

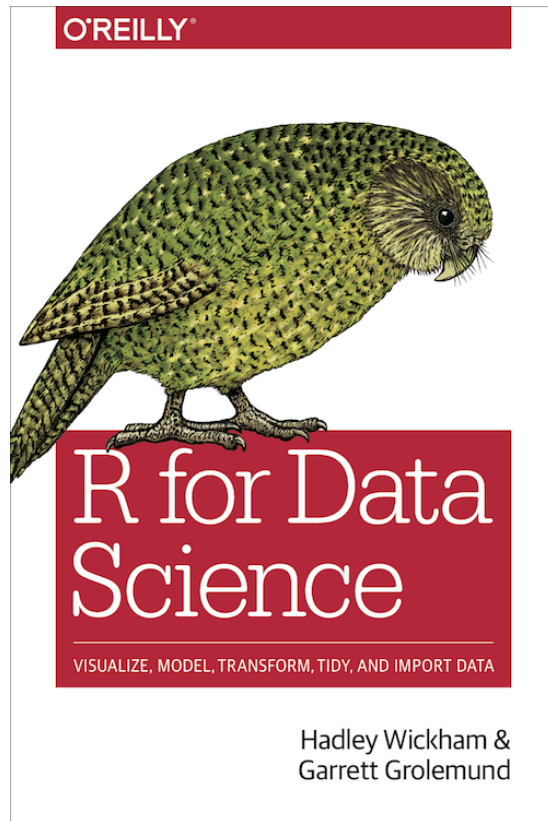


Funciones

R para Ciencia de Datos

04/05/2020

- R for Data Science



- R para Ciencia de Datos



- Las funciones permiten automatizar algunas tareas comunes de una forma más poderosa y general que copiar-y-pegar
- Consejos para:
 - Escribir funciones
 - Estilo de código

Prerrequisitos

- R
- RStudio



Principio DRY - “Do not Repeat Yourself”



- “No repetirse a uno mismo”: Cuanta más repetición tengas en tu código, más lugares tendrás que recordar de actualizar cuando las cosas cambien, y es más probable que crees errores (bugs) a lo largo del tiempo.

- Extraer el código repetido en una función es una buena idea, ya que previene que cometamos errores.
- Otra ventaja de las funciones es que si nuestros requerimientos cambian, solo necesitamos hacer modificaciones en un solo lugar.
- Deberías considerar escribir una función cuando has copiado y pegado un bloque de código más de dos veces.
- Para escribir una función, lo primero que necesitas hacer es analizar el código. ¿Cuántos inputs tiene?
- Es más fácil empezar con código que funciona y luego convertirlo en una función; es más difícil crear la función y luego tratar que funcione.
- Una función devuelve el último valor que calculó.

...Para crear una función nueva

1. Necesitas elegir un nombre para la función.
2. Listar los inputs, o argumentos, de la función dentro de function.
3. Situar el código que has creado en el cuerpo de una función, un bloque de { que sigue inmediatamente a function(...).

```
nome_de_la_funcion <- function(argumentos){  
  cuerpo de la función  
}
```

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

- `rescale01` es el nombre de la función
- `x` es lo unico argumento
- Cuerpo de la función:

```
rng <- range(x, na.rm = TRUE)
```

```
(x - rng[1]) / (rng[2] - rng[1])
```

- Ejemplo de la utilización de la función:

```
rescale01(c(0, 5, 10))
```

```
## [1] 0.0 0.5 1.0
```

- Es una buena idea chequear tu función con algunos inputs diferentes:

```
rescale01(c(-10, 0, 10))
```

```
## [1] 0.0 0.5 1.0
```

```
rescale01(c(1, 2, 3, NA, 5))
```

```
## [1] 0.00 0.25 0.50 NA 1.00
```

- Sobre tests formales y automatizados -> se llama pruebas unitarias (unit testing): lea mas en <https://r-pkgs.org/tests.html>

Las funciones son para ...



...los seres humanos y para las computadoras

- Consejos para estilo de código: cosas que debes tener en mente a la hora de escribir funciones entendibles para otras personas

Nombre de una función



- Debería ser corto, pero que evoque claramente lo que la función hace.
- Generalmente, los nombres de las funciones deberían ser verbos y los argumentos sustantivos, pero hay algunas excepciones.
- Ejemplos del libro:

Muy corto

`f()`

No es un verbo y es poco descriptivo

`mi_funcion_genial()`

Largos, pero descriptivos

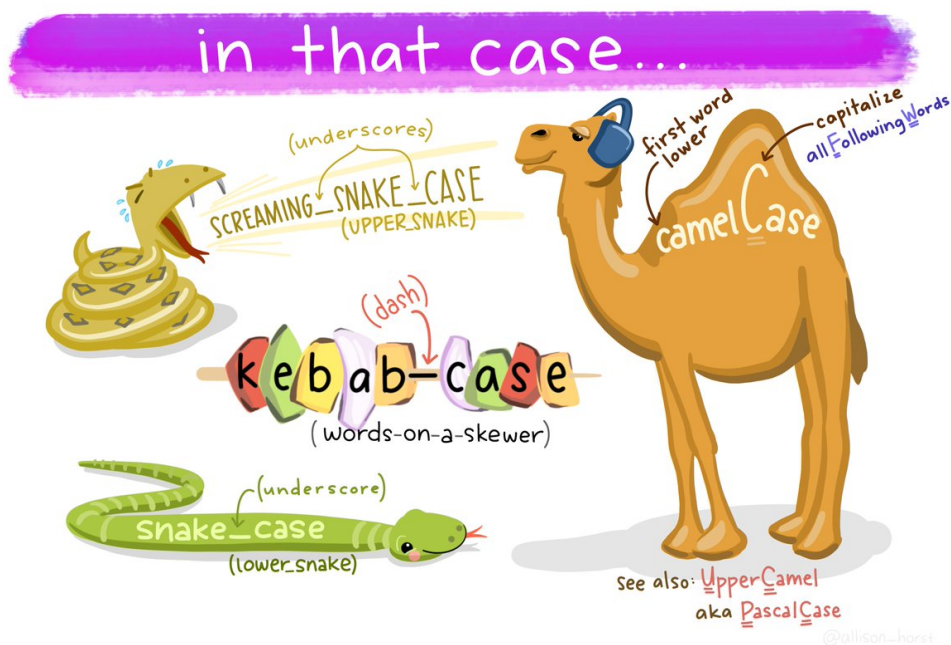
`imputar_faltantes()`

`colapsar_anios()`

Case Style



- Lo importante es que seas consistente: elije uno o el otro y quédate con él.
- Recomendación: `snake_case` - cada palabra en minúscula está separada por un guión bajo



Source: [@allison_horst](#)

- Consejo personal para tibbles



Source: [@allison_horst](#)

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
```

```
iris_clean <- janitor::clean_names(iris)
names(iris_clean)
```

```
## [1] "sepal_length" "sepal_width" "petal_length" "petal_width" "species"
```

Nombres y argumentos consistentes



- Si tienes una familia de funciones que hacen cosas similares, asegúrate de que tengan nombres y argumentos consistentes.
- Utiliza un prefijo común para indicar que están conectadas. Eso es mejor que usar un sufijo común, ya que el autocompletado te permite escribir el prefijo y ver todos los otros miembros de la familia.

Bien

`input_select()`

`input_checkbox()`

`input_text()`

No tan bien

`select_input()`

`checkbox_input()`

`text_input()`

.. ya existentes

- Siempre que sea posible, evita sobrescribir funciones y variables ya existentes.
- No siempre es posible hacer esto, ya que hay un montón de nombres buenos que ya han sido utilizados por otros paquetes.
- Evitar el uso de los nombres más comunes de R base ahorrará confusiones.

```
# ¡No hagas esto!
```

```
T <- FALSE
```

```
c <- 10
```

```
mean <- function(x) sum(x)
```

- Comienzan con #
- Explicar el "porqué" de tu código.
 - Por qué elegiste este enfoque frente a otras alternativas?
 - ¿Qué otra cosa probaste que no funcionó?
 - Es una gran idea capturar este tipo de pensamientos en un comentario.

```
# Esto es un comentario! :)
```

Dividir el código en partes



- Puedes utilizar comentarios y - o =
- Más fácil detectar los fragmentos
- Más fácil de leer

```
# Cargar los datos -----  
# Graficar los datos -----
```


Ejecución condicional



- Una sentencia if (si) te permite ejecutar un código condicional.

```
if (condition) {  
  # el código que se ejecuta cuando la condición es verdadera (TRUE)  
} else {  
  # el código que se ejecuta cuando la condición es falsa (FALSE)  
}
```

Puedes encadenar múltiples sentencias if juntas:

```
if (this) {  
  # haz aquello  
} else if (that) {  
  # haz otra cosa  
} else {  
  #  
}
```

- La condición debe evaluar como TRUE o FALSE:

```
if (c(TRUE, FALSE)) {}
```

```
#> Warning in if (c(TRUE, FALSE)) {: the condition has length > 1 and only the  
#> first element will be used  
#> NULL
```

```
if (NA) {}
```

```
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

Estilo del código



```
# Bien
if (y < 0 && debug) {
  message("Y es negativo")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Mal
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

Estilo del código



- Atajo útil para indentación: `Ctrl + Shift + A`

A screenshot of the RStudio editor interface. The top toolbar includes icons for undo, redo, save, source on save, search, and a dropdown menu. Below the toolbar, the first line of code is displayed: `1 df_titanic %>% filter(sexo == "masculino" & sobreviveu == "sim") %>% arrange(classe)`. A light gray vertical bar highlights the first line of code.

```
1 df_titanic %>% filter(sexo == "masculino" & sobreviveu == "sim") %>% arrange(classe)
```

- 2 conjuntos amplios:
 - Argumentos que proveen los **datos** a computar
 - Argumentos que controlan los **detalles de la computación**.
- Generalmente, argumentos relativos a los datos deben ir primero.
- Cuando llamas una función, generalmente omites los nombres de los argumentos de datos justamente porque son los más comúnmente usados.
- En la sección Help de una función, tenemos las informaciones de argumentos posibles.

Ejemplo:

```
?mean
```

Arguments:

x **(provee los datos a computar)** An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.

trim **(controlan los detalles de la computación)** the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

na.rm **(controlan los detalles de la computación)** a logical value indicating whether NA values should be stripped before the computation proceeds.

... **(controlan los detalles de la computación)** further arguments passed to or from other methods.

Elección de nombres

- Los nombres de los argumentos también son importantes.
- A R no le importa, pero sí a quienes leen tu código.
- En general, deberías preferir nombres largos y más descriptivos. Pero hay un puñado de nombres muy comunes y muy cortos. Vale la pena memorizar estos:

x, **y**, **z**: vectores.

w: un vector de pesos.

df: un data frame.

i, **j**: índices numéricos (usualmente filas y columnas).

n: longitud, o número de filas.

p: número de columnas.

- **El orden** de los argumentos es importante!
- Ejemplo. `mean()` tiene los argumentos: `x`, `trim`, `na.rm`. Si solo usa el primer y el tercer argumento (`x` y `na.rm`), debe nombrar el argumento para no dar un error, de lo contrario, R indicará este input para el segundo argumento (en orden).

```
mean(1:10, TRUE)
```

```
# Error in mean.default(1:10, TRUE): 'trim' deve ser numérico de comprimento 1
```

- La mejor forma:

```
mean(1:10, na.rm = TRUE)
```

```
## [1] 5.5
```


Valores por defecto

- El valor por defecto debería ser casi siempre el valor más común.
- Se especifica un valor por defecto de la misma manera en la que se llama a una función con un argumento nombrado.
- Ejemplo:

```
round(5.55555)
```

```
## [1] 6
```

```
round(5.55555, digits = 1)
```

```
## [1] 5.6
```

```
round(5.55555, digits = 2)
```

```
## [1] 5.56
```

Valores por defecto

- Ejemplo del libro

```
# Computar intervalo de confianza alrededor de la media usando  
# la aproximación normal  
mean_ci <- function(x, conf = 0.95) {  
  se <- sd(x) / sqrt(length(x))  
  alpha <- 1 - conf  
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))  
}
```

```
x <- runif(100)  
mean_ci(x)
```

```
## [1] 0.441236 0.553990
```

```
mean_ci(x, conf = 0.99)
```

```
## [1] 0.4235211 0.5717050
```

- Ten en cuenta que cuando llamas a una función, debes colocar un espacio alrededor de = y siempre poner un espacio después de la coma, no antes (como cuando escribes en español).
- El uso del espacio en blanco hace más fácil echar un vistazo a la función para identificar los componentes importantes.

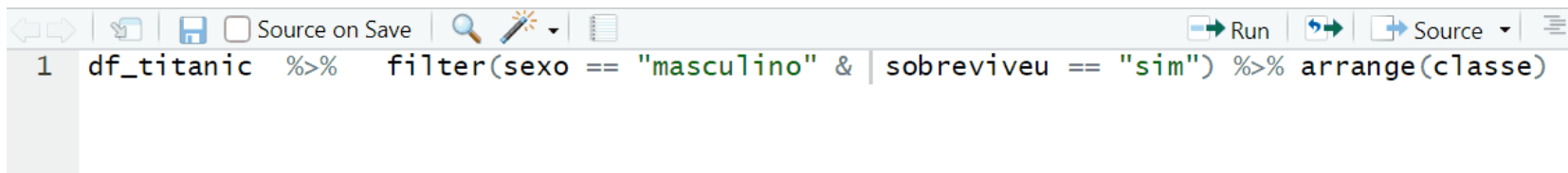
Bien

```
promedio <- mean(pies / 12 + pulgadas, na.rm = TRUE)
```

Mal

```
promedio<-mean(pies/12+pulgadas,na.rm=TRUE)
```

- Atajo útil para esto: `ctrl + shift + A` (de nuevo! 😊)



```
1 df_titanic %>% filter(sexo == "masculino" & sobreviveu == "sim") %>% arrange(classe)
```

Chequear valores

- Chequear si los inputs utilizados son adecuadas para la función.
- Es una buena práctica verificar las condiciones previas importantes y arrojar un error (con `stop()`, parar), si estas no son verdaderas.
- Debe haber un equilibrio entre la cantidad de tiempo que inviertes en hacer que tu función sea sólida y la cantidad de tiempo que pasas escribiéndola.

Chequear valores

Ejemplo:

```
wt_mean <- function(x, w) {  
  if (length(x) != length(w)) {  
    stop("`x` y `w` deben tener la misma extensión", call. = FALSE)  
  }  
  sum(w * x) / sum(w)  
}
```

```
wt_mean(1:10, 1:5)
```

```
# Erro: `x` y `w` deben tener la misma extensión
```

Chequear valores

- Una opción útil es incorporar `stopifnot()`: esto comprueba que cada argumento sea TRUE. En caso contrario genera un mensaje de error. Al usar `stopifnot()` afirmas lo que debería ser cierto en lugar de verificar lo que podría estar mal.

Ejemplo:

```
wt_mean <- function(x, w, na.rm = FALSE) {  
  stopifnot(is.logical(na.rm), length(na.rm) == 1)  
  stopifnot(length(x) == length(w))  
  
  if (na.rm) {  
    miss <- is.na(x) | is.na(w)  
    x <- x[!miss]  
    w <- w[!miss]  
  }  
  sum(w * x) / sum(w)  
}  
  
# wt_mean(1:6, 6:1, na.rm = "foo")  
#> Error in wt_mean(1:6, 6:1, na.rm = "foo"): is.logical(na.rm) is not TRUE
```

Punto-punto-punto (...)

- `...` es un argumento especial: captura cualquier número de argumentos que no estén contemplados de otra forma.
- Cualquier argumento mal escrito no generará un error. Esto hace que sea más fácil que los errores de tipeo pasen inadvertidos.

Ejemplo

```
sum(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] NA
```

```
sum(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 3
```

- Cosas que debes considerar al retornar un valor:

¿Devolver un valor antes hace que tu función sea más fácil de leer?

¿Puedes hacer tu función apta para utilizarla con pipes (`%>%`)?

Sentencias de retorno explícitas

- Puedes optar por devolver algo anticipadamente haciendo uso de la función `return()`

Escribir funciones aptas para un pipe `%>%`

- Hay dos tipos básicos de funciones aptas para pipes: transformaciones y efectos secundarios.
- **Transformaciones** : se ingresa un objeto como primer argumento y se retorna una versión modificada del mismo.

```
iris %>% janitor::clean_names()
```

- **Efectos Secundarios** : el objeto ingresado no es modificado, sino que la función realiza una acción sobre el objeto. -> Las funciones de efectos secundarios deben retornar “invisiblemente” el primer argumento, de manera que aún cuando no se impriman, puedan ser utilizados en una secuencia de pipes.

Ejemplo con **Efectos Secundarios** :

```
mostrar_faltantes <- function(df) {  
  n <- sum(is.na(df))  
  cat("Valores faltantes: ", n, "\n", sep = "")  
  
  invisible(df)  
}
```

```
x <- mostrar_faltantes(iris)
```

```
## Valores faltantes: 0
```

```
class(x)
```

```
## [1] "data.frame"
```

```
dim(x)
```

```
## [1] 150    5
```

Entorno (Environment)



- Cruciales para que algunas funciones trabajen

Ejemplo:

```
f <- function(x) {  
  x + y  
}
```

- Como y no está definida dentro de la función, R mirará dentro del entorno donde la función fue definida

```
y <- 100  
f(10)
```

```
## [1] 110
```

```
y <- 1000  
f(10)
```

```
## [1] 1010
```

- Muchas gracias!
- Diapositivas creadas con el paquete **Xaringan**, con el tema **metropolis** modificado por **Bea Milz**.
- Source: **Funciones. R para Ciencia de Datos** by Hadley Wickham and Garrett Grolemund.