

# Getting to program in R

Alexandre Courtiol

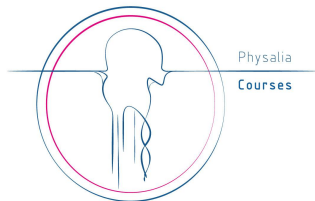
Leibniz Institute of Zoo and Wildlife Research

June 2018



**Leibniz Institute for Zoo  
and Wildlife Research**

IN THE FORSCHUNGSVERBUND BERLIN E.V.



# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# My first function

The best way:

```
my_function <- function(input1, input2) {  
  output <- input1 + input2  
  return(output)  
}
```

```
my_function(input1 = 1, input2 = 3)
```

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

# My first function

The best way:

```
my_function <- function(input1, input2) {  
  output <- input1 + input2  
  return(output)  
}  
  
my_function(input1 = 1, input2 = 3)  
## [1] 4
```

If no return(), then it returns the last row:

```
my_function <- function(input1, input2) {  
  input1 + input2  
}  
  
my_function(input1 = 1, input2 = 3)  
## [1] 4
```

# My first function

The best way:

```
my_function <- function(input1, input2) {
  output <- input1 + input2
  return(output)
}

my_function(input1 = 1, input2 = 3)
## [1] 4
```

If no return(), then it returns the last row:

```
my_function <- function(input1, input2) {
  input1 + input2
}

my_function(input1 = 1, input2 = 3)
## [1] 4
```

Inline shortcut (useful in \*apply()):

```
my_function <- function(input1, input2) input1 + input2

my_function(input1 = 1, input2 = 3)
## [1] 4
```

# My first function

You can set defaults for your inputs:

```
my_function <- function(input1, input2 = 10) {  
  output <- input1 + input2  
  return(output)  
}  
  
my_function(input1 = 1)  
## [1] 11  
  
my_function(input1 = 1, input2 = 0)  
## [1] 1  
  
my_function(input2 = 1)  
## Error in my_function(input2 = 1): argument "input1" is missing, with no default
```

# My first function

You can pass optional arguments to another function via "...":

```
my_function <- function(input1, ...) {  
  output <- mean(input1, ...)  
  return(output)  
}  
  
x <- c(1, 2, 3, 4, 5, NA)  
my_function(input1 = x)  
## [1] NA  
my_function(input1 = x, na.rm = TRUE)  
## [1] 3
```

Note: this is particularly useful when designing plotting functions!

# My first function

You can print things while the function runs:

```
my_function <- function() {  
  print("This function will output 2")  
  return(2)  
}  
  
my_function()  
## [1] "This function will output 2"  
## [1] 2
```

Note: observe also that it is possible to create functions that do not consider any input!



# My first function

You can return only a single object; so if you need several outputs, you must combine them:

```
my_function <- function(input1, ...) {
  output1 <- min(input1, ...)
  output2 <- max(input1, ...)
  output <- list(output1 = output1, output2 = output2)
  return(output)
}

x <- c(1, 2, 3, 4, 5, NA)
my_function(input1 = x, na.rm = TRUE)

## $output1
## [1] 1
##
## $output2
## [1] 5
```

Note: you can also use a vector, a matrix, a data frame...

# My first function

You can return the output “invisibly”:

```
my_function <- function(input1) {  
  output <- min(input1)  
  return(invisible(output))  
}  
  
my_function(input1 = c(1, 2, 3, -4))
```

# My first function

You can return the output “invisibly”:

```
my_function <- function(input1) {  
  output <- min(input1)  
  return(invisible(output))  
}  
  
my_function(input1 = c(1, 2, 3, -4))
```

```
foo <- my_function(input1 = c(1, 2, 3, -4))  
foo  
## [1] -4
```

# My first function

You don't have to return something:

```
my_function <- function(input1) {  
  the_min <- min(input1)  
  print(paste("The minimum is:", the_min))  
  return(invisible(NULL))  
}  
  
my_function(input1 = c(1, 2, 3, -4))  
## [1] "The minimum is: -4"
```

# My first function

You don't have to return something:

```
my_function <- function(input1) {  
  the_min <- min(input1)  
  print(paste("The minimum is:", the_min))  
  return(invisible(NULL))  
}  
  
my_function(input1 = c(1, 2, 3, -4))  
## [1] "The minimum is: -4"
```

```
foo <- my_function(input1 = c(1, 2, 3, -4))  
## [1] "The minimum is: -4"  
  
foo  
## NULL
```

# Programming with R

- 1 Writing simple functions
- 2 **Why programming?**
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Why writing your own R functions?

Using your own functions makes your scripts

- easier to understand
- safer to (re)use
- shorter to write (often)

# What do you prefer?

```
d <- data.frame(proba = c(0.1, 0.5, 0.4), group = factor(c("A", "B", "C")))

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 9

with(data = d, (proba[group == "C"] / (1 - proba[group == "C"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 6

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "C"] / (1 - proba[group == "C"])))
## [1] 1.5
```



# What do you prefer?

```
d <- data.frame(proba = c(0.1, 0.5, 0.4), group = factor(c("A", "B", "C")))

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 9

with(data = d, (proba[group == "C"] / (1 - proba[group == "C"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 6

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "C"] / (1 - proba[group == "C"])))
## [1] 1.5
```

Or

```
odds_ratio <- function(group1, group2){
  with(data = d, (proba[group == group1] / (1 - proba[group == group1])) / (proba[group == group2] / (1 - proba[group == group2])))
}

odds_ratio(group1 = "B", group2 = "A")
## [1] 9

odds_ratio(group1 = "C", group2 = "A")
## [1] 6

odds_ratio(group1 = "B", group2 = "C")
## [1] 1.5
```

# What do you prefer?

Still not convinced? Let's compute all pairwise comparisons:

```
with(data = d, (proba[group == "A"] / (1 - proba[group == "A"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 1

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 9

with(data = d, (proba[group == "C"] / (1 - proba[group == "C"])) / (proba[group == "A"] / (1 - proba[group == "A"])))
## [1] 6

with(data = d, (proba[group == "A"] / (1 - proba[group == "A"])) / (proba[group == "B"] / (1 - proba[group == "B"])))
## [1] 0.1111111

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "B"] / (1 - proba[group == "B"])))
## [1] 1

with(data = d, (proba[group == "C"] / (1 - proba[group == "C"])) / (proba[group == "B"] / (1 - proba[group == "B"])))
## [1] 0.6666667

with(data = d, (proba[group == "A"] / (1 - proba[group == "A"])) / (proba[group == "C"] / (1 - proba[group == "C"])))
## [1] 0.1666667

with(data = d, (proba[group == "B"] / (1 - proba[group == "B"])) / (proba[group == "C"] / (1 - proba[group == "C"])))
## [1] 1.5

with(data = d, (proba[group == "C"] / (1 - proba[group == "C"])) / (proba[group == "C"] / (1 - proba[group == "C"])))
## [1] 1
```

## What do you prefer?

Still not convinced? Let's compute all pairwise comparisons:

```
for (group2 in d$group) {  
  for (group1 in d$group) {  
    print(paste(group1, group2, odds_ratio(group1 = group1, group2 = group2)))  
  }  
}  
  
## [1] "A A 1"  
## [1] "B A 9"  
## [1] "C A 6"  
## [1] "A B 0.111111111111111"  
## [1] "B B 1"  
## [1] "C B 0.666666666666667"  
## [1] "A C 0.166666666666667"  
## [1] "B C 1.5"  
## [1] "C C 1"
```

Note: this is bad code, we will come back on this!

## When to write your own functions?

Don't Repeat Yourself

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics**
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Control structures: if()

```
i <- 1
a <- 2

if (i == 1) {
  a <- 1
}

a

## [1] 1
```

# Control structures: if()

```
i <- 1
a <- 2

if (i == 1) {
  a <- 1
}

a

## [1] 1
```

```
i <- 5
a <- 2

if (i == 1) {
  a <- 1
} else {
  a <- 2
}

a

## [1] 2
```

Note: for help, check `?if`.

# Control structures: loops using for() & while()

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

```
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5
```

```
i <- 1  
  
while (i < 5) {  
  print(i)  
  i <- i + 1  
}
```

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

Note: for help, check ?"for".



# Random number generators

```
runif(5)
## [1] 0.9064717 0.5339812 0.3740594 0.1669095 0.6422982
runif(5)
## [1] 0.8174356 0.3480384 0.9347922 0.7758938 0.9907253
```

# Random number generators

```
runif(5)
## [1] 0.9064717 0.5339812 0.3740594 0.1669095 0.6422982
runif(5)
## [1] 0.8174356 0.3480384 0.9347922 0.7758938 0.9907253
```

You can have reproducible results by setting a seed:

```
set.seed(10132)
runif(5)
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
set.seed(10132)
runif(5)
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
```

# Random number generators

```
runif(5)
## [1] 0.9064717 0.5339812 0.3740594 0.1669095 0.6422982
runif(5)
## [1] 0.8174356 0.3480384 0.9347922 0.7758938 0.9907253
```

You can have reproducible results by setting a seed:

```
set.seed(10132)
runif(5)
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
set.seed(10132)
runif(5)
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
```

Note: check [?Distributions](#) for more distributions.

# Errors and Warnings

```
process_factor <- function(x) {  
  if (!is.factor(x) & !is.character(x)) {  
    stop("Your input for factor should be of class character or factor")  
  }  
  if (!is.factor(x) & is.character(x)) {  
    x <- as.factor(x)  
    warning("Your input has been converted to factor")  
  }  
  return(x)  
}
```

# Errors and Warnings

```
process_factor <- function(x) {  
  if (!is.factor(x) & !is.character(x)) {  
    stop("Your input for factor should be of class character or factor")  
  }  
  if (!is.factor(x) & is.character(x)) {  
    x <- as.factor(x)  
    warning("Your input has been converted to factor")  
  }  
  return(x)  
}
```

```
process_factor(x = factor(c("A", "B")))  
## [1] A B  
## Levels: A B
```

# Errors and Warnings

```
process_factor <- function(x) {  
  if (!is.factor(x) & !is.character(x)) {  
    stop("Your input for factor should be of class character or factor")  
  }  
  if (!is.factor(x) & is.character(x)) {  
    x <- as.factor(x)  
    warning("Your input has been converted to factor")  
  }  
  return(x)  
}
```

```
process_factor(x = factor(c("A", "B")))  
## [1] A B  
## Levels: A B
```

```
process_factor(x = c("A", "B"))  
## Warning in process_factor(x = c("A", "B")): Your input has been converted to factor  
## [1] A B  
## Levels: A B
```

# Errors and Warnings

```
process_factor <- function(x) {  
  if (!is.factor(x) & !is.character(x)) {  
    stop("Your input for factor should be of class character or factor")  
  }  
  if (!is.factor(x) & is.character(x)) {  
    x <- as.factor(x)  
    warning("Your input has been converted to factor")  
  }  
  return(x)  
}
```

```
process_factor(x = factor(c("A", "B")))  
## [1] A B  
## Levels: A B
```

```
process_factor(x = c("A", "B"))  
## Warning in process_factor(x = c("A", "B")): Your input has been converted to factor  
## [1] A B  
## Levels: A B
```

```
process_factor(x = c(2, 3))  
## Error in process_factor(x = c(2, 3)): Your input for factor should be of class character or factor
```

# Boolean logic in R

```
!TRUE
## [1] FALSE

!FALSE
## [1] TRUE

TRUE == 1
## [1] TRUE

TRUE & TRUE
## [1] TRUE

TRUE & FALSE
## [1] FALSE

FALSE & FALSE
## [1] FALSE

c(TRUE, TRUE) & c(TRUE, FALSE)
## [1] TRUE FALSE

FALSE && "foo"
## [1] FALSE

FALSE & "foo"
## Error in FALSE & "foo": operations are possible only for numeric,
logical or complex types
TRUE && "foo"
## Error in TRUE && "foo": invalid 'y' type in 'x && y'
c(TRUE, TRUE) && c(TRUE, FALSE) ## dangerous
## [1] TRUE
```

```
TRUE | TRUE
## [1] TRUE

TRUE | FALSE
## [1] TRUE

FALSE | FALSE
## [1] FALSE

c(TRUE, TRUE) | c(TRUE, FALSE)
## [1] TRUE TRUE

FALSE || "foo"
## Error in FALSE || "foo": invalid 'y' type in 'x || y'
TRUE || "foo"
## [1] TRUE

any(c(FALSE, FALSE, FALSE, TRUE))
## [1] TRUE
```

Note: help available at ?"&".



# Scope

Every object in **R** belongs to an environment:

```
i <- 1
i <- i + 1
i
## [1] 2
```

```
e1 <- environment()
e1
## <environment: R_GlobalEnv>
get("i", envir = e1)
## [1] 2
```

# Scope

As a rule, anything created inside a function stays inside the function:

```
i <- 1

f <- function(i) {
  i <- i + 1
  return(i)
}

f(i)
## [1] 2

i
## [1] 1
```

# Scope

This is because the functions have their own environments:

```
i <- 1

f <- function(i) {
  i <- i + 1
  e <- environment()
  return(list(env = e, i = i))
}

res <- f(i)
res$i
## [1] 2
res$env
## <environment: 0x7f98b70ac6b0>
```

# Scope

This is because the functions have their own environments:

```
i <- 1

f <- function(i) {
  i <- i + 1
  e <- environment()
  return(list(env = e, i = i))
}

res <- f(i)
res$i
## [1] 2
res$env
## <environment: 0x7f98b70ac6b0>
```

```
get("i", envir = globalenv()) ## globalenv() provides the address of the global environment
## [1] 1
get("i", envir = res$env)
## [1] 2
```

# Scope

This is because the functions have their own environments:

```
i <- 1

f <- function(i) {
  i <- i + 1
  e <- environment()
  return(list(env = e, i = i))
}

res <- f(i)
res$i
## [1] 2
res$env
## <environment: 0x7f98b70ac6b0>
```

```
get("i", envir = globalenv()) ## globalenv() provides the address of the global environment
## [1] 1
get("i", envir = res$env)
## [1] 2
```

Challenge: check whether the environment remains the same at each function call or not!

# Scope

Environments can be hacked, but it is usually source of troubles:

```
i <- 1

f <- function(i) {
  i <<- i + 1
  return(i)
}

f(i)
## [1] 1

i
## [1] 2
```

# Scope

Environments can be defined, but it is usually not necessary:

```
i <- 1

f <- function(i) {
  i <- i + 1
  assign("i", i, envir = globalenv())
  return(i)
}

f(i)
## [1] 2

i
## [1] 2
```

# Scope

The best is to use a functional approach:

```
i <- 1

f <- function(i) {
  i <- i + 1
  return(i)
}

i <- f(i)
i
## [1] 2
```



# Scope

There are a few scoping exceptions:

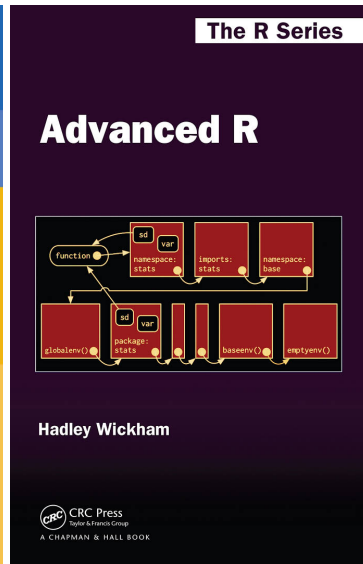
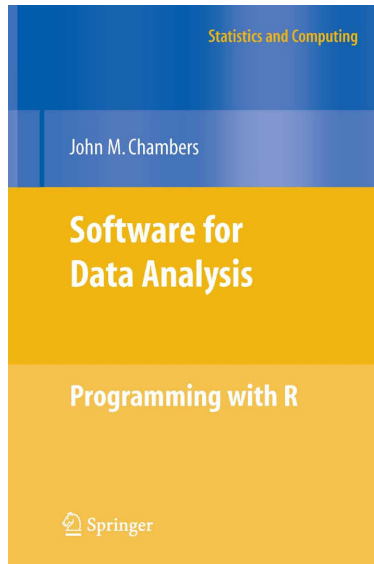
```
i <- 1  
  
for (j in 1:10) {  
  i <- i + 1  
}  
  
i  
## [1] 11
```

And yet, `for` is actually a function:

```
i <- 1  
  
`for`(j, 1:10, i <- i + 1)  
i  
## [1] 11
```

# Scope

For more info:



# R has the same numerical issues as most programming languages

```
x <- 0.7 - 0.4 - 0.3  
x == 0  
## [1] FALSE
```

# R has the same numerical issues as most programming languages

```
x <- 0.7 - 0.4 - 0.3  
x == 0
```

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

```
print(x, digits = 22)
```

```
## [1] -5.551115123125782702118e-17
```

# R has the same numerical issues as most programming languages

```
x <- 0.7 - 0.4 - 0.3
```

```
x == 0
```

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

```
print(x, digits = 22)
```

```
## [1] -5.551115123125782702118e-17
```

```
print(seq(0, 1, 0.1), digits = 22)
```

```
## [1] 0.00000000000000000000 0.10000000000000000055511
```

```
## [3] 0.20000000000000000111022 0.3000000000000000444089
```

```
## [5] 0.4000000000000000222045 0.5000000000000000000000
```

```
## [7] 0.6000000000000000888178 0.7000000000000000666134
```

```
## [9] 0.8000000000000000444089 0.9000000000000000222045
```

```
## [11] 1.00000000000000000000
```

Note 1: same kind of thing can happen in Excel too (<https://support.microsoft.com/en-us/kb/214118>).

Note 2: this kind of problem sometimes kills people

(<http://www-users.math.umn.edu/~arnold/disasters/Patriot-dharan-skeel-siam.pdf>).

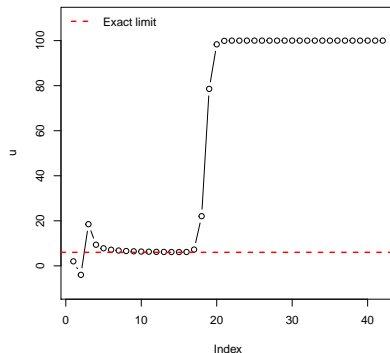
# R has the same numerical issues as most programming languages

The errors sometimes add up:

Example of the J.M Muller's Serie

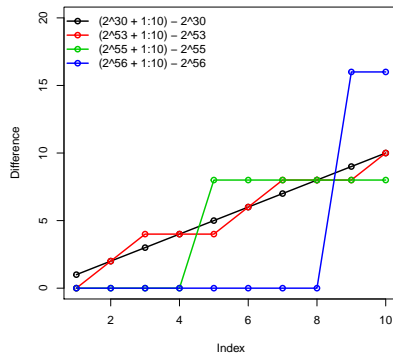
$$u_0 = 2; u_1 = -4; u_{n+1} = 111 - \frac{1130}{u_n} + \frac{3000}{u_n * u_{n-1}}$$

```
u <- c(2, -4)
new.u <- function(u) 111 - 1130/u[length(u)] + 3000/(u[length(u)]*u[length(u) - 1])
for (i in 1:40) u <- c(u, new.u(u))
```



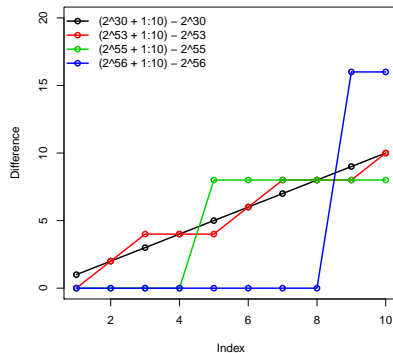
# R has the same numerical issues as most programming languages

And it is not just about fractions:



# R has the same numerical issues as most programming languages

And it is not just about fractions:



```
as.integer(2^31 - 1) ## upper limit in 32 bits coding
## [1] 2147483647
as.integer(2^31)
## Warning: NAs introduced by coercion to integer range
## [1] NA
```



# R has the same numerical issues as most programming languages

Solution: beware of floats and use adequate functions, not boolean tests, when performing comparisons!

```
??"equality"
```

```
Help pages:
base::all.equal          Test if Two Objects are (Nearly) Equal
base::identical          Test Objects for Exact Equality
data.table::all.equal    Equality Test Between Two Data Tables
datasets::airquality     New York Air Quality Measurements
dplyr::all_equal          Flexible equality comparison for data frames
```

```
all.equal(target = 0, current = x)
## [1] TRUE
```

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

## Some key advices

- everything you use in the body must pass through the inputs
- everything you output must pass through the return
- try to write functions that you could reuse in other situations

## Some key advices

- everything you use in the body must pass through the inputs
- everything you output must pass through the return
- try to write functions that you could reuse in other situations

Bad:

```
d <- data.frame(proba = c(0.1, 0.5, 0.4), group = factor(c("A", "B", "C")))

odds_ratio <- function(group1, group2){
  with(data = d, (proba[group == group1] / (1 - proba[group == group1])) / (proba[group == group2] / (1 - proba[group == group2])))
}

for (group2 in d$group) {
  for (group1 in d$group) {
    print(paste(group1, group2, odds_ratio(group1 = group1, group2 = group2)))
  }
}
```

Why is this bad?

# A better implementation of functions to compute odd ratios

Better because it is general and based on functions (example 1):

```
odds_ratio <- function(p1, p2){
  OR <- (p1/(1 - p1)) / (p2/(1 - p2))
  return(OR)
}

all_pairwise <- function(proba, groups){
  groups_id <- unique(groups)

  results <- matrix(NA, ncol = length(groups_id), nrow = length(groups_id))
  colnames(results) <- groups_id
  rownames(results) <- groups_id

  for (group1 in groups_id) {
    for (group2 in groups_id) {
      results[group1, group2] <- odds_ratio(p1 = proba[groups == group1], p2 = proba[groups == group2])
    }
  }

  return(results)
}

all_pairwise(proba = d$proba, groups = d$group)

##      A      B      C
## A 1 0.1111111 0.1666667
## B 9 1.0000000 1.5000000
## C 6 0.6666667 1.0000000
```

# A better implementation of functions to compute odd ratios

Better because it is very general and based on functions (example 2):

```
odds_ratio <- function(p1, p2){
  OR <- (p1/(1 - p1)) / (p2/(1 - p2))
  return(OR)
}

all_pairwise2 <- function(proba, groups){
  groups_id <- unique(groups)
  to_do <- expand.grid(groups_id, groups_id)

  OR <- apply(to_do, 1, function(gr) { ## define anonymous function
    odds_ratio(p1 = proba[groups == gr[1]], p2 = proba[groups == gr[2]])
  })

  return(data.frame(group1 = to_do[, 1],
                    group2 = to_do[, 2],
                    OR = OR))
}

all_pairwise2(proba = d$proba, groups = d$group)
```

##	group1	group2	OR
## 1	A	A	1.0000000
## 2	B	A	9.0000000
## 3	C	A	6.0000000
## 4	A	B	0.1111111
## 5	B	B	1.0000000
## 6	C	B	0.6666667
## 7	A	C	0.1666667
## 8	B	C	1.5000000
## 9	C	C	1.0000000

## Some key advices

- everything you use in the body must pass through the inputs
- everything you output must pass through the return
- try to write functions that you could reuse in other situations
- there are many ways to reach the same outcome; experiment a bit to find something you like/understand

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests**
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R



# Challenge

The t-test and the Mann-Whitney U tests are two tests aiming at comparing 2 groups. We want to compare the risk of false positives and true positives of these two tests in the following conditions:

- assuming that height of males is Gaussian with mean 180 cm and SD 6 cm and that the height of females is Gaussian with mean 170 cm and SD 5 cm, how many individuals (sex-ratio = 1) do I need to get a power of 80% (risk of false negative = 20%)?
- assuming that the null hypothesis is true and that you have sampled 20 males and 20 females, what is the probability of false positives for the threshold  $\alpha = 0.05$ ? (And for any threshold between 0 and 1%?)

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code**
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Learning by mimicking

Looking at code written by others will teach you

- how their functions work
- how to code
- new functions or packages that could be useful for you

# How to get the code behind a function?

Start by simply typing the function name (without brackets):

```
mosaic::oddsRatio  
## function (x, conf.level = 0.95, verbose = !quiet, quiet = TRUE,  
##     digits = 3)  
## {  
##     orrr(x, conf.level = conf.level, verbose = verbose, digits = digits,  
##         relrisk = FALSE)  
## }  
## <bytecode: 0x7f98b9dfae20>  
## <environment: namespace:mosaic>
```

# How to get the code behind a function?

Then, follows the successive calls:

```
mosaic::orrr
## function (x, conf.level = 0.95, verbose = !quiet, quiet = TRUE,
##   digits = 3, relrisk = FALSE)
## {
##   if (any(dim(x) != c(2, 2))) {
##     stop("expecting something 2 x 2")
##   }
##   names(x) <- NULL
##   row.names(x) <- NULL
##   colnames(x) <- NULL
##   rowsums <- rowSums(x)
##   p1 <- x[1, 1]/rowsums[1]
##   p2 <- x[2, 1]/rowsums[2]
##   o1 <- p1/(1 - p1)
##   o2 <- p2/(1 - p2)
##   RR <- p2/p1
##   OR <- o2/o1
##   crit <- qnorm((1 - conf.level)/2, lower.tail = FALSE)
##   names(RR) <- "RR"
##   log.RR <- log(RR)
##   SE.log.RR <- sqrt(sum(x[, 2]/x[, 1]/rowsums))
##   log.lower.RR <- log.RR - crit * SE.log.RR
##   log.upper.RR <- log.RR + crit * SE.log.RR
##   lower.RR <- exp(log.lower.RR)
##   upper.RR <- exp(log.upper.RR)
##   names(OR) <- "OR"
##   log.OR <- log(OR)
##   SE.log.OR <- sqrt(sum(1/x))
##   log.lower.OR <- log.OR - crit * SE.log.OR
##   log.upper.OR <- log.OR + crit * SE.log.OR
##   lower.OR <- exp(log.lower.OR)
##   upper.OR <- exp(log.upper.OR)
```

## How to get the code behind a function?

Sometimes, the code is not directly displayed... e.g. **R** methods (S3):

```
residuals
## function (object, ...)
## UseMethod("residuals")
## <bytecode: 0x7f98b5cf63f0>
## <environment: namespace:stats>
```

`residuals()` is a *generic* function which rely on class specific *methods*:

```
methods(residuals)
## [1] residuals.default*      residuals.glm
## [3] residuals.gls*          residuals.glsStruct*
## [5] residuals.gnls*         residuals.gnlsStruct*
## [7] residuals.HoltWinters*   residuals.isoreg*
## [9] residuals.lm            residuals.lme*
## [11] residuals.lmeStruct*     residuals.lmList*
## [13] residuals.loglm*        residuals.nlmeStruct*
## [15] residuals.nls*          residuals.psych*
## [17] residuals.smooth.spline* residuals.tukeyline*
## see '?methods' for accessing help and source code
```

The methods with a `*` are not exported from their package namespace!

## How to get the code behind a function?

Add the class at the end of the function name to get the code for exported **R** (S3) methods:

```
residuals.lm
## function (object, type = c("working", "response", "deviance",
##      "pearson", "partial"), ...)
## {
##   type <- match.arg(type)
##   r <- object$residuals
##   res <- switch(type, working = , response = r, deviance = ,
##       pearson = if (is.null(object$weights)) r else r * sqrt(object$weights),
##       partial = r)
##   res <- naresid(object$na.action, res)
##   if (type == "partial")
##       res <- res + predict(object, type = "terms")
##   res
## }
## <bytecode: 0x7f98b9706b60>
## <environment: namespace:stats>
```

Note: this requires to know the class of the object you work with!  
You can use `class()` on your input to figure this out.

## How to get the code behind a function?

It is also possible to get the code of non-exported **R** methods (S3):

```
residuals.nls
## Error in eval(expr, envir, enclos): object 'residuals.nls' not found

getAnywhere("residuals.nls") # or getS3method("residuals", "nls")

## A single object matching 'residuals.nls' was found
## It was found in the following places
##   registered S3 method for residuals from namespace stats
##   namespace:stats
## with value
##
## function (object, type = c("response", "pearson"), ...)
## {
##   type <- match.arg(type)
##   if (type == "pearson") {
##     val <- as.vector(object$m$resid())
##     std <- sqrt(sum(val^2)/(length(val) - length(coef(object))))
##     val <- val/std
##     if (!is.null(object$na.action))
##       val <- naresid(object$na.action, val)
##     attr(val, "label") <- "Standardized residuals"
##   }
##   else {
##     val <- as.vector(object$m$lhs() - object$m$fitted())
##     if (!is.null(object$na.action))
##       val <- naresid(object$na.action, val)
##     lab <- "Residuals"
##     if (!is.null(aux <- attr(object, "units")$y))
##       lab <- paste(lab, aux)
##     attr(val, "label") <- lab
##   }
## }
```



# Challenge

Which function actually computes the numbers behind `boxplot()`?

# Challenge

What is the code behind `t.test()`?

## How to get the code behind a function?

Some functions – the interfaces – call functions that are written in other languages.

The source code of these latter functions is not directly visible (spotted as `.C()`, `.Fortran()`, `.Call()`, `.Primitive()`, `.Internal()`, `.External()`).

```
dnorm
## function (x, mean = 0, sd = 1, log = FALSE)
## .Call(C_dnorm, x, mean, sd, log)
## <bytecode: 0x7f98bde43d60>
## <environment: namespace:stats>
```

In these cases, the easiest is to use the read-only mirror for **R** (<https://github.com/wch/r-source>) or the relevant package on Github! (here, the answer lies in `r-source/src/nmath/dnorm.c`)

For more info, check:

<https://stackoverflow.com/questions/19226816/how-can-i-view-the-source-code-for-a-function>.

# Challenge

What is the code really estimating coefficients behind `lm()`?

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging**
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Debugging a faulty function

```
pythagora <- function(x, y) {  
  x2 <- x^2  
  y2 <- y^2  
  hyp <- (x^2 + y^2)^1/2  
  return(hyp)  
}  
  
pythagora(x = 2, y = 2)  
## [1] 4
```

## Debugging a faulty function

```
pythagora <- function(x, y) {  
  x2 <- x^2  
  y2 <- y^2  
  hyp <- (x^2 + y^2)^1/2  
  return(hyp)  
}  
  
pythagora(x = 2, y = 2)  
## [1] 4
```

```
pythagora <- function(x, y) {  
  x2 <- x^2  
  y2 <- y^2  
  browser()  
  hyp <- (x^2 + y^2)^1/2  
  return(hyp)  
}  
  
pythagora(x = 2, y = 2)
```

Note: this is very useful but you need to have access to the code!

# Debugging a faulty function

```
pythagora <- function(x, y) {  
  x2 <- x^2  
  y2 <- y^2  
  hyp <- (x^2 + y^2)^1/2  
  return(hyp)  
}  
  
pythagora(x = 2, y = 2)  
## [1] 4
```

```
debug(pythagora)  
  
pythagora(x = 2, y = 2)  
  
undebug(pythagora) ## when you are done, or use debugonce() above, or reload the function
```

Note: this can work on any function without having to mess with the code!



## Debugging a faulty function

There are plenty more debugging possibilities out there!

Check:

- the nice possibilities with RStudio:  
<https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio>
- `?option` and look at error
- `?traceback`
- `?trace`

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling**
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Remember

*“premature optimization is the root of all evil ”*

Donald Knuth, Computing Surveys, Vol 6, No 4, December 1974

# When to optimize?

- optimize only when the code really works
- optimize only if needed

# Profiling

You can time functions:

```
library(spaMM)

data("Loaloa")

system.time({
  fullfit <- fitme(maxNDVI ~ 1 + Matern(1|latitude+longitude), data = Loaloa)
})

##      user  system elapsed
## 11.485    1.906   14.094
```

# Profiling

You can time functions:

```
library(spaMM)

data("Loaloa")

system.time({
  fullfit <- fitme(maxNDVI ~ 1 + Matern(1|latitude+longitude), data = Loaloa)
})

##      user  system elapsed
## 11.485    1.906    14.094
```

You can profile the code very precisely using profvis:

```
library(profvis)

profvis({
  fullfit <- fitme(maxNDVI ~ 1 + Matern(1|latitude+longitude), data = Loaloa)
})
```

Note: try it out!

## General tips to improve speed

- limit to number of function calls as much as possible!

```
system.time({  
  replicate(1e6, runif(1))  
})  
##      user  system elapsed  
##   3.615    0.382    4.028
```

```
system.time({  
  runif(1e6)  
})  
##      user  system elapsed  
##   0.034    0.001    0.035
```

Note: it is always best to design functions that can work on vectors and not just on single values.

## General tips to improve speed

- limit to number of function calls as much as possible!
- compare alternative implementations

```
d <- data.frame(proba = runif(500), group = factor(1:500))
```

```
system.time({  
  all_pairwise(proba = d$proba, groups = d$group)  
})
```

```
##      user  system elapsed  
##    8.682    0.518    9.573
```

```
system.time({  
  all_pairwise2(proba = d$proba, groups = d$group)  
})
```

```
##      user  system elapsed  
##   10.111    0.485   11.284
```



## General tips to improve speed

- limit to number of function calls as much as possible!
- compare alternative implementations
- code bottlenecks in another language and rely on interfaces (advanced)

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages**
- 10 Writing more advanced functions
- 11 Object systems in R

# What is an R package

An R package is only a way to organise one's data, scripts and documentation!

Benefits:

- easy distribution
- thorough check of the code
- incentive to write up some documentation and examples

Note: it is good practice to write a research project as a package!

## How to create an R package

- you can simply use the function `package.skeleton()` (see help for example)
- you can use the combo RStudio + the R package devtools

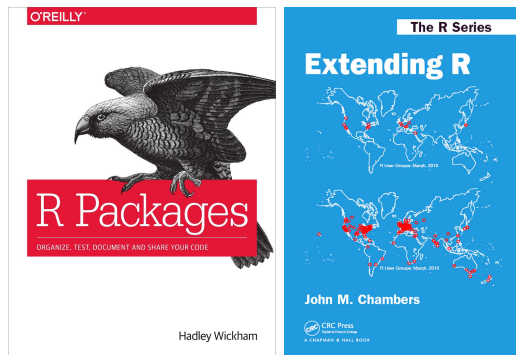
# How to create an R package

- you can simply use the function `package.skeleton()` (see help for example)
- you can use the combo RStudio + the R package devtools

For more info:

<https://cran.r-project.org/doc/manuals/r-release/R-exts.html>

Or:



# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions**
- 11 Object systems in R

# Working on the language

## Crude:

```
set.seed(1)
d <- data.frame(foo = runif(10), bar = runif(10))

my_wilcox <- function(var1, var2, data) {
  w <- wilcox.test(x = data[, var1], y = data[, var2]) ## data$var1 would not work!
  return(w)
}

my_wilcox(var1 = "foo", var2 = "bar", data = d)

##
## Wilcoxon rank sum test
##
## data: data[, var1] and data[, var2]
## W = 47, p-value = 0.8534
## alternative hypothesis: true location shift is not equal to 0
```

# Working on the language

## Better:

```
set.seed(1)
d <- data.frame(foo = runif(10), bar = runif(10))

my_wilcox <- function(var1, var2, data) {
  w <- eval(substitute(wilcox.test(x = data$var1, y = data$var2)))
  return(w)
}

my_wilcox(var1 = "foo", var2 = "bar", data = d)

##
##  Wilcoxon rank sum test
##
## data:  d$foo and d$bar
## W = 47, p-value = 0.8534
## alternative hypothesis: true location shift is not equal to 0
```



## Working on the language

`quote` & `substitute` generate unevaluated pieces of code.  
`eval` can be used to evaluate the piece of code.

```
saved_code <- quote(a + b)
saved_code

## a + b

a <- 2
b <- 3

eval(saved_code)

## [1] 5
```

```
eval_example <- function(){
  a <- 5
  b <- 5
  print(eval(saved_code))
  print(eval(saved_code, envir = .GlobalEnv))
}

eval_example()

## [1] 10
## [1] 5
```

Note: the difference between `quote` & `substitute` is that `substitute` allows for substitution before the evaluation (useful to play with variable names).

## Working on the language

### A few examples:

```
foo <- 1
getNameFromObject <- function(i) paste(substitute(i))
getNameFromObject(foo)
## [1] "foo"
```

```
getValueFromName <- function(i) get(i)
getValueFromName("foo")
## [1] 1
```

```
setObjectFromName <- function(i, x) assign(i, x, envir = globalenv())
setObjectFromName("bar", 2)
bar
## [1] 2
```

Note: there are plenty more possibilities for working on the language in **R**, but it would require to dive quite deeper into how **R** works.

# Recursions

## The Fibonacci example:

```
fibonacci <- function(x) {  
  if (x == 0) return(0)  
  if (x == 1) return(1)  
  return(fibonacci(x - 1) + fibonacci(x - 2))  
}  
  
system.time(foo <- fibonacci(35))  
  
##      user  system elapsed  
## 12.152    0.150   12.721  
  
foo  
## [1] 9227465
```

Note: if you plan to rename functions, you can use "Recall()".!

# Using C++ code within functions

The package Rcpp allows to run C++ code inside **R** on the fly!

```
library(Rcpp) ## you may need to install it (not included in BeginR)
fibRcpp <- cppFunction('
int fibonacci(const int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return (fibonacci(x - 1) + fibonacci(x - 2));
}
')
system.time(foo <- fibRcpp(35)) ## beware: memory issue if x is too large!

##      user system elapsed
##  0.066   0.000   0.066

foo

## [1] 9227465
```

# Programming with R

- 1 Writing simple functions
- 2 Why programming?
- 3 Coding basics
- 4 A few coding tips
- 5 Example: comparing the performances of 2 tests
- 6 Exploring existing code
- 7 Debugging
- 8 Optimisation & Profiling
- 9 Writing R packages
- 10 Writing more advanced functions
- 11 Object systems in R

# Definitions

What is an object?

An instance of a class.

What is a class?

A data structure for which specific methods can be defined.

What is a method?

A function that is designed to work with all objects of a given class.

# R is an object-based system

Rule #1:

Everything that exists in **R** is an object

(John M. Chambers)

# Example

```
res <- 1

res
## [1] 1
class(res)
## [1] "numeric"
methods(class = "numeric")
## [1] [          [<-      all.equal    Arith
## [5] as.data.frame as.Date    as.POSIXct  as.POSIXlt
## [9] as.raster     cbind2      coerce      Compare
## [13] confint       Logic       Ops         rbind2
## see '?methods' for accessing help and source code
```

```
`<-`
## .Primitive("<-")
class(`<-`)
## [1] "function"
methods(class = "function")
## [1] as.data.frame as.list      coef          coerce
## [5] coerce<-      head          plot          print
## [9] tail
## see '?methods' for accessing help and source code
```



# Objects are accessed or modified using references

Reference = a name + an environment:

```
ls()

## [1] "a"                "all_pairwise"
## [3] "all_pairwise2"    "b"
## [5] "bar"              "d"
## [7] "do_slow"          "e1"
## [9] "eval_example"     "f"
## [11] "fibonacci"        "fibRcpp"
## [13] "foo"              "fullfit"
## [15] "getNameFromObject" "getValueFromName"
## [17] "group1"           "group2"
## [19] "i"                "j"
## [21] "Loaloa"           "my_function"
## [23] "my_wilcox"        "new.u"
## [25] "odds_ratio"       "process_factor"
## [27] "pythagora"        "res"
## [29] "saved_code"       "setObjectFromName"
## [31] "u"                "x"

environmentName(pryr::where("res"))

## [1] "R_GlobalEnv"
```

Note: remember, we discussed environments earlier on!

## Objects are accessed or modified using references

Behind one reference there is one memory address:

```
pryr::address(res)
## [1] "0x7f98c3c48090"
```

Behind one memory address there can be several references:

```
res2 <- res
pryr::address(res2)
## [1] "0x7f98c3c48090"
```

Note: as we will see, the address behind a reference can change during computation.

# There are different systems for defining and using objects in R

Native class systems:

- S3 (legacy from S version 3, in *base*)
- S4 (legacy from S version 4, in core package *methods*)
- Reference Class (sometimes referred to as R5, in *methods*)

# There are different systems for defining and using objects in R

Native class systems:

- S3 (legacy from S version 3, in *base*)
- S4 (legacy from S version 4, in core package *methods*)
- Reference Class (sometimes referred to as R5, in *methods*)

Additional class systems:

- R6 (in the package R6, one of the most downloaded package on CRAN!)
- ggproto (in the package *ggplot2*)
- others (proto, ...)

Note: the objects created with one system can contain objects created with another.

## The 2 main object-based programming paradigms

### Functional (Object Oriented) Programming

- suitable for analytical workflows
- S3, S4
- methods defined outside the objects
- objects are not mutable
- $a \xrightarrow{\text{fn}} b \xrightarrow{\text{fn}} c \xrightarrow{\text{fn}} \dots$

## The 2 main object-based programming paradigms

### Functional (Object Oriented) Programming

- suitable for analytical workflows
- S3, S4
- methods defined outside the objects
- objects are not mutable
- $a \xrightarrow{fn} b \xrightarrow{fn} c \xrightarrow{fn} \dots$

### Encapsulated Object Oriented Programming (aka OOP)

- suitable for data that evolve over time (modularity and re-usability)
- RC, R6
- methods defined inside the objects
- objects are mutable
- $a \xrightarrow{fn} b$  &  $a \xrightarrow{fn} \emptyset$

Note: actual programming can borrow from multiple paradigms (pure form is difficult).

## A simple example using S3

```
addone <- function(x) res + 1
res <- 1
addone(res)
## [1] 2
```

## A simple example using S4

```

setClass(Class = "S4_obj", slots = list(value = "numeric"))
setGeneric(name = "addone", def = function(object) stop("only for S4_obj")) ## set generic with default behaviour
## [1] "addone"

setMethod(f = "addone", signature = "S4_obj", definition = function(object) {
  object@value <- object@value + 1
  return(object)
})

resS4 <- new("S4_obj", value = 1)
addone(resS4)

## An object of class "S4_obj"
## Slot "value":
## [1] 2

```



## A simple example using RC

```
RC_obj <- setRefClass(Class = "RC_obj",  
  fields = list(value = "numeric"),  
  methods = list(  
    addone = function() value <- value + 1  
  )  
)
```

```
resRC <- RC_obj$new(value = 1)  
resRC$addone()  
resRC  
  
## Reference class object of class "RC_obj"  
## Field "value":  
## [1] 2
```

# A simple example using R6

```
library(R6)
R6_obj <- R6Class(
  public = list(
    value = NA,
    initialize = function(value) self$value <- value,
    addone = function() self$value <- self$value + 1
  )
)
```

```
resR6 <- R6_obj$new(value = 1)
resR6$addone()
resR6$value
## [1] 2
```

```
resR6$value <- 1
replicate(10, resR6$addone())
## [1] 2 3 4 5 6 7 8 9 10 11
resR6$value
## [1] 11
```

# Encapsulated OOP requires mutability

S3 objects are (generally) not mutable (same for S4):

```
a <- 1:3
a
## [1] 1 2 3
pryr::address(a)
## [1] "0x7f98c18ae8c8"
```

```
a[2] <- 10
a
## [1] 1 10 3
pryr::address(a)
## [1] "0x7f98c0ce31e8"
```

```
b <- a
b
## [1] 1 10 3
pryr::address(b)
## [1] "0x7f98c0ce31e8"
```

```
b[2] <- 11
pryr::address(b)
## [1] "0x7f98c132a508"
pryr::address(a)
## [1] "0x7f98c0ce31e8"
```

```
a
## [1] 1 10 3
```

# Encapsulated OOP requires mutability

R6 objects are mutable (same for RC):

```
resR6$value
## [1] 11
pryr::address(resR6)
## [1] "0x7f98a817a468"
```

```
resR6$value <- 3
resR6$value
## [1] 3
pryr::address(resR6)
## [1] "0x7f98a817a468"
```

```
resR6_bis <- resR6
pryr::address(resR6_bis)
## [1] "0x7f98a817a468"
```

```
resR6_bis$value
## [1] 3
resR6_bis$value <- 4
resR6_bis$value
## [1] 4
```

```
resR6$value
## [1] 4
```

Note: mutability can be prevented if necessary.

## Example: An simple Individual Based Models (IBM)

Context:

- two age classes: children (0–14 yrs), adults (15+ yrs)
- children die at an average rate of 15 deaths per 1000 per year
- adults die at an average rate of 5 deaths per 1000 per year
- children do not reproduce
- adults reproduce at an average rate of 10 births per 1000 per year

Question:

Starting with 1000 individuals (with age following a uniform distribution between 0 and 40 yrs), what is the number of children and adults after 50 years?

## Functional way using S3: functions

```

death <- function(pop) {
  death_children <- rbinom(n = length(pop), size = 1, prob = 15/1000)
  death_adults <- rbinom(n = length(pop), size = 1, prob = 5/1000)
  alive <- rep(1, length(pop))
  alive[pop < 15] <- 1 - death_children[pop < 15]
  alive[pop > 14] <- 1 - death_adults[pop > 14]
  pop <- pop[alive == 1]
  return(pop)
}

birth <- function(pop) {
  adults <- pop[pop > 14]
  babies_nb <- sum(rbinom(n = length(adults), size = 1, prob = 10/1000))
  babies <- rep(0, babies_nb)
  pop <- c(pop, babies)
  return(pop)
}

age <- function(pop) pop <- pop + 1

```

## Functional way using S3: run

```
pop <- round(runif(1000, min = 0, max = 40))

for (i in 1:50) {
  pop <- birth(pop)
  pop <- death(pop)
  pop <- age(pop)
}

table(pop > 14)

##
## FALSE TRUE
## 111 1032
```

# OOP way using R6: definition of the class individual

```
individual <- R6Class(
  public = list(
    age = NA,
    alive = 1,
    initialize = function(age = 0) {self$age <- age},
    die = function() {
      if (self$age < 15 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 15/1000)
      if (self$age > 14 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 5/1000)
    },
    reproduce = function() {
      ifelse(self$alive == 1 & self$age > 14, rbinom(n = 1, size = 1, prob = 10/1000), FALSE)
    },
    aging = function() {
      if (self$alive == 1) {self$age <- self$age + 1; self$die()}
    }
  )
)
```



# OOP way using R6: test

You can test things before creating the population!

```
alex <- individual$new()
alex

## <R6>
##   Public:
##     age: 0
##     aging: function ()
##     alive: 1
##     clone: function (deep = FALSE)
##     die: function ()
##     initialize: function (age = 0)
##     reproduce: function ()
```

```
for (i in 1:200) alex$aging()
alex

## <R6>
##   Public:
##     age: 59
##     aging: function ()
##     alive: 0
##     clone: function (deep = FALSE)
##     die: function ()
##     initialize: function (age = 0)
##     reproduce: function ()
```

# OOP way using R6: definition of the class population

```

population <- R6Class(
  public = list(
    individuals = list(),
    initialize = function(N = 1000) {
      for (i in 1:N)
        self$individuals[[i]] <- individual$new(age = round(runif(n = 1, min = 0, max = 40)))
    },
    repro = function() {
      for (i in 1:length(self$individuals))
        if (self$individuals[[i]]$reproduce() == TRUE)
          self$individuals[[length(self$individuals) + 1]] <- individual$new()
    },
    death = function(){
      alive <- sapply(self$individuals, function(i) i[["alive"]])
      self$individuals[!alive] <- NULL
    },
    aging = function(){
      for (i in 1:length(self$individuals)) self$individuals[[i]]$aging()
    },
    year = function() {self$repro(); self$death(); self$aging()},
    count = function() table(sapply(self$individuals, function(i) i[["age"]])) > 14
  )
)

```

## OOP way using R6: run

```
pop <- population$new()
for (i in 1:50) pop$year()
pop$count()

##
## FALSE TRUE
## 130 980
```

Note: I did not handle the possible population crash, so it may crash :-/

## Our Individual Based Models (IBM): UPDATE

### Setup:

- two age classes: children (0–14 yrs), adults (15+ yrs)
- children die at an average rate of 15 deaths per 1000 per year
- adults die at an average rate of 5 deaths per 1000 per year
- children do not reproduce
- adults reproduce at an average rate of 10 births per 1000 per year
- two sexes, females do not reproduce after 45 yrs (males do not reproduce)

### Question:

Starting with 1000 individuals (with age following a uniform distribution between 0 and 40 yrs), what is the number of children and adults after 50 years?

# OOP way using R6: re-definition of the class individual

```
individual <- R6Class(
  public = list(
    age = NA,
    alive = 1,
    sex = NA,
    initialize = function(age = 0) {
      self$age <- age
      self$sex <- ifelse(runif(1) < 0.5, "male", "female")
    },
    die = function() {
      if (self$age < 15 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 15/1000)
      if (self$age > 14 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 5/1000)
    },
    reproduce = function() {
      ifelse(self$alive == 1 &
        (self$age > 14 & self$age < 44 & self$sex == "female"),
        rbinom(n = 1, size = 1, prob = 10/1000), FALSE)
    },
    aging = function() {
      if (self$alive == 1) {self$age <- self$age + 1; self$die()}
    }
  )
)
```

Note: there is no need to redefine the class population!

## OOP way using R6: re-run

```
pop <- population$new()
for (i in 1:50) pop$year()
pop$count()

##
## FALSE TRUE
##      3  846
```

## Functional way using S3: re-defining functions

We would have to recode everything . . .

# Pros and cons of R6 for IBM

## Pros

- clearer structure (see butterfly example)
- easier to modify once existing
- easier to share classes between projects, packages. . .
- easier to translate to C++

## Cons

- initially difficult for those knowing mostly S3
- much slower (cost can somewhat be reduced with some tweaks)
- additional issues to take care (e.g. side effects)



# A more advanced simulation model using R6

```
?Butterfly
```