Introduction to R and Bioconductor

K. Zimmermann and P. Thomas

WBI

May 30, 2013

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] "" "" "" "" ""
```
```r
> v <- 1:5
> v == 3
[1] FALSE FALSE TRUE FALSE FALSE

> all(v > 2)
[1] FALSE

> any(v < 3)
[1] TRUE

> sum(v > 2.5)
[1] 3

> which(v > 2.5)
```
Matrices can be created using `matrix()`. They are represented as a vector with a dimension attribute.

**Example for `matrix()`**

```r
data <- matrix(1:10, nrow=2)
dim(data)

[1,] 1 3 5 7 9
[2,] 2 4 6 8 10

is.vector(data)

[1] FALSE

as.vector(data)

[1] 1 2 3 4 5 6 7 8 9 10
```
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 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
```
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 `[ , ]`. 

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

```r
tax <- matrix(c(4, 8, 5, 6, 4, 2, 1, 5, 7), nrow=3)
tax
[,1] [,2] [,3]
[1,] 4 6 1
[2,] 8 4 5
[3,] 5 2 7
tax[2,3]
[1] 5
tax[2,]
[1] 8 4 5
```
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
```
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
```
A subscript consisting of a vector of negative integer values is taken to indicate the indices which are not to be extracted.

> 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

Positive and negative subscripts cannot be mixed. But can be used in succession.

> x[-(1:3)][1:3]

   [1]  7  6  5
Assignments with Negative Indexes

- Negative subscripts can appear on the 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
```
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

> x[x > 5]

[1]  6  7  8  9 10
```
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
3
```

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

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.

All 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.
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().
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
```

```r
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 sequence of its argument.

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

```r
> 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
```
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,]

<table>
<thead>
<tr>
<th></th>
<th>C-a</th>
<th>C-b</th>
<th>C-c</th>
<th>C-d</th>
<th>C-e</th>
<th>C-f</th>
<th>C-g</th>
<th>C-h</th>
<th>C-i</th>
<th>C-j</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_1</td>
<td>1</td>
<td>11</td>
<td>21</td>
<td>31</td>
<td>41</td>
<td>51</td>
<td>61</td>
<td>71</td>
<td>81</td>
<td>91</td>
</tr>
<tr>
<td>R_2</td>
<td>2</td>
<td>12</td>
<td>22</td>
<td>32</td>
<td>42</td>
<td>52</td>
<td>62</td>
<td>72</td>
<td>82</td>
<td>92</td>
</tr>
<tr>
<td>R_3</td>
<td>3</td>
<td>13</td>
<td>23</td>
<td>33</td>
<td>43</td>
<td>53</td>
<td>63</td>
<td>73</td>
<td>83</td>
<td>93</td>
</tr>
<tr>
<td>R_4</td>
<td>4</td>
<td>14</td>
<td>24</td>
<td>34</td>
<td>44</td>
<td>54</td>
<td>64</td>
<td>74</td>
<td>84</td>
<td>94</td>
</tr>
<tr>
<td>R_5</td>
<td>5</td>
<td>15</td>
<td>25</td>
<td>35</td>
<td>45</td>
<td>55</td>
<td>65</td>
<td>75</td>
<td>85</td>
<td>95</td>
</tr>
</tbody>
</table>
```

```r
> write.csv(x, "file.csv")
> xi <- read.csv("file.csv", row.names=1, header=TRUE, as.is=TRUE)
> xi[1:5,]

<table>
<thead>
<tr>
<th></th>
<th>C.a</th>
<th>C.b</th>
<th>C.c</th>
<th>C.d</th>
<th>C.e</th>
<th>C.f</th>
<th>C.g</th>
<th>C.h</th>
<th>C.i</th>
<th>C.j</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_1</td>
<td>1</td>
<td>11</td>
<td>21</td>
<td>31</td>
<td>41</td>
<td>51</td>
<td>61</td>
<td>71</td>
<td>81</td>
<td>91</td>
</tr>
<tr>
<td>R_2</td>
<td>2</td>
<td>12</td>
<td>22</td>
<td>32</td>
<td>42</td>
<td>52</td>
<td>62</td>
<td>72</td>
<td>82</td>
<td>92</td>
</tr>
<tr>
<td>R_3</td>
<td>3</td>
<td>13</td>
<td>23</td>
<td>33</td>
<td>43</td>
<td>53</td>
<td>63</td>
<td>73</td>
<td>83</td>
<td>93</td>
</tr>
<tr>
<td>R_4</td>
<td>4</td>
<td>14</td>
<td>24</td>
<td>34</td>
<td>44</td>
<td>54</td>
<td>64</td>
<td>74</td>
<td>84</td>
<td>94</td>
</tr>
<tr>
<td>R_5</td>
<td>5</td>
<td>15</td>
<td>25</td>
<td>35</td>
<td>45</td>
<td>55</td>
<td>65</td>
<td>75</td>
<td>85</td>
<td>95</td>
</tr>
</tbody>
</table>
```

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="T I T L E")
> points(x, y, pch=1, col="red")
```

```
0  2  4  6  8  10
0  20  40  60  80  100
T I T L E
x_label
y_label
● ●
●
●
●
●
●
```

> 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")