(A bit of) Advanced R

Part 1 - R-base programming

Julien Chiquet

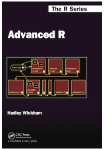
http://github/jchiquet/CourseAdvancedR

Université Paris Dauphine, Juin 2018

Outline

References

Advanced R (Wickham, 2014), http://adv-r.had.co.nz/



A Language and Environment for Statistical Computing (R Core Team, 2017), https://www.R-project.org/

⇝ Many ideas/examples inspired/stolen there.

Prerequisites

Data Structure in base R

1. Atomic vector (integer, double, logical, character)
2. Recursive vector (list)
3. Factors
4. Matrices and array
5. Data Frame

$\rightsquigarrow$ Creation, Basic Operation, Manipulation, Representation

Resources

- Advanced R, chapters I.2, I.3 (Wickham, 2014, http://adv-r.had.co.nz/)
- An introduction to R programming
  http://julien.cremeriefamily.info/teachings_L3BI_ISV51.html

Developement environment I

The Rstudio API

- A full API with code, interpreter, workspace and plots
- Package developement and external code integration are easier
- Notebooks integration with Rmarkdown
- Interface with github $\rightsquigarrow$ required tool for efficent development in `R`

My favorites shortcuts

- `ctrl + return`: execute current selection in console
- `ctrl + 1/2/3/4`: naviguate between panels
- `ctrl + down/up`: naviguate between tabs
- `ctrl + shift + k`: knit current doccument

Developement environment II

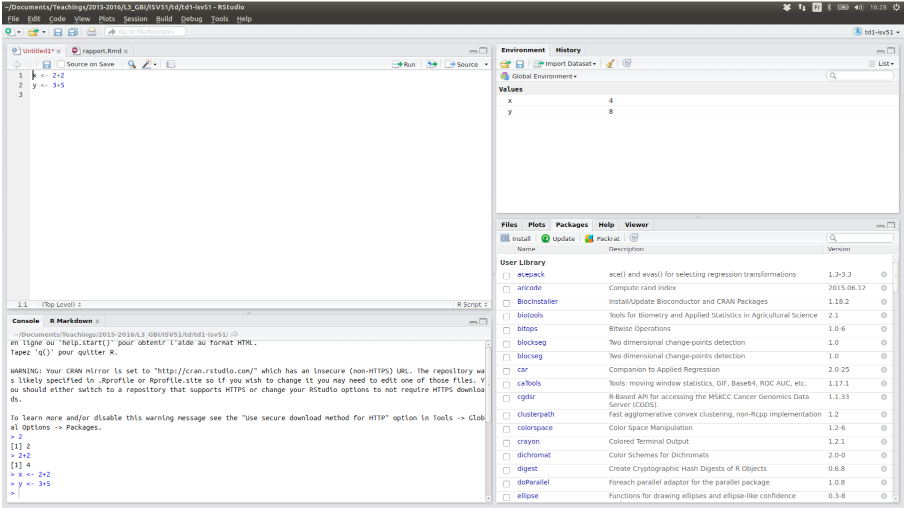


Figure 1: Screenshot of the Rstudio API

Outline

Grouped expressions

Syntax

```
{expr_1; expr_2; ...; expr_n }
{
  expr_1
   ...
  expr_n
}
```

Remarks

- the last value is sent back
- un group statement can be passed to a function

Grouped expressions: examples

Example 1

```r
expr1 <- {a<-3; b<-5; a*b}
expr1
```

```
## [1] 15
```

Example 2

```r
tmp <- 12
expr2 <- {a<-3; b<-5; tmp<-a*b+tmp}
```

```r
expr2
```

```
## [1] 27
```

```r
tmp
```

```
## [1] 27
```

Grouped expressions: examples

Example 1

```
expr1 <- {a<-3; b<-5; a*b}
expr1
```

```
## [1] 15
```

Example 2

```
tmp <- 12
expr2 <- {a<-3; b<-5; tmp<-a*b+tmp}
```

```
expr2
```

```
## [1] 27
```

```
tmp
```

```
## [1] 27
```

Grouped expressions: examples

Example 1

```
expr1 <- {a<-3; b<-5; a*b}
expr1
```

```
## [1] 15
```

Example 2

```
tmp <- 12
expr2 <- {a<-3; b<-5; tmp<-a*b+tmp}
```

```
expr2
```

```
## [1] 27
```

```
tmp
```

```
## [1] 27
```

Conditional statements: `if`, `if/else`, `ifelse`

Standard syntax

```
if (condition) {
  expr_1
} else {
  expr_2
}
```

Vectorial form

```
ifelse(condition, a, b)
```

Remarks

- `condition` is logical, so use &, |, `!`, etc.
- `else` is optional
- `elseif` allows imbricating statements

Conditional statements: example

```
partiel <- 11
DS <- 14
if (partiel > 6 & mean(DS,partiel) > 10) {
    cat("\nreçu(e).")
} else {
    cat("\nrecalé(e).")
}
```

```
##
## reçu(e).
```

Exercice

Use the vectorial `ifelse` to send the full vector of results

```
partiel <- c(11,5,6,12,9,8,14)
DS <- c(14,16,12,12,19,12,7)
```

```
ifelse(partiel > 6 & rowMeans(cbind(DS,partiel)) > 10, "reçu(e)", "recalé(e)")
```

```
## [1] "reçu(e)"   "recalé(e)" "recalé(e)" "reçu(e)"   "reçu(e)"   "recalé(e)"
## [7] "reçu(e)"
```

Conditional statements: example

```r
partiel <- 11
DS <- 14
if (partiel > 6 & mean(DS,partiel) > 10) {
    cat("\nreçu(e).")
} else {
    cat("\nrecalé(e).")
}
```

```
##
## reçu(e).
```

Exercice

Use the vectorial `ifelse` to send the full vector of results

```r
partiel <- c(11,5,6,12,9,8,14)
DS <- c(14,16,12,12,19,12,7)
```

```r
ifelse(partiel > 6 & rowMeans(cbind(DS,partiel)) > 10, "reçu(e)", "recalé(e)")
```

```
## [1] "reçu(e)"   "recalé(e)" "recalé(e)" "reçu(e)"   "reçu(e)"   "recalé(e)"
## [7] "reçu(e)"
```

Conditional statement: `switch`

Syntax

```
switch (expr,
 expr_1 = do_1,
 ...,
 expr_n = do_n,
 do_default
)
```

Remarks

- `expr` is either an integer or a character
- if an integer with value i, the ith expression `do_i` is evaluated
- if a string the expression `do_i` so that `expr == expr_i` is evaluated

switch: examples

integer form

```
expr <- 2
switch(expr, cat("My value is 1"), cat("My value is 2"))
```

```
## My value is 2
```

```
expr <- 3
switch(expr, cat("My value is 2"), cat("My value is 2"))
```

character form

```
stat <- "variance"
f_stat <- switch(stat,
    "mean"   = mean,
    "variance" = var,
NULL)
f_stat(1:10)
```

```
## [1] 9.166667
```

loop statement: `for`

Syntax

```
for (var in set) {
  expr(var)
}
for (var in set) # avoid this syntax!!
  expr(var)
for (var in set) expr(var)
```

Remarks

- `var` is the incremented variable
- `set` is a vector of the successive values
- generally slow compared to matricial/vectorize operation

for loop: examples

Example: C/C++ like

```
for (i in sample(1:5)) cat(i)
```

```
## 23145
```

```
v <- numeric(7)
for (i in seq_along(v))  v[i] <- i*3
```

Exercice:

Use a for loop to display the colnames of the data frame `iris` which are not a factor, by completing the follwoing piece of code

```
data(iris)
for (nom in colnames(iris)) {
  if (!is.factor(iris[,nom])) cat("",nom)
}
```

```
##  Sepal.Length Sepal.Width Petal.Length Petal.Width
```

A more R-style way to do that is

```
cat(colnames(iris)[!sapply(iris, is.factor)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
```

for loop: examples

Example: C/C++ like

```
for (i in sample(1:5)) cat(i)
```

```
## 23145
```

```
v <- numeric(7)
for (i in seq_along(v))  v[i] <- i*3
```

Exercice:

Use a for loop to display the colnames of the data frame `iris` which are not a factor, by completing the follwoing piece of code

```
data(iris)
for (nom in colnames(iris)) {
  if (!is.factor(iris[,nom])) cat("",nom)
}
```

```
##  Sepal.Length Sepal.Width Petal.Length Petal.Width
```

A more R-style way to do that is

```
cat(colnames(iris)[!sapply(iris, is.factor)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
```

for loop: examples

Example: C/C++ like

```
for (i in sample(1:5)) cat(i)
```

```
## 23145
```

```
v <- numeric(7)
for (i in seq_along(v))  v[i] <- i*3
```

Exercice:

Use a for loop to display the colnames of the data frame `iris` which are not a factor, by completing the follwoing piece of code

```
data(iris)
for (nom in colnames(iris)) {
  if (!is.factor(iris[,nom])) cat("",nom)
}
```

```
##  Sepal.Length Sepal.Width Petal.Length Petal.Width
```

A more R-style way to do that is

```
cat(colnames(iris)[!sapply(iris, is.factor)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
```

Loop statement: `while`, `repeat`

Syntax

```
while (condition) {
  expr
}
repeat {
  expr
}
```

Remarks

- avoid imbrication (slow)
- can be interrupted/controlled with with `break`/`next`

```
repeat {
  expr
  if (condition) {break}
}
while (condition1){
  expr_1
  if (condition2) {next}
  expr_2
}
```

Outline

1. Control Statements
2. Functions
3. Functionals

Functions definition

Syntax

```
my_func <- function(arg1,arg2, ...) {
  expression
}
```

Remarks

- The last value of the expression is returned
- One must use a list to send back several objects
- `return()` is used only when you need to send a value at an early stage in the expression
- In R, functions are object like any others and can manipulated as such

Function components I

Most of functions have three parts

- the `body()` (code inside the function)
- the `formals()` (list of arguments)
- the `environment()` (a set of bindings between symbols and objects, i.e, a place to store variables)

```
environment(var)
```

```
## <environment: namespace:stats>
```

```
formals(var)
```

```
## $x
##
##
## $y
## NULL
##
## $na.rm
## [1] FALSE
##
## $use
```

Function components II

```r
body(var)
```

```
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
```

Lexical Scoping I

Definition

Set of rule that governs how R looks up the value of a symbol

Name masking

If a name is not defined inside a function, R looks a level up

```
y <- 2
func <- function(x) c(x,y)
func(4)
```

```
## [1] 4 2
```

This applies to funciton defined in another function

```
x <- 2
func <- function(y) {
  sub_func <- function(z) c(x,y,z)
  sub_func(5)
}
func(3)
```

```
## [1] 2 3 5
```

Lexical Scoping II

function vs variable

R makes the distinction between variable and function names

```
n <- function(x) x/2
f <- function() {n <- 10 ; n(n)}
f()
```

```
## [1] 5
```

Fresh star

An environement is created at *each time a function is called*

```
f <- function() {
  a <- ifelse(exists("a"), a + 1, 1)
  print(a)
}
f()
```

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

```
f()
```

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

```
a <- 4
f()
```

```
## [1] 5
```

```
f()
```

```
## [1] 5
```

Function arguments

Calling function

Arguments can be specified 1. by name 2. by partial name 3. by position
Here are some stupid (but correct) call to `mean(x=,trim=,na.rm=)`

```
mean(1:10, n = T)
```

```
## [1] 5.5
```

```
mean(1:10, , FALSE)
```

```
## [1] 5.5
```

```
mean(1:10, 0.05, FALSE)
```

```
## [1] 5.5
```

```
mean(, TRUE, x = c(1:10, NA))
```

```
## [1] 5.5
```

Exercice

Clarify the following function calls

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

Default arguments

Arguments can have default values in R

```
f <- function(a = 1, b = 2) c(a,b)
f()
```

```
## [1] 1 2
```

The missing function

You can check wether an argument was passed or not with `missing`:

```
f <- function(a = 1, b = 2) c(missing(a),missing(b))
f(a)
```

```
## [1] FALSE  TRUE
```

Hence, you can assign a default value *a posteriori*

```
f <- function(a, b = 2) {
  if (missing(a)) a <- 3
c(a, b)
}
f(a)
```

```
## [1] 4 2
```

Default arguments

Arguments can have default values in R

```
f <- function(a = 1, b = 2) c(a,b)
f()
```

```
## [1] 1 2
```

The `missing` function

You can check wether an argument was passed or not with `missing`:

```
f <- function(a = 1, b = 2) c(missing(a),missing(b))
f(a)
```

```
## [1] FALSE  TRUE
```

Hence, you can assign a default value *a posteriori*

```
f <- function(a, b = 2) {
  if (missing(a)) a <- 3
c(a, b)
}
f(a)
```

```
## [1] 4 2
```

Lazy evaluation

Arguments are evaluated only if they are used, which is known as “lazy evaluation”

```
f <- function(a = 1, b = 4*a) c(a,b)
f()
```

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

```
f(43)
```

```
## [1]  43 172
```

Even better (or worse. . .)

```
f <- function(a = 1, b = d) {
  d <- 4 + 2 * a; c(a,b)
}
f()
```

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

```
f(4)
```

```
## [1]  4 12
```

The ... argument

The argument ... matches any argument not otherwise matched

- useful when collecting argument to call another function
- do not need to precify the name ofd required argument
- the counterpart is that any misspelled argument is passed to ... and show no warning

Example: `plot`

Many argument in `plot` are passed to the `par` function that manages the graphical paramters:

```
plot(1:5, col = "red")
plot(1:5, llty = "dotted")
```

Capturing ...

`list()` can be used to easily captured arguments passed with ...

```
f <- function(...) names(list(...))
f(a = 1, b = 2)
```

```
## [1] "a" "b"
```

Calling function with a list

The `do.call` function constructs and executes a function call from a name or a function and a list of arguments to be passed to it:

```
do.call(mean, list(x = 1:10, trim = 0.05, na.rm = FALSE))
```

```
## [1] 5.5
```

Exercice

Suppose the outputs of 100 simulations are stored in a list like that

```
class(res)
```

```
## [1] "list"
```

```
res[[1]]
```

```
##   method       mse       timing
## 1  lasso 0.9858685   0.6677586
## 2  ridge 0.5847965   0.2446544
## 3  bayes 0.9616895 130.3337640
```

```
length(res)
```

```
## [1] 100
```

How would you store them in a single data frame?

```
head(do.call(rbind, res), 2)
```

```
##   method       mse    timing
## 1  lasso 0.9858685 0.6677586
## 2  ridge 0.5847965 0.2446544
```

Calling function with a list

The `do.call` function constructs and executes a function call from a name or a function and a list of arguments to be passed to it:

```
do.call(mean, list(x = 1:10, trim = 0.05, na.rm = FALSE))
```

```
## [1] 5.5
```

Exercice

Suppose the outputs of 100 simulations are stored in a list like that

```
class(res)
```

```
## [1] "list"
```

```
res[[1]]
```

```
##   method       mse      timing
## 1  lasso 0.9858685   0.6677586
## 2  ridge 0.5847965   0.2446544
## 3  bayes 0.9616895 130.3337640
```

```
length(res)
```

```
## [1] 100
```

How would you store them in a single data frame?

```
head(do.call(rbind, res), 2)
```

```
##   method       mse    timing
## 1  lasso 0.9858685 0.6677586
## 2  ridge 0.5847965 0.2446544
```

Calling function with a list

The `do.call` function constructs and executes a function call from a name or a function and a list of arguments to be passed to it:

```
do.call(mean, list(x = 1:10, trim = 0.05, na.rm = FALSE))
```

```
## [1] 5.5
```

Exercice

Suppose the outputs of 100 simulations are stored in a list like that

```
class(res)
```

```
## [1] "list"
```

```
res[[1]]
```

```
##   method       mse       timing
## 1  lasso 0.9858685   0.6677586
## 2  ridge 0.5847965   0.2446544
## 3  bayes 0.9616895 130.3337640
```

```
length(res)
```

```
## [1] 100
```

How would you store them in a single data frame?

```
head(do.call(rbind, res), 2)
```

```
##   method       mse    timing
## 1  lasso 0.9858685 0.6677586
## 2  ridge 0.5847965 0.2446544
```

Infix functions

Definition

Infix function (contrary to prefix functions) are function where the name comes between the argument (like "-" or "+").
R comes with the following infix functions predefined: `%%`, `%*%`, `%/%`, `%in%`, `%o%`, `%x%`, `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, `<<-`

Example

Can be use to define operator

```
`%+%` <- function(x,y) paste(x,y)
"Université" %+% "Paris" %+% "Dauphine"
```

```
## [1] "Université Paris Dauphine"
```

Exercice

Create infix functions for `intersection`, `union` and `setdiff` and test it on simple vectors.

Primitive functions: definition

- Primitive functions are functions from the `base` package that call `C` code directly
- Primitive functions do not contain `R` code, as so

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
formals(sum)
```

```
## NULL
```

```
body(sum)
```

```
## NULL
```

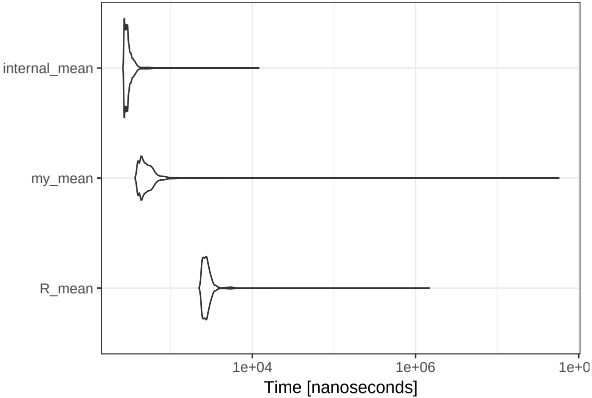
```
environment(sum)
```

```
## NULL
```

Primitive functions: performance

Function defined internally (either with `.Primitive` either called via `.Internal`) are sometimes incredibly faster (written in `C`), but cannot by called directly in packages submitted to CRAN.

```
x <- rnorm(100)
R_mean <- function(x) mean(x)
my_mean <- function(x) sum(x)/length(x)
internal_mean <- function(x) .Internal(mean(x))
```



Compile your functions with `base::compiler` I

The `R` byte code compiler

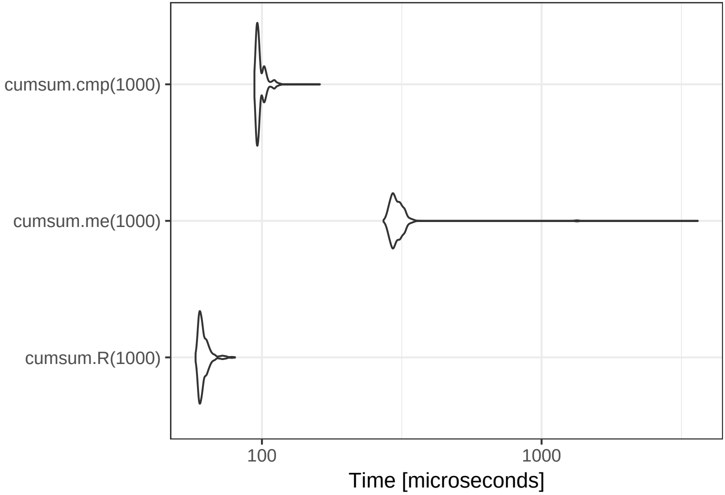
The function `cmpfun` compiles the body of your function and returns a new function with the same formals and the body replaced by the compiled body expression.

```
cumsum.R <- function(n) {
  x <- rnorm(n)
  cumsum(x)
}

cumsum.me <- function(n) {
  x <- rnorm(n)
  res <- 0
  for (i in 1:length(x))
    res <- res + x[i]
  res
}

cumsum.cmp <- compiler::cmpfun(cumsum.me)
```

Compile your functions with `base::compiler` II



- If you cannot avoid a loop, you will save some time
- Can be set automatically with `compiler::enableJIT(3)`

Outline

Functional programming: basics

For loop functional: `lapply`

Definition

Applies a function to each element found in a vector. The vector is defined in a `R`-way, that is, it can be a list.

```
lapply(vector, FUN, ...)
```

Remarks

- `lapply` is the more spred functional in `R`
- Works on vector of *elements*, *numeric indices* and *names*
- Most of other functionals that we will meat build on `lapply`
- Make your code *elegant* and *easier to read*

Implementation

The way `lapply` operates can be understood as follows:

```
my_lapply <-function(vector, FUN, ...) {
  res <- vector("list", length(vector))
  for (i in seq_along(vector))
    res[[i]] <- FUN(vector[[i]], ...)
  res
}
```

Anonymous function

Definition

A function that does not deserve a name, defined 'on the fly' during its use

```
unlist(lapply(datasets::iris, function(x) length(unique(x))))
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##           35           23           43           22            3
```

⇝ Generally very short with few arguments, used in cunjunction with `lapply`-like functionals

```
(function(x) (x^2))(2)
```

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

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

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

Exercises on `lapply`

Consider the dataset `datasets::mtcars`.

Exercise 1

Use `lapply` and an anonymous function to find the coefficient of variation (`sd(x)/mean(x)`) on each column of the dataset.

Exercise 2

Suppose we want to predict `mpg` (consumption) from the regressors `disp` (engine size) and `wt` (weight). We test several linear models whose corresponding `R` formulae are - `mgp ~ 1 + disp` - `mgp ~ I(1/disp)` - `mgp ~ 1 + I(1 / disp) + wt`
Use `lapply` to adjust linear models with such formulaes and extract the coefficient of determination (R^2).

Cousins of `lapply`: vector ouputs

`sapply` and `vapply` extend `lapply` by simplifying the output to an atomic vector rather than a list - `sapply` guesses the types - `vapply` uses a user argument

```
sapply(datasets::iris, is.numeric)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

```
vapply(datasets::iris, is.numeric, logical(1))
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

```
unlist(lapply(datasets::iris, is.numeric))  ## equivalent
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

Remarks

- `sapply` is very permissive and always find a way to output something
- `vapply` will throw an error if the required operation is not possible

$\rightsquigarrow$ `sapply` should be avoid when writing function as it can mask and propagate important error.

Cousins of lapply: repeated evaluation

Sometimes a loop repeats the same operation that does not need the iteration label, for instance when one replicate several simulation involving randomness.

Definition

```
replicate(number_of_repetition, {expression}, simplify = "array")
```

Exercise

Use `replicate` to generate 100 bootstrap samples of the `iris` dataset stocked as a list.

```
n <- nrow(datasets::iris)
boots <- replicate(100, datasets::iris[sample.int(n, n, replace=TRUE), ], simplify = FALSE)
```

Cousins of `lapply`: repeated evaluation

Sometimes a loop repeats the same operation that does not need the iteration label, for instance when one replicate several simulation involving randomness.

Definition

```
replicate(number_of_repetition, {expression}, simplify = "array")
```

Exercise

Use `replicate` to generate 100 bootstrap samples of the `iris` dataset stocked as a list.

```
n <- nrow(datasets::iris)
boots <- replicate(100, datasets::iris[sample.int(n, n, replace=TRUE), ], simplify = FALSE)
```

Cousins of `lapply`: multiple inputs I

Caveat

`lapply` and its vector-input variants like `sapply` and `vapply` works for a single vector to loop over.

Example

Suppose we want to compute the weighted mean for series of 10 couple of vectors $(x, w) \in \mathbb{R}^{2\times 100}$:

```
x <- replicate(5, rnorm(100      ), simplify = FALSE)
w <- replicate(5, runif(100, 0, 1), simplify = FALSE)
```

Solution

It is possible to handle this problem with `lapply`.

```
unlist(
  lapply(seq_along(x), function(i) {
    weighted.mean(x[[i]], w[[i]])
  })
)
```

```
## [1] -0.06751944 -0.15072211 -0.07694802 -0.05949277  0.17029073
```

Cousins of `lapply`: multiple inputs I

Caveat

`lapply` and its vector-input variants like `sapply` and `vapply` works for a single vector to loop over.

Example

Suppose we want to compute the weighted mean for series of 10 couple of vectors $(x, w) \in \mathbb{R}^{2\times 100}$:

```
x <- replicate(5, rnorm(100      ), simplify = FALSE)
w <- replicate(5, runif(100, 0, 1), simplify = FALSE)
```

Solution

It is possible to handle this problem with `lapply`.

```
unlist(
  lapply(seq_along(x), function(i) {
    weighted.mean(x[[i]], w[[i]])
  })
)
```

```
## [1] -0.06751944 -0.15072211 -0.07694802 -0.05949277  0.17029073
```

Cousins of `lapply`: multiple inputs II

A more elegant and readable solution is to rely on `mapply` or `Map`, which let the possibility to pass several vectors to jointly loop over:

Syntax

```
Map(FUN, ...)
mapply(FUN, ..., MoreArgs = NULL, simplify = TRUE, USE.NAMES = TRUE)
```

Remarks

- `Map` and `mapply` are equivalent
- `Map` is more consistent with `lapply` (do not simplify, a bit simpler)

Exercise

Use `Map` and `mapply` to compute the weighted means for the set $\{(x_i, w_i)\}_{i=1,\dots,10}$

```
unlist(Map(weighted.mean, x, w))

## [1] -0.06751944 -0.15072211 -0.07694802 -0.05949277  0.17029073
```

Cousins of `lapply`: multiple inputs II

A more elegant and readable solution is to rely on `mapply` or `Map`, which let the possibility to pass several vectors to jointly loop over:

Syntax

```
Map(FUN, ...)
mapply(FUN, ..., MoreArgs = NULL, simplify = TRUE, USE.NAMES = TRUE)
```

Remarks

- `Map` and `mapply` are equivalent
- `Map` is more consistent with `lapply` (do not simplify, a bit simpler)

Exercise

Use `Map` and `mapply` to compute the weighted means for the set $\{(x_i, w_i)\}_{i=1,\dots,10}$

```
unlist(Map(weighted.mean, x, w))
```

```
## [1] -0.06751944 -0.15072211 -0.07694802 -0.05949277  0.17029073
```

Functional for `matrix` and `arrays`: `apply`

Definition

Applies a function along a dimension of an array (row/columns of matrix).

```
apply(array, dim, FUN, ...)
```

Example

```
mat <- matrix(1:6, 2, 3)
apply(mat, 2, max)
```

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

```
arr <- array(1:12, c(2,3,2))
apply(arr, 3, colMeans)
```

```
##      [,1] [,2]
## [1,]  1.5  7.5
## [2,]  3.5  9.5
## [3,]  5.5 11.5
```

Other array functionals: sweep

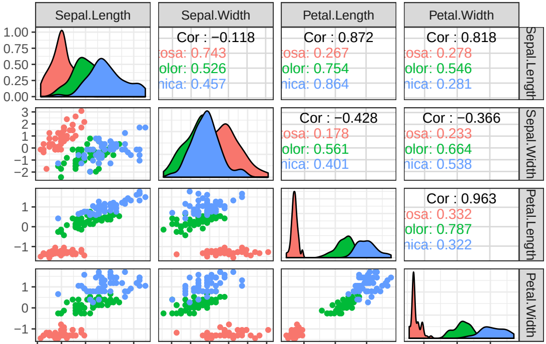
Definition

'Sweep out' a summary statistic along a dimension of an array for a given function

```
sweep(array, dim, stat, FUN, ...)
```

Example

```
out <- sweep(iris[, 1:4], 2, colMeans(iris[, 1:4]), "-") ## center
out <- sweep(out, 2, sqrt(colSums(out^2)/(nrow(out) - 1)), "/") ## scale
iris_sc <- data.frame(out, Species = iris$Species)
GGally::ggpairs(iris_sc, columns = 1:4, ggplot2::aes(colour = Species))
```



Other array functionals: `outer`

Definition

```r
outer(array1, array2, FUN, ...)
```

Example

The more basic example is the kronecker product, but `FUN` can be anything!

```r
outer(1:3, 1:4, "*")
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
```

```r
1:3 %o% 1:4
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
```

Grouped functional: `tapply`

Definition

Applies a function on a vector partioned by a factor: combine a `split` + `lapply` operation

Example

```
with(iris, tapply(Sepal.Length, Species, mean)) # readable
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```
with(iris, lapply(split(Sepal.Length, Species), mean)) # still ok (I think)
```

```
## $setosa
## [1] 5.006
##
## $versicolor
## [1] 5.936
##
## $virginica
## [1] 6.588
```

```
stat <- c() # less readable (and naming is lost)
for (l in levels(iris$Species))
  stat <- c(stat, mean(iris$Sepal.Length[iris$Species == l]))
stat
```

```
## [1] 5.006 5.936 6.588
```

Functional for working on lists: Reduce

Definition

'Reduce' uses a binary function to successively combine the elements of a given vector

```
Reduce(FUN, vector, init, right = FALSE, accumulate = FALSE)
Reduce(f, 1:3) <-> f(f(1,2))
```

Exercise

Consider a list of vectors of integer.

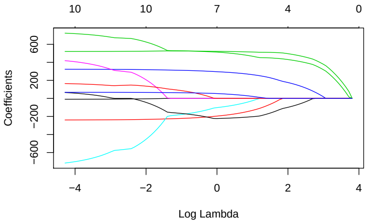
```
my_list <- replicate(10, sample.int(20, 10, replace = TRUE), simplify = FALSE)
```

Find which values occur in every element of the list visited so far, starting from the end.

Example: "jacknifing" a lasso solution path

A single Lasso fit of the diabete data set

```
library(glmnet)
library(lars) # the diabetes data set (part of the lars package)
data(diabetes)
x <- diabetes$x; y <- diabetes$y; n <- length(y)
lasso <- glmnet(x,y)
plot(lasso, xvar = "lambda")
```



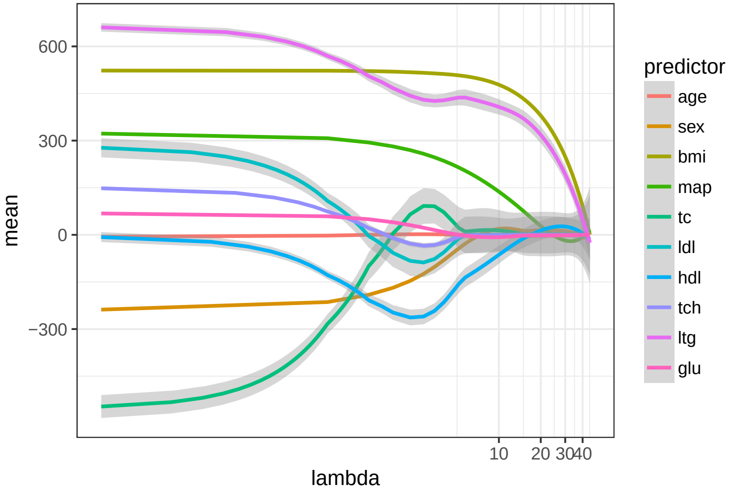
Compute the regularization paths for all subsets, removing one individual at once

```
paths <- lapply(1:n, function(i) {
    glmnet(x[-i, ], y[-i], lambda = lasso$lambda)$beta
})
```

A Reduce example: "jacknifing" a lasso solution path II

Computing the envelop around the average regularization path with Reduce

```
mean.path <- Reduce("+", paths)/n
sdev.path <- sqrt(Reduce("+", lapply(paths, function(path) path**2))/n -
                  mean.path**2)
```



Mathematical functionals I

- functionals are quite natural mathematics!
- R includes a couple of mathematical functionals for univariate functions

integrate

Finds the area under the cruve defined by a function:

```
integrate(sin, 0, pi)
```

```
## 2 with absolute error < 2.2e-14
```

optimise and optim (multivariate)

Find the location of lowest value of the function

```
optimise(sin, c(0, 2*pi))
```

```
## $minimum
## [1] 4.712391
##
## $objective
## [1] -1
```

$\rightsquigarrow$ optim is much more powerful but is out of the scope of this course

Mathematical functionals II

uniroot

Finds where the function hits zero

```
uniroot(sin, c(pi/2, 3*pi/2))
```

```
## $root
## [1] 3.141593
##
## $f.root
## [1] 1.224647e-16
##
## $iter
## [1] 2
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

References

R Core Team. (2017). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Wickham, H. (2014). *Advanced r*. CRC Press. Retrieved from http://adv-r.had.co.nz/