

Programación funcional

Julio 2019

Traducido por Leonardo Collado-Torres

@fellgernon

lcolladotor@gmail.com

lcolladotor.github.io

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

@cvwickham

cwickham@gmail.com

cwick.co.nz

Adapted from *Tidy Tools* by Hadley Wickham



Motivación

Copiar y pegar es una fuente rica de errores

```
# Arregla valores faltantes (Leo: Stata usa -99 en vez de NA)
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -99] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
df$k[df$k == -99] <- NA
```

Como recordatorio, **puedes obtener los materiales con:**
`usethis::use_course("ComunidadBioinfo/cdsb2019")`

Copiar y pegar es una fuente rica de errores

```
# Arregla valores faltantes
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -99] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
df$k[df$k == -99] <- NA
```

Como recordatorio, **puedes obtener los materiales con:**

```
usethis::use_course("ComunidadBioinfo/cdsb2019")
```

Funciones pueden eliminar algunas fuentes de duplicación

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}  
df$a <- fix_missing(df$a)  
df$b <- fix_missing(df$b)  
df$c <- fix_missing(df$c)  
df$d <- fix_missing(df$d)  
df$e <- fix_missing(df$e)  
df$f <- fix_missing(df$f)  
df$g <- fix_missing(df$g)  
df$h <- fix_missing(df$h)  
df$h <- fix_missing(df$i)
```

Como recordatorio, **puedes obtener los materiales con:**
`usethis::use_course("ComunidadBioinfo/cdsb2019")`

Funciones pueden eliminar algunas fuentes de duplicación

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}  
df$a <- fix_missing(df$a)  
df$b <- fix_missing(df$b)  
df$c <- fix_missing(df$c)  
df$d <- fix_missing(df$d)  
df$e <- fix_missing(df$e)  
df$f <- fix_missing(df$f)  
df$g <- fix_missing(df$g)  
df$h <- fix_missing(df$h)  
df$h <- fix_missing(df$i)
```

Como recordatorio, **puedes obtener los materiales con:**
`usethis::use_course("ComunidadBioinfo/cdsb2019")`

Ciclos de for pueden eliminar otras

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
for (i in seq_along(df)) {  
  df[[i]] <- fix_missing(df[[i]])  
}
```

Como recordatorio, **puedes obtener los materiales con:**

```
usethis::use_course("ComunidadBioinfo/cdsb2019")
```

Porque los ciclos
de for son malos

Un detour con panqués

Porque los ciclos
de for son malos

subóptimos

Un detour con panqués

Panqués de vainilla

El libro de cocina
de la panadería
hummingbird

1 cup flour

a scant $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until $\frac{2}{3}$ full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Panqués de chocolate

El libro de cocina
de la panadería
hummingbird

$\frac{3}{4}$ cup + 2T flour

2 $\frac{1}{2}$ T cocoa powder

a scant $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, cocoa, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until $\frac{2}{3}$ full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Panqués de chocolate

El libro de cocina
de la panadería
hummingbird

$\frac{3}{4}$ cup + 2T flour

2 $\frac{1}{2}$ T cocoa powder

a scant $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, **cocoa**, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until $\frac{2}{3}$ full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Panqués de vainilla

El libro de cocina
de la panadería
hummingbird

1 cup flour

a scant $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until $\frac{2}{3}$ full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Panqués de vainilla

El libro de cocina
de la panadería
hummingbird

120g flour

140g sugar

1.5 t baking powder

40g unsalted butter

120ml milk

1 egg

0.25 t pure vanilla extract

Preheat oven to 170°C.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

1. Convierte las
unidades

Panqués de vainilla

El libro de cocina
de la panadería
hummingbird

120g flour

140g sugar

1.5 t baking powder

40g unsalted butter

120ml milk

1 egg

0.25 t pure vanilla extract

Beat flour, sugar, baking powder, salt, and butter until sandy.

Whisk milk, egg, and vanilla. Mix half into flour mixture until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

2. Apóyate en el conocimiento del área

Panqués de vainilla

El libro de cocina
de la panadería
hummingbird

120g flour

140g sugar

1.5 t baking powder

40g butter

120ml milk

1 egg

0.25 t vanilla

Beat **dry ingredients** + butter until sandy.

Whisk together **wet ingredients**. Mix half into dry until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

Panqués de chocolate

El libro de cocina
de la panadería
hummingbird

100g flour

20g cocoa

140g sugar

1.5 t baking powder

40g butter

120ml milk

1 egg

0.25 t vanilla

Beat **dry ingredients** + butter until sandy.

Whisk together **wet ingredients**. Mix half into dry until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

Panqués

Beat **dry ingredients** + butter until sandy.

Whisk together **wet ingredients**. Mix half into dry until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

Vanilla

120g flour

140g sugar

1.5t baking powder

40g butter

120ml milk

1 egg

0.25 t vanilla

Chocolate

100g flour

20g cocoa

140g sugar

1.5t baking powder

40g butter

120ml milk

1 egg

0.25 t vanilla

4. Extrae el código en común

¿Qué tienen estos ciclos de for en común?

```
out1 <- vector("double", ncol(mtcars))  
for(i in seq_along(mtcars)) {  
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)  
}
```

```
out2 <- vector("double", ncol(mtcars))  
for(i in seq_along(mtcars)) {  
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)  
}
```

Los ciclos de for enfatizan a los objetos

```
out1 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}
```

```
out2 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

Y no a las acciones

```
out1 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}
```

```
out2 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

La programación funcional enfatiza a las acciones

```
library(purrr)
```

```
means <- map_dbl(mtcars, mean)
```

```
medians <- map_dbl(mtcars, median)
```

Y de regreso...

Ciclos de for pueden eliminar a otros

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
for (i in seq_along(df)) {  
  df[[i]] <- fix_missing(df[[i]])  
}
```


La PF te permite enfocar en lo que sucede

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
df <- modify(df, fix_missing)
```

Y provee herramientas útiles para la **generalización**

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
df <- modify_if(df, is.numeric, fix_missing)
```

Principio:

Resuelve un solo problema

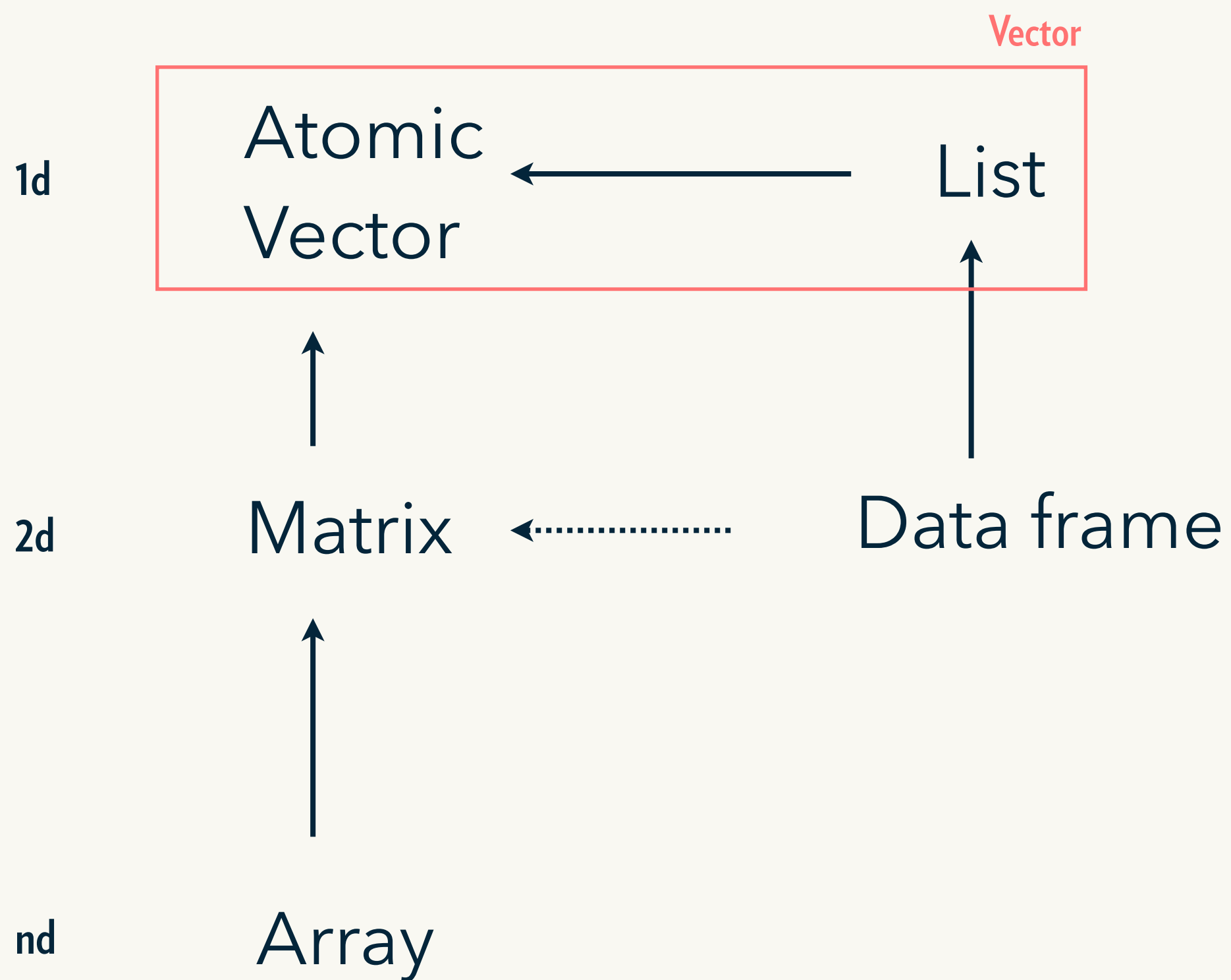
Principio:

Amplia con map & amigos

Calentamientos

Tu turno

- ¿En qué es diferente una list de un atomic vector?
- ¿En qué es diferente un data frame de una list?
- ¿Cómo examinarías la estructura de un objeto?



Mismos tipos

Diferentes tipos

str()

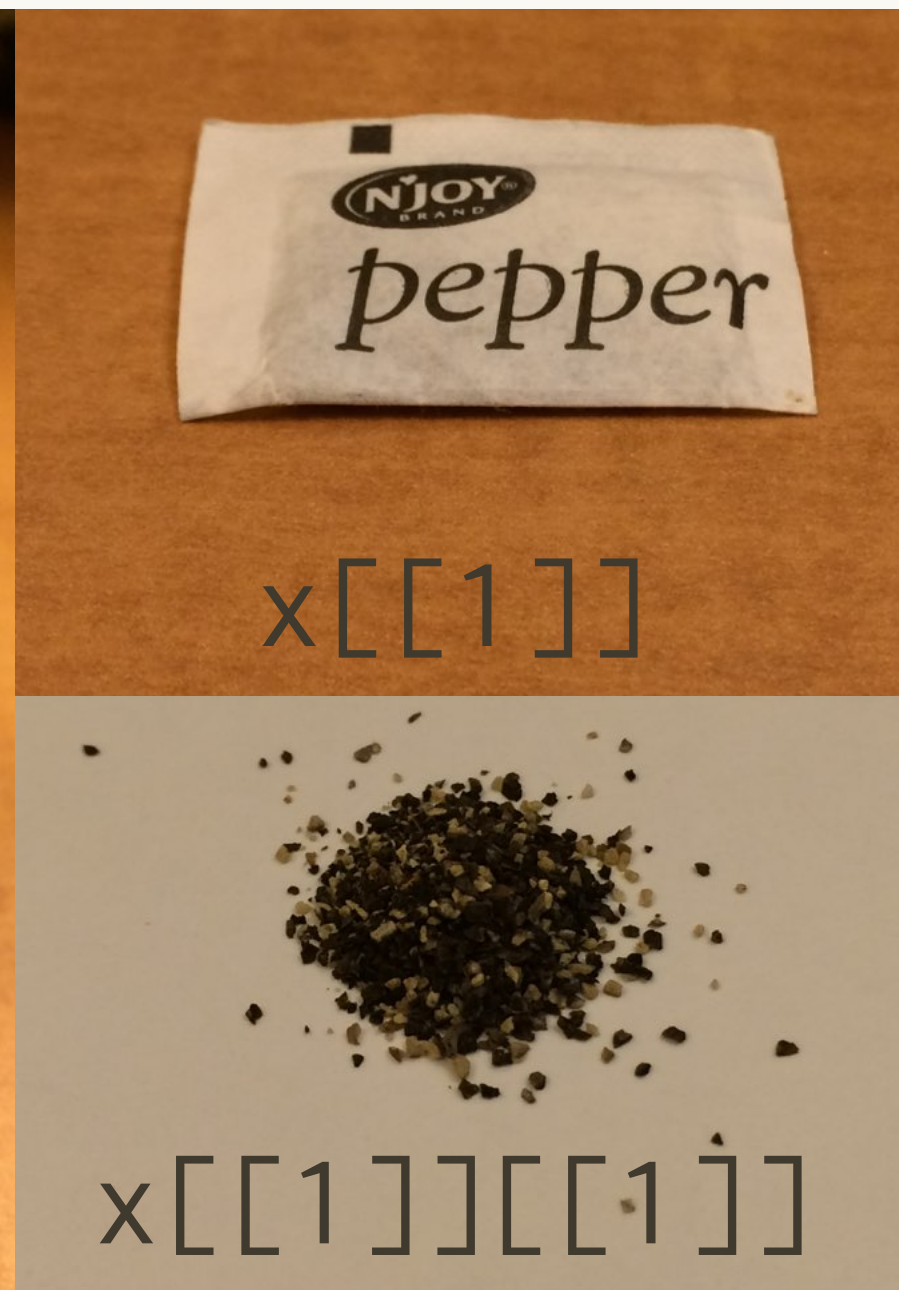
view()

(Si tienes RStudio \geq 1.1)

Tu turno

¿Cuál es la diferencia entre [y [[?

	Sencillo	Múltiple
Vectors	<code>x[[1]]</code>	<code>x[1:4]</code>
Lists	<code>x[[1]]</code> <code>x\$name</code>	<code>x[1]</code>

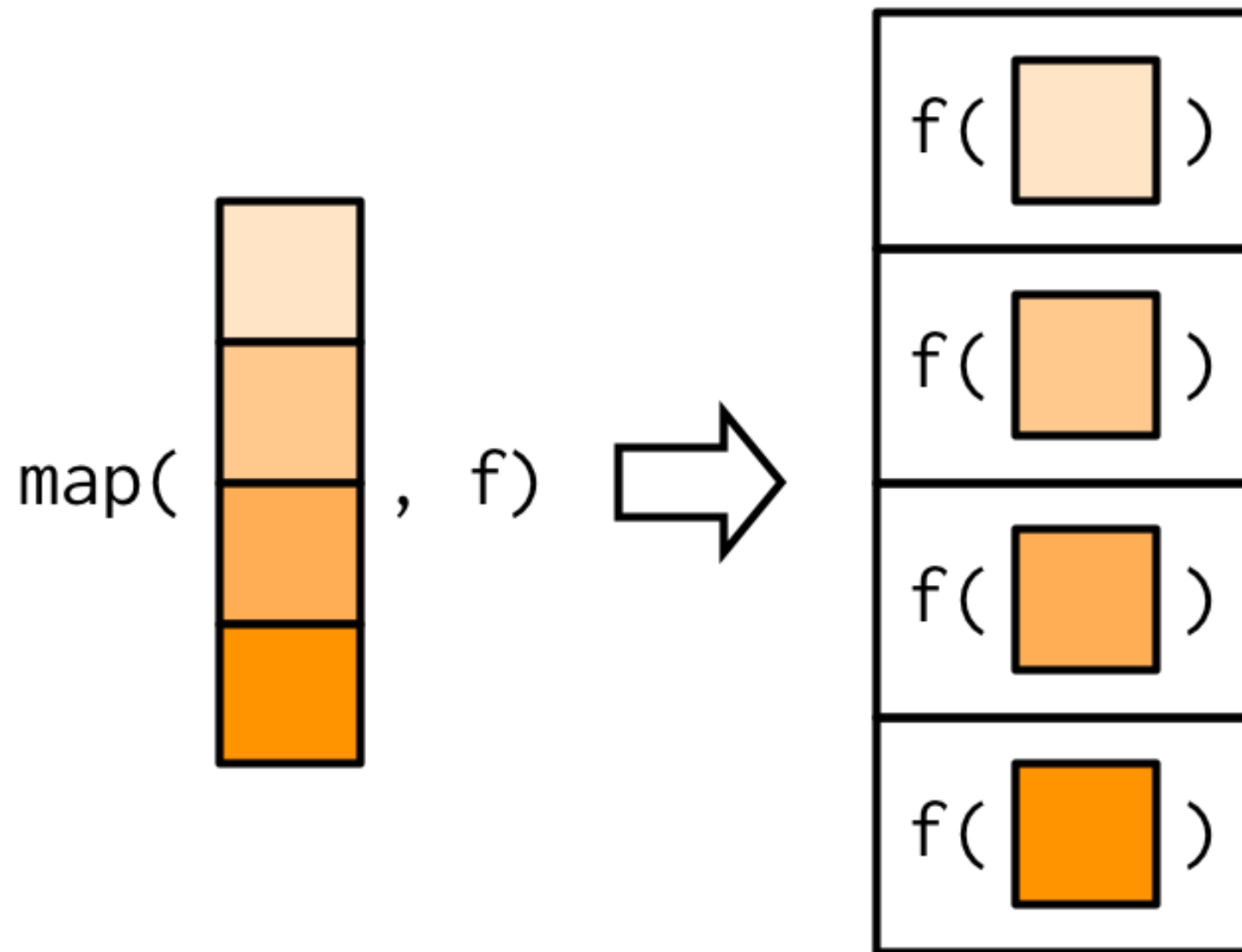


¿Qué hace este código?

```
trans <- list(  
  disp = function(x) x * 0.0163871,  
  am = function(x) {  
    factor(x, labels = c("auto", "manual"))  
  }  
)  
for(var in names(trans)) {  
  mtcars[[var]] <- trans[[var]](mtcars[[var]])  
}
```

Familia map

map(): para cada elemento, aplica f



Estrategia map

Para una tarea iterativa:

1. Resuélvela para un solo `.x`
2. Generaliza la solución con la función `map()` apropiada
3. Simplifica (de ser posible)

Encuentra el primer elemento del texto compuesto

```
strings <- c("a|b", "a|b|c", "d|e", "b|c|d")
```

```
# Queremos:
```

```
# "a" "a" "d" "b"
```

```
# Un objeto intermedio útil
```

```
strings_split <- strsplit(strings, "|", fixed = TRUE)
```

```
# Para cada elemento de strings_split
```

```
# extrae el primer elemento
```

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

```
# [1] "a" "b"
```

```
#
```

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

```
# [1] "a" "b" "c"
```

```
#
```

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

```
# [1] "d" "e"
```

```
#
```

```
# [[4]]
```

```
# [1] "b" "c" "d"
```


1. Resuelve para un solo .x

```
# Extrae un elemento  
.x <- strings_split[[1]]
```

Pronombre especial que map entiende

```
.x  
# [1] "a" "b"  
  
# Obtén el primer elemento  
.x[[1]]  
# ¡Lo resolvimos!
```

2. Generaliza la solución con map()

```
# Solución para un elemento  
    .x[[1]]
```

```
# Conviértela en una receta con ~ y pásala a  
map
```

```
map(strings_split, ~ .x[[1]])
```

Para cada elemento de
strings_split,

tómalo, y extrae el primer
elemento

Estrategia map

Para una tarea iterativa:

1. Resuélvela para un solo `.x`
2. Generaliza la solución con la función `map()` **apropiada**
3. Simplifica (de ser posible)

Cada variante siempre resulta en el mismo tipo de objeto

Función	Valor de salida
map_lgl()	Logical vector
map_int()	Integer vector
map_dbl()	Double vector
map_chr()	Character vector
map()	List
map_dfc()	Data frame (by col)
map_dfr()	Data frame (by row)

Tipo de objeto garantizado, o un error

```
map(strings_split, ~ .x[[1]]) %>% str()
```

```
# List of 4
```

```
# $ : chr "a"
```

```
# $ : chr "a"
```

```
# $ : chr "d"
```

```
# $ : chr "b"
```

```
map_chr(strings_split, ~ .x[[1]])
```

```
# [1] "a" "a" "d" "b"
```

```
map_dbl(strings_split, ~ .x[[1]])
```

```
# Error: Can't coerce element 1 from a  
character to a double
```

Estrategia map

Para una tarea iterativa:

1. Resuélvela para un solo `.x`
2. Generaliza la solución con la función `map()` apropiada
3. **Simplifica** (de ser posible)

Simplifica la extracción de valores

```
map(z, ~ .x[[1]])
```

```
map(z, 1)
```

```
map(z, ~ .x[["string"]])
```

```
map(z, "string")
```

```
map(z, ~ .x[["string"]][[1]] %||% NA)
```

```
map(z, list("string", 1), .default = NA))
```

Simplifica el llamado de funciones

`map(z, ~ f(.x))`

`map(z, f)`

`map(z, ~ f(.x, a = 1, b = 2))`

`map(z, f, a = 1, b = 2)`

`map(z, ~ f(1, .x))`

`map(z, f, first_arg = 1)`

Tu turno

Calcula la media de cada columna en mtcars

Genera 10 valores de la distribución normal al azar usando las siguientes medias: -10, 0, 10, 100

Calcula el número de valores únicos en cada columna de iris

Calcula la media de cada columna en mtcars

```
# Resuelve para un caso
```

```
.x <- mtcars[[1]]
```

```
mean(.x)
```

```
# Generaliza
```

```
map_dbl(mtcars, ~ mean(.x))
```

```
# Simplifica (opcional)
```

```
map_dbl(mtcars, mean)
```

Genera 10 valores de la distribución normales al azar

```
mu <- c(-10, 0, 10, 100)
```

```
# Resuelve para un caso
```

```
.x <- mu[[1]]
```

```
rnorm(10, mean = .x)
```

```
# Generaliza
```

```
map(mu, ~ rnorm(10, mean = .x))
```

```
# Simplifica (opcional)
```

```
map(mu, rnorm, n = 10)
```

Calcula el número de valores únicos en cada columna

```
# Resuelve para un caso
```

```
.x <- iris[[1]]  
length(unique(.x))
```

```
# Generaliza
```

```
map_int(iris, ~ length(unique(.x)))
```

```
# ¿Simplifica?
```

```
nunique <- function(x) length(unique(x))  
map_int(iris, ~ nunique(.x))  
map_int(iris, nunique)
```

¿Por qué no R
base?

Comparado con purrr, funciones de R base:

Tienen nombres inconsistentes (`lapply()` vs. `Map()`)

Tienen un orden inconsistente de los argumentos (`lapply()` vs. `mapply()`)

Requiren funciones (no existe `~`, o ayudantes para extraer)

Son tipo-inestable (`sapply()`) o verbosas (`vapply()`)

No tienen un modo con efectos secundarios (no `walk()`)

No hay mapas pareados (no `map2()`)

No regresan data frames (no `_dfc()`, `_dfr()`)

R base solo provee un conjunto parcial de funciones

Número de valores de entrada

	Valor de salida es escalar	Valor de salida es cualquier cosa	Valor de salida es nada
1	sapply() / vapply()	lapply()	
2			
n	mapply()	Map()	

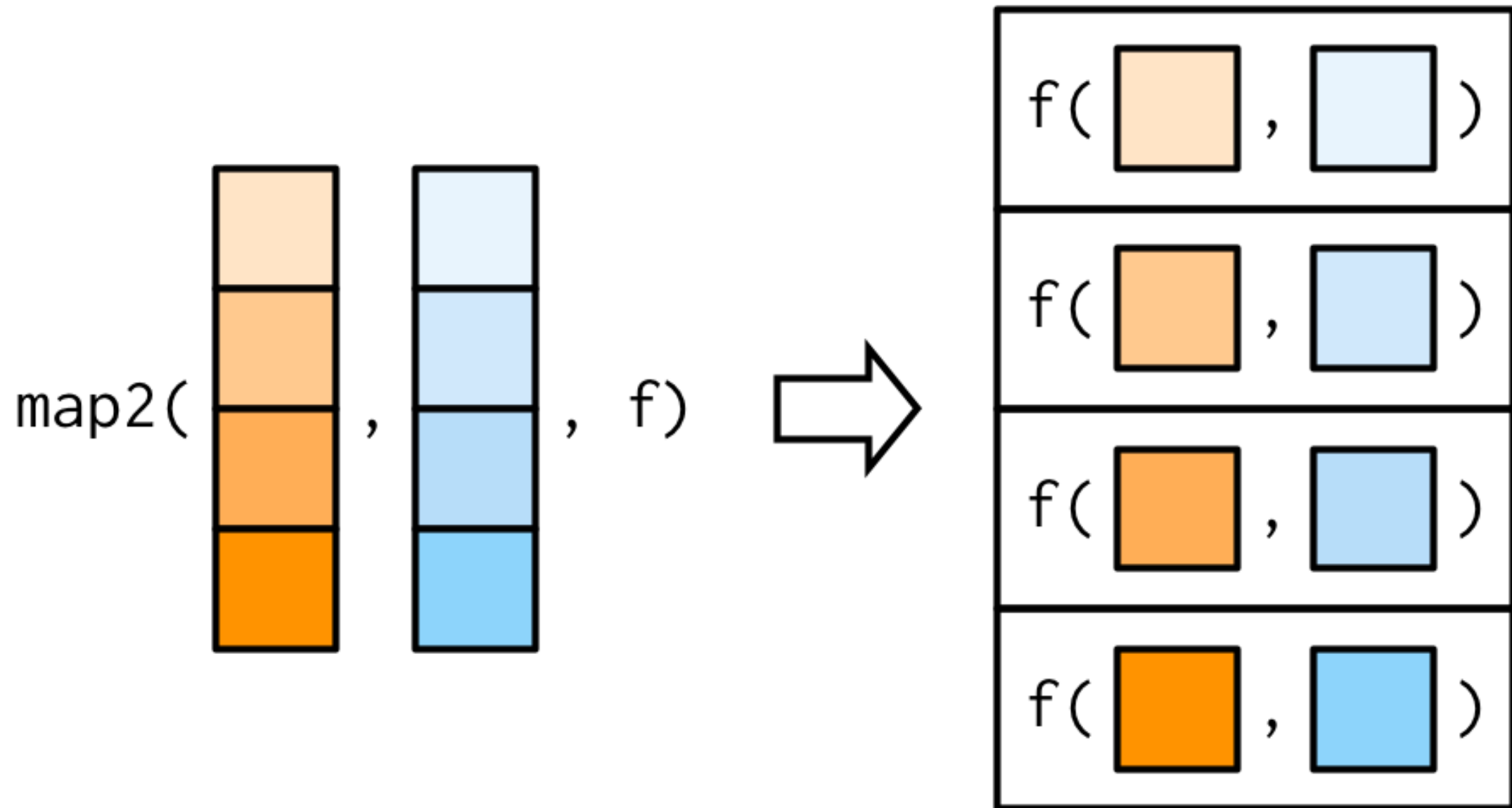
purrr provee un conjunto completo de funciones

Número de valores de entrada

	Valor de salida es escalar	Valor de salida es cualquier cosa	Valor de salida es nada
1	map_lgl(), map_int(), map_dbl(), map_chr()	map()	walk()
2	map2_lgl(), map2_int(), map2_dbl(), map2_chr()	map2()	walk2()
n	pmap_lgl(), pmap_int(), pmap_dbl(), pmap_chr()	pmap()	pwalk()

Mapa pareado

`map2()`: para cada par de elementos, aplica f



Cuando necesitas iterar sobre dos objetos: `map2()`

Para una tarea iterativa:

1. Resuélvela para un solo `.x` & `.y`
2. Generaliza la solución con la función `map2()` apropiada
3. **Simplifica** (de ser posible)

Objetivo: guardar datos en diferentes archivos

```
library(ggplot2)
```

```
# una lista de data frames
```

```
by_color <- split(diamonds, diamonds$color)
```

```
# un vector de archivos
```

```
paths <- paste0(names(by_color), ".csv")
```

1. Resolver para un `.x` & `.y`

```
# Resolver para un caso
```

```
.x <- by_color[[1]]
```

```
.y <- paths[[1]]
```

```
write.csv(.x, .y)
```

2. Generaliza la solución con map2()

```
# write.csv(.x, .y)
map2(by_color, paths, ~ write.csv(.x, .y))
```

```
# Usa una función más apropiada
walk2(by_color, paths, ~ write.csv(.x, .y))
```

```
# Simplifica (opcional)
walk2(by_color, paths, write.csv)
```

```
# Para limpiar
file.remove(paths)
```

Principio:

Realiza funciones de valores
con `map()`; realiza funciones
con efectos usando `walk()`

Cambia al proyecto:

[colsum]

Este paquete automáticamente carga purrr

```
devtools::load_all(".")
```

```
Loading colsum
```

```
Loading required package: purrr
```

```
Attaching package: 'purrr'
```

```
# Porque antes corrí
```

```
usethis::use_package("purrr", "depends")
```

A favor

En contra

Fácilmente
utiliza funciones
de purrr

Afecta el camino
de búsqueda
global

No es aceptable
en CRAN

Tu turno

Escribe la función `col_write(df, path)` que escribe cada columna de `df` en un archivo llamado *nombre_columna.txt*, con un valor en cada línea (es decir, usa `writeLines()`).

Este paquete incluye una unidad de prueba que te permite checar tu trabajo.

Desde `R/col_write.R` puedes correr `devtools::test_file()`, para solo correr las pruebas relevantes al archivo.

Una posible solución

```
col_write <- function(df, path = tempdir()) {  
  filenames <- paste0(path, "/", names(df), ".txt")  
  
  walk2(df, filenames,  
    ~ writeLines(as.character(.x), .y))  
}
```

Otros tipos de iteración

Valores de entrada

1

`map()`

2

`map2()`

1 + index

`imap()`

3+

`pmap()`

functions

`invoke_map()`

Estabilidad de tipos de objetos

¿Por qué sapply es difícil de utilizar al programar?

```
df <- data.frame(  
  a = 1L,  
  b = 1.5,  
  y = Sys.time(),  
  z = ordered(1)  
)
```

Adivina el tipo del objeto del resultado

```
df[1:4] %>% sapply(class) %>% str()  
df[1:2] %>% sapply(class) %>% str()  
df[3:4] %>% sapply(class) %>% str()
```

Principio:

Minimiza el contexto necesario para predecir el tipo de objeto del resultado

El extremo es una función tipo-estable que siempre regresa el mismo tipo de objeto sin importar el valor de entrada.

`map()`

`sapply()`

`data.frame()`



Regresa una list,
o muere en el
intento

Tipo de valor de
salida depend del
tipo de valor de
entrada, longitud y
función

Factor vs character
depende de tu
configuración
global

La alternativa purrr

```
df <- data.frame(  
  a = 1L,  
  b = 1.5,  
  y = Sys.time(),  
  z = ordered(1)  
)
```

Adivina el tipo del objeto del resultado

```
df[1:4] %>% map_chr(class) %>% str()  
df[1:2] %>% map_chr(class) %>% str()  
df[3:4] %>% map_chr(class) %>% str()
```

Un ejemplo más realista

```
# En R/col_means.R
```

```
col_means <- function(df) {  
  numeric <- sapply(df, is.numeric)  
  numeric_cols <- df[, numeric]  
  
  as.data.frame(lapply(numeric_cols, mean))  
}
```

¿Cuál es el problema con col_means?

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
```

```
df <- data.frame(
  x = 1:26,
  y = letters
)
col_means(df)
```

Principio:

Piensa en los invariantes

¿Qué debería siempre ser cierto?

¿Cuáles son los invariantes?

```
# ¿Qué es lo que siempre debería ser cierto del resultado?
```

```
# * debería ser un data frame
```

```
expect_s3_class(out, "data.frame")
```

```
# * un renglón
```

```
expect_equal(nrow(out), 1)
```

```
# * una columna para cada valor numérico del objeto inicial
```

```
expect_equal(ncol(out), sum(map_lgl(in, is.numeric)))
```

sapply y [no son tipo-estables

```
col_means <- function(df) {  
  numeric <- sapply(df, is.logical)  
  numeric_cols <- df[, numeric]  
  
  as.data.frame(lapply(numeric_cols, mean))  
}
```

list o logical vector

vector o data
frame

Una posible solución

```
col_means <- function(df) {  
  numeric <- map_lgl(df, is.numeric)  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  as.data.frame(map(numeric_cols, mean))  
}
```


One possible solution

```
col_means <- function(df) {  
  numeric <- map_lgl(df, is.numeric)  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  as.data.frame(map(numeric_cols, mean))  
}
```

siempre un logical
vector

siempre un
data frame

Podemos simplificar aún más con funciones ayudantes

```
col_means <- function(df) {  
  numeric_cols <- keep(df, is.numeric)  
  map_dfc(numeric_cols, mean)  
}
```

¿Es keep() tipo-estable? Su valor de salida es del mismo tipo que el objeto de entrada

Tipo del objeto
de salida
depende del
de entrada
keep()

map()

sapply()

data.frame()

Regresa una list,
o muere en el
intento

Tipo de valor de
salida depend del
tipo de valor de
entrada, longitud y
función

Factor vs character
depende de tu
configuración
global

Es particularmente elegante con el pipe

```
col_means <- function(df) {  
  df %>%  
    keep(is.numeric) %>%  
    map_dfc(mean)  
}
```

Invariante fallido

```
col_means(data.frame())
```

```
#> data frame with 0 columns and 0 rows
```

```
# Debería ser
```

```
#> data frame with 0 columns and 1 rows
```

```
# ¿Arreglar esto es importante? 🙄
```

Para aprender más

R4DS: <https://r4ds.had.co.nz/iteration.html>

(en español) <https://es.r4ds.hadley.nz/index.html>

Advanced R: <https://adv-r.hadley.nz/functionals.html>

Apply functions with purrr : : CHEAT SHEET



Apply Functions

Map functions apply a function iteratively to each element of a list or vector.

`map(x, fun, ...)` → `fun(x)` → `map(x, fun, ...)` Apply a function to each element of a list or vector. `map(x, is.logical)`

`map2(x, y, fun, ...)` → `fun(x, y)` → `map2(x, y, fun, ...)` Apply a function to pairs of elements from two lists, vectors. `map2(x, y, sum)`

`pmap(list(x, y, z), fun, ...)` → `fun(x, y, z)` → `pmap(list(x, y, z), fun, ...)` Apply a function to groups of elements from list of lists, vectors. `pmap(list(x, y, z), sum, na.rm = TRUE)`

`invoke_map(x, fun, ...)` → `fun(x)` → `invoke_map(x, fun, ...)` Apply function to each list-element of a list or vector. Also `invoke`. `l <- list(var, sd); invoke_map(l, x = 1:9)`

`lmap(x, .f, ...)` Apply function to each list-element of a list or vector. `imap(x, .f, ...)` Apply .f to each element of a list or vector and its index.

OUTPUT

`map()`, `map2()`, `pmap()`, `imap` and `invoke_map` each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. `map2_chr`, `pmap_lgl`, etc.

Use `walk`, `walk2`, and `pwalk` to trigger side effects. Each return its input invisibly.

function	returns
<code>map</code>	list
<code>map_chr</code>	character vector
<code>map_dbl</code>	double (numeric) vector
<code>map_dfc</code>	data frame (column bind)
<code>map_dfr</code>	data frame (row bind)
<code>map_int</code>	integer vector
<code>map_lgl</code>	logical vector
<code>walk</code>	triggers side effects, returns the input invisibly

SHORTCUTS - within a purrr function:

"name" becomes `function(x) x[["name"]]`, e.g. `map(l, "a")` extracts `a` from each element of `l`

`~.x.y` becomes `function(x, y) .x.y`, e.g. `map2(l, p, ~.x+y)` becomes `map2(l, p, function(l, p) l + p)`

`~.x` becomes `function(x) x`, e.g. `map(l, ~.x)` becomes `map(l, function(x) x)`

`~.1..2` etc becomes `function(..1, ..2, etc) ..1..2` etc, e.g. `pmap(list(a, b, c), ~.3 + ..1 - ..2)` becomes `pmap(list(a, b, c), function(a, b, c) c + a - b)`

Work with Lists

FILTER LISTS

`pluck(x, ..., .default=NULL)` Select an element by name or index, `pluck(x, "b")`, or its attribute with `attr_getter`. `pluck(x, "b", attr_getter("n"))`

`keep(x, .p, ...)` Select elements that pass a logical test. `keep(x, is.na)`

`discard(x, .p, ...)` Select elements that do not pass a logical test. `discard(x, is.na)`

`compact(x, .p = identity)` Drop empty elements. `compact(x)`

`head_while(x, .p, ...)` Return head elements until one does not pass. Also `tail_while`. `head_while(x, is.character)`

RESHAPE LISTS

`flatten(x)` Remove a level of indexes from a list. Also `flatten_chr`, `flatten_dbl`, `flatten_dfc`, `flatten_dfr`, `flatten_int`, `flatten_lgl`, `flatten(x)`

`transpose(.l, .names = NULL)` Transposes the index order in a multi-level list. `transpose(x)`

SUMMARISE LISTS

`every(x, .p, ...)` Do all elements pass a test? `every(x, is.character)`

`some(x, .p, ...)` Do some elements pass a test? `some(x, is.character)`

`has_element(x, .y)` Does a list contain an element? `has_element(x, "foo")`

`detect(x, .f, ..., .right=FALSE, .p)` Find first element to pass. `detect(x, is.character)`

`detect_index(x, .f, ..., .right=FALSE, .p)` Find index of first element to pass. `detect_index(x, is.character)`

`vec_depth(x)` Return depth (number of levels of indexes). `vec_depth(x)`

JOIN (TO) LISTS

`append(x, values, after = length(x))` Add to end of list. `append(x, list(d = 1))`

`prepend(x, values, before = 1)` Add to start of list. `prepend(x, list(d = 1))`

`splice(...)` Combine objects into a list, storing S3 objects as sub-lists. `splice(x, y, "foo")`

TRANSFORM LISTS

`modify(x, .f, ...)` Apply function to each element. Also `map`, `map_chr`, `map_dbl`, `map_dfc`, `map_dfr`, `map_int`, `map_lgl`. `modify(x, ~.+2)`

`modify_at(x, .at, .f, ...)` Apply function to elements by name or index. Also `map_at`. `modify_at(x, "b", ~.+2)`

`modify_if(x, .p, .f, ...)` Apply function to elements that pass a test. Also `map_if`. `modify_if(x, is.numeric, ~.+2)`

`modify_depth(x, .depth, .f, ...)` Apply function to each element at a given level of a list. `modify_depth(x, 1, ~.+2)`

WORK WITH LISTS

`array_tree(array, margin = NULL)` Turn array into list. Also `array_branch`. `array_tree(x, margin = 3)`

`cross2(x, y, .filter = NULL)` All combinations of `x` and `y`. Also `cross`, `cross3`, `cross_dfc`. `cross2(1:3, 4:6)`

`set_names(x, nm = x)` Set the names of a vector/list directly or with a function. `set_names(x, c("p", "q", "r"))` `set_names(x, tolower)`

Reduce Lists

`reduce(x, .f, ..., .init)` Apply function recursively to each element of a list or vector. Also `reduce_right`, `reduce2`, `reduce2_right`. `reduce(x, sum)`

`accumulate(x, .f, ..., .init)` Reduce, but also return intermediate results. Also `accumulate_right`. `accumulate(x, sum)`

Modify function behavior

`compose()` Compose multiple functions.

`lift()` Change the type of input a function takes. Also `lift_dbl`, `lift_dv`, `lift_ld`, `lift_lv`, `lift_vd`, `lift_vl`.

`rerun()` Rerun expression n times.

`negate()` Negate a predicate function (a pipe friendly !)

`partial()` Create a version of a function that has some args preset to values.

`safely()` Modify func to return list of results and errors.

`quietly()` Modify function to return list of results, output, messages, warnings.

`possibly()` Modify function to return default value whenever an error occurs (instead of error).



<https://github.com/rstudio/cheatsheets/raw/master/purrr.pdf>

Adapted from *Tidy Tools* by Hadley Wickham

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/](https://creativecommons.org/licenses/by-sa/4.0/)