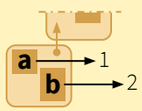
Evaluación *tidy* con rlang : : **GUÍA RÁPIDA**

Vocabulario

Evaluación tidy (**Tidy Eval**) no es un paquete, sino un entorno para la evaluación no estándar (es decir, evaluación tardía) que hace más fácil programar funciones en el ecosistema tidyverse.

pi — **Símbolo** (***symbol***) - un nombre que representa un valor u objeto almacenado en R. *is_symbol(expr(pi))*



Entorno (***env*** de ***Environment***)**-** un objeto de tipo lista que enlaza símbolos (nombres) a objetos almacenados en memoria. Cada entorno ***env*** tiene un enlace a un segundo entorno, **padre *env***, creando una cadena, o camino de búsqueda, de entornos. *is_environment(current_env())*

rlang::**caller_env**(n = 1) Devuelve el entorno llamado en la función en la que está.
rlang::**child_env**(.parent, ...) Crea un nuevo entorno hijo de .parent.
rlang::**current_env**() Devuelve la ejecución en env de la función en la que está.

1 — **Constante** - un valor puro, es decir, un vector de longitud 1. *is_bare_atomic(1)*

abs (**1**) — **Objeto lamada** (***call***)- un vector de símbolos, constantes y/o llamadas que empieza con el nombre de una función, posiblemente seguida de argumentos. *is_call(expr(abs(1)))*

pi — código
3.14 — resultado
Código - una secuencia de símbolos, constantes y/o llamadas que devuelve un resultado al ser evaluada. El código puede ser:
1. Evaluado inmediatamente (**Standard Eval**)
2. Guardado para usar después (**Non-Standard Eval**)

is_expression(expr(pi))

e / **a + b** — **Expresión** (***expression***) - un objeto que almacena código entrecomillado sin evaluarlo. is_*expression(expr(a + b))*

q / **a + b,** [a b] — **Quosure**- un objeto que almacena un código entrecomillado (sin evaluarlo) junto a su entorno. *is_quosure(quo(a + b))*

[a b] rlang::**quo_get_env**(quo) Devuelve el entorno de una ***quosure***.

[a b] rlang::**quo_set_env**(quo, expr) Determina el entorno de una ***quosure***.

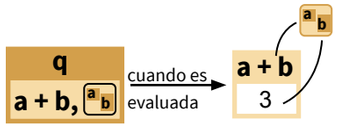
a + b rlang::**quo_get_expr**(quo) Devuelve la expresión de una ***quosure***.

Vector de expresiones - una lista de código entrecomillado creada por las funciones de R base *expression* y *parse*. No confundir con ***expression***.

Entrecomillar código

Entrecomilla código de dos maneras (ante la duda usa una quosure):

QUOSURES



Quosure- Una expresión que ha sido guardada junto con un *entorno* (AKA una *closure*).

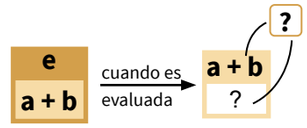
Una ***quosure*** puede ser evaluada más tarde en el entorno con el que fue guardada para devolver un resultado previsible.

rlang::**quo**(expr) Entrecomilla código en una ***quosure***. También **quos** para múltiples argumentos. *a <- 1; b <- 2; q <- quo(a + b); qs <- quos(a, b).*

rlang::**enquo**(arg) Llama dentro de una función para citar lo que el usuario pasó como argumento como una ***quosure***. También **enquos** para múltiples argumentos.
quote_esto < - function(x) enquo(x)
quote_estos < - function(…) enquos(…)

rlang::**new_quosure**(expr, env = caller_env()) Crea una ***quosure*** a partir de una expresión entrecomillada y un entorno.
new_quosure(expr(a + b), current_env())

EXPRESSION



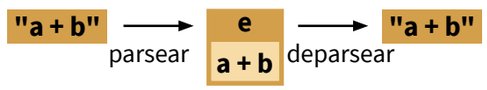
Expresión entrecomillada- Una expresión que ha sido guardada. Una expresión entrecomillada puede ser evaluada luego para devolver un resultado que depende del entorno en el cual es evaluada.

rlang::**expr**(expr) Entrecomilla una expresión. También **exprs** para múltiples expresiones. *a <- 1; b <- 2; e <- expr(a + b); es <- exprs(a, b, a + b)*

rlang::**enexpr**(arg) Llama dentro de una función para citar lo que el usuario pasó como argumento. También **enexprs** para múltiples argumentos.
quote_esto < - function(x) enexpr(x)
quote_estos< - function(…) enexprs(…)

rlang::**ensym**(x) Llama dentro de una función para citar lo que el usuario pasó como argumento como un símbolo (acepta cadena de caracteres). También **ensyms**.
quote_nombre < - function(name) ensym(name)
quote_nombres < - function(…) ensyms(…)

Parseando y deparseando



Parse - Convertir una cadena de caracteres en una expresión guardada.

rlang::**parse_expr**(x) Convertir una cadena de caracteres en una expresión. También **parse_exprs**, **sym**, **parse_quo**, **parse_quos**.
e<-*parse_expr("a+b")*

Deparse - Convertir una expresión guardada en una cadena de caracteres.

rlang::**expr_text**(expr, width = 60L, nlines = Inf) Convertir una expresión en una cadena de caracteres. También **quo_name**.
expr_text(e)

Creando llamadas

rlang::**call2**(.fn, ..., .ns = NULL) Crea una llamada a partir de una función y una lista de argumentos. Usa **exec** para crear y luego evaluar la llamada. (Ver reverso para **!!!**) *args <- list(x = 4, base = 2)*

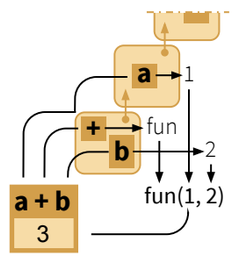
log (x = **4**, base = **2**)

2

call2("log", x = 4, base = 2)
call2("log", !!!args)

exec("log", x = 4, base = 2)
exec("log", !!!args)

Evaluación



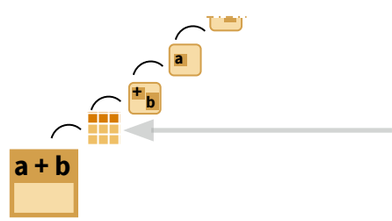
Para evaluar una expresión, R :

1. Busca los símbolos de la expresión en el entorno activo (o en uno suministrado), continuando por los padres del entorno.
2. Ejecuta las llamadas en la expresión**.**

El resultado de una expresión depende del entorno en el cual es evaluada.

EXPRESIÓN ENTRECOMILLADA

rlang::**eval_bare**(expr, env = parent.frame()) Evalúa ***expr*** en ***env***. *eval_bare(e, env =.GlobalEnv)*



a <- 1; b <- 2
*p <- quo(**.data$**a + **!!**b)*
mask <- tibble(a = 5, b = 6)
eval_tidy(p, data = mask)

QUOSURES

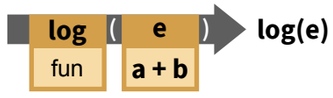
rlang::**eval_tidy**(expr, data = NULL, env = caller_env()) Evalúa ***expr*** en ***env***, usando ***data*** como una **máscara de datos**. Evaluará ***quosures*** en su correspondiente entorno. *eval_tidy(q)*

Máscara de datos - Si ***data*** no es NULL, eval_tidy inserta ***data*** en el camino de búsqueda previo a ***env***, emparejando símbolos a nombres de ***data***.
Usa el pronombre **.data$** para forzar que un símbolo sea emparejado en ***data***, y **!!** (ver reverso) para forzar que un símbolo sea emparejado en los entornos.

Cuasientrecomillado (!!, !!!,:=)

ENTRECOMILLADO

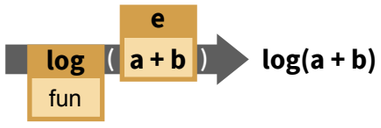
Almacenar una expresión sin evaluarla.
e <- expr(a + b)



expr(log(e))

CUASIENTRECOMILLADO

Entrecomillar algunas partes de una expresión al evaluar y luego insertar los resultados de otras (**desencomillando** otras).
e <- expr(a + b)

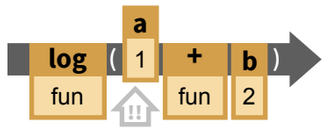


*expr(log(**!!**e))*

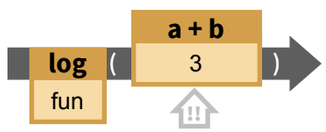
rlang provee **!!**, **!!!**, y **:=** para cuasiencomillar.

!!, **!!!**, y **:=** no son funciones sino sintaxis (símbolos reconocidos por las funciones a las cuales son pasados). Es similar a como
. es usado por magrittr::**%>%()**
. es usado por stats::**lm()**
.x es usado por purrr::**map()**, etc.

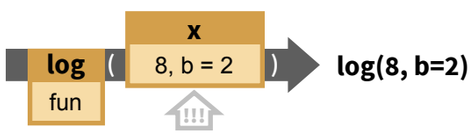
!!, **!!!**, y **:=** son solo reconocidos por algunas funciones de rlang y funciones que usan estas funciones (como las funciones de tidyverse).



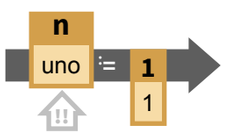
*expr(log(**!!**a + b))*



*expr(log(**!!**(a + b)))*



*expr(log(**!!!**x))*



*tibble::tibble(**!!**n **:=** 1)*

!! Desencomilla el símbolo o llamada que le sigue. Se pronuncia "desencomillar" o "bang-bang."
a <- 1; b <- 2
expr(log(!!a + b))

Combina **!!** con **()** para desencomillar una expresión más larga.
a <- 1; b <- 2
expr(log(!!(a + b)))

!!! Desencomilla un vector o una lista y segmenta (splice) los resultados como argumentos en la llamado circundante. Se pronuncia "unquote splice" o "bang-bang-bang."
x <- list(8, b = 2)
expr(log(!!!x))

:= Reemplaza un = para permitir desencomillado del nombre que aparece del lado izquierdo del =. Usar con **!!**
n <- expr(uno)
tibble::tibble(!!n := 1)

Recetas de programación

Función entrecomilladora- Una función que entrecomilla alguno de sus argumentos internamente para una evaluación tardía en un entorno adecuado. Tienes que seguir pasos específicos para programar de forma segura con estas funciones.

Cómo reconocer una función entrecomilladora?
Una función entrecomilla un argumento si el argumento devuelve un error cuando se ejecuta.

Muchas funciones tidyverse son funciones entrecomilladoras: por ejemplo **filter**, **select**, **mutate**, **summarise**, etc.

dplyr::filter(cars, **speed = = 25**)

speed	dist
1	25
85	

speed == 25
Error!

ESCRIBIR UNA FUNCIÓN QUE RECONOCE CUASI ENTRECOMILLADO (!!,!!!,:=)

1. Capturar el argumento entrecomillado con rlang::**enquo**.
2. Evaluar el argumento con rlang::**eval_tidy**.

```
sumar1 <- function(x) {
  q <- rlang::enquo(x)                1
  rlang::eval_tidy(q) + 1             2
}
```

PROGRAMAR CON UNA FUNCIÓN ENTRECOMILLADORA

```
data_media <- function(data, var) {
  require(dplyr)
  var <- rlang::enquo(var)            1
  data %>%
    summarise(mean = mean(!!var))     2
}
```

1. Capturar el argumento a ser entrecomillado con rlang::**enquo**.
2. Desencomillar el argumento en la función entrecomilladora con **!!**.

PASAR MÚLTIPLES ARGUMENTOS A UNA FUNCIÓN ENTRECOMILLADORA

```
media_grupo <- function(data, var, …) {
  require(dplyr)
  var <- rlang::enquo(var)
  group_vars <- rlang::enquos(…)      1
  data %>%
    group_by(!!!group_vars) %>%       2
    summarise(mean = mean(!!var))
}
```

1. Capturar los argumentos a ser entrecomillados con rlang::**enquos**.
2. Desencomillar y segmentar los argumentos del usuario en la función entrecomilladora con **!!!**.

PASAR A NOMBRES DE ARGUMENTOS DE UNA FUNCIÓN ENTRECOMILLADORA

```
nombrar_v <- function(data, var, nombre){
  require(dplyr)
  var <- rlang::enquo(var)
  nombre <- rlang::ensym(nombre)      1
  data %>%
    summarise(!!nombre := mean(!!var))2
}
```

1. Capturar el a ser entrecomillado con rlang::**ensym**.
2. Desencomillar el argumento del usuario en la función entrecomilladora con **!!** y **:=**.

MODIFICAR ARGUMENTOS DEL USUARIO

```
mi_tarea <- function(f, v, df) {
  f <- rlang::enquo(f)                1
  v <- rlang::enquo(v)
  todo <- rlang::quo((!!f)(!!v))      2
  rlang::eval_tidy(todo, df)          3
}
```

1. Capturar los argumentos con rlang::**enquo**.
2. **Desencomillar** los argumentos en una nueva ***expression*** o ***quosure*** a utilizar.
3. **Evaluar** la nueva ***expression/quosure*** en lugar del argumento original

APLICAR UN ARGUMENTO A UN CONJUNTO DE DATOS

```
subset2 <- function(df, filas) {
  filas <- rlang::enquo(filas)        1
  vals <- rlang::eval_tidy(filas, data = df)
                                      2
  df[vals, , drop = FALSE]
}
```

1. Capturar el argumento con rlang::**enquo**.
2. Evaluar el argumento con rlang::**eval_tidy**. Pasar el conjunto de datos a ***data*** para usar como una máscara de datos.
3. **Sugerir** en tu documentación que los usuarios usen los pronombres **.data** y **.env**.

PASAR LOS CHEQUEOS DE CRAN

```
#' @importFrom rlang .data            1
mutate_y <- function(df) {
  dplyr::mutate(df, y = .data$a +1)   2
}
```

Los argumentos encomillados en funciones de tidyverse pueden provocar una **R CMD check** NOTE sobre variables globales no definidas. Para evitar esto:

1. Importar rlang::**.data** en tu paquete, quizás con la etiqueta de roxygen2. **@importFrom rlang .data**
2. Usar el pronombre **.data$** delante de los nombres de variables de funciones de tidyverse.