(A very short) introduction to R

Alejandro Cáceres[a]
acaceres@creal.cat

[a]Centre for Research in Environmental Epidemiology (CREAL)

May, 2016

Contents

Why R?

- It is free and all the code is open
- It is a high level programming language (object oriented)
- If has flexible syntax that allows compact coding
- It is easy to write software packages that use other packages (community)
- Initially designed as statistical software, now there are packages with the latest methods of statistical inference, graphics, data mining, parallel computing or deploying apps on the Internet.
- Almost everything can be done within R. If not, there are wrappers to call C, Fortran, PERL, etc... so everything can be controlled from R.

course

- There is a virtual machine for each of you with all the code for the course
- It is accessed with a web-browser in Biocould.com
- You can run the code while I explain it
- We'll be using RStudio which is a GUI for R

Installing R

- Go to one of the R mirrors
  http://cran.es.r-project.org/
- or from the R website
  http://www.r-project.org/ (Download CRAN packages -Spain)
- Choose your platform and then your binary
  (you can also download it and compile it)
- check packages link

The R environment

R is a command line program. You type expressions (`2+2`) and functions (`log(34)`)

You access it through a terminal or a GUI window (like RStudio).

- open the terminal (Linux-Mac Terminal or DOS)
- type R

```
R version 2.10.1 (2009-12-14)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

- try different commands: `> 2+2` `> c(1,2,3)+2`

R scripts

Commands can be written in a text (extension .R) file and then run in R by copy paste. You can use any text editor (Emacs or the one build in RStudio).

- open a text editor
- write a set of commands, and save it in a `test.R` file.

```
#assign value to a
a<-c(1,2,3,4,5)
#print value of a
a
#assign value to b
b<-2
#print  (show) sum of a and b
a+b
#list all variables defined in the session
ls()
```

- Note: any line followed by `#` is ignored (comment), `<-` is assignation (=), entering the name of the variable prints it. R does not print if you don't ask.
- Copy paste command lines or type ctrl+Enter in RStudio
- type: `> source(test.R)` to run all the script file

Always use a script the script is your work!

R plots

plots are also obtained from the command line.

- type:

```
a<-1:10
a
b<-a^2
b
plot(a,b)
plot(a,b,col="blue",pch="A")
```

- a new window is automatically open if working on a terminal. The plot is displayed in the corresponding RStudio window.
- recover your work, type: `source(samplePlot.R)`
- you can save it as pdf/png/jpeg

R help

- try Google for generic questions: i.e "substitute missing R" check the `R help archive` hits or any other forum
- if you know the function (say `is.na`) type:
  `?is.na`
- check the examples and copy paste them:
  `is.na(c(1, NA))`
- Manual: An Introduction to R:
  `http://cran.r-project.org/doc/manuals/R-intro.pdf`

Libraries

- look in Google: "read stata file in r", what is the command (function) to do that?
- try: `?read.dta`
- check the library in which it is
- type:

  ```
  library(foreign)
  ?read.dta
  ```
- download `ade4` library and run `dudi.acm` example

vectors

- R operates on data structures. The simplest is a vector of one component (element). All operations are functions like `oneFunction()`.
- try a basis function on an element: `is.vector(17)`
- vectors are of many classes, enquire the class with `class()`

```
class(17)
class("male")
class(TRUE)
class(NaN)
class(Inf)
```

- the function `c()` concatenates elements in a vector of the same class:

```
c(10, 5, NaN, 6, Inf)
class(c(10, 5, NaN, 6, Inf))
is.vector(c(10, 5, NaN, 6, Inf))

c(TRUE,FALSE,TRUE)
class(c(TRUE,FALSE,TRUE))
is.vector(c(TRUE,FALSE,TRUE))
```

vectors -functions

- A fundamental function is `assign()`

```
assign("a",c(1,2,3,4,5))
a
#another way of using assign is with <-
a<-1:5
a
#check the variables you have defined
ls()
```

- another useful function is `seq`

```
seq(1,10)
#or
1:10
#a function within function
assign("b",seq(1,10))
#or a more standard use
b<-1:10
```

vectors -functions

- select elements with `[ ]`

```
b<-seq(1,100,2)
b[5]
b[5]<-999
b

#select and remove many elements
b[c(1,3,4)]
b[1:10]

b[-c(1,3,4)]
b
```

- you can perform arithmetic

```
a<-rep(2,10)
a
b<-a+1
b
b*a
```

Matrix

- A two dimensional array of vectors is a matrix, organized by rows and columns. Matrices can be constructed from vectors by concatenating vectors (and assigned to a variable)
  - concatenating columns `cbind()`

    ```
    matR<-cbind(c(1,2,3,4),c(2,4,6,8))
    matR
    ```

  - concatenating rows `rbind()`

    ```
    matC<-rbind(c(1,2,3,4),c(2,4,6,8))
    matC
    ```

- Formally, matrices as constructed by listing the data and then giving it the appropriate format
  - using `matrix()`

  ```
  mat<-matrix(c(1,2,3,4,5,6,7,8),ncol=2,nrow=4)
  mat
  mat<-matrix(c(1,2,3,4,5,6,7,8),ncol=4,nrow=2)
  mat
  ```

Matrix-functions

Do it yourself:

- Suppose we have a large matrix (read from a file) and want to find out its dimensions, which function do we need to use?
- What are `colnames()` and `rownames()` for?
- Name the rows of `mat<-matrix(1:10,5,2)` with alphabetical letters

other useful functions are `NCOL` and `NROW`

Matrix-functions

Specific elements or subsets of the matrix can be selected with `[,]`

- use `rnorm` to simulate a data set

  ```
  x <- matrix(rnorm(100, 1), ncol = 5)
  colnames(x)<-c("a","b","c","d","e")
  x
  #explore the matrix...
  head(x)
  tail(x)
  dim(x)
  image(x)
  #select one element
  x[1,2]
  ```
- DIY:
  - select columns b,c,d
  - select even rows.
  - from the second rows select those elements that are >0. Replace them by 0.

Matrix-functions

operations can be done on the whole matrix or to subsets of it

- you can add and multiply by a number, compute any function on its elements, perform matrix algebra, etc...
- You can perform complex data manipulation in a single line of code: take the 3rd column of x, find its maximum and replace all the values of that 3rd column by the maximum.
- Some useful functions are `colSums()`, `rowSums()`, `t()` and `diag()`
- What does the following the command do?

```
(x[-1,1:3])[1,1]<-0
```

data.frame

Data frames are special type of matrices that allow any class of vector in its columns

```
mat<-cbind(c("a","b","c"),c(1,2,3))
colnames(mat)<-c("letters","numbers")
mat
mat[,2]
#numbers get coerced into characters
class(mat[,2])
class(mat)
```

There is a sample data set in R: `airquality`

```
airquality
dim(airquality)
head(airquality)
class(airquality)
newCol<-c(rep("yes",130),rep("maybe",10),rep("no",13))
newAir<-cbind(airquality,newCol)
class(newAir[,3])
class(newAir[,7])
#in a data.frame characters are coerced into factors
newAir[,7]
```

data.frame

So how do I create my own `data.frame`?

- to convert a class into another do

```
matFrame<-as.data.frame(mat)
matFrame
#but check thesecond column
matFrame[,2]
# as is not numeric we cannot do
mean(matFrame[,2])
```

- how can we solve this? so if I tell you there is a function `as.numeric()` to change from factor to numeric.
- It is better to define our data.frame well from the beginning (try to do this always).

```
v1<-c("a","b","c")
v2<-c(1,2,3)
matFrame<-data.frame(letters=v1,numbers=v2)
mean(matFrame[,2])
class(matFrame[,2])
class(matFrame[,1])
```

data.frame -functions

- You can select subsets in various ways. `data.frame` is a list of variables (columns). You can select the variables you want like

  ```
  names(airquality)
  airquality$Ozone
  airquality$Day
  airquality[c("Ozone","Day")]
  ```

- But `data.frame` is also like a matrix so you can also use `[]` to select rows and columns

  ```
  airquality[1:10,c(2,5)]
  ```

- select data for only August and November (*Month* $>=$ 8)

  ```
  sel<-airquality$Month>=8
  #whatever matches TRUE rows-wise will be selected
  #cbind(sel, airquality)
  AugustNovAir<-airquality[sel,]
  ```

factor

- factors are vectors that encode categorical variables

```
newCol<-c(rep("yes",130),rep("maybe",10),rep("no",13))
facCol<-as.factor(newCol)
```

- the levels are retrieved with

```
levels(facCol)
#the levels can be renamed
levels(facCol)<-c("a","b","c")
facCol
```

- useful functions for factors are table

```
table(airquality$Month,facCol)
```

- you can get the means for within each factor

```
tapply(airquality$Temp,facCol,mean)
```

List

Lists are collection of variables of different classes, actually `data.frame` is a list

- create a list of with different objects

```
mat<-cbind(c(2,4,6),c(1,2,3))
vec<-c(TRUE,FALSE)
chr<-"hello"
L<-list(M=mat,V=vec,C=chr)
L
```

- you can select the elements of a list with $

```
names(L)
L$M
L$C
```

- or with double brackets [[]]

```
L[[1]]
L[[2]]
```

- change the names of the list elements

```
names(L)<-c("data","output","message")
L
```

Summary -so far

- R objects
  - vectors
  - matrices
  - data.frames
  - factors
  - lists
- R functions
  - assign:
    $<-$
  - select:
    [], [[]],$
  - convert into another class:
    `as.data.frame(), as.factor(), as.numeric(), as.character(), as.matrix(), as.vector()`
  - create objects:
    `c(), cbind(), rbind(), 1:10`
  - explore objects:
    `dim(), head(), tail(), colnames(), rownames(), names()`
  - compute on objects:
    `mean(), table(), *, /, plot(), etc...`

Importing/exporting data

Generally our data is in a file (txt, STATA, EXCEL, etc...)

- read a txt file with `read.table()`

```
dat<-read.table("dat1.txt")
dat[1:5,]
names(dat)
class(dat)
#change options for the read
?read.table
dat<-read.table("dat1.txt",header=TRUE)
dat
names(dat)

dat<-read.table("dat2.txt",header=TRUE,sep=",")
dat
names(dat)
```

- write data back to a txt file

```
write.table(dat,file="myData.txt")
?write.table
write.table(dat,file="myData.txt",quote=FALSE)
```

Importing/exporting data

- Read stata data with the `foreign` library

```
data(swiss)
swiss
write.dta(swiss,file="stataDat.dta")
stataD<-read.dta("stataDat.dta")
stataD
```

- say you have some variables on R that you would like to keep for a future session

```
ls()
save(swiss,dat,file="myResults.RData")
#remove all the variables in your environment
rm(list=ls())
ls()
#load your previous results
load("myResults.RData")
ls()
dat
```

Exercise

- there is a number of functions that can be useful
- find out what `sort()` does
- find out what `order()` does
- find out what == does
- order swiss data respect to increasing levels of education, then remove from the data set those rows with education 2, save your results as text file. Finally check that what you saved is ok.

Functions
Sintax

```
myFunction<-function(x, y, z=10, ...)
{
  code
  otput
}
```

- variables are of any data type (object) and are local to the function
- a variable that is not defined as an argument is taken as global by default
- the output can be any data type (object)
- the function call is flexible

```
a<-1
b<-2

myFunction(a,b)
myFunction(1,2)
myFunction(y=b,x=a,60)
```

Exercise

Write a function that computes $sin(x * y + z)$ where z is 0 by default

Exercise

Write a function that computes $sin(x * y + z)$ where z is 0 by default

```
mysin<-function(x,y,z=0)
{
  out<-sin(x*y+z)
  out
}
```

Loops: lapply

Loops is R are a bit special there is `for` and `while` but `lapply` is more efficient as it outputs the result of a function

```
for(x in 1:10)
{
  mysin(x,y=1)
}
```

To save the result of this for loop, we would need to initialize a variable (list) and store the result in an indexed field.

lapply sintax:

```
out<-lapply(1:10, function(x)
{
  sin(x+10)
})

#shorter

out<-lapply(1:10, mysin, y=1, z=10)
#note that the first argument of mysin is the one that varies
#from 1:10 and is implicit
```

Parallel computing

Parallel computing is done with the library parallel and the function `mclapply` with identical sintax as `lapply`

```
library(parallel)

int<-seq(-pi/2, pi/2, length=10000)
out<-mclapply(int, mysin, y=1, mc.cores=3)

#it is three times faster than
out<-lapply(int, mysin, y=1)
```