Programming with R

Alexandre Courtiol

Leibniz Institute of Zoo and Wildlife Research

June 2018

Table of contents

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?

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

odds_ratio(group1 = "B", group2 = "A", data = d)
## [1] 9
odds_ratio(group1 = "C", group2 = "A", data = d)
## [1] 6
odds_ratio(group1 = "B", group2 = "C", data = d)
## [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, data = d)))
  }
}
```

```
## [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"
```

When to write your own functions?

Don't Repeat Yourself

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?

Usually, by simply typing its name (without brackets). But that is not always sufficient...

```
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: 0x55e3b6ed0a48>
## <environment: namespace:mosaic>
```

How to get the code behind a function?

Usually, by simply typing its name (without brackets). But that is not always sufficient...

```
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?

R methods (S3):

```
residuals
```

```
## function (object, ...)
## UseMethod("residuals")
## <bytecode: 0x55e3b4e72908>
## <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?

Getting the code for exported R methods (S3):

```
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: 0x55e3baff8790>
## <environment: namespace:stats>
```

Note: this requires to know the `class` of the object you work with!

How to get the code behind a function?

Getting the code for 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
##     }
##     val
```

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: 0x55e3bcf01628>
## <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)

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Usual programming commands exist in R

```
for (i in 1:4) {
  print(x = i)
  if (i == 2) print(x = "found 2!")
}
```

```
## [1] 1
## [1] 2
## [1] "found 2!"
## [1] 3
## [1] 4
```

```
?"for"
```

You can write your own functions!

```
OddRatio <- function(a, b) {
  odd.a <- a/(1 - a)
  odd.b <- b/(1 - b)
  return(odd.a/odd.b)
}
```

```
OddRatio(0.1, 0.01)
```

```
## [1] 11
```

Numerical issues common to most programming languages

```
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
```

```
x <- 0.7 - 0.4 - 0.3
print(x, digits = 22)
```

```
## [1] -5.5511151231257827021181e-17
```

```
x == 0
```

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

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

Numerical issues common to most programming languages

```
??"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
```

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

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

R is a programming language. . . with usual limits

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))
```