Tutorial "Cómo usar la familia de funciones map de manera efectiva"

Tabla de contenidos

1 Manual básico de listas, vectores y marcos de datos

Dominar `purrr` y la familia de funciones `map` puede ser un desafío sin una comprensión sólida de listas, vectores y marcos de datos. Las funciones `map` en su mayoría toman estas estructuras de datos como entrada. Si no está familiarizado con ellos, será difícil seguirlos. Hablaré mucho sobre ellos en este tutorial, por lo que es una buena idea repasar estas estructuras de datos.

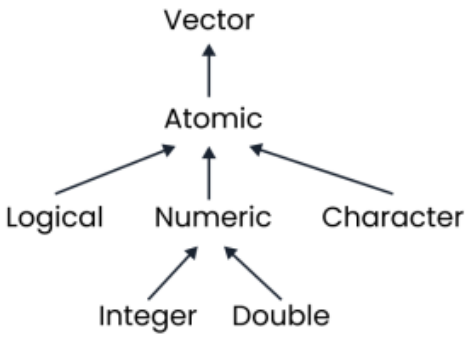
1.1 Una descripción general de las estructuras de datos en `R`

`R` tiene cinco estructuras de datos básicas: vectores atómicos, matrices, matrices, listas y marcos de datos. Son diferentes entre sí de dos maneras: si contienen el mismo tipo de datos o no, y si son unidimensionales o si son bidimensionales. No profundizaré demasiado en matrices y arreglos en este tutorial, ya que no son tan importantes para purrr como las otras estructuras de datos.

1.1.1 Vectores atómicos

Los vectores atómicos son solo un tipo de vector, siendo las listas el otro. Tanto los vectores atómicos como las listas tienen una cosa en común: son unidimensionales. Piénselo así: si comparamos las dimensiones con las direcciones, un elemento en una estructura de datos unidimensional solo puede moverse hacia adelante y hacia atrás (o hacia arriba y hacia abajo). Pero un elemento en una estructura de datos bidimensional puede moverse en cualquier dirección.

Tenemos cuatro tipos de vectores atómicos: lógicos (TRUE o FALSE), caracteres (texto), enteros (números enteros como 1) y dobles (números con decimales como 1.31). Los números enteros y los dobles también se denominan vectores atómicos numéricos.




```
c(TRUE, FALSE) # lógico
c("a", "bunch", "of strings") # carácter
c(4, 5, 9) # entero
c(5.1, 8.2) # doble
```

Se vuelve interesante cuando intentamos crear vectores no homogéneos (o heterogéneos). Por defecto, R intentará convertir todos los elementos del vector al mismo tipo de datos. Aquí hay un ejemplo donde mezclamos un doble y un carácter:

```
c(2.1, "8") %>% typeof()
```

```
[1] "character"
```

Se observa que la salida es un vector de caracteres. En `R` o terminología de programación, diríamos que el doble fue forzado (coerción) a un caracter.

La coerción ocurre cuando combinamos elementos vectoriales de diferentes tipos. En el siguiente ejemplo, hemos combinado un lógico, un doble y un carácter:

```r
c(TRUE, 8.1, "house")
```

```
[1] "TRUE"  "8.1"   "house"
```

Nuevamente, obtenemos un vector de caracteres. Lo que vemos aquí son reglas de coerción. Estas reglas determinan en qué orden se deben coaccionar los tipos de datos. Las reglas de coerción en `R` son las siguientes: logical < integer < numeric < complex < character < list

Al combinar varios tipos de datos en un vector, el tipo de datos a la izquierda de las reglas de coerción se convertirá al tipo de datos a la derecha. Por ejemplo, en los ejemplos anteriores se coaccionaba el doble y el lógico a un carácter. Por eso, cuando combinamos un lógico y un doble, el lógico se convertirá en un doble:

```r
c(TRUE, 5.1, FALSE) %>% typeof()
```

```
[1] "double"
```

Como puede ver en el ejemplo anterior, cuando se coacciona a un doble, TRUE se representa como 1 y FALSE como 0.

1.1.2 Listas

Lo que diferencia a las listas de los vectores atómicos son dos cosas. Primero, las listas son heterogéneas, lo que significa que sus elementos pueden ser de diferentes tipos de datos. En segundo lugar, las listas son **recursivas**, lo que significa que los elementos de la lista pueden contener otros objetos. Aquí hay un ejemplo de una lista que muestra ambas propiedades:

```r
my_list <- list(
    a = 3,
    b = "Some text",
    c = list(
        x = c(4, 5, 6)),
    c(TRUE, FALSE)
    ) %>%
    print()
```

```
$a
[1] 3
```

```
$b
[1] "Some text"

$c
$c$x
[1] 4 5 6


[[4]]
[1]  TRUE FALSE
```

En el ejemplo anterior, la lista contiene tres elementos (a, b y c). Cada elemento es de un tipo de datos diferente: a es un vector numérico, b es un vector de caracteres y c es una lista. Este es un ejemplo de una estructura de datos heterogénea. La recursividad también se demuestra en este ejemplo. El tercer elemento c contiene una lista, que a su vez contiene otros elementos. Esto no es posible con vectores atómicos.

Para leer mejor una estructura de datos profunda como esta, podemos usar la función `str()` que nos brinda información sobre los tipos de datos de los elementos:

```r
str(my_list)
```

```
List of 4
 $ a: num 3
 $ b: chr "Some text"
 $ c:List of 1
  ..$ x: num [1:3] 4 5 6
 $  : logi [1:2] TRUE FALSE
```

Dirija su atención a los elementos de la lista anterior. El elemento etiquetado como c es una lista. Los elementos a y x (dentro de c) contienen vectores numéricos y el elemento b contiene un vector de caracteres. También puede notar que el último elemento de la lista, el vector lógico, no tiene nombre. Esto es válido en listas.

Lo anterior nos lleva al siguiente tema sobre listas: los subconjuntos. Subconjunto se refiere al acto de recuperar un elemento de una lista. Pueden existir cinco formas de los elementos de una lista:

```r
my_list[1]
my_list[[1]]
my_list$a
my_list[["a"]]
```

```
my_list["a"]
```

Aquí hay una regla para recordar. Siempre que subjunte un elemento de lista usando un corchete ([), el resultado será una lista. Por ejemplo, si tuviera que crear un subconjunto del primer elemento de la lista usando `my_list[1]` o `my_list["a"]`, el resultado sería una lista. Volviendo a nuestro organizador visual, la lista devuelta contiene un vector numérico:

```
my_list[1]
```

```
$a
[1] 3
```

```
my_list[1] %>% typeof()
```

```
[1] "list"
```

```
my_list["a"]
```

```
$a
[1] 3
```

```
my_list["a"] %>% typeof()
```

```
[1] "list"
```

En cambio cuando creamos un subconjunto de un elemento de lista usando corchetes dobles ([[), recuperamos el elemento en sí, en nuestro caso, un vector doble:

```
my_list[[1]]
```

```
[1] 3
```

```
my_list[[1]] %>% typeof()
```

```
[1] "double"
```

```r
my_list[["a"]]
```

```
[1] 3
```

```r
my_list[["a"]] %>% typeof()
```

```
[1] "double"
```

De manera similar, si un elemento de la lista tiene un nombre, podemos subdividirlo directamente usando su nombre, lo que equivale a usar [[:

```r
my_list$a
```

```
[1] 3
```

Lo que no te dije todavía es cómo funciona la indexación en R para listas y vectores. La indexación es un método para organizar los elementos en una lista o vector y asignarles un valor numérico. Por ejemplo:

```r
my_list[[4]]
```

```
[1]  TRUE FALSE
```

Lo que estamos diciendo aquí es esto: "Dame el cuarto elemento de esa lista". En `R`, las listas siempre comienzan a indexarse en 1, por lo que `my_list[[1]]` devolvería el primer elemento de la lista. La indexación es crucial para acceder a elementos que no tienen nombre.

Los desarrolladores del paquete `purrr` han abordado este problema creando la función `pluck()`, que funciona de manera similar al uso de corchetes dobles [[para listas. Una característica útil de esta función es que siempre devuelve un valor, incluso si el elemento no existe. Por ejemplo, si `my_list` solo contiene cuatro elementos, intentar elegir el décimo elemento no generará un error (como normalmente lo haría). En su lugar, `purrr` devuelve `NULL` para el elemento que no existe. Aquí hay un ejemplo:

```r
pluck(
    .x = my_list,
    10
    )
```

```
NULL
```

El primer argumento (`.x`) toma una lista. Todos los argumentos posteriores toman índices, que pueden ser números enteros o el nombre de un elemento. En el siguiente ejemplo, extraigo el primer elemento, etiquetado como x, de la lista c dentro de my_list (eche otro vistazo a my_list si no está claro):

```
pluck(
    .x = my_list,
    "c", 1
    )
```

```
[1] 4 5 6
```

Lo anterior es similar a:

```
my_list[["c"]][[1]]
```

```
[1] 4 5 6
```

1.1.3 Marcos de datos

Los marcos de datos son un tipo especializado de lista que puede contener elementos heterogéneos. Sin embargo, existen dos diferencias clave entre las listas y los marcos de datos. En primer lugar, los elementos o columnas de un marco de datos deben tener la misma longitud. Este no es un requisito para los elementos de la lista. En segundo lugar, los marcos de datos son bidimensionales y consisten en filas y columnas. Son esencialmente listas de vectores de la misma longitu. Al igual que las listas, los marcos de datos se pueden dividir en subconjuntos haciendo referencia a los nombres de sus elementos.

```
my_data_frame <- tibble(
    a = c(1, 3, 4),
    b = c("A", "few", "words")
    )

my_data_frame$a
```

```
[1] 1 3 4
```

Incluso las reglas de creación de subconjuntos para marcos de datos son similares a las de las listas. El uso de un solo conjunto de corchetes devolverá una lista, mientras que los corchetes dobles devolverán el elemento o la columna específicos:

```
my_data_frame[1] %>% typeof()
```

```
[1] "list"
```

```
my_data_frame[[1]] %>% typeof()
```

```
[1] "double"
```

El marco de datos anterior se creo usando la función `tibble()`, introducido por los creadores del paquete tidverse. Puede pensar en tibbles como marcos de datos de mejor apariencia y mejor comportamiento que los marcos de datos creados a partir de la función `data.frame()`.

Con la fución `tibble()` se pueden crear marcos de datos cononocido como tibble anidado. Estos tibbles anidados serán importantes más adelante en el tutorial cuando presentemos las funciones map. El siguiente ejemplo muestra un tibble anidado:

```
my_tibble <- tibble(
    a = c(1, 3, 4),
    b = c("A", "few", "words"),
    c = list(a = 3, b = TRUE, c = c(3, 7))
    ) %>%
    print()
```

```
# A tibble: 3 x 3
      a b     c
  <dbl> <chr> <named list>
1     1 A     <dbl [1]>
2     3 few   <lgl [1]>
3     4 words <dbl [2]>
```

El resultado anterior es una columna c que contiene una lista. Esta columna se comporta de manera similar a una lista, con los mismos métodos de creación de subconjuntos e indexación que hemos cubierto anteriormente:

```r
# Devuelve el elemento de la lista como una lista de longitud 1
my_tibble$c[1]
my_tibble$c["a"]

# Devuelve los elementos de la lista
my_tibble$c[[1]]
my_tibble$c$a
pluck(my_tibble$c, 1)
```

2 Bucles `for` en `R`

Al comienzo del tutorial, mencioné que la familia de funciones `map` itera sobre elementos de ciertos tipos de datos. Para la mayoría de los estudiantes, la primera técnica para iterar elementos es el bucle for. Esto se debe a que los bucles `for` son explícitos, muestran claramente cada paso computacional y son un concepto fundamental que todos los programadores deben comprender. Para proporcionar una conexión clara entre los bucles `for` y las funciones `map`, presentaré brevemente los bucles `for` en este punto del tutorial.

Como ejemplo, suponga que desea utilizar un ciclo `for` para calcular el cuadrado de cada elemento en un vector numérico:

```r
.input <- c(2, 4, 6)
output <- vector(mode = "numeric", length = length(length(.input)))

for (i in seq_along(.input)) {
    output[[i]] <- .input[[i]]^2
    }

output
```

```
[1]  4 16 36
```

`.input` es un vector atómico de tipo numérico o entero. El `output` es un vector atómico vacío de tipo numérico con la misma longitud que `.input`. El parámetro `length` se utiliza para asignar una cantidad específica de espacio para ese vector. Inicialmente, el `output` está vacío. El bucle `for` usa `seq_along()`, que crea una secuencia de valores desde 1 hasta el número de elementos en `.input`. Por ejemplo, si la entrada tiene tres elementos, la secuencia sería 1, 2, 3:

```r
seq_along(.input)
```

```
[1] 1 2 3
```

Dentro del bucle for, `i` corresponde al elemento actual de la secuencia creada por `seq_along()`:

```
for (i in seq_along(.input)) {
    print(i)
    }
```

```
[1] 1
[1] 2
[1] 3
```

Por últmo, dentro del cuerpo del ciclo `for`, calculamos el cuadrado de cada elemento del vector y almacenamos el resultado en el mismo índice en el vector de salida (`output[[i]] <- .input[[i]]^2`).

Los bucles `for` y las funciones `map` tienen algunas similitudes y se pueden traducir fácilmente, como se demuestra en el paquete `loopurrr` (https://github.com/TimTeaFan/loopurrr) creado por Tim Tiefenbach. Comparten las siguientes características:

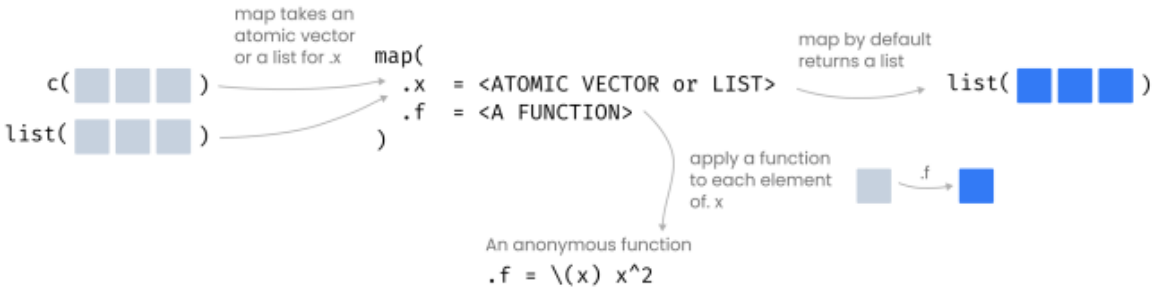
- Tanto los bucles `for` como las funciones `map` toman la entrada en forma de lista, vector, marco de datos o tibble.
- Ambos iteran sobre cada elemento de estas estructuras de datos y aplican un cálculo a ese elemento.
- Ambos crean o devuelven un nuevo objeto.

Es importante tener en cuenta que estas similitudes solo se aplican al bucle `for` específico que acabamos de crear. Ahora que hemos cubierto los bucles `for`, estamos listos para comenzar a trabajar con funciones `map`.

3 Introducción de la familia de funciones `map`

`map` es una familia de funciones, que consta de al menos cinco funciones diferentes. De forma similar a los bucles `for`, las funciones `map` iteran sobre los elementos, aplican una función a cada elemento y devuelven los resultados en una estructura de datos de su elección. El siguiente es un resumen visual de las funciones `map`:

- .x: es una lista o un vector atómico. Los marcos de datos también se pueden usar como entrada, en cuyo caso el mapa iterará sobre cada columna del marco de datos.



- .f: una función proporcionada a .f se aplicará a cada elemento de .x. Se considera una buena práctica escribir esta función en forma de una función R anónima (`\(x) <CUERPO DE LA FUNCIÓN>`) en lugar de usar la tilde `~`, que se considera una mala práctica.