- c(10,11)
> x

   n1    n2    n3    n4    n5    n6    n7    n8    n9   n10
10.00 11.00 -0.84  1.60  0.33 -0.82  0.49  0.74  0.58 -0.31

> x[4:6] <- NA
> x

   n1    n2    n3    n4    n5    n6    n7    n8    n9   n10
10.00 11.00 -0.84    NA    NA    NA  0.49  0.74  0.58 -0.31

> is.na(x)              # returns TRUE if the position is NA

   n1    n2    n3    n4    n5    n6    n7    n8    n9   n10
FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

> x <- na.omit(x)       # To delete NA values
> x

   n1    n2    n3    n7    n8    n9   n10
10.00 11.00 -0.84  0.49  0.74  0.58 -0.31
attr(,"na.action")
n4 n5 n6
 4  5  6
attr(,"class")
[1] "omit"
```

4.3 Vectorization

R is intented to handle large data sets and to retrieve information using a concise syntax. Thanks to the internal feature of R, called *vectorization*, numerous operation can be written without a loop:

```
> x <- 0:10
> y <- 20:30
> x+y

 [1] 20 22 24 26 28 30 32 34 36 38 40

> x^2

 [1]   0   1   4   9  16  25  36  49  64  81 100

> sqrt(x)

 [1] 0.000000 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [9] 2.828427 3.000000 3.162278
```

5 Objects of class: `factor`, `Matrix`, `data.frame` and `list`.

5.1 factor

This object looks like a vector. It is used to store categorical variables. A vector can be converted to a factor using the `as.factor` function. The `levels` function can be used to extract the names of the categories and to rename them.

```
> x <- rep(c("good","bad"),5)
> x

 [1] "good" "bad"  "good" "bad"  "good" "bad"  "good" "bad"  "good" "bad"

> x <- as.factor(x)
> x

 [1] good bad  good bad  good bad  good bad  good bad
Levels: bad good

> x <- as.factor(x)
> x

 [1] good bad  good bad  good bad  good bad  good bad
Levels: bad good

> levels(x)

[1] "bad"  "good"
```

```
> levels(x) <- 0:1
> x

 [1] 1 0 1 0 1 0 1 0 1 0
Levels: 0 1

> table(x)

x
0 1
5 5
```

5.2 Matrix

Matrix are intended to store 2-dimensional datasets. Each value will be of the same mode. As with vectors, one can use names, numeric vectors or a logical vector for indexing this object. One can index rows or columns or both.

```
> x <- matrix(1:10,ncol=2)
> colnames(x) <- c("ctrl","trmt")
> row.names(x) <- paste("gene", 1:5, sep="_")
> x

       ctrl trmt
gene_1    1    6
gene_2    2    7
gene_3    3    8
gene_4    4    9
gene_5    5   10

> x[,1]                       # first column

gene_1 gene_2 gene_3 gene_4 gene_5
     1      2      3      4      5

> x[1,]                       # first row

ctrl trmt
   1    6

> x[1,2]                      # row 1 and column 2

[1] 6

> x[c(T,F,T,T,T),]
```

```
       ctrl trmt
gene_1    1    6
gene_3    3    8
gene_4    4    9
gene_5    5   10
```

Note that the syntax below that use a logical matrix is also frequently used to extract or replace part of a matrix.

```
> x < 2 | x > 8
```

```
        ctrl  trmt
gene_1  TRUE FALSE
gene_2 FALSE FALSE
gene_3 FALSE FALSE
gene_4 FALSE  TRUE
gene_5 FALSE  TRUE
```

```
> x[x < 2 | x > 8] <-  NA
```

5.3 data.frame

This object is very similar to the matrix except that each column can contain a given mode (a column with characters, a column with logicals, a column with numerics,...).

5.4 list

Object of class list can store any type of object. They should be indexed with the "[[" or $ operators.

```
> l1 <- list(A=x, B=rnorm(10))
> l1
```

```
$A
       ctrl trmt
gene_1   NA    6
gene_2    2    7
gene_3    3    8
gene_4    4   NA
gene_5    5   NA

$B
 [1]  1.51178117  0.38984324 -0.62124058 -2.21469989  1.12493092 -0.04493361
 [7] -0.01619026  0.94383621  0.82122120  0.59390132
```

```
> is(l1[[1]])
```

```
[1] "matrix"    "array"     "structure" "vector"
```

```
> is(l1[[2]])
```

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

```
> l1$A
```

```
       ctrl trmt
gene_1   NA    6
gene_2    2    7
gene_3    3    8
gene_4    4   NA
gene_5    5   NA
```

5.5 The "Apply" family of functions

They are used to loop through row and columns of a matrix (or dataframe) or through elements of a list.
The `apply` function

```
> x <- matrix(1:10,ncol=2)
> apply(x,1, sd)        # 1 stands for rows
```

```
[1] 3.535534 3.535534 3.535534 3.535534 3.535534
```

```
> apply(x,2, sd)        # 2 stands for colums
```

```
[1] 1.581139 1.581139
```

The `lapply` function The `lapply` is used for list (or data.frame).

```
> lapply(l1, is)        # The types of the object stored in the list.
```

```
$A
[1] "matrix"    "array"     "structure" "vector"

$B
[1] "numeric" "vector"
```

5.6 The `tapply` function

This function tipically takes a vector and a factor as arguments. For instance, we have data that fall in three caterogies ("good", "bad", "medium"). We can compute different statistics related to the category:

```
> cat <- rep(c("good", "bad", "medium"),5)
> cat <- as.factor(cat)
> x <- rnorm(length(cat))
> x[cat == "good"] <- x[cat == "good"]+2
> x[cat == "medium"] <- x[cat == "medium"]+1
> boxplot(x~cat)
> tapply(x,cat,sd)
```

```
      bad      good    medium
1.0842461 1.1309181 0.5898764

> tapply(x,cat,mean)

      bad      good    medium
0.2472168 1.9158887 0.6120878

> tapply(x,cat,length)

   bad   good medium
     5      5      5
```

5.7 Graphics with R

5.8 Graphics

R offers a large variety of high-level graphics functions (`plot`, `boxplot`, `barplot`, `hist`, `pairs`, `image`, ...). The generated graphics can be modified using low-level functions (**points**, **text**, **line**, **abline**, **rect**, **legend**, ...).

5.9 A simple example using two colour microarray data processed with basic R functions.

Loading and visualizing the data

```
> # Create an OS-independent path to the directory "swirldata"
> # located in the package "marray"
> path <- system.file("swirldata", package = "marray")
> path

[1] "/usr/lib/R/library/marray/swirldata"

> getwd()                   # the current working directory

[1] "/win/D/COMMUNICATION/PERSONNEL/cours/LES COURS/2008/brazil/R_tuto"

> setwd(path)               # set working directory to "path"
> getwd()                   # The working directory has changed

[1] "/usr/lib/R/library/marray/swirldata"

> dir()                     # list the working directory

 [1] "fish.gal"               "RQuickTutorial-029.eps" "RQuickTutorial-029.pdf"
 [4] "RQuickTutorial-030.eps" "RQuickTutorial-030.pdf" "swirl.1.spot"
 [7] "swirl.2.spot"           "swirl.3.spot"           "swirl.4.spot"
[10] "SwirlSample.txt"
```

```
> #file.show("swirl.1.spot") # this file contains a Header
> d <- read.table("swirl.1.spot", header=T, sep="\t", row.names=1)
> is(d)

[1] "data.frame" "list"       "oldClass"   "vector"

> colnames(d)

 [1] "grid.r"            "grid.c"            "spot.r"
 [4] "spot.c"            "area"              "Gmean"
 [7] "Gmedian"           "GIQR"              "Rmean"
[10] "Rmedian"           "RIQR"              "bgGmean"
[13] "bgGmed"            "bgGSD"             "bgRmean"
[16] "bgRmed"            "bgRSD"             "valleyG"
[19] "valleyR"           "morphG"            "morphG.erode"
[22] "morphG.close.open" "morphR"            "morphR.erode"
[25] "morphR.close.open" "logratio"          "perimeter"
[28] "circularity"       "badspot"

> G <- d[,"Gmedian"]
> R <- d[,"Rmedian"]
> plot(R,G, pch=16, cex=0.5, col="red") # low values are densely packed
>                                       # in the UL corner
> library(geneplotter)
> smoothScatter(R,G)                    # even more marked
> boxplot(R,G)                          # 1rst , 2nd (median) and 3rd quartile
> R <- log2(R)
> G <- log2(G)
> plot(R,G, pch=16, cex=0.5, col="red")
> boxplot(R,G)
```

MA plot A MA plot without normalization.

```
> M <- R-G
> A <- R+G
> plot(A,M, pch=16,cex =0.5)
> low <- lowess(M~A)
> lines(low,col="blue", lwd = 2)       # lwd: line width
> abline(h=0, col="red")              # h: horizontal
> abline(h=-1, col="green")
> abline(h=1, col="green")
> # Genes considered as "good" have an
> # absolute ratio of 1 (2 in linear scale).
> good <- abs(M) > 1
> points(A[good], M[good], col="red")
> # They aren't so good...
> gn <- 1:nrow(d)
> text(A[good], M[good], lab=gn[good],cex=0.4,pos=2)
```

5.10 S4 objects in R

In the last example, we have used Gmedian and Rmedian data. However, we should have also considered the background signal. The problem is that we should have created four vectors (or four matrix if several microarrays were analyzed). As it is rather difficult to manipulate the four matrices we will design a class ("micBatch") that will contain all information within a single object.

```
> # micBatch
> # "representation" corresponds to all attributes of the object:
> # "prototype" corresponds corresponds to default values:
>
> setClass("micBatch",
+     representation(
+         R="matrix",
+         G="matrix",
+         Rb="matrix",
+         Gb="matrix",
+         phenotype="matrix",
+         genes="character",
+         description="character"),
+     prototype=list(
+         R=matrix(nr=0,nc=0),
+         G=matrix(nr=0,nc=0),
+         Rb=matrix(nr=0,nc=0),
+         Gb=matrix(nr=0,nc=0),
+         phenotype=matrix(nr=0,nc=0),
+         description=new("character")
+             )
+     )

[1] "micBatch"

> myMA=new("micBatch")
> slotNames(myMA)

[1] "R"           "G"           "Rb"          "Gb"          "phenotype"
[6] "genes"       "description"

> getClassDef("micBatch")

Class âmicBatchâ

Slots:

Name:            R           G          Rb          Gb   phenotype       genes
Class:      matrix      matrix      matrix      matrix      matrix   character

Name:  description
Class:   character
```

```
> myMA@R <- matrix(rnorm(20),nc=5)
> myMA@R <- matrix(rnorm(20),nc=5)
> myMA@Rb <- matrix(rnorm(20),nc=5)
> myMA@Gb <- matrix(rnorm(20),nc=5)
> # let's visualize the object.
> # (implicit call to the "show" method).
> myMA

An object of class âmicBatchâ
Slot "R":
           [,1]       [,2]        [,3]      [,4]       [,5]
[1,]  1.9803999 -0.1350546  0.02800216 1.4655549 -0.7099464
[2,] -0.3672215  2.4016178 -0.74327321 0.1532533  0.6107264
[3,] -1.0441346 -0.0392400  0.18879230 2.1726117 -0.9340976
[4,]  0.5697196  0.6897394 -1.80495863 0.4755095 -1.2536334

Slot "G":
<0 x 0 matrix>

Slot "Rb":
             [,1]       [,2]       [,3]       [,4]      [,5]
[1,]  0.291446236 -0.5895209 -1.5235668 -0.3041839 1.2078678
[2,] -0.443291873 -0.5686687  0.5939462  0.3700188 1.1604026
[3,]  0.001105352 -0.1351786  0.3329504  0.2670988 0.7002136
[4,]  0.074341324  1.1780870  1.0630998 -0.5425200 1.5868335

Slot "Gb":
           [,1]        [,2]       [,3]       [,4]       [,5]
[1,]  0.5584864 -0.47340064  0.1580288  0.9101742 -0.4616447
[2,] -1.2765922 -0.62036668 -0.6545846  0.3841854  1.4322822
[3,] -0.5732654  0.04211587  1.7672873  1.6821761 -0.6506964
[4,] -1.2246126 -0.91092165  0.7167075 -0.6357365 -0.2073807

Slot "phenotype":
<0 x 0 matrix>

Slot "genes":
character(0)

Slot "description":
character(0)

> # Let's define the method "show" for an object of class
> # micBatch
>
> setMethod("show", signature("micBatch"),
+         function(object){
```

```
+         cat("An instance of class micBatch\n")
+         cat("Nb sample=", ncol(object@R),"\n")
+         cat("Nb genes=", nrow(object@R),"\n")
+ })

[1] "show"

> # Let's call the "show" method
> myMA

An instance of class micBatch
Nb sample= 5
Nb genes= 4
```

As shown in this example we can easily define methods for the class `micBatch`. Just remind that we could have define a method for the "[" operator. This would have been particularly interesting to extract rows (genes) and columns (arrays) of the object.

6 Bioconductor

Bioconductor offers a set of libraries of functions that are useful to biologists (see http://bioconductor.org for more informations)

- cDNA Microarray analysis:
  -marray, ArrayQuality, arrayMagic, limma, sma
- Affymetrix GeneChip analysis:
  -Affy, simpleaffy
- Probe Metadata:
  - Annotate, hgu133aprobe, hgu95av2probe, ABPkgBuilder
- High level plotting functions - geneplotter
- Microarray data filtering:
  -Genefilter
- Statistical analysis:
  -SAMR, siggenes, multtest, DEDS, pickgene
- Interpretation:
  -GO, Gostats, goCluster, geneplotter
- Graphs:
  -graph, Rgraphviz, biocGraph,
- Training datasets:
  -golubEsets, fibroEseT...
- Flow cytometry:
  -flowCore, flowViZ...

- Proteomics -MassSpecWavelet, PROcess, xcms...
- Image analysis:
  -EBImage

6.1 Example: 2 color Arrays

Affymetrix data can be analyzed using the `affy` package.
You can run this example using raw data of the GSE2004 dataset (Gene Expression Omnibus).

```
> library(marray)
> path <- system.file("swirldata", package = "marray")
> setwd(path)
> m <- read.Spot()
> plot(m[,1])
> maImage(m[,1])
> n <- maNorm(m,norm="printTipLoess")
> plot(n[,1])
> plot(n[,2])
```

6.2 Example: Affy Data normalization

Affymetrix data can be analyzed using the `affy` package.
You can run this example using raw data of the GSE2004 dataset (Gene Expression Omnibus).

```
> setwd("/where/the/cel/files/are")
> library(affy)
> d <- ReadAffy()
> is(d)
> image(d[,1])
> n <- rma(d)
> write.exprs(n,"normalized.txt")
```

7 Further readings

- R for beginners
  http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

- Bioinformatics and Computational Biology Solutions Using R and Bioconductor. Robert Gentleman, Vincent Carey, Wolfgang Huber, Rafael Irizarry, Sandrine Dudoit.ISBN: 978-0-387-25146-2
  . http://www.springer.com/computer/computational+biology+and+bioinformatics/book/978-0-387-25146-2