

# Evaluación "tidy":

Programando con ggplot2 y dplyr

Julio 2019

Traducido por Leonardo Collado-Torres

@feligernon

lcolladotor@gmail.com

lcolladotor.github.io

Desarrollado por Hadley Wickham para rstudio::conf(2019)

@hadleywickham

Chief Scientist, RStudio



Escribir funciones

**Regla de tres:** crea una función si haz copiado y pegado tres veces

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$c))$

**Regla de tres:** crea una función si haz copiado y pegado tres veces

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$c))$

**Regla de tres:** crea una función si haz copiado y pegado tres veces

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$d))$

Primero, identifica las parques que podrían cambiar

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$d))$

# Luego dales nombres

x

x

x

x

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$d))$

# Crear el templado de la función

```
rescale01 <- function(x) {
```

```
}
```



Luego copia un ejemplo

```
rescale01 <- function(x) {  
  (df$a - min(df$a)) / (max(df$a) - min(df$a))  
}
```

Ahora usa la variable

```
rescale01 <- function(x) {  
  (x - min(x)) / (max(x) - min(x))  
}
```

Y tal vez cambia el código un poco

```
rescale01 <- function(x) {  
  rng <- range(x)  
  (x - rng[1]) / (rng[2] - rng[1])  
}
```

# Ahora agrega más casos

```
rescale01 <- function(x) {  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  (x - rng[1]) / (rng[2] - rng[1])  
}
```

**Regla de tres:** crea una función si haz copiado y pegado tres veces

$(df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a))$

$(df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b))$

$(df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c))$

$(df\$d - \min(df\$d)) / (\max(df\$d) - \min(df\$d))$

**Regla de tres:** crea una función si haz copiado y pegado tres veces

```
rescale01(df$a)
```

```
rescale01(df$b)
```

```
rescale01(df$c)
```

```
rescale01(df$d)
```

¿Por qué crear una función? Porque una función:

1. Previene inconsistencias
2. Enfatiza que es lo que cambia
3. Facilita los cambios
4. Puede tener un nombre informativo

# Motivación



# Tratemos con un poco de código de dplyr

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
```

```
df %>% group_by(x2) %>% summarise(mean = mean(y2))
```

```
df %>% group_by(x3) %>% summarise(mean = mean(y3))
```

```
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

# Tu turno

Identifica las partes que cambias.

Dales nombres.

Crea una función.

¿Por qué no funciona?

# Tratemos con un poco de código de dplyr

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
```

```
df %>% group_by(x2) %>% summarise(mean = mean(y2))
```

```
df %>% group_by(x3) %>% summarise(mean = mean(y3))
```

```
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Primero identifica las partes que cambian

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))  
df %>% group_by(x2) %>% summarise(mean = mean(y2))  
df %>% group_by(x3) %>% summarise(mean = mean(y3))  
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

# Luego dales nombres

df

group\_var

summary\_var

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))  
df %>% group_by(x2) %>% summarise(mean = mean(y2))  
df %>% group_by(x3) %>% summarise(mean = mean(y3))  
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Ahora crea una función

```
grouped_mean <- function(df, group_var, summary_var) {  
  df %>%  
    group_by(group_var) %>%  
    summarise(mean = mean(summary_var))  
}
```

No funciona 😭

```
grouped_mean <- function(df, group_var, summary_var) {  
  df %>%  
    group_by(group_var) %>%  
    summarise(mean = mean(summary_var))  
}
```

```
grouped_mean(mtcars, cyl, mpg)  
#> Error: Column `group_var` is unknown
```

# Vocabulario



# Necesitamos un poco de vocabulario nuevo

**Evaluado** usando las reglas de  
R usuales

```
(x - min(x)) / (max(x) - min(x))
```

```
mtcars %>%
```

```
  group_by(cyl) %>%
```

```
  summarise(mean = mean(mpg))
```

Automáticamente **citado (quoted)** y  
evaluado de una forma "no estándar"

# Ya estás familiarizado con la siguiente idea

```
df <- data.frame(  
  y = 1,  
  var = 2  
)
```

```
df$y
```

```
var <- "y"  
df$var
```

¡Predice el resultado!

# \$ automáticamente agrega citas al nombre de la variable

```
df <- data.frame(  
  y = 1,  
  var = 2  
)
```

```
df$y  
#> [1] 1
```

```
var <- "y"  
df$var  
#> [1] 2
```

Si quieres referirte a ellas de forma indirecta, tienes que usar [[

```
df <- data.frame(  
  y = 1,  
  var = 2  
)
```

```
var <- "y"  
df[[var]]  
#> [1] 1
```

Citado  
(quoted)

Evaluated

Directo

`df$y`

`???`

Indirecto

`???`

```
var <- "y"  
df[[var]]
```

Citado

Evaluated

Direct

`df$y`

`df[["y"]]`

Indirect

???

```
var <- "y"  
df[[var]]
```

Citado

Evaluated

Direct

df\$y

df[["y"]]

Indirect



```
var <- "y"  
df[[var]]
```

# Identifica cuales argumentos son automáticamente citados

```
library(MASS)
```

```
mtcars2 <- subset(mtcars, cyl == 4)
```

```
with(mtcars2, sum(vs))
```

```
sum(mtcars2$am)
```

```
rm(mtcars2)
```



¿No puedes determinarlo? Intenta correr el código

```
library(MASS)
```

```
#> Works
```

```
MASS
```

```
#> Error: object 'MASS' not found
```

```
# -> El primer argumento de library() es citado
```

¿No puedes determinarlo? Intenta correr el código

```
subset(mtcars, cyl == 4)
```

```
#> Funciona
```

```
cyl == 4
```

```
#> Error: object 'cyl' not found
```

```
# -> El segundo argumento de subset() es citado
```

Ahora podemos identificar a los argumentos citados

```
library(MASS)
```

```
mtcars2 <- subset(mtcars, cyl == 4)
```

```
with(mtcars2, sum(vs))
```

```
sum(mtcars2$am)
```

```
rm(mtcars2)
```

# R base tiene 3 formas primarias de "des-citar" ("unquote")

Citado/Directo	Evaluable/Indirecto
<code>df\$<u>y</u></code>	<code>x &lt;- "y"</code> <code>df[[<u>x</u>]]</code>
<code>library(<u>MASS</u>)</code>	<code>x &lt;- "MASS"</code> <code>library(<u>x</u>, character.only = TRUE)</code>
<code>rm(<u>mtcars</u>)</code>	<code>x &lt;- "mtcars"</code> <code>rm(list = <u>x</u>)</code>



`rm(list = ls())`

<https://www.tidyverse.org/articles/2017/12/workflow-vs-script/>

# Identifica cuáles argumentos son automáticamente citados

```
library(tidyverse)
```

```
mtcars %>% pull(am)
```

```
by_cyl <- mtcars %>%  
  group_by(cyl) %>%  
  summarise(mean = mean(mpg))
```

```
ggplot(by_cyl, aes(cyl, mpg)) +  
  geom_point()
```


# Identifica cuáles argumentos son automáticamente citados

```
library(tidyverse)
```


```
mtcars %>% pull(am)
```

```
by_cyl <- mtcars %>%  
  group_by(cyl) %>%  
  summarise(mean = mean(mpg))
```

```
ggplot(by_cyl, aes(cyl, mpg)) +  
  geom_point()
```

	Citado	Evaluated	Limpio (Tidy)
Directo	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>	<code>pull(df, <u>y</u>)</code>
Indirecto		<code>var &lt;- "y"</code> <code>df[[<u>var</u>]]</code>	???



	Citado	Evaluated	Clean (Tidy)
Direct	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>	<code>pull(df, <u>y</u>)</code>
Indirect		<code>var &lt;- "y"</code> <code>df[[<u>var</u>]]</code>	<code>var &lt;- quo(<u>y</u>)</code> <code>pull(df, !!<u>var</u>)</code>

# En cualquier lugar del tidyverse usa !! para des-citar (unquote)

Pronunciado bang-bang

```
x_var <- quo(cyl)
```

```
y_var <- quo(mpg)
```

```
by_cyl <- mtcars %>%
```

```
  group_by(!!x_var) %>%
```

```
  summarise(mean = mean(!!y_var))
```

```
ggplot(by_cyl, aes(!!x_var, !!y_var)) +
```

```
  geom_point()
```

```
x_var <- "cyl"
```

```
y_var <- "mpg"
```

```
by_cyl <- mtcars %>%
```

```
  group_by("cyl") %>%
```

```
  summarise(mean = mean("mpg"))
```

Envolviendo funciones  
que *citan*

# Nuevo: Identifica arguments citados vs evaluados

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
```

```
df %>% group_by(x2) %>% summarise(mean = mean(y2))
```

```
df %>% group_by(x3) %>% summarise(mean = mean(y3))
```

```
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

# Nuevo: Identifica arguments citados vs evaluados

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
```

```
df %>% group_by(x2) %>% summarise(mean = mean(y2))
```

```
df %>% group_by(x3) %>% summarise(mean = mean(y3))
```

```
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Luego identifica las partes que podrían cambiar

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))  
df %>% group_by(x2) %>% summarise(mean = mean(y2))  
df %>% group_by(x3) %>% summarise(mean = mean(y3))  
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Estas partes se convierten en argumentos

df

group\_var

summary\_var

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))  
df %>% group_by(x2) %>% summarise(mean = mean(y2))  
df %>% group_by(x3) %>% summarise(mean = mean(y3))  
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```



Ahora escribe la función templado e identifica los argumentos citados

```
grouped_mean <- function(df, group_var, summary_var) {  
  
  df %>%  
    group_by(group_var) %>%  
    summarise(mean = mean(summary_var))  
}
```

**Nuevo:** Envuelve cada argumento citado en enquo()

```
grouped_mean <- function(df, group_var, summary_var) {  
  group_var <- enquo(group_var)  
  summary_var <- enquo(summary_var)  
  
  df %>%  
    group_by(group_var) %>%  
    summarise(mean = mean(summary_var))  
}
```

**Nuevo:** Y después des-cita (unquote) con !!

```
grouped_mean <- function(df, group_var, summary_var) {  
  group_var <- enquos(group_var)  
  summary_var <- enquos(summary_var)  
  
  df %>%  
    group_by(!!group_var) %>%  
    summarise(mean = mean(!!summary_var))  
}
```

Usa la expresión almacenada en la variable, y  
no "summary\_var" de forma literal

```
grouped_mean(mtcars, cyl, mpg)
```

```
grouped_mean <- function(df, group_var, summary_var) {  
  group_var <- enquo(group_var)  
  summary_var <- enquo(summary_var)  
  
  df %>%  
    group_by(!!group_var) %>%  
    summarise(mean = mean(!!summary_var))  
}
```

```
grouped_mean(mtcars, cyl, mpg)
```

```
grouped_mean <- function(df, group_var, summary_var) {  
  group_var <- quo(cyl)  
  summary_var <- quo(mpg)  
  
  df %>%  
    group_by(!!group_var) %>%  
    summarise(mean = mean(!!summary_var))  
}
```

```
grouped_mean(mtcars, cyl, mpg)
```

```
grouped_mean <- function(df, group_var, summary_var) {
```

```
  df %>%
```

```
    group_by(cyl) %>%
```

```
    summarise(mean = mean(mpg))
```

```
}
```

# ¿Vale la pena?

1:23 PM - 1:43 PM

Friday

## Session 5 / programming / Lazy evaluation

The "tidy eval" framework is implemented in the rlang package and is rolling out in packages across the tidyverse and beyond. There is a lively conversation these days, as people come to terms with tidy eval and share their struggles and successes with the community. Why is this such a big deal? For starters, never before have so many people engaged with R's lazy evaluation model and been encouraged and/or required to manipulate it. I'll cover some background fundamentals that provide the rationale for tidy eval and that equip you to get the most from other talks.

Speakers: [Jenny Bryan](#)

Nos ahorra mucho código que tendríamos que escribir

```
filter(diamonds, x > 0 & y > 0 & z > 0)
```

# vs

```
diamonds[  
  diamonds$x > 0 &  
  diamonds$y > 0 &  
  diamonds$z > 0,  
]
```



Nos ahorra mucho código que tendríamos que escribir

```
filter(diamonds, x > 0 & y > 0 & z > 0)
```

# vs

```
diamonds[  
  diamonds[["x"]] > 0 &  
  diamonds[["y"]] > 0 &  
  diamonds[["z"]] > 0,  
]
```

# Nos permite traducir el código a otros lenguajes

```
mtcars_db %>%
```

```
  filter(cyl > 2) %>%
```

```
  select(mpg:hp) %>%
```

```
  head(10) %>%
```

```
  show_query()
```

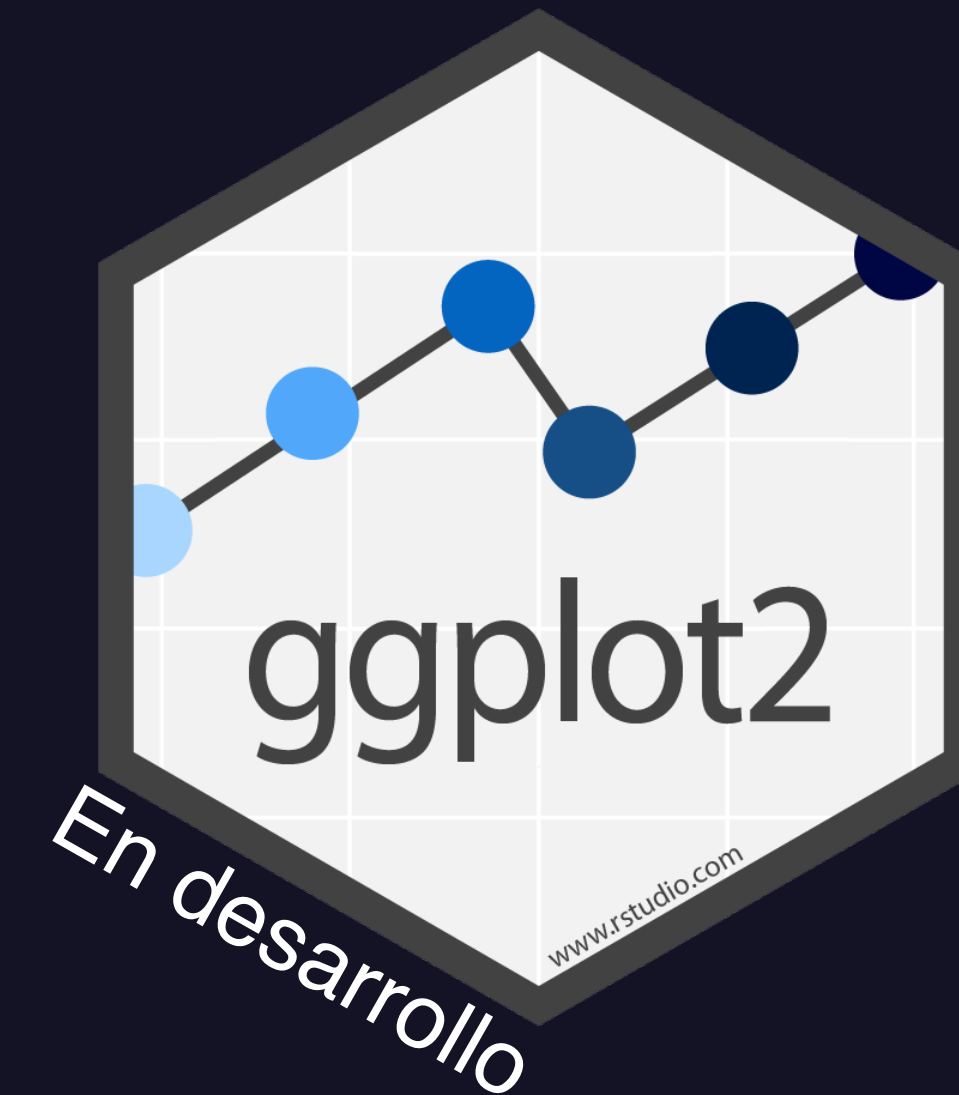
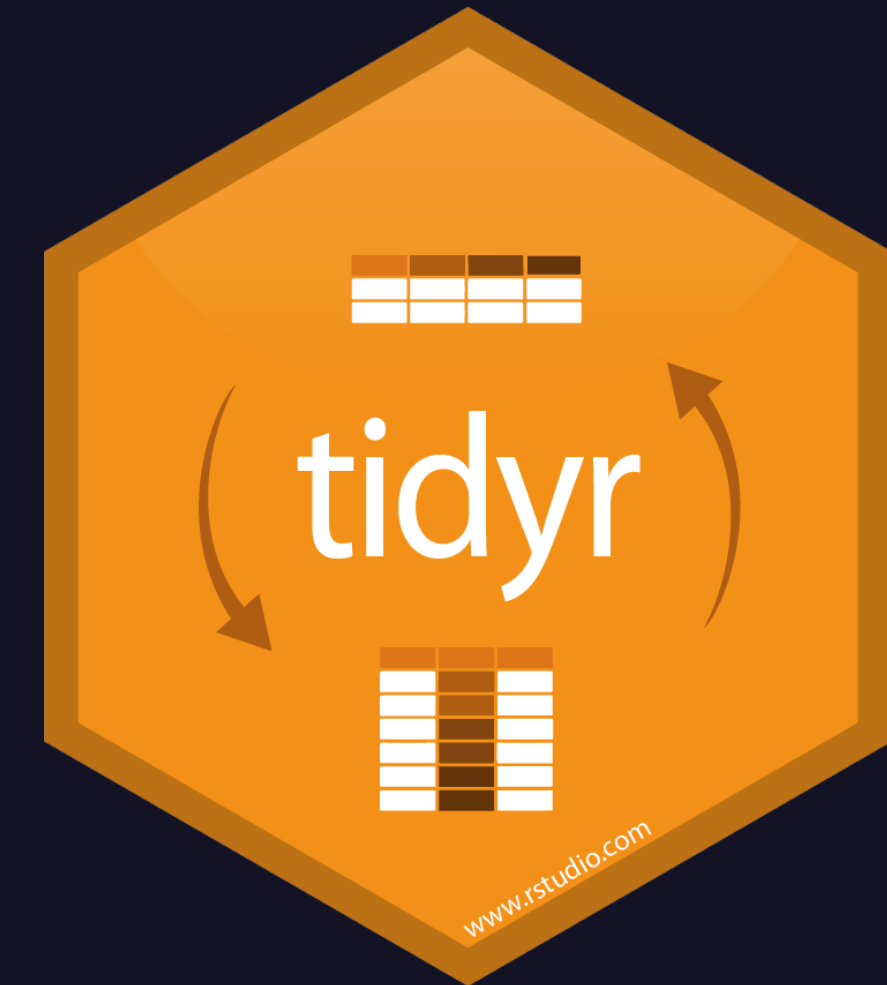
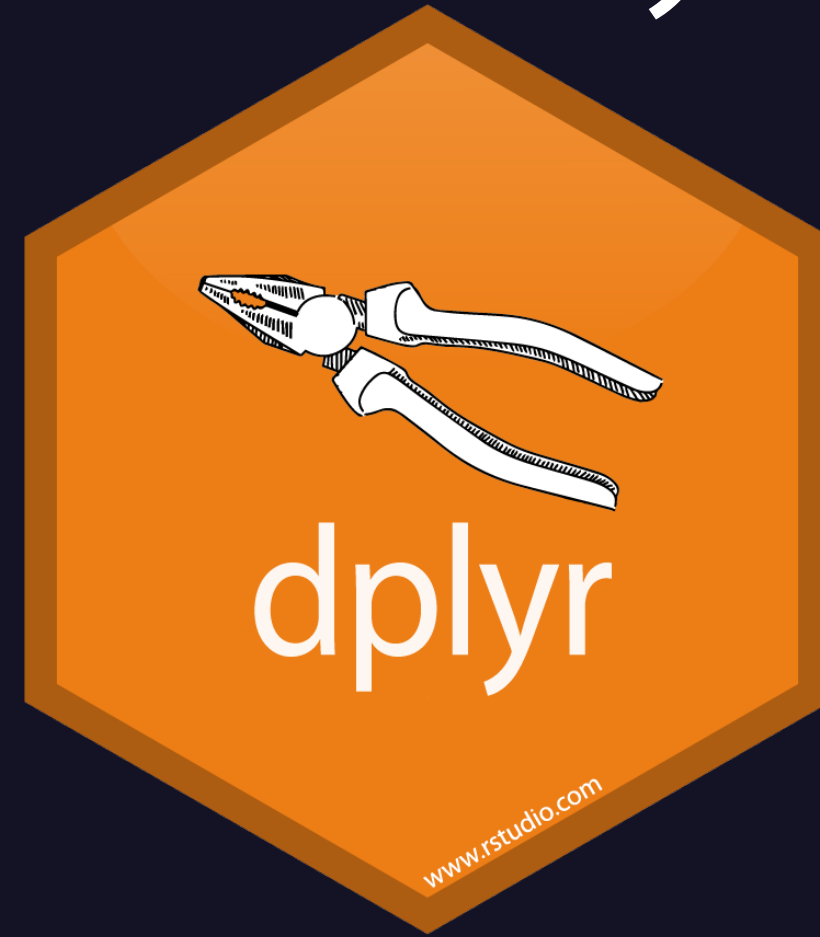
```
#> SELECT `mpg`, `cyl`, `disp`, `hp`
```

```
#> FROM `mtcars`
```

```
#> WHERE (`cyl` > 2.0)
```

```
#> LIMIT 10
```

Evaluación "Tidy" (limpia) = evaluación no-estandar con principios  
(principles NSE)



# Ahora un poco de teoría ~~de juegos~~

1. El código de R es un árbol
2. Des-citar construye árboles
3. Ambientes ligan nombres a valores

Práctica

# Reduce las copias aquí

```
df <- data.frame(  
  g = rep(c("a", "b", "c"), c(3, 2, 2)),  
  b = runif(7),  
  a = runif(7),  
  c = runif(7)  
)
```

```
summarise(df, mean = mean(a), sd = sd(a), n = n())  
summarise(df, mean = mean(b), sd = sd(b), n = n())  
summarise(df, mean = mean(c), sd = sd(c), n = n())
```

```
stat_sum <- function(df, var) {  
  var <- enquo(var)  
  
  summarise(df,  
    mean = mean (!!var),  
    sd = sd (!!var),  
    n = n()  
  )  
}
```



# Tu turno

```
# Este código es frecuentemente usado para calcular
# la proporción de una suma agrupada.
# Completa la siguiente función para simplificar
# este patrón útil

mtcars %>% count(cyl) %>% mutate(prop = n / sum(n))

prop <- function(df, x = n) {
  x <- enquos(x)
  ...
}
```



```
prop <- function(df, x = n) {  
  x <- enquos(x)  
  df %>% mutate(prop = !!x / sum(!!x, na.rm = TRUE))  
}
```

```
prop <- function(df, x = n) {  
  x <- enquos(x)  
  df %>% mutate(prop = prop.table(!!x))  
}
```

```
count_and_prop <- function(df, ..., sort = TRUE) {  
  if ("n" %in% names(df)) {  
    # ¿¿¿hacer algo???  
  }  
  
  df %>%  
    count(..., sort = sort) %>%  
    mutate(n = n / sum(n))  
}
```

# Crea una función que se pueda re-usar para este patrón

```
counts <- starwars %>%  
  group_by(g = homeworld) %>%  
  summarise(n = n()) %>%  
  head(10) %>%  
  mutate(g = reorder(g, n))
```

```
counts %>%  
  ggplot(aes(g, n)) +  
  geom_col() +  
  coord_flip() +  
  xlab("homeworld")
```

# Checa el templado en la siguiente diapositiva

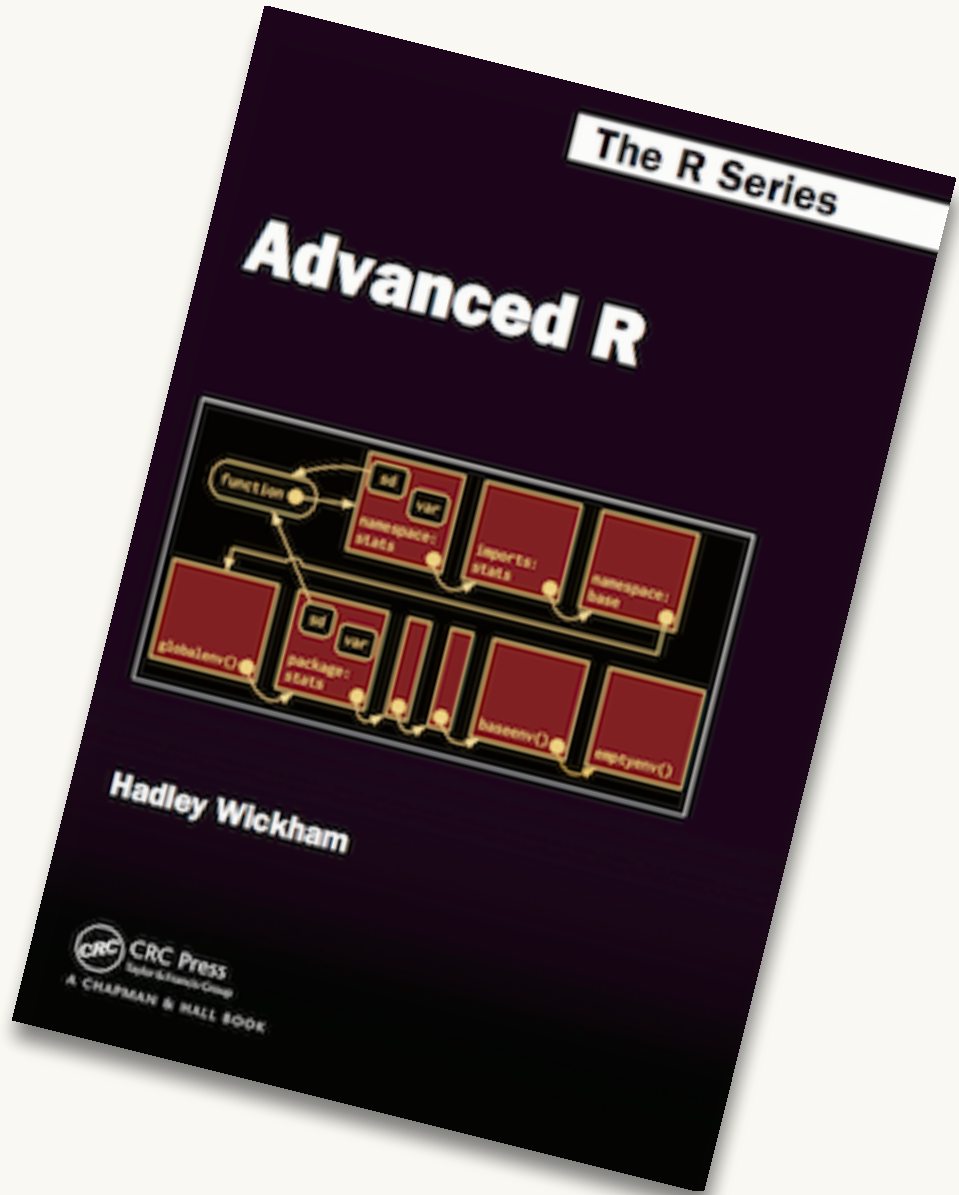
<https://twitter.com/JustTheSpring/status/1082515899821617152>

```
top_n <- function(df, x, n = 10) {  
  
}
```

```
# Reto: ¿puedes cambiar el caso base para  
# manejar mejor los empates?
```

Apr<sup>e</sup>nde m<sup>á</sup>s

# Teoría



<https://adv-r.hadley.nz/expressions.html>  
<https://adv-r.hadley.nz/quasiquotation.html>  
<https://adv-r.hadley.nz/evaluation.html>

<https://youtu.be/nERXS3ssntw>

# Práctica

<https://tidyeval.tidyverse.org>

(siguen desarrollando esta parte)

### Tidy evaluation with rlang : : CHEAT SHEET

#### Vocabulary

**Tidy Evaluation (Tidy Eval)** is not a package, but a framework for doing non-standard evaluation (i.e. delayed evaluation) that makes it easier to program with tidyverse functions.

**Symbol** - a name that represents a value or object stored in R, i.e. `symbol(expr)`

**Environment** - a list-like object that binds symbols (names) to objects stored in memory. Each env contains a link to a second **parent** env, which creates a chain, or search path, of environments, i.e. `environment(current_env())`

**rlang::caller\_env()** Returns calling env of the function it is in.

**rlang::child\_env()** Call from within a function to quote what the user passed to an argument as a quosure. Also **new\_env** as child of parent. Also **env**.

**rlang::current\_env()** Returns execution env of the function it is in.

**Constant** - a bare value (i.e. an atomic vector of length 1), i.e. `run_atomic()`

**Call object** - a vector of symbols/constants/calls that begins with a function name, possibly followed by arguments, i.e. `call(expr, args)`

**Code** - a sequence of symbols/constants/calls that will return a result if evaluated. Code can be:  
1. Evaluated immediately (Standard Eval)  
2. Quoted to use later (Non-standard Eval) i.e. `expression(expr)`

**Expression** - an object that stores quoted code without evaluating it, i.e. `expression(expr)`

**Quosure** - an object that stores both quoted code (without evaluating it) and the code's environment, i.e. `quosure(expr, env)`

**rlang::quo\_get\_env()** Return the environment of a quosure.

**rlang::quo\_set\_env()** Set the environment of a quosure.

**rlang::quo\_get\_expr()** Return the expression of a quosure.

**Expression Vector** - a list of pieces of quoted code created by base R's expression and parse functions. Not to be confused with **expression**.

#### Quoting Code

Quote code in one of two ways (if in doubt use a quosure):

**QUOSURES**

**rlang::quo(expr)** Quote contents as a quosure. Also **quos** to quote multiple expressions, e.g. `quo(a <- 1, b <- 2, c <- quo(a + b))` or `quos(a, b)`

**rlang::enquo(arg)** Call from within a function to quote what the user passed to an argument as a quosure. Also **enquos** for multiple args, e.g. `quo_this <- function(x) enquos(x)` or `quos_these <- function(...) enquos(...)`

**rlang::new\_quosure(expr, env = caller\_env())** Build a quosure from a quoted expression and an environment, e.g. `new_quosure(expr + b, current_env())`

#### Parsing and Deparsing

**Parse** - Convert a string to a saved expression.

**rlang::parse\_expr()** Convert a string to an expression. Also **parse\_exprs**, **sym**, **parse\_quo**, **parse\_quos**, `rlang::parse_expr("a + b")`

**Deparse** - Convert a saved expression to a string.

**rlang::expr\_text()** Convert an expression to a string. Also **expr\_name**, **expr\_text()**

#### Building Calls

**rlang::call2(fn, ..., ns = NULL)** Create a call from a function and a list of args. Use **exec** to create and then evaluate the call. (See back page for full expr = list <- 4, base = 2)

**log** (x = 4, base = 2)  
2

**call2("log", x = 4, base = 2)**  
`call2("log", x = 4, base = 2)`  
`exec("log", x = 4, base = 2)`  
`exec("log", !!log)`

#### EXPRESSION

**rlang::expr(expr)** Quote contents. Also **exprs** to quote multiple expressions, e.g. `expr(a <- 1, b <- 2, c <- quo(a + b))` or `expr(a, b, a + b)`

**rlang::enexpr(arg)** Call from within a function to quote what the user passed to an argument as a symbol, except strings. Also **enexprs**.

**rlang::enexprs()** Call from within a function to quote what the user passed to an argument as a symbol, except strings. Also **enexprs**.

**rlang::enexprs()** Call from within a function to quote what the user passed to an argument as a symbol, except strings. Also **enexprs**.

#### Quoted Expression

An expression that has been saved by itself. A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in.

**rlang::eval\_bare(expr, env = parent.frame())** Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval, tidy.

**rlang::eval\_tidy(expr, data = NULL, env = caller\_env())** Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval, tidy.

**Data Mask** - If data is non-NULL, eval, tidy inserts data into the search path before env, matching symbols to names in data.

Use the pronoun **data** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

#### QUOSURES (and quoted exprs)

**rlang::eval\_bare(expr, env = parent.frame())** Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval, tidy.

**rlang::eval\_tidy(expr, data = NULL, env = caller\_env())** Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval, tidy.

**Data Mask** - If data is non-NULL, eval, tidy inserts data into the search path before env, matching symbols to names in data.

Use the pronoun **data** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

#### Evaluation

To evaluate an expression, R:


- Looks up the symbols in the expression in the active environment or a supplied env, followed by the environment's parents
- Executes the calls in the expression

The result of an expression depends on which environment it is evaluated in.

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 864-448-1212 • rstudio.com • Learn more at <https://www.tidyverse.org> • rlang 0.3.0 • Updated: 2018-11

Revisemos algunos cambios desde enero del 2019

<https://ryo-n7.github.io/2019-07-21-user2019-reflections/>

HOME 日本語 VISUALIZATION GALLERY

## My useR! 2019 Highlights & Experience: Shiny, R Community, {packages}, and more!

*Posted on July 21, 2019*

The useR! Conference was held in Toulouse, France and for me this was my second useR! after my first in Brisbane last year. This time around I wanted to write about my experiences and some highlights similar to my post on the [RStudio::Conference 2019 & Tidyverse Dev Day](#) earlier this year. This blog post will be divided into 4 sections: **Programming, Shiny, {Packages}, and Touring Toulouse.**





This work is licensed as  
Creative Commons  
Attribution-ShareAlike 4.0  
International

To view a copy of this license, visit  
<https://creativecommons.org/licenses/by-sa/4.0/>