R Programming Basics

Here, we described the basics you should know about **R programming**, including :

- Performing basic arithmetic operations and using basic arithmetic functions
- Creating and subsetting basic data types in R

Basic arithmetic operations

R can be used as a calculator.

The basic arithmetic operators are:

1. **+** (addition)
2. **-** (subtraction)
3. ***** (multiplication)
4. ***/*** (division)
5. and **^** (exponentiation).

Type directly the command below in the console:

```
# Addition
3 + 7
[1] 10
# Substraction
7 - 3
[1] 4
# Multiplication
3 * 7
[1] 21
# Divison
7/3
[1] 2.333333
# Exponentiation
2^3
[1] 8
# Modulo: returns the remainder of the division of 8/3
8 %% 3
[1] 2
```

Note that, in R, '#' is used for adding comments to explain what the R code is about.

Basic arithmetic functions

1. **Logarithms and Exponentials**:

```
log2(x) # logarithms base 2 of x
log10(x) # logaritms base 10 of x
exp(x) # Exponential of x
```

3. **Trigonometric functions**:

```
cos(x) # Cosine of x
sin(x) # Sine of x
tan(x) #Tangent of x
acos(x) # arc-cosine of x
asin(x) # arc-sine of x
atan(x) #arc-tangent of x
```

4. **Other mathematical functions**

```
abs(x) # absolute value of x
```

```
sqrt(x) # square root of x
```

Assigning values to variables

A variable can be used to store a value.

For example, the R code below will store the price of a lemon in a variable, say “lemon_price”:

```
# Price of a lemon = 2 euros
lemon_price <- 2
# or use this
lemon_price = 2
```

Note that, it’s possible to use **<-** or **=** for variable assignments.

Note that, R is case-sensitive. This means that *lemon_price* is different from *Lemon_Price*.

To print the value of the created object, just type its name:

```
lemon_price
[1] 2
```

or use the function **print()**:

```
print(lemon_price)
[1] 2
```

R saves the object *lemon_price* (also known as a variable) in memory. It’s possible to make some operations with it.

```
# Multiply lemon price by 5
5 * lemon_price
[1] 10
```

You can change the value of the object:

```
# Change the value
lemon_price <- 5
# Print again
lemon_price
[1] 5
```

The following R code creates two variables holding the width and the height of a rectangle. These two variables will be used to compute of the rectangle.

```
# Rectangle height
height <- 10
# rectangle width
width <- 5
# compute rectangle area
area <- height*width
print(area)
[1] 50
```

The function **ls()** can be used to see the list of objects we have created:

```
ls()
[1] "area"        "height"      "info"        "lemon_price" "PACKAGES"
"R_VERSION"
[7] "width"
```

The collection of objects currently stored is called the **workspace**.

Note that, each variable takes some place in the computer memory. If you work on a big project, it's good to clean up your workspace.

To remove a variable, use the function **rm**():

```
# Remove height and width variable
rm(height, width)
# Display the remaining variables
ls()
[1] "area"        "info"        "lemon_price" "PACKAGES"    "R_VERSION"
```

Basic data types

Basic data types are **numeric**, **character** and **logical**.

```
# Numeric object: How old are you?
my_age <- 28
# Character  object: What's your name?
my_name <- "Nicolas"
# logical object: Are you a data scientist?
# (yes/no) <=> (TRUE/FALSE)
is_datascientist <- TRUE
```

Note that, character vector can be created using double (") or single (') quotes. If your text contains quotes, you should escape them using"\" as follow.

```
'My friend\'s name is "Jerome"'
[1] "My friend's name is \"Jerome\""
# or use this
"My friend's name is \"Jerome\""
[1] "My friend's name is \"Jerome\""
```

It's possible to use the function **class**() to see what type a variable is:

```
class(my_age)
[1] "numeric"
class(my_name)
[1] "character"
```

You can also use the functions **is.numeric**(), **is.character**(), **is.logical**() to check whether a variable is numeric, character or logical, respectively. For instance:

```
is.numeric(my_age)
[1] TRUE
is.numeric(my_name)
[1] FALSE
```

If you want to change the type of a variable to another one, use the **as.*** functions, including: **as.numeric**(), **as.character**(), **as.logical**(), etc.

```
my_age
[1] 28
# Convert my_age to a character variable
as.character(my_age)
[1] "28"
```

Note that, the conversion of a character to a numeric will output NA (for not available). R doesn't know how to convert a numeric variable to a character variable.

Vectors

A vector is a combination of multiple values (numeric, character or logical) in the same object. In this case, you can have **numeric vectors**, **character vectors** or **logical vectors**.

Create a vector

A vector is created using the function **c()** (for *concatenate*), as follow:

```
# Store your friends'age in a numeric vector
friend_ages <- c(27, 25, 29, 26) # Create
friend_ages # Print
[1] 27 25 29 26
# Store your friend names in a character vector
my_friends <- c("Nicolas", "Thierry", "Bernard", "Jerome")
my_friends
[1] "Nicolas" "Thierry" "Bernard" "Jerome"
# Store your friends marital status in a logical vector
# Are they married? (yes/no <=> TRUE/FALSE)
are_married <- c(TRUE, FALSE, TRUE, TRUE)
are_married
[1]  TRUE FALSE  TRUE  TRUE
```

It's possible to give a name to the elements of a vector using the function **names()**.

```
# Vector without element names
friend_ages
[1] 27 25 29 26
# Vector with element names
names(friend_ages) <- c("Nicolas", "Thierry", "Bernard", "Jerome")
friend_ages
Nicolas Thierry Bernard  Jerome 
     27      25      29      26 
# You can also create a named vector as follow
friend_ages <- c(Nicolas = 27, Thierry = 25, 
                 Bernard = 29, Jerome = 26)
friend_ages
Nicolas Thierry Bernard  Jerome 
     27      25      29      26 
```

Note that a vector can only hold elements of the same type. For example, you cannot have a vector that contains both characters and numeric values.

- **Find the length of a vector** (i.e., the number of elements in a vector)

```
# Number of friends
length(my_friends)
[1] 4
```

Case of missing values

I know that some of my friends (Nicolas and Thierry) have 2 child. But this information is not available (NA) for the remaining friends (Bernard and Jerome).

In R **missing values** (or missing information) are represented by NA:

```
have_child <- c(Nicolas = "yes", Thierry = "yes", 
                Bernard = NA, Jerome = NA)
have_child
Nicolas Thierry Bernard  Jerome 
  "yes"   "yes"      NA      NA 
```

It’s possible to use the function **is.na**() to check whether a data contains missing value. The result of the function **is.na**() is a logical vector in which, the value TRUE specifies that the corresponding element in x is NA.

```
# Check if have_child contains missing values
is.na(have_child)
Nicolas Thierry Bernard  Jerome 
  FALSE   FALSE    TRUE    TRUE 
```

Note that, there is a second type of **missing values** named **NaN** (“Not a Number”). This is produced in a situation where mathematical function won’t work properly, for example 0/0 = NaN.

Note also that, the function **is.na**() is TRUE for both NA and NaN values. To differentiate these, the function **is.nan**() is only TRUE for NaNs.

Get a subset of a vector

Subsetting a vector consists of selecting a part of your vector.

- **Selection by positive indexing**: select an element of a vector by its position (index) in square brackets

```
# Select my friend number 2
my_friends[2]
[1] "Thierry"
# Select my friends number 2 and 4
my_friends[c(2, 4)]
[1] "Thierry" "Jerome"
# Select my friends number 1 to 3
my_friends[1:3]
[1] "Nicolas" "Thierry" "Bernard"
```

Note that, **R indexes from 1**, NOT 0. So your first column is at [1] and not [0].

If you have a named vector, it's also possible to use the name for selecting an element:

```
friend_ages["Bernard"]
Bernard
     29
```

- **Selection by negative indexing**: Exclude an element

```
# Exclude my friend number 2
my_friends[-2]
[1] "Nicolas" "Bernard" "Jerome"
# Exclude my friends number 2 and 4
my_friends[-c(2, 4)]
[1] "Nicolas" "Bernard"
# Exclude my friends number 1 to 3
my_friends[-(1:3)]
[1] "Jerome"
```

- **Selection by logical vector**: Only, the elements for which the corresponding value in the selecting vector is TRUE, will be kept in the subset.

```
# Select only married friends
my_friends[are_married == TRUE]
[1] "Nicolas" "Bernard" "Jerome"
# Friends with age >=27
my_friends[friend_ages >= 27]
[1] "Nicolas" "Bernard"
# Friends with age different from 27
my_friends[friend_ages != 27]
[1] "Thierry" "Bernard" "Jerome"
```

If you want to remove missing data, use this:

```
# Data with missing values
have_child
Nicolas Thierry Bernard  Jerome
  "yes"   "yes"      NA      NA
# Keep only values different from NA (!is.na())
have_child[!is.na(have_child)]
Nicolas Thierry
  "yes"   "yes"
# Or, replace NA value by "NO" and then print
have_child[!is.na(have_child)] <- "NO"
have_child
Nicolas Thierry Bernard  Jerome
   "NO"    "NO"      NA      NA
```

Note that, the “logical” comparison operators available in R are:

- **<**: for less than
- **>**: for greater than
- **<=**: for less than or equal to
- **>=**: for greater than or equal to
- **==**: for equal to each other
- **!=**: not equal to each other

Calculations with vectors

Note that, all the basic arithmetic operators (+, -, *, / and ^) as well as the common arithmetic functions (log, exp, sin, cos, tan, sqrt, abs, …), described in the previous sections, can be applied on a numeric vector.

If you perform an operation with vectors, the operation will be applied to each element of the vector. An example is provided below:

```
# My friends' salary in dollars
salaries <- c(2000, 1800, 2500, 3000)
names(salaries) <- c("Nicolas", "Thierry", "Bernard", "Jerome")
salaries
Nicolas Thierry Bernard  Jerome
   2000    1800    2500    3000
# Multiply salaries by 2
salaries*2
Nicolas Thierry Bernard  Jerome
   4000    3600    5000    6000
```

As you can see, R multiplies each element in the salaries vector with 2.

Now, suppose that you want to multiply the salaries by different coefficients. The following R code can be used:

```
# create coefs vector with the same length as salaries
coefs <- c(2, 1.5, 1, 3)
# Multiply salaries by coeff
salaries*coefs
Nicolas Thierry Bernard  Jerome
   4000    2700    2500    9000
```

Note that the calculation is done element-wise. The first element of salaries vector is multiplied by the first element of coefs vector, and so on.

Compute the square root of a numeric vector:

```
my_vector <- c(4, 16, 9)
sqrt(my_vector)
[1] 2 4 3
```

Other useful functions are:

```
max(x) # Get the maximum value of x
min(x) # Get the minimum value of x
# Get the range of x. Returns a vector containing
# the minimum and the maximum of x
range(x)

length(x) # Get the number of elements in x

sum(x) # Get the total of the elements in x

prod(x) # Get the product of the elements in x

# The mean value of the elements in x
# sum(x)/length(x)
mean(x)
sd(x) # Standard deviation of x
var(x) # Variance of x
# Sort the element of x in ascending order
sort(x)
```

For example, if you want to compute the total **sum** of salaries, type this:

```
sum(salaries)
[1] 9300
```

Compute the **mean** of salaries:

```
mean(salaries)
[1] 2325
```

The range (minimum, maximum) of salaries is:

```
range(salaries)
[1] 1800 3000
```

Matrices

A **matrix** is like an Excel sheet containing multiple rows and columns. It's used to combine vectors with the same type, which can be either numeric, character or logical. Matrices are used to store a data table in R. The rows of a matrix are generally individuals/observations and the columns are variables.

Create and naming matrix

To create easily a matrix, use the function **cbind**() or **rbind**() as follow:

```
# Numeric vectors
col1 <- c(5, 6, 7, 8, 9)
col2 <- c(2, 4, 5, 9, 8)
col3 <- c(7, 3, 4, 8, 7)
# Combine the vectors by column
my_data <- cbind(col1, col2, col3)
my_data
     col1 col2 col3
[1,]    5    2    7
[2,]    6    4    3
[3,]    7    5    4
[4,]    8    9    8
[5,]    9    8    7
# Change rownames
rownames(my_data) <- c("row1", "row2", "row3", "row4", "row5")
my_data
     col1 col2 col3
row1    5    2    7
row2    6    4    3
row3    7    5    4
row4    8    9    8
row5    9    8    7
```

- **cbind()**: combine R objects by columns
- **rbind()**: combine R objects by rows
- **rownames()**: retrieve or set row names of a matrix-like object
- **colnames()**: retrieve or set column names of a matrix-like object

If you want to transpose your data, use the function **t**():

```
t(my_data)
     row1 row2 row3 row4 row5
col1    5    6    7    8    9
col2    2    4    5    9    8
col3    7    3    4    8    7
```

Note that, it's also possible to construct a matrix using the function **matrix()**.

The simplified format of **matrix()** is as follow:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
```

- **data**: an optional data vector
- **nrow**, **ncol**: the desired number of rows and columns, respectively.
- **byrow**: logical value. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.
- **dimnames**: A list of two vectors giving the row and column names respectively.

In the R code below, the input data has length 6. We want to create a matrix with two columns. You don't need to specify the number of rows (here nrow = 3). R will infer this automatically. The matrix is filled column by column when the argument **byrow = FALSE**. If you want to fill the matrix by rows, use **byrow = TRUE**.

```
mdat <- matrix(
           data = c(1,2,3, 11,12,13),
           nrow = 2, byrow = TRUE,
           dimnames = list(c("row1", "row2"), c("C.1", "C.2", "C.3"))
           )
mdat
     C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

Dimensions of a matrix

The R functions **nrow**() and **ncol**() return the number of rows and columns present in the data, respectively.

```
ncol(my_data) # Number of columns
[1] 3
nrow(my_data) # Number of rows
[1] 5
dim(my_data) # Number of rows and columns
[1] 5 3
```

Get a subset of a matrix

- **Select rows/columns** by positive indexing

rows and/or columns can be selected as follow: my_data[row, col]

```
# Select row number 2
my_data[2, ]
col1 col2 col3
```

```
   6    4    3
# Select row number 2 to 4
my_data[2:4, ]
     col1 col2 col3
row2    6    4    3
row3    7    5    4
row4    8    9    8
# Select multiple rows that aren't contiguous
# e.g.: rows 2 and 4 but not 3
my_data[c(2,4), ]
     col1 col2 col3
row2    6    4    3
row4    8    9    8
# Select column number 3
my_data[, 3]
row1 row2 row3 row4 row5
   7    3    4    8    7
# Select the value at row 2 and column  3
my_data[2, 3]
[1] 3
```

- **Select by row/column names**

```
# Select column 2
my_data[, "col2"]
row1 row2 row3 row4 row5
   2    4    5    9    8
# Select by index and names: row 3 and olumn 2
my_data[3, "col2"]
[1] 5
```

- **Exclude rows/columns** by negative indexing

```
# Exclude column 1
my_data[, -1]
     col2 col3
row1    2    7
row2    4    3
row3    5    4
row4    9    8
row5    8    7
```

- **Selection by logical**: In the R code below, we want to keep only rows where col3 >=4:

```
col3 <- my_data[, "col3"]
my_data[col3 >= 4, ]
     col1 col2 col3
row1    5    2    7
row3    7    5    4
row4    8    9    8
row5    9    8    7
```

Calculations with matrices

- It's also possible to perform **simple operations on matrice**. For example, the following R code multiplies each element of the matrix by 2:

```
my_data*2
     col1 col2 col3
row1   10    4   14
row2   12    8    6
row3   14   10    8
row4   16   18   16
row5   18   16   14
```

Or, compute the log2 values:

```
log2(my_data)
         col1     col2     col3
row1 2.321928 1.000000 2.807355
row2 2.584963 2.000000 1.584963
row3 2.807355 2.321928 2.000000
row4 3.000000 3.169925 3.000000
row5 3.169925 3.000000 2.807355
```

- **rowSums()** and **colSums()** functions: Compute the total of each row and the total of each column, respectively.

```
# Total of each row
rowSums(my_data)
row1 row2 row3 row4 row5 
  14   13   16   25   24 
# Total of each column
colSums(my_data)
col1 col2 col3 
  35   28   29 
```

If you are interested in row/column means, you can use the function **rowMeans**() and **colMeans**() for computing row and column means, respectively.

Note that, it's also possible to use the function **apply**() to apply any statistical functions to rows/columns of matrices.

The simplified format of **apply**() is as follow:

```
apply(X, MARGIN, FUN)
```

- X: your data matrix
- MARGIN: possible values are 1 (for rows) and 2 (for columns)
- FUN: the function to apply on rows/columns

Use **apply**() as follow:

```
# Compute row means
apply(my_data, 1, mean)
    row1     row2     row3     row4     row5 
4.666667 4.333333 5.333333 8.333333 8.000000 
# Compute row medians
apply(my_data, 1, median)
row1 row2 row3 row4 row5 
   5    4    5    8    8 
# Compute column means
apply(my_data, 2, mean)
col1 col2 col3 
 7.0  5.6  5.8 
```

Factors

Factor variables represent categories or groups in your data. The function **factor**() can be used to create a factor variable.

Create a factor

```
# Create a factor variable
friend_groups <- factor(c(1, 2, 1, 2))
friend_groups
[1] 1 2 1 2
Levels: 1 2
```

The variable *friend_groups* contains two categories of friends: 1 and 2. In R terminology, categories are called **factor levels**.

It’s possible to access to the factor levels using the function **levels()**:

```
# Get group names (or levels)
levels(friend_groups)
[1] "1" "2"
# Change levels
levels(friend_groups) <- c("best_friend", "not_best_friend")
friend_groups
[1] best_friend     not_best_friend best_friend     not_best_friend
Levels: best_friend not_best_friend
```

Note that, R orders factor levels alphabetically. If you want a different order in the levels, you can specify the levels argument in the factor function as follow.

```
# Change the order of levels
friend_groups <- factor(friend_groups, 
                      levels = c("not_best_friend", "best_friend"))
# Print
```

```
friend_groups
[1] best_friend     not_best_friend best_friend     not_best_friend
Levels: not_best_friend best_friend
```

Note that:

- The function **is.factor**() can be used to check whether a variable is a factor. Results are TRUE (if factor) or FALSE (if not factor)
- The function **as.factor**() can be used to convert a variable to a factor.

```
# Check if friend_groups is a factor
is.factor(friend_groups)
[1] TRUE
# Check if "are_married" is a factor
is.factor(are_married)
[1] FALSE
# Convert "are_married" as a factor
as.factor(are_married)
[1] TRUE  FALSE TRUE  TRUE
Levels: FALSE TRUE
```

Calculations with factors

- If you want to know the number of individuals in each levels, use the function **summary()**:

```
summary(friend_groups)
not_best_friend     best_friend
              2               2
```

- In the following example, I want to compute the mean salary of my friends by groups. The function **tapply**() can be used to apply a function, here **mean**(), to each group.

```
# Salaries of my friends
salaries
Nicolas Thierry Bernard  Jerome
   2000    1800    2500    3000
# Friend groups
friend_groups
[1] best_friend     not_best_friend best_friend     not_best_friend
Levels: not_best_friend best_friend
# Compute the mean salaries by groups
mean_salaries <- tapply(salaries, friend_groups, mean)
mean_salaries
not_best_friend     best_friend
           2400            2250
# Compute the size/length of each group
tapply(salaries, friend_groups, length)
not_best_friend     best_friend
              2               2
```

- It's also possible to use the function **table**() to create a frequency table, also known as a contingency table of the counts at each combination of factor levels.

```
table(friend_groups)
friend_groups
not_best_friend     best_friend
            2               2
# Cross-tabulation between
# friend_groups and are_married variables
table(friend_groups, are_married)
                 are_married
friend_groups     FALSE TRUE
  not_best_friend     1    1
  best_friend         0    2
```

Data frames

A data frame is like a matrix but can have columns with different types (numeric, character, logical). Rows are observations (individuals) and columns are variables.

Create a data frame

A data frame can be created using the function **data.frame()**, as follow:

```
# Create a data frame
friends_data <- data.frame(
  name = my_friends,
  age = friend_ages,
  height = c(180, 170, 185, 169),
  married = are_married
)
# Print
friends_data
           name age height married
Nicolas Nicolas  27    180    TRUE
Thierry Thierry  25    170   FALSE
Bernard Bernard  29    185    TRUE
Jerome   Jerome  26    169    TRUE
```

To check whether a data is a data frame, use the **is.data.frame**() function. Returns TRUE if the data is a data frame:

```
is.data.frame(friends_data)
[1] TRUE
is.data.frame(my_data)
[1] FALSE
```

The object “friends_data” is a data frame, but not the object “my_data”. We can convert-it to a data frame using the **as.data.frame**() function:

```
# What is the class of my_data? --> matrix
class(my_data)
[1] "matrix"
# Convert it as a data frame
my_data2 <- as.data.frame(my_data)
# Now, the class is data.frame
class(my_data2)
[1] "data.frame"
```

As described in **matrix** section, you can use the function **t**() to transpose a data frame:

```
t(friends_data)
```

Subset a data frame

To select just certain columns from a data frame, you can either refer to the columns by name or by their location (i.e., column 1, 2, 3, etc.).

1. **Positive indexing** by name and by location

```
# Access the data in 'name' column
# dollar sign is used
friends_data$name
[1] Nicolas Thierry Bernard Jerome
Levels: Bernard Jerome Nicolas Thierry
# or use this
friends_data[, 'name']
[1] Nicolas Thierry Bernard Jerome
Levels: Bernard Jerome Nicolas Thierry
# Subset columns 1 and 3
friends_data[ , c(1, 3)]
           name height
Nicolas Nicolas    180
Thierry Thierry    170
Bernard Bernard    185
Jerome   Jerome    169
```

2. **Negative indexing**

```
# Exclude column 1
friends_data[, -1]
        age height married
Nicolas  27    180    TRUE
Thierry  25    170   FALSE
Bernard  29    185    TRUE
Jerome   26    169    TRUE
```

3. **Index by characteristics**

We want to select all friends with age >= 27.

```
# Identify rows that meet the condition
friends_data$age >= 27
[1]  TRUE FALSE  TRUE FALSE
```

TRUE specifies that the row contains a value of age >= 27.

```
# Select the rows that meet the condition
friends_data[friends_data$age >= 27, ]
           name age height married
Nicolas Nicolas  27    180    TRUE
Bernard Bernard  29    185    TRUE
```

The R code above, tells R to get all rows from friends_data where age >= 27, and then to return all the columns.

If you don't want to see all the column data for the selected rows but are just interested in displaying, for example, friend names and age for friends with age >= 27, you could use the following R code:

```
# Use column locations
friends_data[friends_data$age >= 27,  c(1, 2)]
           name age
Nicolas Nicolas  27
Bernard Bernard  29
# Or use column names
friends_data[friends_data$age >= 27, c("name", "age")]
           name age
Nicolas Nicolas  27
Bernard Bernard  29
```

If you're finding that your selection statement is starting to be inconvenient, you can put your row and column selections into variables first, such as:

```
age27 <- friends_data$age >= 27
cols <- c("name", "age")
```

Then you can select the rows and columns with those variables:

```
friends_data[age27, cols]
           name age
Nicolas Nicolas  27
Bernard Bernard  29
```

It's also possible to use the function **subset**() as follow.

```
# Select friends data with age >= 27
subset(friends_data, age >= 27)
           name age height married
Nicolas Nicolas  27    180    TRUE
```

```
Bernard Bernard  29    185    TRUE
```

Another option is to use the functions **attach**() and **detach**(). The function **attach**() takes a data frame and makes its columns accessible by simply giving their names.

The functions **attach**() and **detach**() can be used as follow:

```
# Attach a data frame
attach(friends_data)
# === Data manipulation ====
friends_data[age>=27, ]
# === End of data manipulation ====
# Detach the data frame
detach(friends_data)
```

Extend a data frame

Add new column in a data frame

```
# Add group column to friends_data
friends_data$group <- friend_groups
friends_data
           name age height married          group
Nicolas Nicolas  27    180    TRUE     best_friend
Thierry Thierry  25    170   FALSE not_best_friend
Bernard Bernard  29    185    TRUE     best_friend
Jerome   Jerome  26    169    TRUE not_best_friend
```

It's also possible to use the functions **cbind**() and **rbind**() to extend a data frame.

```
cbind(friends_data, group = friend_groups)
```

Calculations with data frame

With numeric data frame, you can use the function **rowSums**(), **colSums**(), **colMeans**(), **rowMeans**() and **apply**() as described in **matrix** section.

Lists

A list is an ordered collection of objects, which can be vectors, matrices, data frames, etc. In other words, a list can contain all kind of R objects.

Create a list

```
# Create a list
my_family <- list(
  mother = "Veronique",
  father = "Michel",
  sisters = c("Alicia", "Monica"),
  sister_age = c(12, 22)
  )
# Print
my_family
$mother
[1] "Veronique"
$father
[1] "Michel"
$sisters
[1] "Alicia" "Monica"
$sister_age
[1] 12 22
# Names of elements in the list
names(my_family)
[1] "mother"     "father"     "sisters"    "sister_age"
# Number of elements in the list
length(my_family)
[1] 4
```

The list object "my_family", contains four components, which may be individually referred to as my_family[[1]], as_family[[2]] and so on.

Subset a list

It's possible to select an element, from a list, by its name or its index:

- my_family$mother is the same as my_family[[1]]
- my_family$father is the same as my_family[[2]]

```
# Select by name (1/2)
my_family$father
[1] "Michel"
# Select by name (2/2)
my_family[["father"]]
[1] "Michel"
# Select by index
my_family[[1]]
[1] "Veronique"
my_family[[3]]
[1] "Alicia" "Monica"
# Select a specific element of a component
# select the first ([1]) element of my_family[[3]]
my_family[[3]][1]
[1] "Alicia"
```

Extend a list

Note that, it's possible to extend an original list.

In the R code below, we want to add the components "grand_father" and "grand_mother" to *my_family* list object:

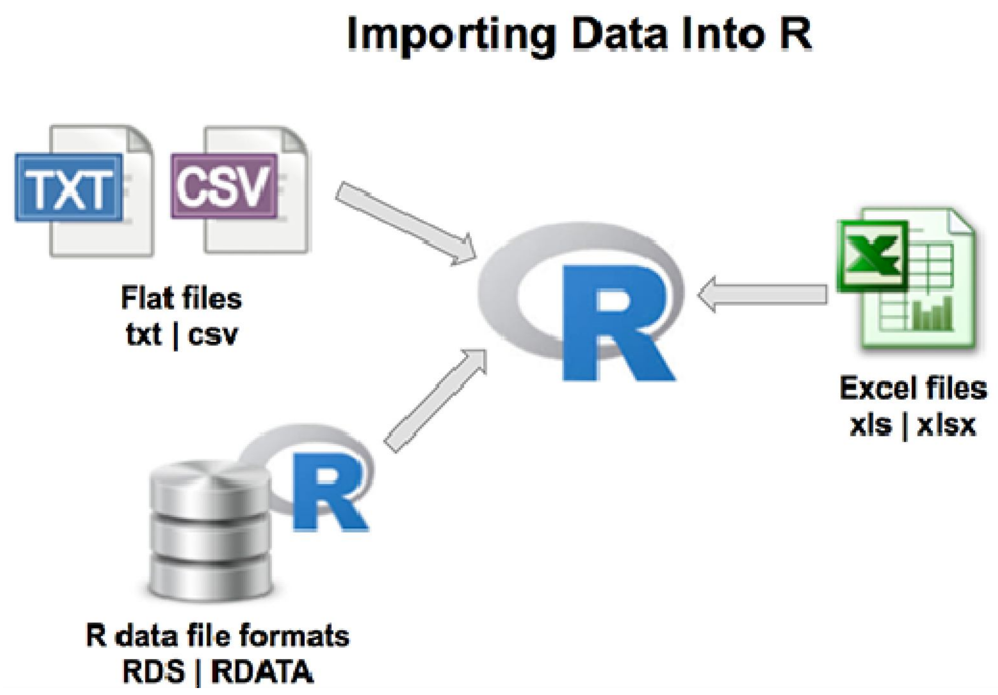
```
# Extend the list
my_family$grand_father <- "John"
my_family$grand_mother <- "Mary"
# Print
my_family
$mother
[1] "Veronique"
$father
[1] "Michel"
$sisters
[1] "Alicia" "Monica"
$sister_age
[1] 12 22
$grand_father
[1] "John"
$grand_mother
[1] "Mary"
```

You can also concatenate two lists as follow:

```
list_abc <- c(list_a, list_b, list_c)
```

The result is a list also, whose components are those of the argument lists joined together in sequence.

Importing Data Into R

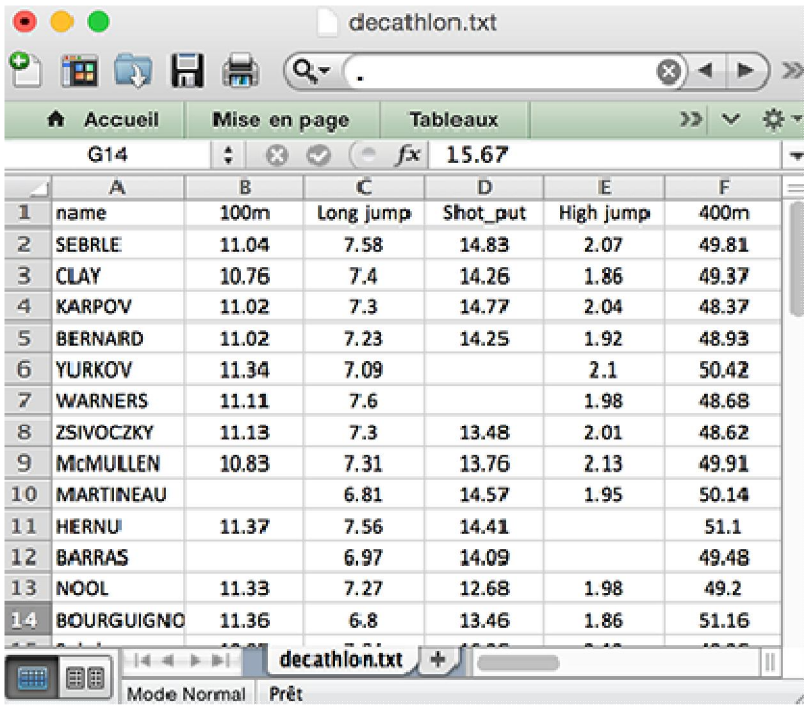


Best Practices in Preparing Data Files for Importing into R

In the previous chapter we provided the essentials of R programming including installation, launching, basic data types and arithmetic functions. In the next articles you will learn how to **import data** into **R**. To avoid errors during the **importation** of a file into R, you should make sure that your data is well prepared.
In this article we'll describe some best practices for **preparing** your data before **importing** into **R**.

Open your file

We suppose that you open and prepare your file with Excel as follow.



	A	B	C	D	E	F
1	name	100m	Long jump	Shot_put	High jump	400m
2	SEBRLE	11.04	7.58	14.83	2.07	49.81
3	CLAY	10.76	7.4	14.26	1.86	49.37
4	KARPOV	11.02	7.3	14.77	2.04	48.37
5	BERNARD	11.02	7.23	14.25	1.92	48.93
6	YURKOV	11.34	7.09		2.1	50.42
7	WARNERS	11.11	7.6		1.98	48.68
8	ZSIVOCZKY	11.13	7.3	13.48	2.01	48.62
9	McMULLEN	10.83	7.31	13.76	2.13	49.91
10	MARTINEAU		6.81	14.57	1.95	50.14
11	HERNU	11.37	7.56	14.41		51.1
12	BARRAS		6.97	14.09		49.48
13	NOOL	11.33	7.27	12.68	1.98	49.2
14	BOURGUIGNO	11.36	6.8	13.46	1.86	51.16

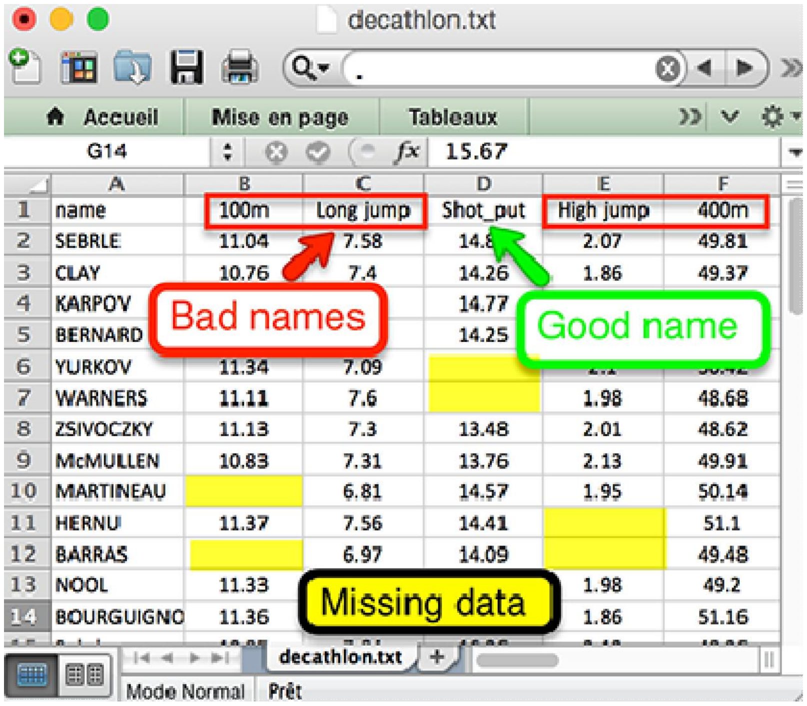
Prepare your file

1. **Row and column names**:

- Use the first row as column **headers** (or **column names**). Generally, columns represent **variables**.

- Use the first column as **row names**. Generally rows represent **observations**.
- Each row name should be unique, so remove duplicated names.

Column names should be compatible with R naming conventions. As illustrated below, our data contains some issues that should be fixed before importing:



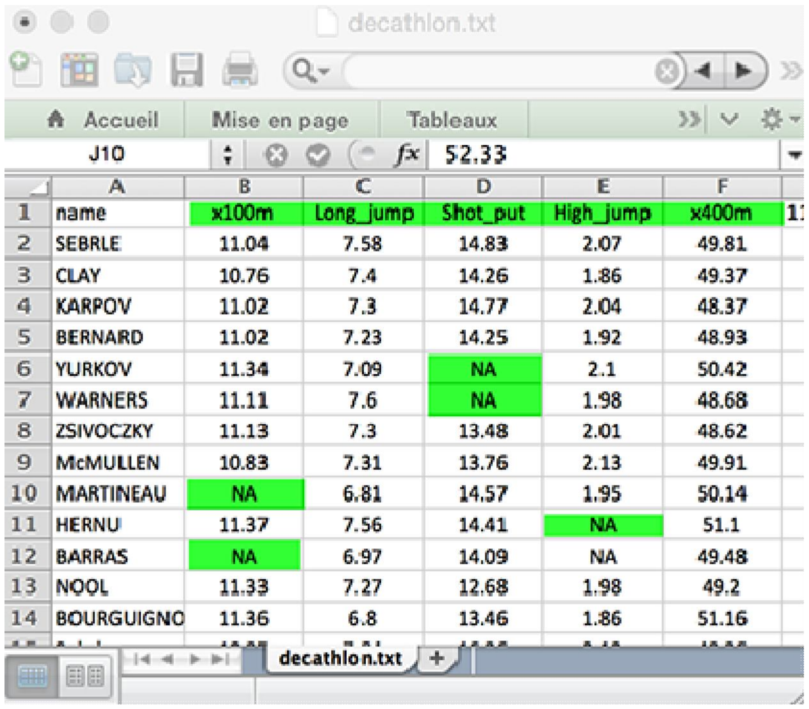
	A	B	C	D	E	F
1	name	100m	Long jump	Shot_put	High jump	400m
2	SEBRLE	11.04	7.58	14.8	2.07	49.81
3	CLAY	10.76	7.4	14.26	1.86	49.37
4	KARPOV			14.77		
5	BERNARD			14.25		
6	YURKOV	11.34	7.09		[illegible]	[illegible]
7	WARNERS	11.11	7.6		1.98	48.68
8	ZSIVOCZKY	11.13	7.3	13.48	2.01	48.62
9	McMULLEN	10.83	7.31	13.76	2.13	49.91
10	MARTINEAU		6.81	14.57	1.95	50.14
11	HERNU	11.37	7.56	14.41		51.1
12	BARRAS		6.97	14.09		49.48
13	NOOL	11.33			1.98	49.2
14	BOURGUIGNO	11.36			1.86	51.16

2. **Naming conventions**:

- Avoid names with blank spaces. Good column names: *Long_jump* or *Long.jump*. Bad column name: *Long jump*.
- Avoid names with special symbols: ?, $, *, +, #, (,), -, /, }, {, |, >, < etc. Only underscore can be used.
- Avoid beginning variable names with a number. Use letter instead. Good column names: sport_100m or x100m. Bad column name: 100m
- Column names must be unique. Duplicated names are not allowed.
- R is case sensitive. This means that Name is different from Name or NAME.
- Avoid blank rows in your data
- Delete any comments in your file
- Replace missing values by **NA** (for not available)
- If you have a column containing date, use the four digit format. Good format: 01/01/2016. Bad format: 01/01/16

3. **Final file**:

Our finale file should look like this:



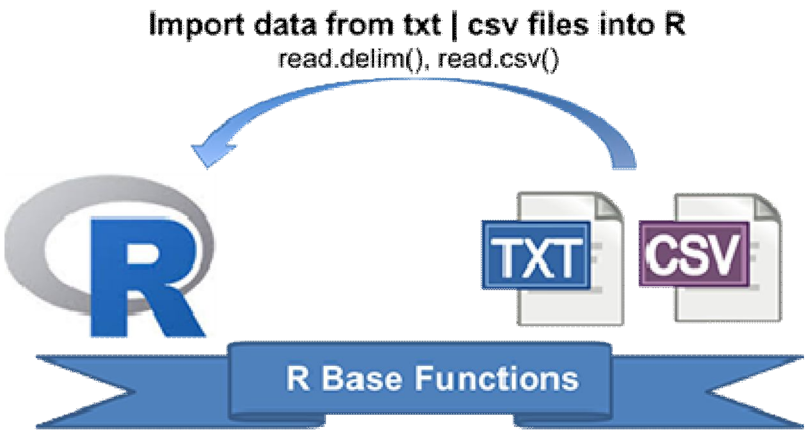
decathlon.txt

J10 52.33

	A	B	C	D	E	F
1	name	x100m	Long_jump	Shot_put	High_jump	x400m
2	SEBRLE	11.04	7.58	14.83	2.07	49.81
3	CLAY	10.76	7.4	14.26	1.86	49.37
4	KARPOV	11.02	7.3	14.77	2.04	48.37
5	BERNARD	11.02	7.23	14.25	1.92	48.93
6	YURKOV	11.34	7.09	NA	2.1	50.42
7	WARNERS	11.11	7.6	NA	1.98	48.68
8	ZSIVOCZKY	11.13	7.3	13.48	2.01	48.62
9	McMULLEN	10.83	7.31	13.76	2.13	49.91
10	MARTINEAU	NA	6.81	14.57	1.95	50.14
11	HERNU	11.37	7.56	14.41	NA	51.1
12	BARRAS	NA	6.97	14.09	NA	49.48
13	NOOL	11.33	7.27	12.68	1.98	49.2
14	BOURGUIGNO	11.36	6.8	13.46	1.86	51.16

--

Reading Data From TXT|CSV Files: R Base Functions



R base functions for importing data

The R base function **read.table**() is a general function that can be used to read a file in table format. The data will be imported as a data frame.

Note that, depending on the format of your file, several variants of **read.table**() are available to make your life easier, including **read.csv**(), **read.csv2**(), **read.delim**() and **read.delim2**().

- **read.csv**(): for reading **"comma separated value"** files (".csv").
- **read.csv2**(): variant used in countries that use a comma "," as decimal point and a semicolon ";" as field separators.
- **read.delim**(): for reading *"tab-separated value"* files (".txt"). By default, point (".") is used as decimal points.
- **read.delim2**(): for reading *"tab-separated value"* files (".txt"). By default, comma (",") is used as decimal points.

The simplified format of these functions are, as follow:

```
# Read tabular data into R
read.table(file, header = FALSE, sep = "", dec = ".")
# Read "comma separated value" files (".csv")
read.csv(file, header = TRUE, sep = ",", dec = ".", ...)
# Or use read.csv2: variant used in countries that
# use a comma as decimal point and a semicolon as field separator.
read.csv2(file, header = TRUE, sep = ";", dec = ",", ...)
# Read TAB delimited files
read.delim(file, header = TRUE, sep = "\t", dec = ".", ...)
read.delim2(file, header = TRUE, sep = "\t", dec = ",", ...)
```

- **file**: the path to the file containing the data to be imported into R.
- **sep**: the field separator character. "\t" is used for tab-delimited file.
- **header**: logical value. If TRUE, **read.table()** assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument **header = FALSE**.
- **dec**: the character used in the file for decimal points.

Reading a local file

- To import a local .txt or a .csv file, the syntax would be:

```
# Read a txt file, named "mtcars.txt"
my_data <- read.delim("mtcars.txt")
# Read a csv file, named "mtcars.csv"
my_data <- read.csv("mtcars.csv")
```

The above R code, assumes that the file “mtcars.txt” or “mtcars.csv” is in your current working directory. To know your current working directory, type the function **getwd**() in R console.

- It’s also possible to choose a file interactively using the function **file.choose**(), which I recommend if you’re a beginner in R programming:

```
# Read a txt file
my_data <- read.delim(file.choose())
# Read a csv file
my_data <- read.csv(file.choose())
```

If you use the R code above in RStudio, you will be asked to choose a file.

If your data contains column with text, R may assume that columns as a factors or grouping variables (e.g.: “good”, “good”, “bad”, “bad”, “bad”). If you don’t want your text data to be converted as factors, add **stringsAsFactor = FALSE** in **read.delim**(), **read.csv**() and **read.table**() functions. In this case, the data frame columns corresponding to string in your text file will be character.

For example:

```
my_data <- read.delim(file.choose(),
                      stringsAsFactor = FALSE)
```

- If your field separator is for example “|”, it’s possible use the general function **read.table**() with additional arguments:

```
my_data <- read.table(file.choose(),
                      sep ="|", header = TRUE, dec =".")
```

Reading a file from internet

It’s possible to use the functions **read.delim**(), **read.csv**() and **read.table**() to import files from the web.

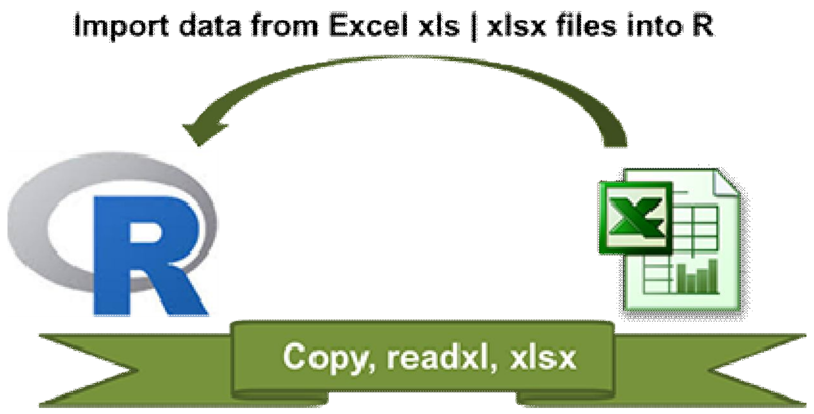
```
my_data <- read.delim("http://www.sthda.com/upload/boxplot_format.txt")
head(my_data)
   Nom variable Group
1 IND1       10     A
2 IND2        7     A
3 IND3       20     A
4 IND4       14     A
5 IND5       14     A
6 IND6       12     A
```

Summary

- Import a local .txt file: **read.delim**(file.choose())
- Import a local .csv file: **read.csv**(file.choose())
- Import a file from internet: **read.delim**(url) if a txt file or **read.csv**(url) if a csv file

--

Reading Data From Excel Files (xls|xlsx) into R



In this article, you'll learn how to **read data** from **Excel xls** or **xlsx** file formats into **R**. This can be done either by:

- **copying data** from Excel
- using **readxl** package
- or using **xlsx** package

Preleminary tasks

1. **Launch RStudio** as described here: Running RStudio and setting up your working directory
2. **Prepare your data** as described here: Best practices for preparing your data

Copying data from Excel and import into R

On Windows system

1. **Open** the Excel file containing your data: **select** and **copy the data** (ctrl + c)
2. Type the R code below to import the copied data from the **clipboard** into R and store the data in a data frame (my_data):

```
my_data <- read.table(file = "clipboard",

              sep = "\t", header=TRUE)
```

Installing and loading readxl package

- Install

```
install.packages("readxl")
```

- Load

```
library("readxl")
```

Using readxl package

The **readxl** package comes with the function **read_excel**() to read xls and xlsx files

1. **Read both xls and xlsx files**

```
# Loading
library("readxl")
# xls files
my_data <- read_excel("my_file.xls")
# xlsx files
my_data <- read_excel("my_file.xlsx")
```

- It's also possible to choose a file interactively using the function **file.choose**(), which I recommend if you're a beginner in R programming:

```
my_data <- read_excel(file.choose())
```

2. **Specify sheet with a number or name**

```
# Specify sheet by its name
my_data <- read_excel("my_file.xlsx", sheet = "data")

# Specify sheet by its index
my_data <- read_excel("my_file.xlsx", sheet = 2)
```

3. **Case of missing values: NA (not available)**. If NAs are represented by something (example: "—") other than blank cells, set the na argument:

```
my_data <- read_excel("my_file.xlsx", na = "---")
```

Importing Excel files using xlsx package

The **xlsx** package, a java-based solution, is one of the powerful R packages to **read**, **write** and **format Excel files**.

Installing and loading xlsx package

- Install

```
install.packages("xlsx")
```

- Load

```
library("xlsx")
```

Using xlsx package

There are two main functions in **xlsx** package for reading both xls and xlsx Excel files: **read.xlsx**() and **read.xlsx2**() [faster on big files compared to read.xlsx function].

The simplified formats are:

```
read.xlsx(file, sheetIndex, header=TRUE)
read.xlsx2(file, sheetIndex, header=TRUE)
```

- **file**: file path
- **sheetIndex**: the index of the sheet to be read
- **header**: a logical value. If TRUE, the first row is used as column names.

Example of usage:

```
library("xlsx")
my_data <- read.xlsx(file.choose(), 1)  # read first sheet
```