Introduction to R and Bioconductor

K. Zimmermann and P. Thomas

WBI

December 2, 2010

1By courtesy of Karl Kugler UMIT-Hall in Tirol, Institute for Bioinformatics and Translational Research
1 Introduction

2 R Basics
   - Data Structures
   - Naming
   - Subsetting
   - Getting Help
   - Packages
   - Package Bioconductor
   - Control Flow
   - Apply
   - Functions

3 Plotting
R-Project: R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS.

http://www.r-project.org/

Bioconductor is an open source and open development software project for the analysis and comprehension of genomic data.

http://www.bioconductor.org/

Books:

- Bioconductor Case Studies
  Florian Hahne, Wolfgang Huber, Robert Gentleman, Seth Falcon

- Programmieren mit R
  Uwe Ligges

- R Programming for Bioinformatics
  Robert Gentleman

Manuals, Vignettes, R-Help
- **vector**: arrays of the same type
- **list**: can contain objects of different types
- **data.frame**: table-like
- **factor**: categorical
- **Classes**: arbitrary record type
- **function**: add functionality to the language
In R, the basic data types are vectors, not scalars.

A vector contains an indexed set of values that are all of the same type:
- logical
- numeric
- character

The numeric type can be further broken down into integer, single, and double types (but this is only important when making calls to foreign functions, e.g., C or Fortran.)
There are two symbols that can be used for assignment: `<-` and `=.

```r
> v <- 123
> v

[1] 123

> s = "a string"
> s

[1] "a string"

> t <- TRUE
> t

[1] TRUE

> letters

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

> length(letters)

[1] 26
```
Functions to create vectors are:

- `c()` – concatenate
- `:` – integer sequence, `seq()` general sequence
- `rep()` – repetitive patterns
- `vector()` – vector with certain mode

```r
> c(1,3,4,7,9)
[1] 1 3 4 7 9

> seq(1,3)
[1] 1 2 3

> 1:3
[1] 1 2 3

> rep(1:2, 3)
[1] 1 2 1 2 1 2

> vector(mode="character", length=5)
[1] "" "" "" "" ""
```
> a <- c(T, F)
> a
[1] TRUE FALSE

> mode(a)
[1] "logical"

> a <- as.numeric(a)
> a
[1] 1 0

> a <- as.character(a)
> a
[1] "1" "0"
Matrices and arrays can be created using `matrix()` and `array()`. They are represented as a vector with a dimension attribute.

**Example for `matrix()`**

```r
> x <- matrix(1:10, nrow=2)
> dim(x)
[1] 2 5
> x
[1,]  1  3  5  7  9
[2,]  2  4  6  8 10
> is.vector(x)
[1] FALSE
> as.vector(x)
[1] 1 2 3 4 5 6 7 8 9 10
```
Example for array():

```r
> a1 <- array(1:3, c(2,4,3))
> a1

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

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

, , 3
[1,]  2  1  3  2
[2,]  3  2  1  3
```
In addition to atomic vectors, R has a number of recursive data structures. Among the important members of this class are lists and environments.

A list is an ordered set of elements that can be arbitrary R objects (vectors, other lists, functions, . . .). In contrast to atomic vectors, which are homogeneous, lists and environments can be heterogeneous.

```r
> lst = list(a=1:3, b = "ciao", c = sqrt)
> lst

$a
[1] 1 2 3

$b
[1] "ciao"

$c
function (x) .Primitive("sqrt")

> lst$c(81)
[1] 9
```
One difference between lists and environments is that there is no concept of ordering in an environment. All objects are stored and retrieved by name.

```R
> e1 = new.env()
> e1["a"] <- 1:3
> assign("b", "ciao", e1)
> ls(e1)

[1] "a" "b"
```

Random access to large environment can be sped up by using hashing (see the manual page of new.env).

Names must match exactly (for lists, partial matching is used for the `$` operator).
Data frames are a special R structure used to hold a set of spreadsheet-like table. In a data.frame, the observations are the rows and the covariates are the columns.

- Data frames can be treated like matrices and be indexed with two subscripts. The first subscript refers to the observation, the second to the variable.
- Data frames are really lists, and list subsetting can also be used on them.
```R
> df <- data.frame(type=rep(c("case", "control"), c(2, 3)), time=rexp(5))
> df

     type     time
   1  case 0.04823183
   2  case 0.19438853
   3 control 2.86068777
   4 control 1.65556862
   5 control 0.32826093

> df$time or df[,,"time"]
[1] 0.04823183 0.19438853 2.86068777 1.65556862 0.32826093

> names(df)
[1] "type" "time"

> rn <- paste("id", 1:5, sep="")
> rownames(df) <- rn
> df[1:2, ]

     type     time
   id1  case 0.04823183
   id2  case 0.19438853
```
The elements of a vector can (and often should) be given names. Names can be specified:

- **at creation time**

```r
> x <- c(a=0, b=2)
> x
a b
0 2

> names(x) <- c("Australia", "Brazil")
> x
Australia    Brazil
     0       2
```
later by using names(), dimnames(), rownames(), colnames()

```r
> x <- matrix(c(4, 8, 5, 6, 4, 2, 1, 5, 7), nrow=3)
> x
[,1] [,2] [,3]
[1,]  4  6  1
[2,]  8  4  5
[3,]  5  2  7

> dimnames(x) <- list(year = c("2005", "2006", "2007"), "mode of transport" = c("plane", "bus", "boat"))
> x

   mode of transport
year plane bus boat
2005  4  6  1
2006  8  4  5
2007  5  2  7
```
One of the most powerful features of R is its ability to manipulate subsets of vectors and arrays.

Subsetting is indicated by [ , ].

Note that [ is actually a function (try get("[")). \( x[2, 3] \) is equivalent to \\
"["(x, 2, 3). Its behavior can be customized for particular classes of objects.

The number of indices supplied to [ must be either the dimension of x or 1.

\[
\begin{align*}
> a1[1,2,3] \\
[1] 1 \\
> a1[2] \\
[1] 2 \\
> \text{# equivalent to} \\
> \text{as.vector(a1)[2]} \\
[1] 2
\end{align*}
\]
Subsetting with Positive Indices

- A subscript consisting of a vector of positive integer values is taken to indicate a set of indices to be extracted.
  
  ```r
  > x <- 1:10
  > x[2]
  [1] 2
  > x[1:3]
  [1] 1 2 3
  ```

- A subscript which is larger than the length of the vector being subsetted produces an NA in the returned value.
  
  ```r
  > x[9:11]
  [1] 9 10 NA
  ```
Subset expressions can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```r
> x[2] <- 200
> x[8:10] <- 10
> x
```

```
[1] 1 200 3 4 5 6 7 10 10 10
```

If a zero or NA occurs as a subscript in this situation, it is ignored.

```r
> x[c(NA,0:3)] <- 15
> x
```

```
[1] 15 15 15 4 5 6 7 10 10 10
```
Subsetting with Negative Indexes

- A subscript consisting of a vector of negative integer values is taken to indicate the indices which are not to be extracted.
  ```r
  > x <- 10:1
  > x
  [1] 10  9  8  7  6  5  4  3  2  1
  > x[-(1:3)]
  [1]  7  6  5  4  3  2  1
  ```

- Subscripts which are zero are ignored and produce no corresponding values in the result.

- NA subscripts are not allowed.

- Positive and negative subscripts cannot be mixed.
Assignments with Negative Indexes

- Negative subscripts can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```r
> x[-(8:10)] = 10
> x
[1] 10 10 10 10 10 10 10 3 2 1
```

- Zero subscripts are ignored.
- NA subscripts are not permitted.
Subsetting by Logical Predicates

- Vector subsets can also be specified by a logical vector of TRUEs and FALSEs.
  
  ```r
  > x <- 1:10
  > x > 5
  
  [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
  
  > x[x > 5]
  
  [1] 6 7 8 9 10
  
  - NA values used as logical subscripts produce NA values in the output.
If a vector has named elements, it is possible to extract subsets by specifying the names of the desired elements.

```r
x <- c(a=1, b=2, c=3)
x[c("c", "a", "foo")]
```

```
c  a <NA>
3 1 NA
```

If several elements have the same name, only the first of them will be returned.

Specifying a non-existent name produces an `NA` in the result.
When subsetting a matrix, missing subscripts are treated as if all elements are named; so \( x[1,] \) corresponds to the first row and \( x[,3] \) to the third column.

For arrays, the treatment is similar, for example \( y[,1,] \).

These can also be used for assignment, \( x[1,] = 20 \).
Lists are useful as containers for grouping related things together (many R functions return lists as their values).

Because lists are a recursive structure it is useful to have two ways of extracting subsets.

The `[ ]` form of subsetting produces a sub-list of the list being subsetted.

The `[[ ]]` form of subsetting can be used to extract a single element from a list.
List Subsetting Examples

- **Using the `[]` operator to extract a sublist.**
  ```r
  sub <- lst[1]
  sub
  
  $a
  [1] 1 2 3
  
  typeof(sub)
  [1] "list"
  ```

- **Using the `[[ ]]` operator to extract a list element.**
  ```r
  sub <- lst[[1]]
  sub
  
  [1] 1 2 3
  
  typeof(sub)
  [1] "integer"
  ```

- **As with vectors, indexing using logical expressions and names is also possible.**
The dollar operator provides a short-hand way of accessing list elements by name. This operator is different from all other operators in R, it does not evaluate its second operand (the string).

\[ \text{lst}$a \]
\[ [1] \ 1 \ 2 \ 3 \]

\[ \text{lst}["a"] \]
\[ [1] \ 1 \ 2 \ 3 \]

For $ partial matching is used, for [[it is not by default, but can be turned on.
There are a number of ways of getting help:

- `help.start` and the HTML help button in the Windows GUI
- `help` and `?:` `help("data.frame")`
- `help.search`, `apropos`, `??`
- `RSiteSearch` (requires internet connection)
- Online manuals
- Mailing lists
- Vignettes
In R the primary mechanism for distributing software is via packages.

CRAN is the major repository for packages.

You can either download packages manually or use `install.packages` or `update.packages` to install and update packages.

In addition, on Windows and other GUIs, there are menu items that facilitate package downloading and updating.

It is important that you use the R package installation facilities. You cannot simply unpack the archive in some directory and expect it to work.
Bioconductor packages are hosted in CRAN-style repositories and are accessible using `install.packages`.

The most reliable way to install Bioconductor packages (and their dependencies) is to use `biocLite`.

Bioconductor has both a release branch and a development branch. Each Bioconductor release is compatible with its contemporary R release.

Bioconductor packages have vignettes.
R has a standard set of control flow functions:

- **Looping:** for, while and repeat.
  - `for (var in seq) expr`
  - `while (cond) expr`
  - `repeat expr`
  - `next`: halts the processing of the current iteration and advances the looping index.
  - `break`: breaks out of a for, while or repeat loop. The control is transferred to the first statement outside the inner-most loop.

- **Conditional evaluation:** if and switch.
  - `if(cond) expr`
  - `if(cond) cons.expr else alt.expr`
  - `switch(EXPR, ...):` If the value of `EXPR` is an integer between 1 and `nargs()-1` then the corresponding element of `...` is evaluated and the result returned.
- **cond**: A length-one logical vector that is not `NA`. Conditions of length greater than one are accepted with a warning, but only the first element is used. Other types are coerced to logical if possible, ignoring any class.

- **var**: A syntactical name for a variable.

- **seq**: An expression evaluating to a vector (including a list and an expression) or to a pairlist or `NULL`. A factor value will be coerced to a character vector.

- **expr, cons.expr, alt.expr**: An *expression* in a formal sense. This is either a simple expression or a so called *compound expression*, usually of the form `{expr1; expr2}`.
A natural programming construct in R is to apply the same function to elements of a list, of a vector, rows of a matrix, or elements of an environment.

The members of this family of functions are different with regard to the data structures they work on and how the answers are dealt with.

Some examples, `apply()`, `sapply()`, `lapply()`, `mapply()`, `eapply()`.
apply() applies a function over the margins of an array.

- Computes the mean of each row of matrix x

```r
> x <- matrix(1:100, nrow = 10)
> x
[1,]  1  11  21  31  41  51  61  71  81  91
[2,]  2  12  22  32  42  52  62  72  82  92
[3,]  3  13  23  33  43  53  63  73  83  93
[4,]  4  14  24  34  44  54  64  74  84  94
[5,]  5  15  25  35  45  55  65  75  85  95
[6,]  6  16  26  36  46  56  66  76  86  96
[7,]  7  17  27  37  47  57  67  77  87  97
[8,]  8  18  28  38  48  58  68  78  88  98
[9,]  9  19  29  39  49  59  69  79  89  99
[10,] 10  20  30  40  50  60  70  80  90 100

> apply(x, 1, mean)
[1] 46 47 48 49 50 51 52 53 54 55

- Computes the mean of each column of matrix x

```r
> apply(x, 2, mean)
[1] 5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
```
apply is usually much faster than a for loop and it is more elegant.

```r
>a = matrix(runif(1e6), ncol=10)
s1 = apply(a, 1, sum)

> sum(s1)
[1] 499380.8

s2 = numeric(nrow(a))
for(i in 1:nrow(a)){
  s2[i] = sum(a[i,])
}

> sum(s2)
[1] 499380.8
```
Writing R functions provides a means of adding new functionality to the language.

Functions that a user writes have the same status as those which are provided with R.

Reading the functions provided with the R system is a good way to learn how to write functions.
Here is a function that computes the fibonacci of its argument.

```r
> fib <- function(x) {
+  if (x == 1 || x == 0) {
+    return(x)
+  }
+  else {
+    return(fib(x - 1) + fib(x - 2))
+  }
+}
> fib(7)
[1] 13
```

Calculate the fibonacci for a vector of arguments.

```r
> fib(2:8)
[1]  1  3  5  7  9 11 13
```

Once a function is defined, it is possible to call it from other functions.

```r
> sumfib = function(x) sum(fib(x))
> sumfib(2:8)
[1] 49
```
- Any single R object can be returned as the value of a function; including a function.
- If you want to return more than one object, you should put them in a list (usually with names), or an S4 object, and return that.
- The value returned by a function is either the value of the last statement executed, or the value of an explicit call to `return()`.
- `return()` takes a single argument, and can be called from any where in a function.
**Useful String Functions**

- **Concatenate strings: `paste()`**
  ```
  > s <- c("apple", "banana", "lychee")
  > paste(s, "X", sep="_")
  [1] "apple_X"  "banana_X"  "lychee_X"
  > paste(s, collapse="", ")
  [1] "apple, banana, lychee"
  ```

- **Search strings: `grep()`**
  ```
  > library("ALL")
  > data(ALL)
  > class(ALL$mol.biol)
  [1] "factor"
  > ALL$mol.biol[1:16]
  [1] BCR/ABL NEG BCR/ABL ALL1/AF4 NEG NEG NEG NEG
  [9] NEG BCR/ABL BCR/ABL NEG E2A/PBX1 NEG BCR/ABL NEG
  Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG NUP-98 p15/p16
  > negIdx <- grep("NEG", ALL$mol.biol)
  > negIdx[1:10]
  [1]  2  5  6  7  8  9 12 14 16 21
  > ALL$mol.biol[negIdx]
  [1] NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG
  [20] NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG
  [39] NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG
  [58] NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG NEG
  Levels: ALL1/AF4 BCR/ABL E2A/PBX1 NEG NUP-98 p15/p16
A very common way to exchange data are so called *comma separated values (CSV)* files.

```r
> rownames(x)<-paste("R",1:nrow(x),sep="_")
> colnames(x)<-paste("C",letters[1:ncol(x)],sep="-")
> x[1:5,]
      C-a C-b C-c C-d C-e C-f C-g C-h C-i C-j
R_1  1  11  21  31  41  51  61  71  81  91
R_2  2  12  22  32  42  52  62  72  82  92
R_3  3  13  23  33  43  53  63  73  83  93
R_4  4  14  24  34  44  54  64  74  84  94
R_5  5  15  25  35  45  55  65  75  85  95
```

```r
> write.csv(x,"file.csv")
> xi<-read.csv("file.csv",row.names=1,header=TRUE, as.is=TRUE)
> xi[1:5,]
      C.a C.b C.c C.d C.e C.f C.g C.h C.i C.j
R_1  1  11  21  31  41  51  61  71  81  91
R_2  2  12  22  32  42  52  62  72  82  92
R_3  3  13  23  33  43  53  63  73  83  93
R_4  4  14  24  34  44  54  64  74  84  94
R_5  5  15  25  35  45  55  65  75  85  95
```

Sometimes the files are separated with a different character than comma (e.g. tab, semicolon, space,...).

- use `read.csv(..., sep=";")` to specify the separation character.
- See also `write.table` and `read.table`
Use `plot()` to visualize data

- Plotting the `square()` function for values from 1:10.

```r
x = 0:10
y = square(x)
plot(x = x, y = y, type = "l", col = "blue", xlab = "x_label", ylab = "y_label", main = "TITLE")
points(x, y, pch = 1, col = "red")
```
> library(RColorBrewer)
> data <- rnorm(1000,0,10)
> cols <- brewer.pal(5,"Pastel2")
> par(mfrow=c(2,2))
> hist(data,col=cols)
> boxplot(data, col="blue")
> pdf("plot.pdf")
> plot(x=x,y=y, type="l",col="blue",xlab="x",ylab="y", main="x^2")
> points(x,y,pch=1,col="red")
> dev.off()
> plot(x=x,y=log(x), type="l",col="red",xlab="x",ylab="y", main="log(x)")
> points(x,y=log(x),pch=2,col="green")