BST02: Using R for Statistics in Medical Research

Part C: Functions and Programming

Nicole Erler   Eleni-Rosalina Andrinopoulou

Department of Biostatistics, Erasmus Medical Center

✉ n.erler@erasmusmc.nl   ✉ e.andrinopoulou@erasmusmc.nl

24 - 28 February 2020

Erasmus MC
University Medical Center Rotterdam

Recap Part B

Objects

- `vector`
- `matrix`
- `data.frame`
- `list`

Data Structures

- `numeric`
- `character`
- `integer`
- `logical`
- `factor`

Recap Part B

Objects

- `vector`
- `matrix`
- `data.frame`
- `list`

Operators

- `+`, `-`, `*`, `/`
- `<-`, `=`
- `<`, `>`, `==`

Data Structures

- `numeric`
- `character`
- `integer`
- `logical`
- `factor`

Special Values

- `NA`
- `NaN`
- `Inf`, `-Inf`

Recap Part B

Objects

- `vector`
- `matrix`
- `data.frame`
- `list`

Operators

- `+, -, *, /`
- `<-, =`
- `<, >, ==`

Data Structures

- `numeric`
- `character`
- `integer`
- `logical`
- `factor`

Special Values

- `NA`
- `NaN`
- `Inf, -Inf`

Data Transformations

- rounding (`format()`)
- convert to factor (`factor()`)

Recap Part B

Objects

- `vector`
- `matrix`
- `data.frame`
- `list`

Operators

- +, −, ∗, /
- <−, =
- <, >, ==

Data Structures

- `numeric`
- `character`
- `integer`
- `logical`
- `factor`

Special Values

- `NA`
- `NaN`
- `Inf`, `-Inf`

Data Transformations

- rounding (`format()`)
- convert to factor (`factor()`)

Data Exploration

- `mean()`, `median()`, `sd()`, `IQR()`, ...

Data Visualizations

- plotting packages
- plot types (`plot()`, `barplot()`, ...)

Recap Part B

Objects

- `vector`
- `matrix`
- `data.frame`
- `list`

Operators

- +, −, ∗, /
- <-, =
- <, >, ==

Data Structures

- `numeric`
- `character`
- `integer`
- `logical`
- `factor`

Special Values

- `NA`
- `NaN`
- `Inf`, `-Inf`

Data Transformations

- rounding (`format()`)
- convert to factor (`factor()`)

Data Exploration

- `mean()`, `median()`, `sd()`, `IQR()`, . . .

Data Visualizations

- plotting packages
- plot types (`plot()`, `barplot()`, ...)

Subsetting

- `[[...]]`, `[...]`, `$`, . . .

In this Section

- What are functions?
- Useful functions for data exploration
- Useful functions for data manipulations
- Writing functions
- Control-flow constructs
- The `apply` family
- Lots of practising

Functions

Sometimes we want to perform the same action / manipulation on several objects.

- Option 1: copy & paste
  - a lot of work
  - susceptible to mistakes

Functions

Sometimes we want to perform the same action / manipulation on several objects.

- Option 1: copy & paste
  - a lot of work
  - susceptible to mistakes
- Option 2: **functions**

Functions

Sometimes we want to perform the same action / manipulation on several objects.

- Option 1: copy & paste
  - a lot of work
  - susceptible to mistakes

- Option 2: **functions**

What are functions?

- a group of (organized) R commands
- a (small) program with flexible (= not pre-specified) input

Almost all commands in R are functions!

Functions

Some examples:

- `mean()`
- `sum()`
- `plot()`
- ...

```
class(mean)
## [1] "function"
class(sum)
## [1] "function"
class(plot)
## [1] "function"
```

Functions

Some examples:

- `mean()`
- `sum()`
- `plot()`
- ...

```r
class(mean)
## [1] "function"
class(sum)
## [1] "function"
class(plot)
## [1] "function"
```

Even `class()` is a function:

```r
class(class)
```

```
## [1] "function"
```

Useful Functions for Data Exploration

Demos

- Functions for Data Exploration R html

Practicals

- Exploring and Summarizing Data html

Useful Functions for Data Exploration

Dimension

- `dim()`
- `nrow()`, `ncol()`
- `length()`

Data Structure

- `str()`
- `names()`,
- `head()`, `tail()`
- `is.data.frame()`, `is.list()`, `is.matrix()` `is.numeric()`, `is.ordered()`, ...

Descriptives for Continuous Variables

- `summary()`
- `min()`, `max()`, `range()`
- `mean()`, `median()`, `quantile()`, `IQR()`
- `sd()`, `var()`
- `ave()`

Tables

- `table()`, `prop.table()`
- `addmargins()`, `ftable()`

for `matrix` & `data.frame`

- `summary()`
- `var()`, `cor()`, `cov2cor()`
- `colSums()`, `colMeans()`, `rowSums()`, `rowMeans()`

Duplicates & Comparison

- `duplicated()`
- `unique()`

Useful functions for Data Manipulation

Demo

- Functions for Data Manipulation R html

Practicals

- Merging Data html

Useful functions for Data Manipulation

Transformations

- `log()`, `log2()`, `log10()`
- `exp()`, `sqrt()`, `plogis()`

Splitting & Combining

- `split()`, `cut()`
- `cbind()`, `rbind()`
- `merge()`
- `subset()`
- `c()`
- `paste()`

Sorting

- `sort()`, `order()`, `rev()`, `rank()`

Repetition & Sequence

- `rep()`, `seq()`
- `expand.grid()`

Converting Objects

- `t()`
- `unlist()`, `unname()`
- `as.numeric()`, `as.matrix()`, `as.data.frame()`

Writing Functions

To write your own function:

```
myfun <- function(arguments) {
  syntax
}
```

Writing Functions

To write your own function:

```
myfun <- function(arguments) {
  syntax
}
```

For example:

```
square <- function(x) {
  x^2
}
```

```
square(3)
```

```
## [1] 9
```

Writing Functions

Functions do not always need an argument:

```
random <- function() {
  rnorm(n = 1)
}
```

```
random()
## [1] 0.4291247
random()
## [1] 0.5060559
random()
## [1] -0.57474
```

Writing Functions

Functions can use **multiple arguments**:

```r
subtract <- function(x, y) {
  x - y
}
```

```r
subtract(x = 5.2, y = 3.3)
```

```
## [1] 1.9
```

Writing Functions

Multiple arguments are interpreted in the **pre-defined order**, unless they are named:

```r
subtract(5.2, 1.2)
```

```
## [1] 4
```

is equivalent to

```r
subtract(x = 5.2, y = 1.2)
```

```
## [1] 4
```

Writing Functions

Multiple arguments are interpreted in the **pre-defined order**, unless they are named:

```r
subtract(5.2, 1.2)
```

```
## [1] 4
```

is equivalent to

```r
subtract(x = 5.2, y = 1.2)
```

```
## [1] 4
```

But this is different:

```r
subtract(y = 5.2, x = 1.2)
```

```
## [1] -4
```

Writing Functions

We can also define **default values** for arguments.

```
multiply <- function(x, y = 2) {
  x * y
}
```

The default value is used when the user does not specify a value for that argument:

```
multiply(x = 3, y = 3)
```

```
## [1] 9
```

```
multiply(x = 3)
```

```
## [1] 6
```

Writing Functions

Practical

- Rolling the Dice **html**

Control-flow Constructs: `if()`

Sometimes, we may want to execute code only **if a certain condition is fulfilled**.

To do this, we can use an `if` statement

```
if (condition) {expression}
```

Control-flow Constructs: `if()`

Sometimes, we may want to execute code only **if a certain condition is fulfilled**.

To do this, we can use an `if` statement

```r
if (condition) {expression}
```

For example:

```r
x <- rnorm(n = 10)
if (length(x) > 5) {mean(x)}
```

```
## [1] -0.3850297
```

Control-flow Constructs: `if()`

Sometimes, we may want to execute code only **if a certain condition is fulfilled**.

To do this, we can use an `if` statement

```
if (condition) {expression}
```

For example:

```
x <- rnorm(n = 10)
if (length(x) > 5) {mean(x)}
```

```
## [1] -0.3850297
```

```
x <- rnorm(n = 5)
if (length(x) > 5) {mean(x)}
```

If the condition is not fulfilled, `NULL` is returned.

Control-flow Constructs: `if()` and `else`

We can also specify an expression that is evaluated **if the condition is not fulfilled**:

```
if (cond) {expression} else {alternative expression}
```

Control-flow Constructs: `if()` and `else`

We can also specify an expression that is evaluated **if the condition is not fulfilled**:

```
if (cond) {expression} else {alternative expression}
```

For example:

```
if (length(x) > 5) {
  mean(x)
} else {
  x
}
```

```
## [1] -0.9111954 -0.8371717  2.4158352  0.1340882 -0.4906859
```

Conditional Element Selection: `ifelse()`

A similar function is `ifelse()`, which performs **conditional element selection**:

```
ifelse(test, yes, no)
```

Conditional Element Selection: `ifelse()`

A similar function is `ifelse()`, which performs **conditional element selection**:

```
ifelse(test, yes, no)
```

For example:

```
(x <- round(rnorm(n = 6), 2))
```

```
## [1] -0.44  0.46 -0.69 -1.45  0.57 -1.02
```

```
ifelse(x > 0, ">0", "<0")
```

```
## [1] "<0" ">0" "<0" "<0" ">0" "<0"
```

Conditional Element Selection: `ifelse()`

A similar function is `ifelse()`, which performs **conditional element selection**:

```
ifelse(test, yes, no)
```

For example:

```
(x <- round(rnorm(n = 6), 2))
```

```
## [1] -0.44  0.46 -0.69 -1.45  0.57 -1.02
```

```
ifelse(x > 0, ">0", "<0")
```

```
## [1] "<0" ">0" "<0" "<0" ">0" "<0"
```

Note:

- `if()` expects **one** condition
- `ifelse()` expects a **vector of conditions**

Control-flow Constructs: `for()`-loop

To perform an operation multiple times, we can use a **`for`-loop**

```
for (variable in sequence) {expression}
```

For example:

```
for (i in 1:5) {
  print(2 * i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

Control-flow Constructs: `for()`-loop

In a `for`-**loop**, the `variable` does not need to be used in the `expression`:

```r
for (i in 1:5) {
  print('test')
}
```

```
## [1] "test"
## [1] "test"
## [1] "test"
## [1] "test"
## [1] "test"
```

Note that when using `for()`, always the full `sequence` is used, i.e., we cannot skip iterations.

Control-flow Constructs: `while()`-loop

The function `while()` repeatedly evaluates an expression as long as a condition is fulfilled:

```
while (condition) {expression}
```

Careful:
If your condition is never `FALSE` this will run forever!!!
(or until you stop it manually)

Note:
`for()` and `while()` loops will not print output, unless we specifically use the function `print()`.

Control-flow Constructs: `while()`-loop

For example:

```r
s <- 0
while (s < 7) {
  x <- rnorm(n = 1, mean = 0, sd = 2)
  s <- s + abs(x)
  print(s)
}
```

```
## [1] 2.414131
## [1] 2.96899
## [1] 5.137872
## [1] 9.829268
```

Control-flow Constructs

Demo

- Control Flow R html

Practical

- Control Flow and Functions html

Summary: Writing Functions

```r
function_name <- function(arguments) {
  "function body"
}
```

- can have 0, 1, 2, …arguments
- arguments are interpreted in the **pre-specified order**, unless the **names are used**
- we can specify **default values**

Summary: Control-flow Constructs

- `if (condition) expression`:
  evaluates the ‘expression‘ only if the ‘condition‘ is ‘TRUE‘
- `if (condition) expression1 else expression2`:
  evaluates ‘expression1‘ if the ‘condition‘ is ‘TRUE‘ and ‘expression2‘ if the ‘condition‘ is ‘FALSE‘
- `ifelse(test, yes, no)`:
  expects a vector of ‘test‘s
- `for()` and `while()` loops:
  can be used to repeatedly perform the same action
- to print output from within `for()` and `while()` we need to use `print()`

What is the apply Family

Manipulate **vectors** or slices of data from **matrices**, **data frames** and **lists** in a repetitive way avoiding explicit use of loop-constructs

- An aggregating function, like for example the mean, or the sum
- Other transforming or subsetting functions
- Other vectorized functions, which return more complex structures like lists, vectors and matrices

What is the apply Family (cont’d)

apply(), lapply() , sapply(), tapply(), mapply()

But how and when should we use these?

How To Use apply() in R

- Operates on `matrix` and `data.frame`

- By column

```r
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```r
apply(mat, 2, sum)
```

```
[1]  6 15  6
```

How To Use apply() in R

- Operates on `matrix` and `data.frame`

- By column

```r
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```r
apply(mat, 2, sum)
```

```
[1]  6 15  6
```

- By row

```r
apply(mat, 1, sum)
```

```
[1]  6  9 12
```

How To Use apply() in R (cont'd)

- Operates on `matrix` and `data.frame`

- By column

```r
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```r
apply(mat, 2, mean)
```

```
[1] 2 5 2
```

How To Use apply() in R (cont’d)

- Operates on `matrix` and `data.frame`

- By column

```
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```
apply(mat, 2, mean)
```

```
[1] 2 5 2
```

- By row

```
apply(mat, 1, mean)
```

```
[1] 2 3 4
```

How To Use apply() in R (cont'd)

- You can also apply your own functions

- By column

```r
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```r
apply(mat, 2, function(x)
          sum(x)/(length(x)-1))
```

```
[1] 3.0 7.5 3.0
```

How To Use apply() in R (cont'd)

- You can also apply your own functions

- By column

```r
mat <- matrix(1:6, 3, 3)
mat
```

```
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    2    5    2
[3,]    3    6    3
```

```r
apply(mat, 2, function(x)
          sum(x)/(length(x)-1))
```

```
[1] 3.0 7.5 3.0
```

- By row

```r
apply(mat, 1, function(x)
          sum(x)/(length(x)-1))
```

```
[1] 3.0 4.5 6.0
```

How To Use lapply() in R

- Apply a given function to every element of a `list` and return a `list`

- The difference with `apply()`:
  - It can be used for other objects like `vector`, `data.frame` or `list`
  - The output returned is a list

How To Use lapply() in R (cont’d)

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
myList
```

```
$x
[1] 1 2 3 4 5 6

$y
[1] "m" "f"

$z
[1] 30  4 23
```

How To Use lapply() in R (cont’d)

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
myList
```

```
$x
[1] 1 2 3 4 5 6

$y
[1] "m" "f"

$z
[1] 30  4 23
```

- Use pre-specified functions

```r
lapply(myList, length)
```

```
$x
[1] 6

$y
[1] 2

$z
[1] 3
```

How To Use lapply() in R (cont'd)

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))

myList
```

```
$x
[1] 1 2 3 4 5 6

$y
[1] "m" "f"

$z
[1] 30  4 23
```

- ▶ Use pre-specified functions

```r
lapply(myList, median)
```

```
$x
[1] 3.5

$y
[1] NA

$z
[1] 23
```

- ▶ You can also apply your own functions!

How To Use sapply() in R

- `sapply()` is similar to `lapply()`, but it tries to simplify the output

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
myList
```

```
$x
[1] 1 2 3 4 5 6

$y
[1] "m" "f"

$z
[1] 30  4 23
```

How To Use sapply() in R

- `sapply()` is similar to `lapply()`, but it tries to simplify the output

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
myList
```

```
$x
[1] 1 2 3 4 5 6

$y
[1] "m" "f"

$z
[1] 30  4 23
```

- Use pre-specified functions

```r
sapply(myList, length)
```

```
x y z
6 2 3
```

```r
sapply(myList, median)
```

```
   x    y    z
 3.5   NA 23.0
```

- You can also apply your own functions!

How To Use tapply() in R

- Apply a function to subsets of a `vector` - The subsets are defined by some other `vector`, usually a factor

```
tapply(pbc$bili, pbc$sex, mean)
```

```
       m        f
2.865909 3.262567
```

```
tapply(pbc$age, pbc$sex, median)
```

```
       m        f
54.00137 50.19302
```

How To Use tapply() in R (cont'd)

- You can also apply your own functions

```r
tapply(pbc$bili, pbc$sex, function(x) sum(x)/(length(x)-1))
```

```
       m        f 
2.932558 3.271314 
```

How To Use mapply() in R

- Multivariate apply
- Its purpose is to be able to vectorize arguments to a function that is not usually accepting `vectors` as arguments
- `mapply()` applies a function to multiple `list` or multiple `vector` arguments

```
mapply(length, pbc)
```

```
      id     time   status      trt      age      sex  ascites   hepato   sp
     418      418      418      418      418      418      418      418
    trig platelet  protime    stage
     418      418      418      418
```

How To Use mapply() in R (cont’d)

- Overlapping between functions

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
mapply(length, myList,
       SIMPLIFY = FALSE)
```

```
$x
[1] 6

$y
[1] 2

$z
[1] 3
```

How To Use mapply() in R (cont’d)

- Overlapping between functions

```r
myList <- list(x = c(1:6),
               y = c("m", "f"),
               z = c(30, 4, 23))
mapply(length, myList,
       SIMPLIFY = FALSE)
```

```
$x
[1] 6

$y
[1] 2

$z
[1] 3
```

```r
lapply(myList, length)
```

```
$x
[1] 6

$y
[1] 2

$z
[1] 3
```

- You can also apply your own functions!

Useful Summary: Apply Family

Vectors

- `tapply()`
- `mapply()`

Matrices

- `apply()`
- `tapply()`
- `lapply()`
- `sapply()`
- `mapply()`

Data frames

- `apply()`
- `tapply()`
- `lapply()`
- `sapply()`
- `mapply()`

Lists

- `lapply()`
- `sapply()`
- `mapply()`

Useful Summary: Apply Family

- Use the following webpage to further investigate the apply family https://emcbiostatistics.shinyapps.io/the_apply_family/
- The **R** code for the shiny app is also available:

Demos

- Shiny app apply family R

In order to run the app you will need to install the packages:

- `survival`
- `shiny`

Useful Summary: Apply Family (cont'd)

Demos

- The Apply Family R html

Practicals

- The Apply Family html