II.
R Programming

Kazumi Wada
National Statistics Center, JAPAN

Contents

1 Programming with R

1. 1 Basic operators

The basic arithmetic operators are + for summation, - for subtraction, * for multiplication, / for division and ^ for raising to a power. Comparative operators and logical operators shown in the following tables are used in such as conditional execution.

1. 2 Calculation of vectors and matrices

Comparative operator	Meaning
==	Equal
!=	Inequal
>=	Equal or less (≥)
>	less than
<=	Equal or greater (≤)
<	Greater than

Logical operator	Meaning
!	NOT
&&	AND (conditional execution)
||	OR (conditional execution)
&	AND (logical)
|	OR (logical)

```
A1 <- matrix(1:6, nc=2)
A2 <- matrix(1:6, nc=3)

A1 %*% A2                    #  inner product
A2 %*% A1

t(A1)                        #  transposition

A1 %o% A2                    #  outer product

A3 <- A1 %*% A2
diag(A3)                     # create diagonal vector
solve(A3)                    #  inverse matrix
svd(A3)                      #  singular value decomposition
qr(A3)                       #  QR decomposition
chol(A3)                     #  Choleski factorization
eigen(A3)                    # eigen vector and values
det(A3)                      #  determinant
```

1. 3 Basic statements

(1) Repetitive execution: *for* loops, *while* and *repeat*

- *For* (frequency){statements}: Function *for* repeats the statements in the braces "{}" with the specified number of times in the parenthesis "()". The braces can be omitted if the function repeats one statement.

```
# Case 1
    for (i in 1:5) print("Hello")

# Case 2
    nm1 <-c("Mio", "Ryo", "John", "Yuki", "Taro")
    for (i in 1:5) {
        print(paste("Hello, ", nm1[i], "!", sep="") )
    }
```

- *While* (condition){statements}: Function *while* repeats the statements in the braces until the condition in the parenthesis is TRUE.

```
i<-5
while (i> 0){
   print("Hello!")
   i<-i-1
}
```

- *repeat*{statements}: This function repeats statements in the braces.

```
i<-5
repeat{
   print("Hello!")
   i<-i-1
   if (i < 0) break
}
```

(2) Conditional execution: if function

- if (condition) {statements A} else {statements B}: If the condition(s) in the parenthesis is TRUE, statements A are executed. And if it is FALSE, statements B are executed.

```
a <- rnorm(1)   # a random number from the standard normal distribution
if (a > 0) {
    print("a > 0")
} else {
    print ("a <= 0")
}
a
```

- ifelse(condition, statements A, statements B)

```
# Same results with the previous example
result <-ifelse(a > 0, print("a > 0"), print("a <= 0"))
```

1. 4 Workspace

All objects created and manipulated by R are stored in the workspace. The workspace is created when R starts and erased when R terminates.

```
A1 <- A2 <- 0
ls()                          # everything on the workspace
objects()                     # the same result as ls()
rm(list=ls())                 # delete everything
B1 <- 1;    B2 <- 2;    B3 <- 3
rm(B1, B3)                    # delete B1 and B3 from the workspace
```

1. 5 Files and directories manipulation

```
# show current directory
getwd()
#list files in the directory
list.files()
#change directory
setwd("mydirectory")
#make directory
dir.create("mydirectory")
#delete file
file.remove("myfile")
```

1. 6 Missing values, etc.

When reading files into R, blank cells are automatically interpreted as missing values and "NA"s are inserted in the data frame. Any operation about NA results in NA, therefore comparative operator "==" cannot be used to detect NAs. Instead, there is a special function, *is.na* to detect NA data.

There are similar data like *NAN*, *Null,* and *Inf* in addition to NA. Those special data often become a cause of error.

■ NA: Not Avaialbe

```
x1 <-c(1,,5,2,4)     #-> error
x1                   #-> error
x1 <-c(1,NA,5,2,4)
x1                   #-> 「[1] 1 NA 5 2 4」

x1 == 5              #-> 「[1] FALSE NA TRUE FALSE FALSE」

which(x1 == 5)       # -> 「[1] 3」

which(x1 == NA)    # -> 「integer(0)」 NA cannot be detected.

x1 == NA             #-> 「[1] NA NA NA NA NA」

sum(x1)              # 「[1] NA」
```

■ Manipulation of NAs

```
x1[!is.na(x1)]          # remove NA                  #->[1] 1 5 2 4
sum(x1[!is.na(x1)])   # sum after removing NA      #->[1] 12
x2 <-ifelse(is.na(x1), 0, x1)        # replace NA as 0
sum(x2)                #->[1] 12
x2                     #->[1] 1 0 5 2 4
```

■ NAN, Null, and Inf

- NaN: "Not a number." It is produced by numerical computation, such as "0/0" or "Inf - Inf." Function *is.na* returns TRUE both for *NA* and *NaN* values.
- Null: Null means the variable is empty. The null variables cannot have a structure such as vector and matrix.
- Inf, -Inf: Plus and minus infinity. It can be obtained by dividing 5 by 0, for an example.

Check function	Object
is.nan()	NaN
is.null()	NULL
is.infinite(), is.finite()	Inf

1. 7 GUI preferences

The default settings of the R GUI such as font, size, console colors and so on from the menu bar by selecting "Edit," and the "GUI preferences." Do not forget to save the changed settings. Otherwise the new settings will be lost when you terminate the R session.

1. 8 Other points for programming

R code can vary for a transaction because of the flexibility of the language. It enables us to develop code along our line of thinking; however, it also makes us difficult to understand code developed by others, or even by myself after time passes.

We can write shorter code with experience. And shorter code is easy to maintain, in general. But if we pursuit shorter code too much, it results in very difficult code to understand.

To cope with those problems described above, it is worth adding appropriate comments and paying attention to readability of the code. Sharing information and standardize among colleagues about routine tasks will also help.

2 Import and export data

2. 1 Workspace in a binary file

Function *ls* shows all objects currently on your workspace. Those objects will be disappeared when you terminate the R session without saving them. There are three different ways to save them as a binary file as follows. The extension of the binary file is ".rdata."

a) From the menu bar of the console, select "File," "Save Workspace" and

then type the output file name.

b) When terminating R, answer "yes" to the dialog box "Save your workspace? " and then type the output file name.

c) Save the image using function *save.image*. An example is shown below.

```
ls()                          # all objects in the workspace
setwd("d:/Rdemo/")            # set current directory where the file is saved
save.image("Test1.rdata")     # save the workspace as a specified file name
```

Function *ls* shows all objects currently on your workspace. Those objects will be disappeared when you terminate the R session without saving them. There are three different ways to save the workspace as follows.

a) From the menu bar of the console, select "File," "Load Workspace" and then specify the binary file to load.

b) Click on the binary file to load, and then a new R session, in which the binary file is loaded, will be opened.

c) Save the image using function *save.image*. An example is shown below.

```
setwd("d:/Rdemo/")      # set current directory where the file is saved
load("Test1.rdata")     # save the workspace as a specified file name
```

2. 2 Data objects in a binary file

It is not recommended to save everything in the workspace as shown in 2.1 since the workspace may contain unnecessary objects and therefore, the file size could be very large.

Instead of saving everything in a file, we can specify objects to save as follows.

```
setwd("d:/Rdemo/")          # change the current directory
data(iris)                  # load the iris dataset
save(iris, file="iris.rdata")   # Save the data set as a binary file
data(cars)                  # load the cars dataset
save(iris, cars, file="iris_cars.rdata")      # save them together in a binary file
# ---------------------------------------
# Terminate and restart R, and then load iris_cars.rdata
setwd("d:/Rdemo/")          # set the directory where the file was saved
load(iris_cars.rdata)       # load the binary file
ls()                        # confirm there are cars and iris
```

2. 3 Command history

All the commands we entered during the session is also kept, but it will disappear when we terminate the session. Following is an example to save and load the history as a text file. The default file extension is ".Rhistory".

```
history()                   # display last 25 commands
history(max.show=Inf)       # display all previous commands

setwd("d:/Rdemo/")
savehistory(file="Myhist")    # saved as Myhist.Rhistory

# close the session and start another one
setwd("d:/Rdemo/")          # set the location of the history file
loadhistory(file="Myhist")
```

2. 4 Input data from R console

■ Making cars dataset [50 observation, 2 variables]

- They are numeric variables of speed (mph) and stopping distance (ft).
- Give values as a vector to each variable *speed* and *dist*.
- Then combine it as *dat1* as a matrix.

```
# speed
speed <- c(4, 4, 7, 7, 8, 9, 10, 10, 10, 11, 11, 12, 12, 12, 12, 13, 13, 13, 13,
14, 14, 14, 14, 15, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 18, 19, 19, 19, 20,
20, 20, 20, 20, 22, 23, 24, 24, 24, 24, 25)
```

```
# distance
dist <- c(2, 10, 4, 22, 16, 10, 18, 26, 34, 17, 28, 14, 20, 24, 28, 26, 34, 34, 46,
26, 36, 60, 80, 20, 26, 54, 32, 40, 32, 40, 50, 42, 56, 76, 84, 36, 46, 68, 32,
48, 52, 56, 64, 66, 54, 70, 92, 93, 120, 85)

# make a matrix by combining these vectors
dat1 <- cbind(speed,dist)
```

2. 5 Data editor

① Making an empty data frame named *dat2*.

② From the menu bar, select "edit" and then "Data Editor." Then the data editor starts, and we can input necessary data in it.

③ Instead of selecting from menu bar, function *fix* can also be used.

④ Function edit can also be used.

```
dat2 = data.frame()        # making an empty data frame "dat2"
# From menu, "Edit" => "Data editor"
# The following command has a same feature
fix(dat2)
```

☆ Elements in a matrix have to be the same data type. A data frame can store different types of data by column.

2. 6 Fixed width text files

Function read.fwf is for reading a fixed width text data without delimiter. Please make a data file using Memopad looks like the box in the right side.

There are three records which have 3 variables with 5 characters for each. There is no header. Please save it as FWT01.txt.

```
  123  abc   def
  456   gh     i
    7    j   klm
```

```
wt1 <- rep(5, 3)              # 3 variables with 5 characters for each
cn1 <-c("X", "Y", "Z")        # names of variables
FWT01 <-read.fwf("FWT01.txt", widths=wt1, col.names=cn1)
head(FWT01)
tail(FWT01)          # Sometimes there are blank data in the tail of the text file
```

2. 7 CSV files

A csv file is a comma delimited text file. Following is an example to save iris dataset as a csv file using function *write.csv* and read it.

```
# export a data frame as a csv file
setwd("d:/Rdemo/")            # set the location to save the target file
data(iris)                     # load iris dataset
write.csv(iris, file="iris1.csv")      # the data frame is saved as iris.csv
write.csv(iris, file="iris2.csv", quote=FALSE, row.names=FALSE)
       # without row label, no quotation for character data

# import the csv file
dt1 <- read.csv("iris1.csv")
dt2 <- read.csv("iris2.csv")
dim(iris);      dim(dt1);     dim(dt2)
# [1] 150   5
# [1] 150   6    # the first variable is the row label
# [1] 150   5
```

2. 8 TSV files

A tsv file is a tab delimited text file. Following is an example to save iris dataset as a tsv file using function *write.table,* and read it.

```
setwd("d:/Rdemo/")            # set the location to save the target file
data(iris)                     # load iris dataset
write.table(iris, file="iris1.tsv", sep="¥t")   # the data frame is saved as iris.tsv
write.table(iris, file="iris2.tsv", sep="¥t", row.names=FALSE)
                                                # without row label

dt3 <- read.table("iris1.tsv", sep="¥t")
dt4 <- read.table("iris2.tsv", sep="¥t")
```

2. 9 Further learning

- The R Manuals: R Data Import/Export
  https://cran.r-project.org/doc/manuals/r-release/R-data.html

- "This R Data Import Tutorial Is Everything You Need" by DataCamp
  https://www.datacamp.com/community/tutorials/r-data-import-tutorial

3 Write R functions

We already used many functions such as *sum*, *read.csv*, *write.table*, and so on. They are built-in functions. Functions can also be created by ourselves.

3. 1 Basic syntax

```
FunctionName <- function(Opt1, Opt2, …){
      Statement1
      Statement2…
      return(a.List)
}
```

3. 2 Examples

- Case 1: Simple additive function

Please make a following function in your workplace, and then type "ls()". You will see the created function is also treated as an object in your workspace. Typing the function name and press return key display the source code.

```
A.plus.B <- function(A, B){
      C <- A + B
      return(C)
}

ls()
A.plus.B

#1 Use the function
```

```
A.plus.B(10, 25)

#2 Use the function
d1 <- 1:10
d2 <- 11:20
d3 <- A.plus.B(d1, d2)
```

- Case 2: Plural results

```
#1 Make a function
GetPluralResults <- function(A, B){
      C <- A + B
      D <- A * B
      E <- A^B
      return(C, D, E)
}


#1 Use the function
GetPluralResults (10, 25)
     # Plural results cannot be returned!

#2 Improve the function
GetPluralResults <- function(A, B){
      C <- A + B
      D <- A * B
      E <- A^B
      F <- list(C, D, E)
      return(F)
}

#2 Use the function
GetPluralResults (10, 25)      # It works!

#3 Use the function
d1 <- 1:10
d2 <- 11:20
d3 <- GetPluralResults (d1, d2)
```

3. 3 Save and load functions

① Make a following text file using Memopad or any text editor, and then save the file as "Fun1.r".

```
# Function 1: A.plus.B

A.plus.B <- function(A, B){
      C <- A + B
      return(C)
}


# Function 2: GetPluralResults
GetPluralResults <- function(A, B){
      C <- A + B
      D <- A * B
      E <- A^B
      F <- list(C, D, E)
      return(F)
}
```

② Start another R session and load the file using function *source* as follows.

```
# This is a new session and no objects in the workspace
setwd("d:/Rdemo/")            # set the location of the function file
ls()     # there is no object

source("Fun1.r")

ls()     # functions are loaded as objects
```

③ Another way to use those functions are to copy the function code and paste on the console before use.

4 For further learning

4. 1 How to get a help

- Function *help*

```
help(Name.of.function, package = package.name)
?summary        # ? plus function name also get the same help
#In the latter case we need to load necessary package(s)
```

- Find related functions and packages

```
help.search("obs")
```

- Information about a package

```
library( help = MASS )
```

- Keyword search

```
apropos("norm")        # keyword search using object name
help.search("LATEX")   # keyword search from help file
```

4. 2 Search engine

Finding necessary information of R is often difficult since the name of R consists of a character. There is a special search engine to cope with the problem.

- rseek.org : rstats search engine http://seekr.jp/

4. 3 Useful official documentation on R

- Manuals http://cran.r-project.org/manuals.html
  - An Introduction to R
  - R Data Import/Export
  - R Installation and Administration

- Writing R Extensions
- The R language definition (draft)
- R Internals
- The R Reference Index

- Non-English documents in CRAN
  https://cran.r-project.org/other-docs.html
- Books and other publications related to R.
  http://www.r-project.org/doc/bib/R-publications.html
  http://www.r-project.org/other-docs.html
- FAQ http://cran.r-project.org/doc/FAQ/R-FAQ.html