# R programming, a gentle introduction

M1 IREF, M1 ROAD

Laurent Bergé

B$_X$SE, University of Bordeaux

Fall 2022

## Your choice:

- proceed with the introduction
- skip to the main outline

## Who am I?

### Laurent Bergé

- Assistant Prof., B$_X$SE, Univ. of Bordeaux
- @lrberge
- laurent.berge@u-bordeaux.fr

## What do I do?

### My fields

- Applied economics ( $=$ Data + methods)
- Economics of Innovation ( $=$ Large data)
- (a bit of) Statistics ( $=$ Computational methods)

## R and me

### The story

- I've met R during my master in 2010
- since then... it's a love story

### The outcome of our relationship

- 7 packages, 6 of which are public
- the packages cover:
    - econometrics
    - data handling
    - statistical models
    - package development
    - graphics

## Why R?

- it is free
- it is a programming language (e.g. $\neq$ Stata), which includes: a coherent & smart syntax, loops and conditions $\Rightarrow$ you do not need more!
- extremely easy integration of low level languages as C++ (Rcpp, cpp11)
- it has a super (super) smart IDE (Rstudio)

## Pros and Cons

### Cons

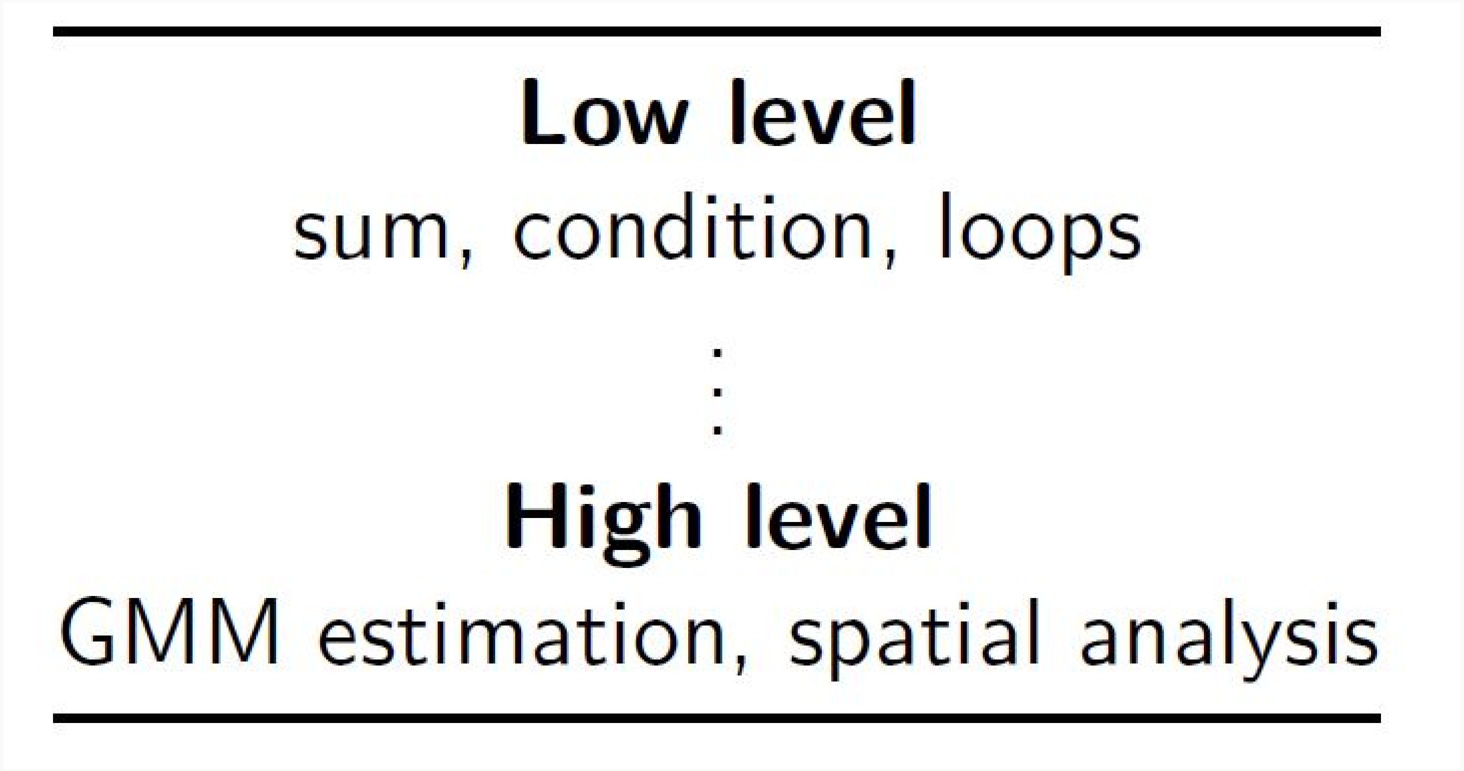
- Many functionalities depend on user-created functions $\Rightarrow$ quality control depends on the goodwill (and time) of the programmer
- It is not "point and click" $\Rightarrow$ bigger learning cost

## Pros and Cons

### Pros

- huge active community $\Rightarrow$ many answers to your questions around
- you can easily code whatever you want (great for simulation and complex data management)
- it has some very (very) smart and convenient coding features
- very well integrated document authoring system with Rstudio: to create webpages, reports, presentations
- Shiny: https://trackingpatentconcepts.shinyapps.io/shiny_diffusionApp/
- you can create your own packages easily

## Where is this course?



**Low level**

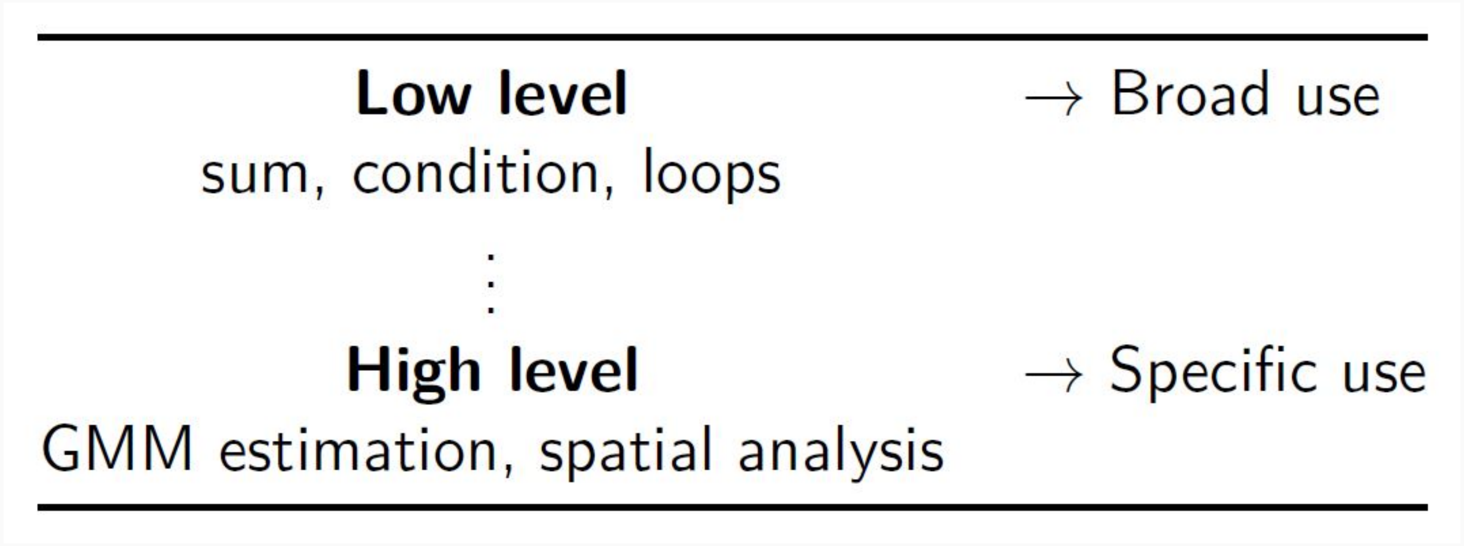
sum, condition, loops

⋮

**High level**

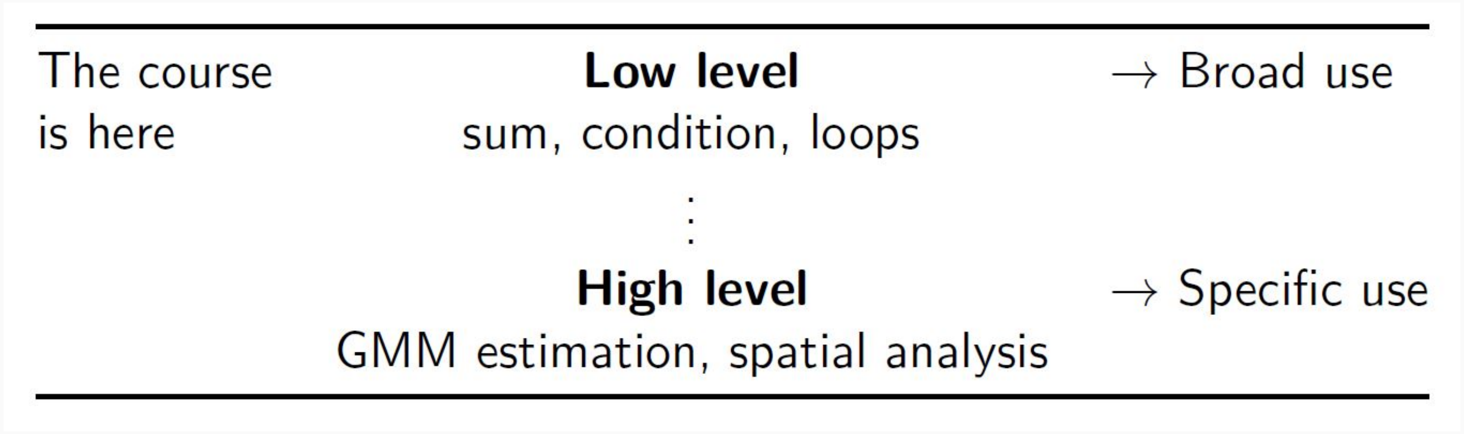
GMM estimation, spatial analysis

## Where is this course?



| | |
|---|---|
| **Low level** | $\rightarrow$ Broad use |
| sum, condition, loops | |
| $\vdots$ | |
| **High level** | $\rightarrow$ Specific use |
| GMM estimation, spatial analysis | |

## Where is this course?



| | | |
|---|---|---|
| The course is here | **Low level** sum, condition, loops | $\rightarrow$ Broad use |
| | $\vdots$ | |
| | **High level** GMM estimation, spatial analysis | $\rightarrow$ Specific use |

## Language

A programming language is really a language:

- words (thousands of functions to know),
- grammar (specific syntax),
- style ("good" vs "bad" code).

## Aim

- This course will strive to:
  - give you the keys to programming in R
  - help you to manage data efficiently
  - make OK graphs

## Outline

1. Basic R programming
2. Data management
3. Statistics (basic)
4. Graphics

## Basic R programming

## What R is

A big calculator

```
1 + 1
#> [1] 2
sqrt(4)
#> [1] 2
exp(log(2))
#> [1] 2
2**2
#> [1] 4
2^2
#> [1] 4
pi
#> [1] 3.141593
```

## What R is

A programming Language

- loops
- conditions
- coherent syntax
- smart function definition

## What R is

R likes vectors and matrices:

- Easy & efficient handling of matrix operations $\Rightarrow$ think matrices!

## What if I need help?

- Every user-level function is documented. The quality of the documentation depends on the author of the function though.
- type `?function` to get the help page of the function

## Syntax in R

**Rule 1**

The syntax rules you shall obey, there is no other way.

In concrete terms:

- R is case sensitive $\Rightarrow$ `x1` is different from `X1` and they will be treated as two different variables
- every symbol has a specific meaning: You shall not use a parenthesis for a bracket! $( \neq \{ \neq [$
- everything open must be closed (paren./bracket)
- 1 line = 1 instruction. And nothing more!
- use one character sign for another and the code will break

## Syntax in R: The two other rules of programming

**Rule 2**

Most errors are syntax errors.

**Rule 3**

If you think you have found a bug in the software, please refer to Rule 2.

## A word of caution: This course is not about tidyverse

If you knew R before this course, you must have heard of the `tidyverse`.

This is a set of functions, for data management and much more, created by talented developers which changed the R paradigm by focusing on user-friendliness.

### The problem

`tidyverse` introduced a lot of *non-standard evaluation* to make the typing much lighter and intuitive.

The tradeoff is that it increased the complexity to program with it.☆

And if you're starting with R, it will give you terrible habits and expectations; and you won't be able to understand non-tidyverse users' code!

Before using *non-standard evaluation*, you should first properly understand what *standard evaluation* is!

☆: It's a bit like Stata: best-in-class for end-users, but as regards programming.. I wouldn't recommend it!

## Programming 001:

functions, a refresher

# What is a function?

- input $\longrightarrow$ function $\longrightarrow$ output

```
exp(2)
#> [1] 7.389056
```

- `exp()` is a (simple) function that takes only one numeric argument as input and gives one numeric output:
  2 $\longrightarrow$ `exp` $\longrightarrow$ 7.389056

- While there can be multiple arguments as inputs, the output must be a unique object!

```
round(pi, 2)
#> [1] 3.14
```

- Here the function `round()` takes two arguments as inputs:
  $(\text{pi}, 2)$ $\longrightarrow$ `round` $\longrightarrow$ 3.14

# Functions in R

- To call a function: `function_name(arg1, arg2, ...., argN)`
- Each argument has a name:

```
round(x = pi, digits = 1)
#> [1] 3.1
```

- You can omit the argument names: if so R automatically inserts argument names following the function's definition order.

```
round(pi, 1)
#> [1] 3.1
round(digits = 1, pi)
#> [1] 3.1
```

- When arguments have default values, you can omit them:

```
round(pi)
#> [1] 3
round(pi, 4)
#> [1] 3.1416
```

## Assignation

- R typing is dynamic: you don't have to declare an object before creating it (contrary to c)
- To assign a value to a variable use either: "`=`" or "`<-`"

```r
x  =  2
x  <-  2
```

- There is a slight difference between the two methods (I'll come back to it later)
- Please use "`=`", like in any other programming language★

★: There is no objective reason, except fad and custom (if you think they are objective), to use `<-`.

## R Programming 101

## Is your friend your friend?

### Google is your friend

By googling on the internet, you can quickly find code snippets to do quite complex analysis. ML, spatial analysis, you name it.

Even if you're a bloody beginner.

### The problem of this friend

What's the point of *"knowing"* how to do complex things if you don't know how to do basic things?

This will generate a lot of frustration when you'll want to perform simple things (export/loop/select/etc).

### My plot, revealed

OK, in fact I just wanted to motivate a very tedious outline in the next slide! ;-)

# R: The stuff you just must know

**What we'll cover**

- vectors
- matrices
- lists
- data frames

**What we'll see on the way**

- subsetting (vec, mat, list, df)
- loops and conditions
- writing functions
- R pitfalls (recycling, factors)

Q: Does this look tedious?

A: Absolutely!

Q: Is this necessary?

A: Nope, **indispensable**!

## Vectors

## Basic types - Vectors

- To create a vector, use the function `c()`:

```
x = c(1, 5, 6)
x
#> [1] 1 5 6
```

- You can also create vectors of character strings:

```
s = c("bonne", "nuit", "les petits")
s
#> [1] "bonne"      "nuit"       "les petits"
```

- You can add elements to an existing vector with `c()`:

```
y = c(x, 8, 9)
y
#> [1] 1 5 6 8 9
```

## Regular vectors

There are some tools to create regular vectors:

1. the colon "`:`"
2. `rep()`
3. `seq()`

## Regular vectors: `:`

- Use a colon to create sequences of integers with unitary increments:

```
1:5
#> [1] 1 2 3 4 5
-2:2
#> [1] -2 -1  0  1  2
```

Q: What happens here?

```
1:2+1
```

A:The colon operator has **precedence** over other operations!

```
#> [1] 2 3
```

## Regular vectors: `rep()`

- Use `rep()` to replicate values or vectors:

```r
rep(1, 3) # identical to rep(x = 1, times = 3)
#> [1] 1 1 1
rep(1:2, 3)
#> [1] 1 2 1 2 1 2
```

- You can use `rep`'s argument `each`:

```r
rep(1:2, each = 3)
#> [1] 1 1 1 2 2 2
```

- Argument `times` can be a vector of the same length as argument `x`:

```r
rep(1:2, 2:3)
#> [1] 1 1 2 2 2
```

# Regular vectors: `seq()`

- Use `seq()` to create regular sequences:

```
seq(1, 5, 1) # identical to seq(from = 1, to = 5, by = 1)
#> [1] 1 2 3 4 5
seq(1, 5, by = 2)
#> [1] 1 3 5
```

- Say you want to create a vector from 1 to 5 with 7 equidistant points, use argument `length.out`:

```
seq(1, 5, length.out = 7)
#> [1] 1.000000 1.666667 2.333333 3.000000 3.666667 4.333333 5.0000
```

## Operations with vectors

Operations between scalars and vectors are term to term:

```
x = 1:3
x + 1
#> [1] 2 3 4
x * 2
#> [1] 2 4 6
x**3 # equivalent to x^3
#> [1]  1  8 27
log(x)
#> [1] 0.0000000 0.6931472 1.0986123
```

Operations between vectors are also term to term:

```
y = 10**(1:3) # equivalent to y = c(10, 100, 1000)
x + y
#> [1]   11  102 1003
x * y
#> [1]   10  200 3000
```

## Operations with vectors

Some functions on vectors:

```
x = 1:4
length(x) # nber of elements of x
#> [1] 4
mean(x)
#> [1] 2.5
sd(x)
#> [1] 1.290994
var(x)
#> [1] 1.666667
sum(x)
#> [1] 10
cumsum(x) # cumulative sum
#> [1]  1  3  6 10
diff(x) # next element - current element
#> [1] 1 1 1
```

## Exercises: Vectors

Create the following vectors:

- $(-2, -1, 0, 1, 2)$ in two ways
- $(1, 1.5, 2, 2.5, 3, 3.5)$
- $(4, 4, 4, 3, 3, 2)$
- $(3, 2, 1, 1, 2, 3)$

## Subsetting I

Say you have a vector $x$ and you want to select specific elements from it.

Variable $index$ represents the elements you want to select from $x$.

The syntax is as follows (note the square brackets):

```
x[index]
```

The $index$ can be of *only* three types, either:

1. a vector of integers (whatever its length),
2. a vector of character strings (whatever its length),
3. a *logical* vector of the exact same length as $x$.

## Subsetting: Example

Ex: You want to select the 4th and 5th elements:

```
x = 1:5
# two types of indexes yielding the same results:
index_nb = c(4, 5)
index_logic = c(FALSE, FALSE, FALSE, TRUE, TRUE)
x[index_nb]
#> [1] 4 5
x[index_logic]
#> [1] 4 5
```

## Subsetting

With an index in number, you can take several times the same value from $x$:

```
x = 5:1
x[c(4, 4, 5, 1, 1)]
#> [1] 2 2 1 5 5
```

With a logical index, *you just can't*.

You can also use negative numbers to drop observations. If so, *all numbers of the index must be negative*:

```
x[-1] # drops first element
#> [1] 4 3 2 1
x[-(3:length(x))] # drops the third to the last element
#> [1] 5 4
```

## Subsetting II: Using character strings

If the vector has names: then you can use a character vector as an index:

```
x = 1:5
names(x) = letters[1:5]
x
#> a b c d e
#> 1 2 3 4 5
x[c("b", "b", "d")]
#> b b d
#> 2 2 4
```

## Subsetting

- At first sight, the logical vector looks like impractical, however it is the one you gonna use the most!
- Why? Because logical operations on vectors yield logical vectors.

# Logical operations and subsetting

A logical vector is returned when you perform logical operations on a vector.

- lower than: `<`, lower or equal: `<=`, greater than: `>`, greater or equal: `>=`, equal: `==`, different: `!=`

To combine the results of several logical operations:

- AND: `&` (ampersand), OR: `|` (pipe), NOT: `!`

## Logical operations examples

```
x = 1:5
x > 2
#> [1] FALSE FALSE  TRUE  TRUE  TRUE
x != 5
#> [1]  TRUE  TRUE  TRUE  TRUE FALSE
x > 2 & x != 5
#> [1] FALSE FALSE  TRUE  TRUE FALSE
!(x > 2 & x != 5)
#> [1]  TRUE  TRUE FALSE FALSE  TRUE
```

## Logical operations and subsetting

**Exercize**

Use logical vectors to find an approximation of the probability that:

$$|x| > 3, \quad x \sim N(0,1)$$

```r
set.seed(1) # for replicability
x = rnorm(1000) # 1000 draws from N(0,1)
is_large = abs(x) > 3
sum(is_large) # nber of times abs(x) > 3
#> [1] 3
x[is_large] # we see the 'large' elements
#> [1] -3.008049  3.810277  3.055742
```

For information: $2 \times \Phi(-3) = 0.0027$, or in `R` terms:

```r
2 * pnorm(-3)
#> [1] 0.002699796
```

## Note on Logicals I

Logicals are just 0/1-like numbers, you can use them in arithmetic operations.

```
a = c(TRUE, FALSE)
a
#> [1]  TRUE FALSE
a + 1 # logical is converted to numeric
#> [1] 2 1
```

# Note on logicals II

Beware: logical operations have the lowest precedence (i.e. they come last.)

**Example:** we want to set the values of x to 0 if y is negative.

## Question

What's the result of:

```
x = 1:5
y = -2:2
x * y>0
```

```
x * y>0
#> [1] FALSE FALSE FALSE  TRUE  TRUE
```

```
x * (y > 0)
#> [1] 0 0 0 4 5
```

# Exercises: subsetting

- The function `which()` returns the indexes of a logical vector which are `TRUE`:

```
which( c(TRUE, FALSE, FALSE, TRUE) )
#> [1] 1 4
set.seed(1)
x = rnorm(6)
which(x > 0)
#> [1] 2 4 5
```

With a pen and paper:

1. Suggest a line of code that does the same as `which(x > 0)` but without using it.
2. Without using any function, create `x_abs` which is the absolute value of `x = -3:3` (you'll only need subsetting).

## Loop: for

The syntax of a `for` loop is:

```r
for(index in vector){
    # do stuff
    # the variable 'index' will successively take
    # each value in 'vector'
}
```

```r
for(i in c("Monique", "Esteban", "Francis")){
    # cat: function used to print msg on console,
    # \n: means return to the line (otherwise
    # everything is in one line)
    cat("Hello ", i, "!\n", sep = "")
}
#> Hello Monique!
#> Hello Esteban!
#> Hello Francis!
```

## While

The syntax for a `while` loop:

```
while(condition){
  # do stuff
}
```

```
i = 2
while(i <= 100){
  cat(i, "^2 = ", i^2, "\n", sep = "")
  i = i**2
}
#> 2^2 = 4
#> 4^2 = 16
#> 16^2 = 256
cat("i out of the loop is", i)
#> i out of the loop is 256
```

*Of course, try to avoid infinite loops!*

## Break and next

Use `break` to escape a loop (either `for` or `while`):

```r
a = 5
while(TRUE){
  a_next = a^a
  if(!is.finite(a_next)){
    break
  }
  a = a_next
}
cat(a, " is finite but ", a, "^", a, " = ", a_next, sep = "")
#> 3125 is finite but 3125^3125 = Inf
```

## Break and next

In a loop, to go to the next iteration, use `next`:

```
for(i in 1:100){
  if(i < 99){
    next
  }
  print(i)
}
#> [1] 99
#> [1] 100
```

## Conditions: I

The syntax for a condition is as follows:

```
if(condition_1){
    # stuff
} else if(condition_2){
    # stuff
} else {
    # stuff
}
```

## Conditions: II

Any condition MUST be of length 1! i.e. you cannot use logical *vectors*.

```
x = 1:5
# BAD:
if(x == 1){
    # meaningless! An error will be raised.
    # (In R < 4.0.0 it only led to a warning.)
}

# GOOD:
if(x[1] == 1){
    # Now it's clear what you test
}

if(any(x == 1)){
  # any() returns TRUE if there is at least
  # one TRUE in a logical vector
}

if(all(x == 1)){
  # all() returns TRUE if all the values are TRUE
}
```

## Logical operations in conditions

Say you want to test whether the 5th element of a vector is greater than 12:

```
if(x[5] > 12) print("ok")
```

**Problem**

If `x` is of length lower than 5 $\Rightarrow$ problem.

**Solution**

Use a logical `and`:

```
if(length(x) >= 5 & x[5] > 12) print("ok")
```

## Logical operations in conditions: Short-circuit

### ~~Solution~~ Problem

```r
if(length(x) >= 5 & x[5] > 12) print("ok")
```

*It works, but not for the good reasons!!!!!*

### Question

What do you think is going to happen?

```r
x = 1:3
if(length(x) >= 5 & stop("This has been evaluated")) print("ok")
```

```r
x = 1:3
if(length(x) >= 5 & stop("This has been evaluated")) print("ok")
#> Error in eval(expr, envir, enclos): This has been evaluated
```

## Logical operations in conditions: Short-circuit

~~Solution~~ Problem

```r
if(length(x) >= 5 & x[5] > 12) print("ok")
```

The previous code will:

1. evaluate `length(x) >= 5`
2. evaluate `x[5] > 12`
3. aggregate the two logical elements with `&`

***This is not what we want!***

## Logical operations in conditions: Short-circuit

What do we want?

Evaluate `length(x) >= 5`

1. if `TRUE`, evaluate `x[5] > 12` and return its value
2. if `FALSE`, return `FALSE`

This means that we don't want `x[5]` to be evaluated if the length of `x` is lower than 5!

**Solution (*the real one*)**

Use logical operators with short-circuit: `&&` and `||`:

```r
if(length(x) >= 5 && x[5] > 12) print("ok")
```

## Two types of logical operators

### The `&` and `|` operators

1. they always evaluate left and right
2. they are vectorised

### The `&&` and `||` operators

1. they evaluate the right side only if needed (i.e. *conditionally*)
2. they accept only scalars on both sides!

## Conditions: Weird behavior?

### Question

*Now knowing how `&` works.*

Why does this produces and error:

```
x = 1:3
if(x[5] > 12) print("ok")
#> Error in if (x[5] > 12) print("ok"): missing value where TRUE
```

… but not this?

```
x = 1:3
if(length(x) >= 5 & x[5] > 12) print("ok")
```

A: It has to do with how `&` deals with missing values.

## Conditions: Behavior with missing values

The operator `&` and NAs:

```
TRUE & NA
#> [1] NA
FALSE & NA
#> [1] FALSE
```

The operator `|` and NAs:

```
TRUE | NA
#> [1] TRUE
FALSE | NA
#> [1] NA
```

## Conditions: Behavior with missing values

Beware the behavior of NAs when subsetting!!!!

```
set.seed(1)
x = rnorm(8)
x[c(3, 7)] = NA
y = round(rnorm(8), 1)
y[x > 0]
#> [1] -0.3   NA  0.4 -0.6   NA  0.0
```

Remember that `FALSE & NA` leads to `FALSE`? You need to use `is.na` to identify the NAs:

```
y[!is.na(x) & x > 0]
#> [1] -0.3  0.4 -0.6  0.0
```

# Conditions: Behavior with missing values

*NAs can be sneaky!!*

Here say we generate two random variables: `x` along a uniform law, `y` along a Normal law, and we create `z`, equal to `y` power `x`.

Finally, we subset `y` on the values taken by `z`.

```
set.seed(1)
x = runif(8)
y = round(rnorm(8), 1)
z = y ^ x
# we want the y's for which z >= 0.7
y[z >= 0.7]
#> [1] 0.3  NA 0.7 0.6  NA 1.5
```

The right way is:

```
y[!is.na(z) & z >= 0.7]
#> [1] 0.3 0.7 0.6 1.5
```

## Conditions: Example

```r
names = c("Monique", "Esteban", "Francis")
for(i in 1:3){
  if(i == 2){
    text = "Not hello "
  } else {
    text = "Hello "
  }
  cat(text, names[i], "!\n", sep = "")
}
#> Hello Monique!
#> Not hello Esteban!
#> Hello Francis!
```

## Loops and conditions: comment

For single instruction loops and conditions, you can omit the brackets:

```r
for(i in 1:100) if(i == 55) print("55 is reached")
#> [1] "55 is reached"
```

Although it looks like two operations are executed in the `for` loop, it is really only one (the `if`).

*I do NOT advise* using this shorthand $\Rightarrow$ the code looses in clarity.

Yet useful for QnD[★] stuff.

★: Quick and dirty.

# Exercise: Loops and conditions

With a pen and paper:

1. Compute the mean of `x = 1:5` with a loop.
2. Compute the exponential of 1 with a loop. Remind that

   $$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

   Use the function `factorial()` and go only until `i = 20`.
3. Do the previous exercise without loop.
4. Discover for which integer `factorial` becomes infinite. Do it twice: with a `for` and then a `while` loop.
5. Find the first divisor of 1234567 (use `%%` to get the rest of the Euclidian division$^\star$).
6. Now let's apply the solutions in Rstudio.

$^\star$: ex: `14 %% 5` yields 4, `8 %% 3` yields 2.

## Random Rstudio tips

- To run current line: ctrl + enter
- To comment / uncomment: ctrl + shift + c
- To create sections, add "####" to the end of a comment (ex: # Section 1 ####).
- ctrl + alt + up or down: duplicates the current line
- ctrl + alt + click: duplicates the cursors
- Create your own macros: https://rstudio.github.io/rstudioaddins/

## Matrices

## Matrices I

To create a $2 \times 2$ matrix full of ones:

```
matrix(data = 1, nrow = 2, ncol = 2)
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    1    1
```

## Matrices II

Let's create a matrix with numbers from 1 to 4:

```
matrix(1:4, 2, 2)
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

R fills the matrix by columns. To fill it by row:

```
matrix(1:4, 2, 2, byrow = TRUE)
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
```

# Matrices III

You can create matrices by "binding" vectors, using functions `rbind` and `cbind`:

```
rbind(1:3, 3:1) # row bind
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    3    2    1
cbind(1:3, 3:1) # column bind
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    2
#> [3,]    3    1
cbind(1:2, 2:1, 3:4, 4:3) # you can have any number of args
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    2    1    4    3
```

## Matrix operations I

Like for vectors, all operations are term to term:

```
X = matrix(1:4, 2, 2)
X + 2
#>      [,1] [,2]
#> [1,]    3    5
#> [2,]    4    6
X ** 2
#>      [,1] [,2]
#> [1,]    1    9
#> [2,]    4   16
exp(X)
#>          [,1]     [,2]
#> [1,] 2.718282 20.08554
#> [2,] 7.389056 54.59815
```

## Matrix operations II

Even when you multiply by a matrix:

```
X = matrix(1:4, 2, 2)
Y = matrix(1, 2, 2)
X * Y
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

Does vector $\times$ matrix multiplication work?

```
Y * 1:4
```

## Matrix operations III

Yes it works:

```
Y * 1:4
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

Weird behavior, isn't it? When a vector multiplies a matrix, R first (kind of) transforms the vector into a matrix of same dimensions before applying the operation.

Here's the logic for `Y * 1:4`:

1. dimension of `Y` is $2 \times 2$
2. `1:4` is transformed into `matrix(1:4, 2, 2)`
3. the following operation is performed: `Y * matrix(1:4, 2, 2)`

## Matrix operations IV

To perform matrix multiplication, we need to use the following symbol: "`%*%`"

```
X = matrix(1:4, 2, 2)
Y = matrix(1, 2, 2)
X %*% Y
#>      [,1] [,2]
#> [1,]    4    4
#> [2,]    6    6
X %*% (1:2)
#>      [,1]
#> [1,]    7
#> [2,]   10
```

Now the vector MUST be of the appropriate dimensions:

```
X %*% 1:4
#> Error in X %*% 1:4: non-conformable arguments
```

# Matrix operations V

- To transpose a matrix: `t(X)`
- to get $X'X$, `crossprod(X)` is faster than `t(X) %*% X`
- to get $XX'$, `tcrossprod(X)` is faster than `X %*% t(X)`
- to get $X^{-1}$: `solve(X)`
- to get the sum of each line: `rowSums(X)`
- to get the sum of each column: `colSums(X)`
- to get the dimension of a matrix: `dim(X)`
- to get the nber of rows (columns) of a matrix: `nrow()` (`ncol()`)

## Main problem - OLS regression

1. Generate data (100 points) according to the following relation:

$$y_i = 2 + 5x_i + \epsilon_i \quad x_i \sim N(0,1) \quad \epsilon_i \sim N(0,1)$$

2. Estimate the coefficients of the constant and of $x$. Recall that:

$$\hat{\beta} = (X'X)^{-1}X'Y$$

## Random number generation (briefly)

- Uniform distribution: `runif(n)`
- Normal distribution: `rnorm(n)`
- Other distributions: `?Distributions`
- Integers: `sample(n, k, replace = TRUE)` draw $k$ numbers among $n$ with replacement.
- To replicate random number creation: `set.seed(m)`, with $m$ a number (when you launch R, default is roughly like `set.seed(Sys.time())`, so that everytime the seed will be different.)

## R major pitfall: recycling

## !!!! Recycling !!!!

What happens if I do:

```r
x = 1:5
x[c(TRUE, FALSE)]
```

```
#> [1] 1 3 5
```

It works! yet it's not good news...

**Explanation**

If an operation requires a vector of length $n$ and you give a vector of length $m < n$, R tries hard to make the second vector match length $n$, so that the operation works.

Basically, it usually replicates the vector until it fits.

**DANGER**

You may not notice mistakes!

## Recycling: examples

```r
rep(1, 6) + 0:1
```

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

```r
matrix(1:3, 3, 2)
```

```
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    2
#> [3,]    3    3
```

```r
matrix(1:3, 2, 3)
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    3    2
#> [2,]    2    1    3
```

## Recycling: more examples

```r
matrix(1, 2, 2) + 0:1
```

```
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    2
```

```r
matrix(1, 2, 2) + 0:2 # Now lengths don't match => warning
```

```
#> Warning in matrix(1, 2, 2) + 0:2: longer object length is not a multiple of
#> shorter object length
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    1
```

## Back to matrices

## Subsetting matrices

Three ways to extract subsets of a matrix:

1. `X[index_row, index_column]` $\Rightarrow$ yields a matrix
2. `X[index_vector]` $\Rightarrow$ yields a vector
3. `X[index_matrix]` $\Rightarrow$ yields a vector

# Subs. mat.: row & col indexes

Both indexes `index_row` and `index_column` are similar to vector indexes: they can either be logical or numeric (or character).

Leaving an index empty means all rows/columns.

```
X = matrix(1:9, 3, 3, byrow = TRUE)
X[1, 2:3] # 1st line, two last columns => 1x2 mat.
#> [1] 2 3
# simplified by R to a vector
X[c(1,3), 2:3] # 1st & 3rd lines, two last cols => 2x2 mat.
#>      [,1] [,2]
#> [1,]    2    3
#> [2,]    8    9
X[1, ] # 1st line, all columns
#> [1] 1 2 3
X[X[, 1] <= 4, ] # all lines such that 1st element <= 4
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
```

## Subsetting matrices: Using vector indexes

You can use a logical/numeric vector going through all the elements of a matrix:

```r
X = matrix(1:9, 3, 3, byrow = TRUE)
X[X<5]
#> [1] 1 4 2 3
X[5] # the 5th element of the matrix
#> [1] 5
```

Logic? The matrix is ex ante transformed into a vector, then the subsetting is done.

Here is an example of what it does:

1. `X_tmp = as.vector(X)`
2. `X_tmp[X_tmp < 5]`

## Subsetting matrices: Using matrix indexes

Sub-setting with a matrix index (`index_matrix`).

`index_matrix`:

1. must be a two columns matrix
2. must contain only integers
3. each row refers to a matrix cell

```
X = matrix(1:9, 3, 3, byrow = TRUE)
# getting the diagonal:
index_matrix = cbind(1:3, 1:3) # 2 column matrix
X[index_matrix]
#> [1] 1 5 9
# getting the other diagonal
X[cbind(1:3, 3:1)]
#> [1] 3 5 7
```

# Matrix: Beware the default!

**Question**

What is the type of `y`:

```r
x = matrix(1:4, 2, 2)
y = x[, 1]
class(y)
```

```
#> [1] "integer"
```

***It's a vector!***

Subsets of matrices leading to something of dimension 1 (either row or column) lead to a conversion to vector. It may not be what you want!

## Matrix: Beware the default!

And the help-page explaining this default behavior is super hard to find!

You need to type: `?"["`.

There you find the solution:

```r
x = matrix(1:4, 2, 2)
y = x[, 1, drop = FALSE]
class(y)
#> [1] "matrix" "array"
class(x[, 1])
#> [1] "integer"
```

# Exercise: recycling

Let $X$ be the $10 \times 4$ matrix such that $X_{ik} = i \times k$.

We want to create $\tilde{X}$ such that $\tilde{X}_{ik} = X_{ik} - \bar{X}_k$, with $\bar{X}_k$ the average of the kth column of $X$.

1. Create $X$ in at least two ways.
2. Compute the column means in three ways: i) `colSums()`, ii) matrix multiplication, iii) `colMeans()`
3. Compute $\tilde{X}$.

## Lists

## Lists I

List are (sort of) vectors of objects that can be of any type.

```
a = list(1:5, c("je", "dors"), matrix(1:4, 2, 2))
a
#> [[1]]
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "je"   "dors"
#>
#> [[3]]
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

As we can see the list named `a` contains three elements: a numeric vector, a character vector and a matrix.

## Lists II

You can give names to the elements of a list ⇒ makes it clearer:

```
a = list(vec = 1:5, charvec = c("je", "dors"),
         mat = matrix(1:4, 2, 2))
a
#> $vec
#> [1] 1 2 3 4 5
#>
#> $charvec
#> [1] "je"   "dors"
#>
#> $mat
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

## Lists III

A list **is not** a vector. (Sorry for the pleonasm.)

You cannot perform regular operations with lists.
Why? Because it does not make sense.

```
a = list(1:5) # list made of just one vector
a * 2 # Error!
#> Error in a * 2: non-numeric argument to binary operator
```

## Lists IV

To add an element to an existing list:

```
a = list() # empty list
a$x = 1:5 # create the first element named x
a[["y"]] = 2 # creates a 2nd element named y
a[[3]] = 66 # creates a 3rd element NOT named
```

Note that `a$x` is *exactly* equivalent to `a[["x"]] = 1:5`.

## Subsetting lists

Two ways to extract elements from a list:

1. Methods returning a list:
    1. `a[index_vector]` ⇒ it always returns a list
2. Methods extracting ***one single element*** from a list:
    1. `a[[index_or_name]]`
    2. `a$name`

## Subsetting lists: Single square bracket method

```
a = list(numvec = 1:5, charvec = c("bon", "jour"))
a[1:2] # a list
#> $numvec
#> [1] 1 2 3 4 5
#>
#> $charvec
#> [1] "bon"  "jour"
a[c(FALSE, TRUE)]
#> $charvec
#> [1] "bon"  "jour"
a["charvec"] # still a list
#> $charvec
#> [1] "bon"  "jour"
a["numvec"] * 2 # it's a list => error is raised
#> Error in a["numvec"] * 2: non-numeric argument to binary operatc
```

## Subsetting lists: Single elements

Now assume you want to extract single elements to perform some operations with them:

```r
a = list(numvec = 1:5, charvec = c("bon", "jour"))
a[["numvec"]] * 2 # we use the vector -- and not the list
#> [1]  2  4  6  8 10
a$numvec # other way
#> [1] 1 2 3 4 5
a[[1]] # yet another
#> [1] 1 2 3 4 5
```

**Note**

When a list doesn't have names, you must use numeric/logical subsets.

## data.frame

## Limitation of matrices for data

So far we've seen only vectors, matrices and lists.

Matrices are good for doing numeric applications, yet they only accept numeric data.

Lists are interesting but their content can be too heterogeneous.

What can we use for data analysis?

## Data frames

A data frame is:

1. a **list** of **vectors**, *all of the same length*
2. the vectors can be of any type

It's a table of a given number of rows, each column being of a specific type.

```
data(iris) # internal R data used for examples
head(iris, 3) # first 3 rows
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
class(iris) # data.frame indeed
#> [1] "data.frame"
dim(iris) # dimension of the data
#> [1] 150   5
```

## Data frames: creation

You can create data frames with… `data.frame`:

```
data.frame(id = 1:3, name = c("John", "Mary", "Tim"))
#>   id name
#> 1  1 John
#> 2  2 Mary
#> 3  3  Tim
```

## Data.frames: General information

To get general information on the variables of a `data.frame`, you can use the function `summary`:

```
summary(iris)
#>   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
#>  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
#>  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
#>  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
#>  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
#>  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
#>  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
#>        Species  
#>  setosa    :50  
#>  versicolor:50  
#>  virginica :50  
#> 
#> 
#> 
```

# Data.frames: General information

The function `head` still works:

```
head(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

## Subsetting data.frames

Data frames are a special beast.

They can be subsetted either like lists, either like matrices.

## Subsetting data.frames: like matrices

```
iris[1:2, 2:3]
#>   Sepal.Width Petal.Length
#> 1         3.5          1.4
#> 2         3.0          1.4
iris[2:3, c("Sepal.Length", "Species")]
#>   Sepal.Length Species
#> 2          4.9  setosa
#> 3          4.7  setosa
```

## Subsetting data.frames: like lists

```r
head(iris[1], 2) # single square bracket: DF is returned
#>   Sepal.Length
#> 1          5.1
#> 2          4.9
head(iris[[1]]) # vector is returned
#> [1] 5.1 4.9 4.7 4.6 5.0 5.4
head(iris[["Sepal.Length"]]) # vector is returned
#> [1] 5.1 4.9 4.7 4.6 5.0 5.4
head(iris$Sepal.Length) # vector is returned
#> [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

## Subsetting: Beware the default!

As for matrices, if the subset of a `data.frame` leads to something of dimension 1, it is simplified into a vector.

You need to use the (hidden) argument `drop` to change the behavior:

```
class(iris[, 1])
#> [1] "numeric"

class(iris[, 1, drop = FALSE])
#> [1] "data.frame"
```

## Column and row names of a data.frame

A data.frame **must** have column names. To both access it and set it, use `names()`.

```
df = iris
names(df)
#> [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
names(df) = 1:ncol(df)
names(df)
#> [1] "1" "2" "3" "4" "5"
```

To access column names in a matrix, use `colnames()`.

## Row names of a data.frame

To access and set the row names of a data.frame, use `row.names()`:

```
df = data.frame(age = c(32, 12, 27), other = 3:1)
row.names(df) = c("Lara", "Julia", "Paul")
df["Paul", ]
#>      age other
#> Paul  27     1
```

To access row names in matrices, use `rownames()` (*no dot!*).

## Creation of new variables

```r
df = data.frame(name = c("Lara", "Julia", "Paul"),
                age = c(32, 12, 27))
df$young = df$age <= 18
# initialize the variable with some value
df$ageSq_if_young = df$age
# modify it conditionnaly
df$ageSq_if_young[df$young] = df$age[df$young] ** 2
# Delete variable:
df$age = NULL
```

## Apply function to DFs

Use the function `apply()` to apply a function to the rows/columns of a matrix/data.frame.

```
# MARGIN: 1: row, 2:column
apply(iris[, 1:4], MARGIN = 2, FUN = median) # get the median for 4
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>         5.80         3.00         4.35         1.30

head(apply(iris[, 1:4], 1, max)) # max value for each obs
#> [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

## R pitfall: factor variables

## Factor variables

In your R programming journey, you may encounter a strange creature: `factor`s.

```
class(iris$Species)
#> [1] "factor"
head(iris$Species)
#> [1] setosa setosa setosa setosa setosa setosa
#> Levels: setosa versicolor virginica
```

Factors are in fact a way to encode categorical data:

- they look like character but are in fact integers
- they're weird!

## Factors and data.frames: Beware the R version

**DANGER**

In R $<$ `4.0.0`, when creating a `data.frame`, the default of argument `stringsAsFactors` is `TRUE`!

Back in the days (check your R version!) upon the creation of a DF, character vectors are converted to factors:

```r
df = data.frame(name = c("Lara", "Julia", "Paul"),
                age = c(32, 12, 27), stringsAsFactors = TRUE)
df$name
#> [1] Lara  Julia Paul
#> Levels: Julia Lara Paul
class(df$name)
#> [1] "factor"
```

# Factor variables II

Illustration of the problem:

```r
df = data.frame(name = c("Lara", "Julia", "Paul"),
                stringsAsFactors = TRUE)
age = c(Lara = 32, Julia = 12, Paul = 27) # named vector
```

What does the following yields?

```r
df$age = age[df$name]
df
```

```
#>    name age
#> 1  Lara  12
#> 2 Julia  32
#> 3  Paul  27
```

***WTF???***

## Factor variables III

Why this behavior? Because factors are treated as integers!

Here's a sketch of what's done to the character variable `df$name`:

1. `name_sorted_unik = sort(unique(df$name))`
2. `dict = 1:length(name_sorted_unik)`
3. `names(dict) = name_sorted_unik`
4. `df$name = dict[df$name]`

```
unclass(df$name) # integers reflect the "character" order
#> [1] 2 1 3
#> attr(,"levels")
#> [1] "Julia" "Lara"  "Paul"
```

## Factor variables IV

To have the appropriate result:

```
df$age = age[as.character(df$name)]
df # we had to reconvert into character
#>    name age
#> 1  Lara  32
#> 2 Julia  12
#> 3  Paul  27
```

Factor variables can be useful in some context (we'll see when later).

## Functions

## Functions, a philosophy

- Think functions.
- Why thinking functions? If you have a piece of code that you use at least twice, it is worth making a function out of it so that the next time, you just use one line of code. Direct productivity gain.
- Big strenght of R: very easy to create *flexible* functions. Cost of making functions is low.
- Even if you think the problem you're dealing with is specific, try to see it as a special case of a broader context.
- This way, you'll be able to create a *broad* function that'll be able to deal with your specific problem but also many others.
- One drawback is that making broad functions requires abstract thinking. Yet it's usually worth the investment.

## Function syntax I

A function is just a set of instructions applied to objects given in input.

To create a function, the structure is as follows:

```r
functionName = function(arg1, arg2){
  # the instructions to perform
  return(output) # the stuff to be returned
}
```

## Function syntax II

To lighten notation, R allows you to avoid the use of `return()` to return something from a function.

By default, the last element of a function is returned.

```r
add1 = function(x){
  res = x + 1
  return(res) # return(x + 1) also works
}
add1_bis = function(x){
  x + 1
}
add1(1)
#> [1] 2
add1_bis(1)
#> [1] 2
```

## Function syntax III

If the function bumps into a `return()`, it stops right away and returns the object:

```r
test = function(x){
  return(1)
  return(2)
  return(3)
}
test()
#> [1] 1
```

## Function syntax VI

You can write functions with only one instruction in one line (no need of brackets):

```r
add1_ter = function(x) x + 1
add1_ter(2)
#> [1] 3
```

## Function syntax: Arguments

You can add default values to the arguments:

```r
funName = function(arg1 = default1, arg2 = default2, arg3){
  # here arg1 and arg2 have default values
}
```

```r
add1_quar = function(x = 0) x + 1
add1_quar()
#> [1] 1
```

## Function syntax: Arguments

Arguments need not be used in the function:

```r
happy = function(x, y, z){
  print("I'm happy.")
}
happy()
#> [1] "I'm happy."
happy("I", "am not", "happy") # yields same result
#> [1] "I'm happy."
```

In the `happy()` function, no error is raised: although missing, arguments $x$, $y$ and $z$ are not used.

When an argument is missing and used $\Rightarrow$ an error will be raised:

```r
funSquare = function(x) x**2
funSquare(2) # works
#> [1] 4
funSquare() # x is missing and used! Error
#> Error in funSquare(): argument "x" is missing, with no default
```

## Exercises: Function

1. Create `myCov(X)`, a function to compute the covariance of a matrix $X$. Remind that:

$$V(X) = \frac{1}{n-1}\tilde{X}'\tilde{X}, \quad \tilde{X}_{ik} = X_{ik} - \bar{X}_k$$

   Apply it to `X = cbind(rnorm(100, sd = 2), rnorm(100, sd = 10))`.

2. Create `myOLS(y, x)`, a function that returns the coef. of an OLS estimation of vector $x$ on vector $y$.

## Function dot dot dot

- You have a special argument called dot dot dot: `...`
- Extremely versatile and useful
- R specific

## What's `...`?

```r
showDot = function(...){
  dots = list(...)
  print(dots)
}
showDot(arg1 = 1:5, "test stuff",
        b = "another", list(test_still = 2))
#> $arg1
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "test stuff"
#>
#> $b
#> [1] "another"
#>
#> [[4]]
#> [[4]]$test_still
#> [1] 2
```

## What's `...`?

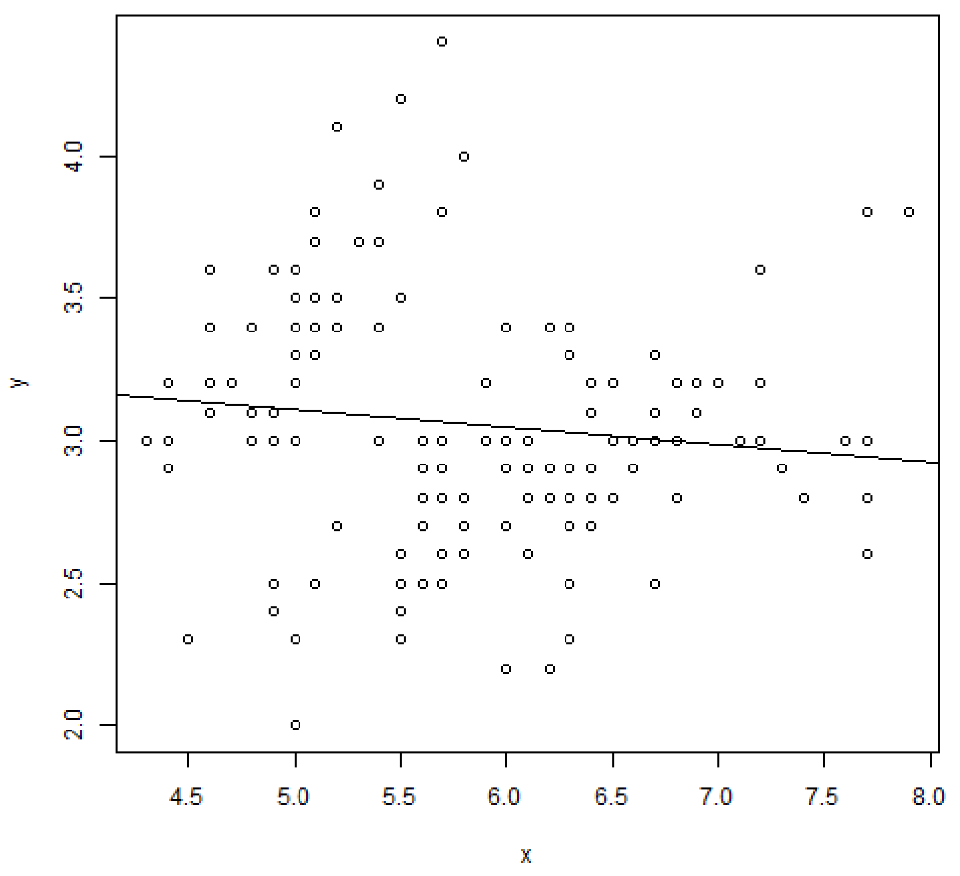
- you can access an unlimited number of arguments via the `...`!
    - it puts all these arguments into a named list
- OK but what's the point?
- it's powerful
- **very** powerful

## Example ...

```r
plotCor = function(x, y){
  # linear regression
  reg = lm(y ~ x)
  # plotting the correlation...
  plot(x, y)
  # with the fit
  abline(reg)
}
```

## Example ...

```
plotCor(iris$Sepal.Length, iris$Sepal.Width)
```



## Example `...` II

What if I want:

- to change the limits of the plot?
- change the color of the points?
- Change the axis labels?

- change other stuff??

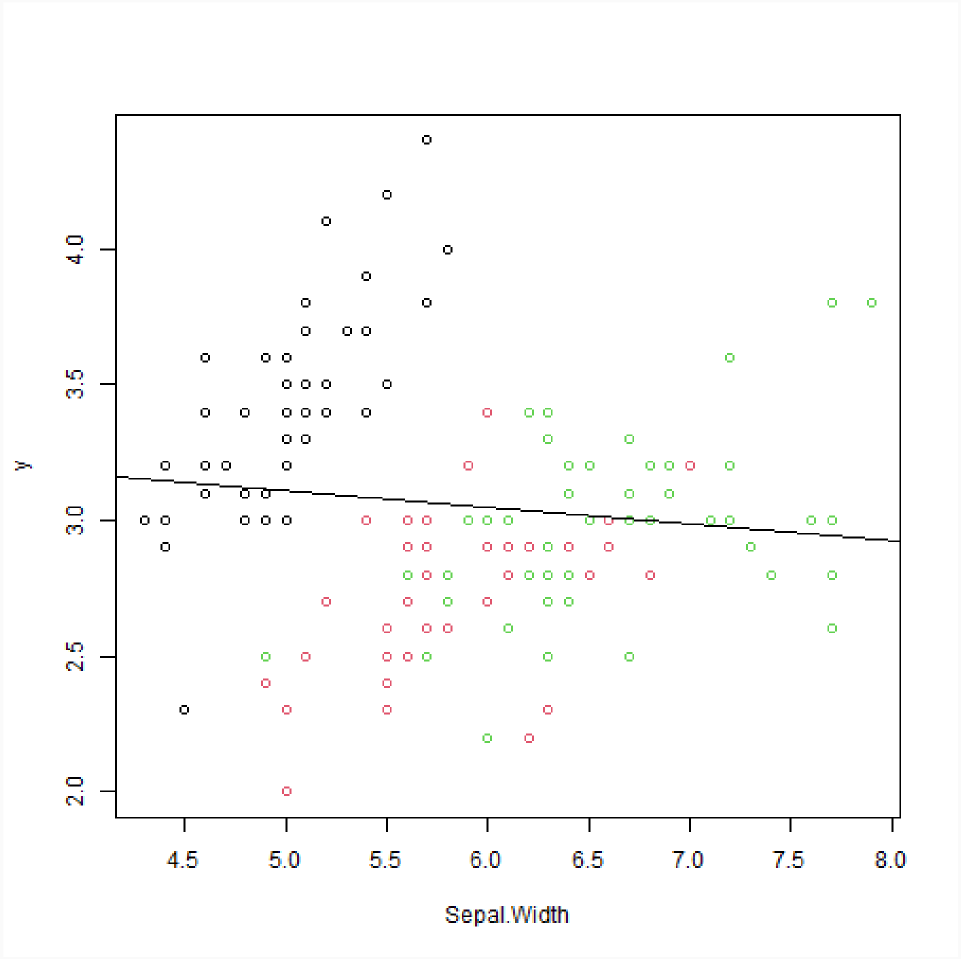
- you can't.

## Example `...`

```r
plotCor = function(x, y, ...){
    # linear regression
    reg = lm(y ~ x)
    # plotting the correlation...
    plot(x, y, ...)
    # with the fit
    abline(reg)
}
```

## Example . . .

```r
plotCor(iris$Sepal.Length, iris$Sepal.Width,
        col = iris$Species, xlab = "Sepal.Width")
```



## How does `...` work?

How does it work?

- all arguments that are NOT `plotCor()` arguments are gathered in `...` and passed on to the plot function
- in the example, the plot performed in the function is: `plot(x, y, col = iris$Species, xlab = "Sepal.Width")`
- allows to create flexible functions, with a lightweight syntax

## Understanding functions: Namespaces

```r
x = 5
f1 = function() print(x)
```

```r
# Does it work?
f1()
#> [1] 5
```

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

- when R doesn't find a variable in a function, it goes on top of it to find it.
- in the context of `f1()`, x is a global variable.

## Understanding functions: Namespaces

```r
x = 1
f1 = function() print(x)

f2 = function() {
  x = 2
  f1()
}

f3 = function(){
  x = 3
  f4 = function() print(x)
  f4()
}

f1()
#> [1] 1
f2()
#> [1] 1
f3()
#> [1] 3
```

- R looks up in the stack where the function was **defined**.

# lapply and sapply

The functions `lapply()` and `sapply()` are very handy, you should have them in your toolbox.

```r
# loops over each element of X and apply a function to it:
lapply(X = iris[, c(1, 5)], FUN = class)
#> $Sepal.Length
#> [1] "numeric"
#>
#> $Species
#> [1] "factor"
# sapply is the same but coerce the result into a vector:
sapply(iris[, c(1, 5)], class)
#> Sepal.Length      Species
#>    "numeric"     "factor"
```

## Why are they useful? I

Let's numerically compute the probability that $x > 0$ when $x \sim N(0,1)$ for varying number of observations: 10, 20, 30, 40, etc, 100.

1. Exercise: Do it using a loop.
2. With sapply:

```
myfun = function(n) mean(rnorm(n) > 0)
sapply(10 * 1:10, myfun)
#>  [1] 0.6000000 0.3500000 0.6000000 0.5500000 0.2800000 0.5166667
#>  [8] 0.5250000 0.4222222 0.4900000
```

## Why are they useful? II

Now let's get the variance of the results for 100 repetitions of the experiment:

```
myfun = function(n, nRepeat){
  var(replicate(nRepeat, mean(rnorm(n) > 0)))
}
sapply(10 * 1:10, myfun, nRepeat = 100)
#>  [1] 0.027485859 0.013802020 0.008594725 0.005618434 0.00536565
#>  [7] 0.003145970 0.002894003 0.002501359 0.003228071
```