

Diseño de un API

Julio 2019

Traducido por Leonardo Collado-Torres

@fellgernon


lcolladotor@gmail.com

lcolladotor.github.io


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

@hadleywickham

Chief Scientist, RStudio



El API define como
interactúas con el código



La interface, no el diseño interior

Caso de estudio

¿Qué es lo que hace difícil de aprender a las funciones de R base?

strsplit(x, split, ...)

grep(pattern, x, value = FALSE, ...)

grepl(pattern, x, ...)

sub(pattern, replacement, x, ...)

gsub(pattern, replacement, x, ...)

regexpr(pattern, text, ...)

gregexpr(pattern, text, ...)

regexec(pattern, text, ...)

substr(x, start, stop)

nchar(x, type, ...)

Algunos problemas

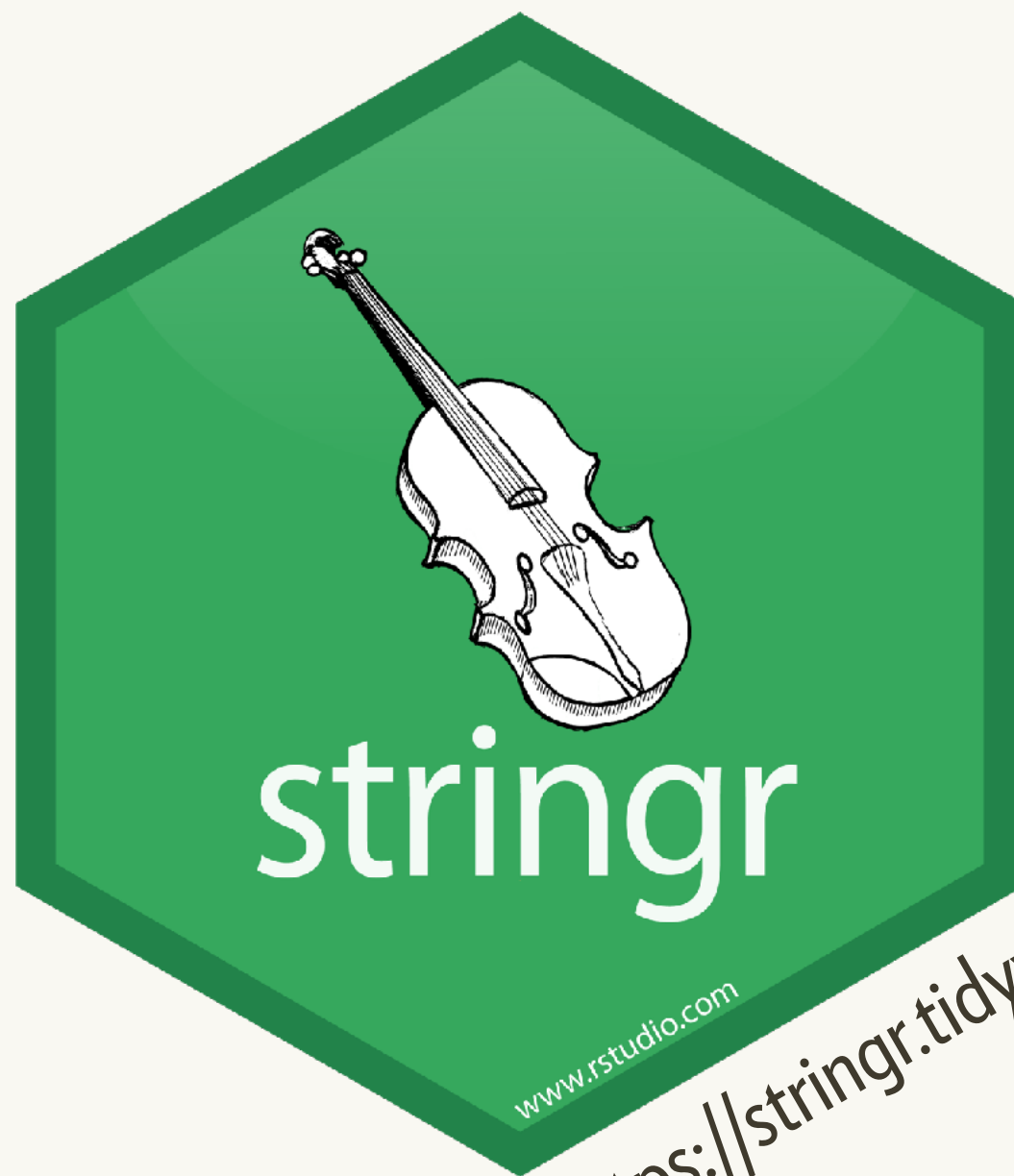
Nombres: Los nombres de las funciones no tienen un tema en común, no hay prefijo común. Los nombres son concisos a cuesta de la expresividad.

Argumentos: Los nombres de los argumentos y su orden no son consistentes, y data no es el primer argumento (o el segundo o el tercero). A veces text, a veces x.

Estabilidad de tipo de valores: `grep()` no es estable: a veces regresa un string o un integer. No puedes usar el valor de salida de `gregexpr()` como valor inicial para `substr()`



Cada problema individual es
pequeño



<https://stringr.tidyverse.org>

“Cada [función] es perfecta tal y como está ... y podría salir beneficiada de algunas mejoras.”

—*Shunryu Suzuki*

Contempla los
nombres con cuidado

“Una rosa por cualquier
otro nombre olería igual de
dulce.”

— *Shakespeare*



“Una **función** por cualquier
otro nombre **no** olería igual de
dulce.”

— *Hadley*



Principio:

Usa verbos para funciones
que cumplen alguna acción

stringr usa verbos evocativos

`str_split()`

`str_detect()`

`str_locate()`

`str_subset()`

`str_extract()`

`str_replace()`

Pero buenos verbos no siempre existen

`str_to_lower()`

`str_to_upper()`

ggplot2 usa sustantivos

`geom_line()`

`scale_x_continuous()`

`coord_fixed()`

Errores pasados

Evita verbos con doble significado

`filter()`

`weather()`

`cleave()`

Evita verbos con variantes en inglés de GB/EEUU

`summarise()` / `summarize()`

`scale_colour_grey()` / `scale_color_grey()`

Principio:

Usa prefijos para agrupar
funciones relacionadas

La mayoría de las funciones de stringr empiezan con str_

`str_split()`

`str_detect()`

`str_locate()`

`str_replace()`

...

Principio:

Usa sufijos para variaciones
de un tema

Usa sufijos para variaciones de un tema

`str_extract()`

`str_extract_all()`

`str_replace()`

`str_replace_all()`

`str_split()`

`str_split_fixed()`

¿Por qué no argumentos?

`str_extract(all = TRUE)`

`str_split(fixed = TRUE)`

Tu turno

¿Qué funciones de stringr violan estos principios?

¿Qué otras funciones del tidy verse violan estos principios?

No empiezan con str_

invert_match()

word()

fixed()

regexp()

No son verbos

str_which()

str_c()

str_length()

str_sub()

Anticipa el uso del
pipe %>%

¿Por qué es útil el pipe?

```
library(dplyr)
library(nycflights13)

by_dest <- group_by(flights, dest)
dest_delay <- summarise(by_dest,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
big_dest <- filter(dest_delay, n > 100)
arrange(big_dest, desc(delay))
```

Pero escoger buenos nombres es una trabajo difícil

```
foo <- group_by(flights, dest)
foo <- summarise(foo,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo <- filter(foo, n > 100)
arrange(foo, desc(delay))
```

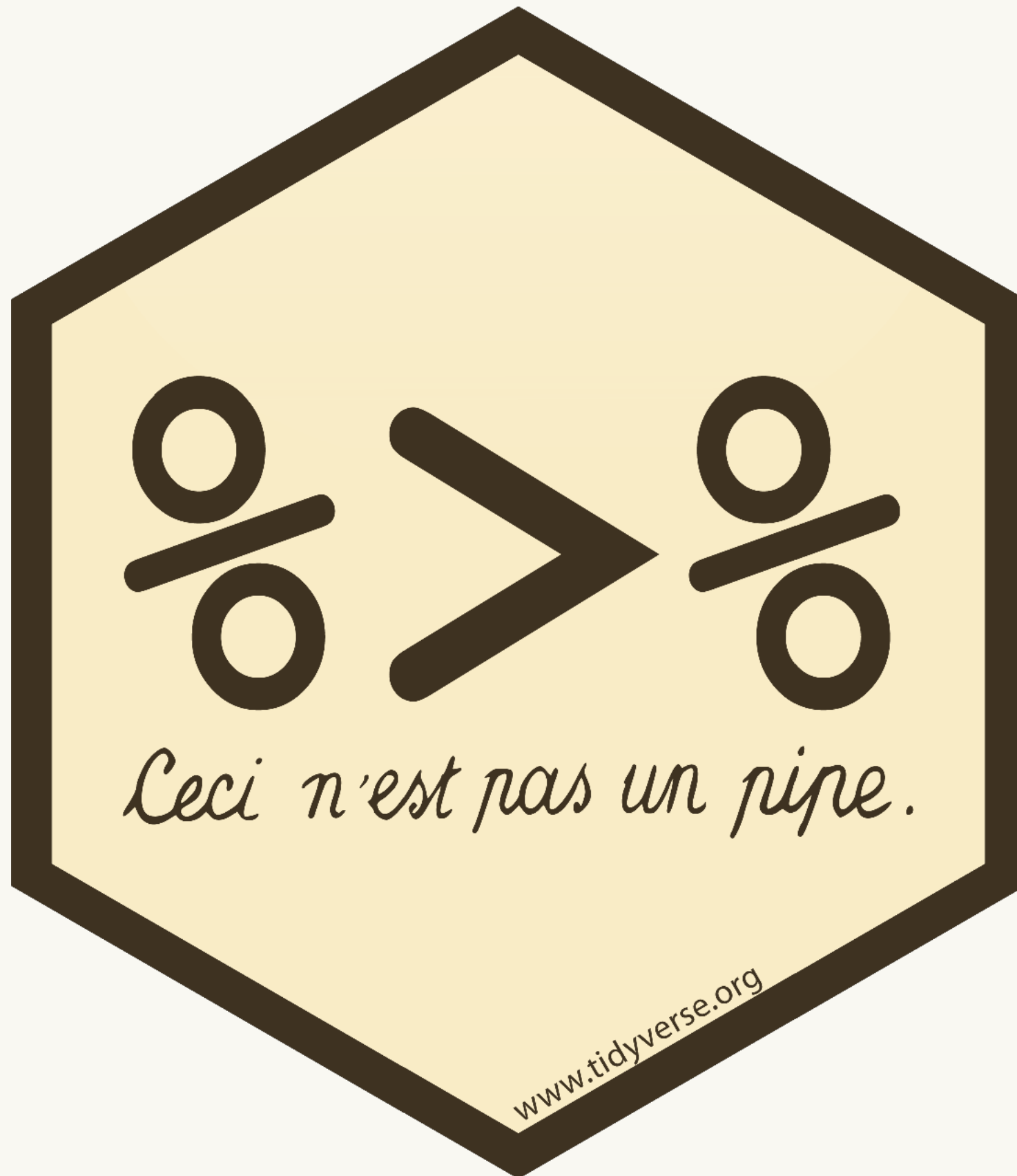
Pero escoger buenos nombres es una trabajo difícil

```
foo1 <- group_by(flights, dest)
foo2 <- summarise(foo1,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo3 <- filter(foo2, n > 100)
arrange(foo2, desc(delay))
```

Alternativamente, podrías usar funciones anidadas

```
arrange(  
  filter(  
    summarise(  
      group_by(flights, dest),  
      delay = mean(dep_delay, na.rm = TRUE),  
      n = n()  
    ),  
    n > 100  
  ),  
  desc(delay)  
)
```


magrittr provee una tercera opción



Sin intermediarios; leer de izquierda a derecha

```
flights %>%  
  group_by(dest) %>%  
  summarise(  
    delay = mean(dep_delay, na.rm = TRUE),  
    n = n()  
  ) %>%  
  filter(n > 100) %>%  
  arrange(desc(delay))
```

| | Lectura izquierda a derecha | Puede omitir nombres intermediarios | No-lineal |
|-----------------------------------|-----------------------------------|---|-----------|
| $y \leftarrow f(x)$ $g(y)$ | ✓ | | ✓ |
| $g(f(x))$ | | ✓ | ✓ |
| $x \%>\%$ $f() \%>\%$ $g()$ | ✓ | ✓ | |

Principio:

Argumentos de datos (data)
deben venir al principio

La mayoría de los argumentos caen en dos clases

| Datos | Detalles |
|----------------------------------|-------------------------|
| Requeridos | Opcionales |
| Datos clave | Opciones adicionales |
| Frecuentemente vectorizados | Un número |
| Frecuentemente llamados x o data | Nombres son importantes |

Ellos afectan como utilizas a las funciones

Omite nombres de los argumentos de datos

```
ggplot(mtcars, aes(x = disp, y = cyl))
```

No

```
ggplot(data = mtcars, mapping = aes(...))
```

Provee nombres de los argumentos de detalles

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

No

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

Nunca uses nombres parciales

(Leo: acuérdense de que modificamos nuestro ~/.Rprofile)

Tu turno

¿Cuáles son los argumentos de datos en grepl()?

¿Cuáles son los de detalles?

¿Cuáles son los argumentos de datos en strsplit()?

¿Cuáles son los de detalles?

¿Cuáles son los argumentos de datos en substr()?

¿Cuáles son los de detalles?

¿Cuáles son los argumentos de datos en merge()?

¿Cuáles son los de detalles?

Con los pipe, usa . para cambiar la posición

```
x %>%
```

```
  str_replace("a", "A") %>%
```

```
  str_replace("b", "B")
```

```
x %>%
```

```
  gsub("a", "A", .) %>%
```

```
  gsub("b", "B", .)
```


Principio:

Haz que los resultados y los
valores de entrada
coincidan

Tu turno

```
x <- c("bbaab", "bbb", "bbaaba")  
loc <- regexpr("a+", x)
```

```
# ¿Qué regresa regexpr()?
```

```
# ¿Qué estructura de datos utiliza?
```

```
# ¿Cómo usas substr() con el resultado de
```

```
# regexpr() para extraer la parte que identificó?
```

El resultado de `regexp()` no es compatible con `substr()`

```
x <- c("bbaab", "bbb", "bbaaba")
```

```
regexpr("a+", x)
```

```
loc <- regexpr("a", x)
```

```
substr(x, loc, loc + attr(loc, "match.length") - 1)
```

```
# Solo funciona porque esto regresa ""
```

```
substr(x, -1, -3)
```

```
# regmatches() tiene otro problema
```

```
regmatches(x, loc)
```

El código equivalente en stringr es mucho más sencillo

```
library("stringr")  
x <- c("bbaab", "bbb", "bbaaba")  
str_sub(x, str_locate(x, "a+"))
```

Todas las partes que contienen nuestra expresión regular

```
loc <- str_locate_all(x, "a+")  
library("purrr")  
map2(x, loc, str_sub)
```

Principio: Estabilidad de tipos



¿Por qué no usa `str_replace()` un argumento?

En vez de sufijos

`str_replace()`

`str_replace_all()`

podría usar un argumento

`str_replace(n = 1)`

`str_replace(n = Inf)`

que generalize mejor

`str_replace(n = 2)`

`str_replace(n = -1)`

pero queremos que `str_replace()` sea

consistente con `str_extract()`

str_extract() vs. str_extract_all()?

```
strings <- c("x1 y", "x2 y x3", "z")
```

```
str_extract(strings, "x.")
```

```
#> [1] "x1" "x2" NA
```

```
str_extract_all(strings, "x.")
```

```
#> [[1]]
```

```
#> [1] "x1"
```

```
#>
```

```
#> [[2]]
```

```
#> [1] "x2" "x3"
```

```
#>
```

```
#> [[3]]
```

```
#> character(0)
```

Y queremos que las funciones tengan resultados consistentes

```
str_extract(x, pattern, n = 1)
```

```
# ¿extraer un character?
```

```
str_extract(x, pattern, n = 2)
```

```
# ¿una character matrix?
```

```
str_extract(x, pattern, n = Inf)
```

```
# ¿una list?
```

```
x <- str_extract(x, pattern, n = n)
```

```
# ¿qué es x?
```


This work is licensed under the
Creative Commons Attribution-Noncommercial 3.0
United States License.

To view a copy of this license, visit
<http://creativecommons.org/licenses/by-nc/3.0/us/>