Programming with R

Alexandre Courtiol

Leibniz Institute of Zoo and Wildlife Research

June 2018

Table of contents

Table of contents

My first function

The best way:

```r
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 (usefull 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 usefull 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:

```
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
```

So if you need several outputs, you must combine them (as vector, matrix, dataframe, list...)!

My first function

You can return return things “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 return things “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
```

Table of contents

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?

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

```r
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

```r
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

Table of contents

Control flow: if()

```r
i <- 1
a <- 2

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

a
## [1] 1
```

Control flow: 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
```

Control flow: for()

```
for (i in 1:5) {
  print(i)
  }
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Control flow: while()

```r
i <- 1

while (i < 5) {
  print(i)
  i <- i + 1
  }
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Random number generators

```r
runif(5)
## [1] 0.96463192 0.49758115 0.55721983 0.04147941 0.29552567
runif(5)
## [1] 0.3423236 0.2061392 0.6612697 0.6651510 0.8388332
```

Random number generators

```r
runif(5)
```

```
## [1] 0.96463192 0.49758115 0.55721983 0.04147941 0.29552567
```

```r
runif(5)
```

```
## [1] 0.3423236 0.2061392 0.6612697 0.6651510 0.8388332
```

You can have reproducible results by setting a seed:

```r
set.seed(10132)
runif(5)
```

```
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
```

```r
set.seed(10132)
runif(5)
```

```
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
```

Random number generators

```r
runif(5)
```

```
## [1] 0.96463192 0.49758115 0.55721983 0.04147941 0.29552567
```

```r
runif(5)
```

```
## [1] 0.3423236 0.2061392 0.6612697 0.6651510 0.8388332
```

You can have reproducible results by setting a seed:

```r
set.seed(10132)
runif(5)
```

```
## [1] 0.7258731 0.3086039 0.4393603 0.7952702 0.5325773
```

```r
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 is not a 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 is not a 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 is not a 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

```r
process_factor <- function(x) {
  if (!is.factor(x) & !is.character(x)) {
    stop("Your input for factor is not a factor")
  }
  if (!is.factor(x) & is.character(x)) {
    x <- as.factor(x)
    warning("Your input has been converted to factor")
  }
  return(x)
}
```

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

```r
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
```

```r
process_factor(x = c(2, 3))
## Error in process_factor(x = c(2, 3)):  Your input for factor is not a factor
```

Scope

```r
i <- 1
i <- i + 1
i
## [1] 2
e1 <- environment()
environmentName(e1)
## [1] "R_GlobalEnv"
get("i", envir = e1)
## [1] 2
```

Note: every object in R belongs to an environment!

Scope

```r
i <- 1

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

e2 <- f(i)
i
## [1] 1
e2
## <environment: 0x55749b7a2670>
get("i", envir = e1)
## [1] 1
get("i", envir = e2)
## [1] 2
```

As a rule, anything created inside a function and not exported stays inside the function!

Scope

There are a few 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
```

R has the same numerical issues as most programming languages

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

R has the same numerical issues as most programming languages

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

```r
print(x, digits = 22)
## [1] -5.551115123125782702118e-17
```

R has the same numerical issues as most programming languages

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

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

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

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

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

```
##  [1] 0.0000000000000000000000 0.1000000000000000055511
##  [3] 0.2000000000000000111022 0.3000000000000000444089
##  [5] 0.4000000000000000222045 0.5000000000000000000000
##  [7] 0.6000000000000000888178 0.7000000000000000666134
##  [9] 0.8000000000000000444089 0.9000000000000000222045
## [11] 1.0000000000000000000000
```

NB: same kind of thing can happen in Excel too (`https://support.microsoft.com/en-us/kb/214118`)

R has the same numerical issues as most programming languages

```r
??"equality"
```

```
Help files with alias or concept or title matching ‘equality’ using
fuzzy matching:


FactoMineR::prefpls      Scatter plot and additional variables with
                         quality of representation contour lines
base::all.equal          Test if Two Objects are (Nearly) Equal
base::identical          Test Objects for Exact Equality
datasets::airquality     New York Air Quality Measurements
```

```r
?all.equal
all.equal(target = 0, current = x)
```

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

R has the same numerical issues as most programming languages

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))
```

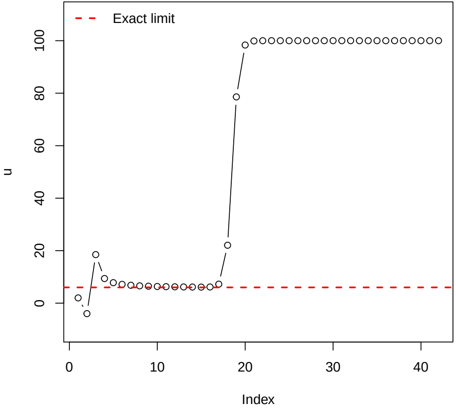

Table of contents

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 the function does not rely on `d`:

```
odds_ratio <- function(group1, group2, data){
  p1 <- data$proba[data$group == group1]
  p2 <- data$proba[data$group == group2]
  OR <- (p1/(1 - p1)) / (p2/(1 - p2))
  return(OR)
}

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

... but it stills rely on the name of the variable `proba` and `group`,
and the second part of the code is not a function.

A better implementation of functions to compute odd ratios

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

```
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)
}

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

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):

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

  OR <- apply(to_do, 1, function(gr) {
    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))
}

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

Table of contents

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 hyptohesis 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%?)

Table of contents

Learning by mimicking

Looking at code writen by others will teach you

- how their functions work
- how to code
- new functions or packages that could be usefull 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: 0x5574a1d02158>
## <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: 0x557498868908>
## <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: 0x5574a278fc38>
## <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: 0x55749d43f748>
## <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)

Challenge

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

Table of contents

Debugging a faulty function

```r
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

```r
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
```

```r
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 usefull but you need to have access to the code!

Debugging a faulty function

```r
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

```r
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
```

```r
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 possibility with R studio: `https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio`
- `?option` and look at `error`
- `?traceback`
- `?trace`

Table of contents

Table of contents

Table of contents

Working on the language

Recursions

Using C++ code on the fly!

Table of contents

There are other object system than S3!