R

Alejandra E. Medina Rivera
Licenciatura en Ciencias Genómicas.
Centro de Ciencias Genómicas, UNAM

Cuernavaca, Mexico
Feb, 2010

Introduction to R and Statistics

1 Revision

2 Lists and Data Frame

3 Read Files

4 Accessing data: Indexes

5 Data Manipulation

Searching for Help

- THE function for getting help is `help()`
  For example, lets say you don't know what the names function does, so you can get info about this using `help("names")` or some other short ways: `?names` and `?"names"`.
- For a deeper search you can use `help.search()`, this function looks inside the manuals for a word or words. Example, `help.search("names")`
- If you are looking for a function and you are not sure of the name use `apropos()`. Example, `apropos("names")`. Other usefull functions are:
  - `help.start()`,
  - `RSiteSearch()`,
  - `args()`

Searching for Help

  - `example()`.
- An other option is to be included on the help mail list, where you can ask more specific questions. They'll request information regarding your session. How can you obtain this information?

Data Types

R has a rich set of self-describing data structures.

```
> z <- "z"
> class(z)

[1] "character"
```

There is no need to declare the types of the variables.

Data Structures

The principal data structures in `R` are:

- `vector`- array of objects of the same type
- `matrix`- array of vectors
- `list`- can contain objects of different types
- `environment`- hashtable
- `data.frame`-array of vectors, lists or both.
- `factor`- categorical
- `fucntion`

Packages as `Bioconductor` provide other types of data structures.

Atomic Data Structures

In R, the basic data types are vectors, not scalars

A vector contains an index set of values that are all of the same type:

- *logical*
- *numeric*
- *complex*
- *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, eg. C or Fortran)

Variables - Vectors

- Remember R is a vector language, this means all variables are vectors.
- R es c() is a useful function. You can create a vector containing different types of variables.

```
> v1 <- c(1:10)
> v2 <- runif(10)
> v3 <- sample(c("A", "C", "G", "T"),
+      size = 10, replace = TRUE)
> v4 <- v3 %in% c("A", "G")
> v5 <- c("foo", 2, TRUE)
> v6 <- c(2, "3")
```

- mode function indicates the type of variable contained in the vector.

Variables - Vectors

- as function can change the type of the vector.
  Example: change the *numeric* mode of vectors v5 y v6:

```
> mode(v5)
> as.numeric(v5)
> as.numeric(v6)
> help(as.vector)
> help(as)
```

Vectors Recycling

- In `R` most fo the functions are vectorize. Example: `x = 2; y = 3; x + y` is in fact $x[i] + y[i], i \in 1, ...\mathtt{max}\{|x|, |y|\}$
- If the length of to vectors is not the same, `R` recycles the shortest till it reaches the length of the longest.

Example (Recycling)

```
c(2,3) + c(3,4,5)
```

```
> c(2, 3) + c(3, 4, 5)
[1] 5 7 7
```

and compare it with `c(2,3) + c(3,4,5,8)`

```
> c(2, 3) + c(3, 4, 5, 8)
[1]  5  7  7 11
```

Categories=Factors

- Data in statistics is usually classified.
- The variables that let us store categorized data are called *factores*.

```
> pain <- c(0, 3, 2, 2, 1)
> fpain <- factor(pain, levels = 0:3)
> levels(fpain) <- c("none", "mild",
+      "medium", "severe")
> fpain
```

```
[1] none   severe medium medium mild
Levels: none mild medium severe
```

What will happen with the factor if I use `as.numeric()` on it?

```
> levels(fpain)
```

Categories=Factors

```
[1] "none"   "mild"   "medium" "severe"
> as.numeric(fpain)
[1] 1 4 3 3 2
```

If you don't specify the `levels` in the `factor()` functions, the levels will be taken from the sorted unique values represented in the vector. Don't forget this cause you'll need this tip on the future.

Lists

- Data in statistics tends to be classified or subdivided.
- Lists are an easy way to combine different objects in one.
- Remember we can store categorized date in *factores*.

Numeric Reference

Elements on a list are always *enumerated*. If `Lst` is a list with four elements, one element is `Lst[[4]]` and if this element is a vector you can access the first element using: `Lst[[4]][1]`

Name Reference

Elements on a list can be access by name: `list$name`

Lists

```
> Lst <- list(name = "Fred", wife = "Mary",
+      no.children = 3, child.ages = c(4,
+          7, 9))
> Lst$name

[1] "Fred"

> Lst[[1]]

[1] "Fred"

> Lst$wife

[1] "Mary"

> Lst$child.ages[1]

[1] 4
```

Lists

```
> Lst[[4]][1]
[1] 4
> length(Lst)
[1] 4
```

Data Frame

Data Frames

In R *data frames* are very important objects. A `data.frame` is a table composed by one or more vectors and/or factors of the same length and different data types.

- `dataframe$variable` o `dataframe[["variable"]]`.
- Functions `attach` and `detach` can add variables from a `data frame` to the R environment.[1]; function `with(data.frame, command)` does the same.
- You can display the first and last elements of a `data.frame` or *array* using functions `head()` or `tail()`.

[1] Not recommendable if you are going to modify values of the `data frame` or if you have variables with the same names

Read Files

The principal function in R to read files is `read.table()`.

Read a `data.frame` from a file

The `read.table()` function reads a table from a file and stores it into a `data.frame` if :

- The first line is a header with the name of the variable in each column of the `data.frame`. If the header is not provided R automatically assigns variables `V1`,`V2...Vn` to each column.
- Each line has to have a unique ID, `row.name`
- If a `data.frame` is not necesary, you can change the format using function `as.format()` .

Read Files

Example

Read table

```
> arch <- "/Users/amedina/Documents/CCG/Cursos/Compu_
> heartatk <- read.table(file = arch,
+       header = TRUE)
```

- Other useful functions to read files are:
  - `scan()`
  - `read.table()`,
  - `read.csv()`
  - `source()`.
- to learn more: `help(read.csv)`
- `scan()` is useful when you don't know the structure of your data..

Read Files

- `source()` command used to read scripts `R` and execute them inside the current session.

Directories

- Sometimes you want to access several files from the same directory of folder
- Maybe you don't want to open all of them but some, so you have to look for a common pattern in the names.
- The automatic way to do it:

```
> setwd("/Users/amedina/Documents/CCG/Cursos/Compu_St
> files <- list.files(pattern = ".txt")
> for (i in files) {
+     x <- read.table(i, header = TRUE)
+ }
```

Read Files

The principal function in `R` to read files is `read.table()`.

Read a `data.frame` from a file

The `read.table()` function reads a table from a file and stores it into a `data.frame` if :

- The first line is a header with the name of the variable in each column of the `data.frame`. If the header is not provided `R` automatically assigns variables `V1,V2...Vn` to each column.
- Each line has to have a unique ID, `row.name`
- If a `data.frame` is not necesary, you can change the format using function `as.format()` .

Read Files

Example

Read table

```
> arch <- "/Users/amedina/Documents/CCG/Cursos/Compu_
> heartatk <- read.table(file = arch,
+       header = TRUE)
```

- Other useful functions to read files are:
    - `scan()`
    - `read.table()`,
    - `read.csv()`
    - `source()`.
- to learn more: `help(read.csv)`
- `scan()` is useful when you don't know the structure of your data..

Read Files

- `source()` command used to read scripts R and execute them inside the current session.

Directories

- Sometimes you want to access several files from the same directory of folder
- Maybe you don't want to open all of them but some, so you have to look for a common pattern in the names.
- The automatic way to do it:

```
> setwd("/Users/amedina/Documents/CCG/Cursos/Compu_Stat_II/2009_2/R_advan
> files <- list.files(pattern = ".txt")
> for (i in files) {
+     x <- read.table(i, header = TRUE)
+ }
```

Logical Operators

Logical Operators

In R there are several logical operators, in general these work as in any other language but there are small differences.

```
> x <- c(1:5)
> x < 5
> x > 1 & x < 5
> x > 1 && x < 5
> x > 1 || x < 5
> x == 3
> x != 3
> !x == 3
> x == c(2, 4)
```

Using Indexes

In R you can select an element in an array in different ways.

- As in perl or C with numerical indexes, beginning with [1].
- Segments of the array.
- Ignore one or several elements.
- Select a regular expression.

Example

Using Indexes

Access Vectors

```
> x <- c(2, 7, 9, 2, NA, 5)
> x[1:3]

[1] 2 7 9

> x[-1]

[1]  7  9  2 NA  5

> OddNum <- seq(1, 6, 2)
> x[OddNum]

[1]  2  9 NA

> x[seq(1, 6, 2)]

[1]  2  9 NA
```

Logic Index

Example

Access vectors with logical operators

```
> requireLogic <- c(TRUE, TRUE, FALSE,
+       FALSE, FALSE, FALSE)
> x[requireLogic]

[1] 2 7

> x[x < 5]

[1]  2  2 NA
```

Index Matrices

Matrices can be access similar to vectors.

```
> y <- matrix(c(2, 7, 9, 2, NA, 5),
+       nrow = 2)
> y

     [,1] [,2] [,3]
[1,]    2    9   NA
[2,]    7    2    5

> y[, c(1, 3)]

     [,1] [,2]
[1,]    2   NA
[2,]    7    5
```

Index NA

Select NA data.

Example

NA Data

```
> is.na(x)

[1] FALSE FALSE FALSE FALSE  TRUE FALSE

> x[is.na(x)] <- 0
> x

[1] 2 7 9 2 0 5
```

Example

Index NA

NA in Matrices

```
> is.na(y)

      [,1]  [,2]  [,3]
[1,] FALSE FALSE  TRUE
[2,] FALSE FALSE FALSE

> str(is.na(y))

 logi [1:2, 1:3] FALSE FALSE FALSE FALSE TRUE FALSE

> y[is.na(y)] <- -1
> y

     [,1] [,2] [,3]
[1,]    2    9   -1
[2,]    7    2    5
```

NA data

Na data

Not always NA data can be managed as Zero value, so is necessary to ignore them. In `R` the functions to mange NA data are:

- `na.fail()`,
- `na.omit()`,
- `na.exclude()`,
- `na.pas()` .

Access data.frames

As shown before you can access `lists` and `data.frames` using `arreglo$name`. However, is also possible to access them with numeric inexes.

Example

Acceso a `data.frame`

```
> names(heartatk)
[1] "Patient"    "DIAGNOSIS" "SEX"
[4] "DRG"        "DIED"      "CHARGES"
[7] "LOS"        "AGE"
```

```
> heartatk[2, ]
```

Access data.frames

```
  Patient DIAGNOSIS SEX DRG DIED CHARGES LOS
2       2     41041   F 122    0    3941   6
  AGE
2  34

> mode(heartatk[2, ])

[1] "list"

> heartatk[2, 3]

[1] F
Levels: F M

> heartatk[2, ][3]

  SEX
2   F
```

Access data.frames

```
> heartatk[2, "AGE"]

[1] 34

> heartatk$AGE[1:5]

[1] 79 34 76 80 55

> grep(pattern = "F", as.vector(heartatk$SEX[][1:6]))

[1] 1 2 3 4

> grp <- grep(pattern = "F", as.vector(heartatk$SEX[]
> heartatk[grp, ]
```

Access data.frames

```
  Patient DIAGNOSIS SEX DRG DIED CHARGES LOS
1       1     41041   F 122    0    4752  10
2       2     41041   F 122    0    3941   6
3       3     41091   F 122    0    3657   5
4       4     41081   F 122    0    1481   2
  AGE
1  79
2  34
3  76
4  80
```

Transform data

Manipulating and extracting data from a data frame can be logical but a bit cumbersome, this applies to the process of adding transformed variables to a data frame. For this tasks we can use the functions:

- `subset`,
- `transform`

```
> library(ISwR)
> data(thuesen)
> ls()
```

Transform data

```
 [1] "Lst"          "OddNum"
 [3] "arch"         "args"
 [5] "fname"        "fpain"
 [7] "grp"          "heartatk"
 [9] "pain"         "requireLogic"
[11] "thuesen"      "x"
[13] "y"            "z"

> head(thuesen)

  blood.glucose short.velocity
1          15.3           1.76
2          10.8           1.34
3           8.1           1.27
4          19.5           1.47
```

Transform data

```
5            7.2            1.27
6            5.3            1.49
```

Now we want to keep the data only of persons with a blood.glucose less than 7.

```
> thue2 <- subset(thuesen, blood.glucose <
+     7)
> thue2
```

```
   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
```

Transform data

In which other way would you solve this problem?

Transform data

```
> thuesen[(thuesen[, 1] < 7), ]
   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
```

Transform data

Now we want to add a column with the logarithm of the blood.glucose.

```
> thue3 <- transform(thuesen, log.gluc = log(blood.glucose))
```

Notice that the variables used in the expression for new variable or for subsetting are evaluated with variables taken from the data frame. Can you think in an other way of doing this?

Transform data

```
> log_blood <- log(thuesen$blood.glucose)
> thue3$log_blood2 <- log_blood
> head(thue3)

  blood.glucose short.velocity log.gluc
1          15.3           1.76 2.727853
2          10.8           1.34 2.379546
3           8.1           1.27 2.091864
4          19.5           1.47 2.970414
5           7.2           1.27 1.974081
6           5.3           1.49 1.667707
  log_blood2
1   2.727853
2   2.379546
3   2.091864
```

Transform data

```
4    2.970414
5    1.974081
6    1.667707
```

Grouped data and data frames

The natural way of sorting grouped data in a data frame is to have the data themselves in one column and in otherl to that to have a factor telling which data are from which group everithing

```
> data(energy)
> head(energy)

  expend stature
1   9.21   obese
2   7.53    lean
3   7.48    lean
4   8.08    lean
5   8.09    lean
6  10.15    lean

> class(energy)
```

Grouped data and data frames

```
[1] "data.frame"
> class(energy[, 1])
[1] "numeric"
> class(energy[, 2])
[1] "factor"
```

How could we separate from this data frame into two vectors the energy depending on the value of the factor?

```
> exp.lean <- energy$expend[energy$stature ==
+      "lean"]
> exp.obese <- energy$expend[energy$stature ==
+      "obese"]
```

Grouped data and data frames

How could you transform this data frame into a list with this two vectors?

```
> list <- split(energy$expend, energy$stature)
> list

$lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40
 [7] 10.88  6.13  7.90  7.05  7.48  7.58
[13]  8.11

$obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69
[8]  9.68  9.19
```

Sort data

This is trivial on a vector

```
> head(heartatk)

  Patient DIAGNOSIS SEX DRG DIED   CHARGES
1       1     41041   F 122    0      4752
2       2     41041   F 122    0      3941
3       3     41091   F 122    0      3657
4       4     41081   F 122    0      1481
5       5     41091   M 122    0      1681
6       6     41091   M 121    0 6378.6400
  LOS AGE
1  10  79
2   6  34
3   5  76
4   2  80
```

Sort data

```
5    1   55
6    9   84
```

Sort data

```
> age <- heartatk[, 8]
> head(age)
```

```
[1] 79 34 76 80 55 84
```

```
> head(sort(age))
```

```
[1] 20 21 23 23 24 24
```

But we usually don't want something as easy as this, we usually want to order a data frame based on one column or columns.

```
> dim(heartatk)
```

```
[1] 12844     8
```

```
> order_age <- order(heartatk[, 8])
> head(order_age)
```

Sort data

```
[1]  5411 10853  4126 10738  4247  5199

> head(heartatk[order_age, ])

      Patient DIAGNOSIS SEX DRG DIED
5411     5411     41041   M 122    0
10853   10853     41091   F 122    0
4126     4126     41041   M 122    0
10738   10738     41011   M 121    0
4247     4247     41091   F 122    0
5199     5199     41041   M 121    0
        CHARGES LOS AGE
5411       6214   4  20
10853 6726.2700   4  21
4126      10781   8  23
10738      <NA>   8  23
```

Sort data

```
4247      10672   6  24
5199       7596   8  24

> order_age_charge <- order(heartatk[,
+     8], heartatk[, 6])
> head(order_age_charge)

[1]  5411 10853  4126 10738  4247  8454

> head(heartatk[order_age_charge, ])
```

Sort data

```
      Patient DIAGNOSIS SEX DRG DIED
5411     5411     41041   M 122    0
10853   10853     41091   F 122    0
4126     4126     41041   M 122    0
10738   10738     41011   M 121    0
4247     4247     41091   F 122    0
8454     8454     41091   M 121    0
        CHARGES LOS AGE
5411       6214   4  20
10853 6726.2700   4  21
4126      10781   8  23
10738      <NA>   8  23
4247      10672   6  24
8454      14950  10  24
```

Sort data

The second variable will be used when the order can not be determined form the first variable