Iteración

Manuel Hernández

9 de junio de 2021

Beneficios de reducir código:

1. La mirada se centra en lo que es diferente y no en lo que es igual.
2. Si se quiere hacer un cambio, se hace en un solo lugar
3. Una línea (o un pedazo) de código se recicla y se usa más veces (menos chance de error).

La iteración como herramienta va en la misma línea que el escribir una función.

Programación imperativa y programación funcional.

For loops

Componentes de un “for loop”:

1. La salida
2. La secuencia
3. El cuerpo

```r
library(tidyverse)
library(datos)

df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}

output
```

```
[1] 0.08038629 0.24528198 0.06934612 0.14862489
```

Es recomendable no hacer crecer la salida en la iteración. ¿Siempre se puede hacer esto?

Ejercicio

```
medias <- c(-10, 0, 10, 100)
output <- vector('list', length(medias))
for (i in seq_along(medias)) {
  output[[i]] <- rnorm(10, mean = medias[i])
}
```

```
output

[[1]]
 [1]  -9.645213 -11.995838 -11.531545  -9.285282 -11.170774 -10.958776
 [7]  -8.277685 -10.771890  -9.300486  -8.765334

[[2]]
 [1]  0.3084540 -0.3016025  1.9028652  0.5151496  1.2389715  0.4853460
 [7] -1.1010287  0.5979683  0.3624208  0.3834574

[[3]]
 [1]  8.895915 11.860792 10.869282  8.960748  9.036125 10.184384  9.678609
 [8]  9.101786  9.997024  9.287663

[[4]]
 [1] 100.13019  98.55579  99.87416  98.80059 100.48139  99.21567 100.47226
 [8] 100.82422 100.26410 100.78792
```

Ejercicio

```
output %>%
  data.table::as.data.table()

            V1         V2        V3        V4
 1:  -9.645213  0.3084540  8.895915 100.13019
 2: -11.995838 -0.3016025 11.860792  98.55579
 3: -11.531545  1.9028652 10.869282  99.87416
 4:  -9.285282  0.5151496  8.960748  98.80059
 5: -11.170774  1.2389715  9.036125 100.48139
 6: -10.958776  0.4853460 10.184384  99.21567
 7:  -8.277685 -1.1010287  9.678609 100.47226
 8: -10.771890  0.5979683  9.101786 100.82422
 9:  -9.300486  0.3624208  9.997024 100.26410
10:  -8.765334  0.3834574  9.287663 100.78792

output %>%
  bind_cols()

New names:
NA -> ...1
NA -> ...2
NA -> ...3
NA -> ...4
# A tibble: 10 x 4
     ...1   ...2  ...3  ...4
    <dbl>  <dbl> <dbl> <dbl>
 1  -9.65  0.308  8.90 100.
 2 -12.0  -0.302 11.9   98.6
 3 -11.5   1.90  10.9   99.9
 4  -9.29  0.515  8.96  98.8
 5 -11.2   1.24   9.04 100.
 6 -11.0   0.485 10.2   99.2
 7  -8.28 -1.10   9.68 100.
 8 -10.8   0.598  9.10 101.
 9  -9.30  0.362 10.0  100.
```

Hay cuatro variaciones del bucle for básico:

1. Modificar un objeto existente, en lugar de crear un nuevo objeto.
2. Iterar sobre nombres o valores, en lugar de índices.
3. Manejar outputs de longitud desconocida.
4. Manejar secuencias de longitud desconocida.

Ejemplo de 1 y 2

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

for (nm in names(df)) {
  df[[nm]] <- rescale01(df[[nm]])
}

for (col in df){
 print(col)
}

 [1] 0.6595492 0.0000000 0.2430628 1.0000000 0.4356174 0.6745473 0.1995
 [8] 0.3979880 0.6905337 0.5132611
 [1] 0.9390897 0.9270551 0.2540175 0.5691313 0.5557492 0.9876692 0.9564
 [8] 0.0000000 0.2634342 1.0000000
 [1] 0.9971237 1.0000000 0.4059595 0.7454746 0.8826924 0.7628612 0.7908
 [8] 0.6122925 0.0000000 0.9788927
 [1] 0.65276299 0.30108158 0.74442244 0.92609174 0.99748626 0.00000000
 [7] 0.06450181 0.58014556 0.42782066 1.00000000
```

Longitud de output desconocida.

En vez de hacer crecer un vector progresivamente:

```
out <- vector("list", length(medias))
for (i in seq_along(medias)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, medias[[i]])
}
str(out)
```

```
List of 4
 $ : num [1:94] -9.07 -10.86 -11.39 -9.5 -10.73 ...
 $ : num [1:54] -1.0842 -0.9118 -1.1979 -0.3386 -0.0437 ...
 $ : num [1:15] 10.81 7.98 10.06 9.47 9.07 ...
 $ : num [1:9] 99.4 100.8 99 100.4 100.4 ...
```

```
str(unlist(out))
```

```
 num [1:172] -9.07 -10.86 -11.39 -9.5 -10.73 ...
```

Longitud de secuencia desconocida.

A veces, podemos recurrir a los bucles *while*:

```
lanzamiento <- function() sample(c("S", "C"), 1)

lanzamientos <- 0
ncaras <- 0

while (ncaras < 3) {
  if (lanzamiento() == "C") {
    ncaras <- ncaras + 1
  } else {
    ncaras <- 0
  }
  lanzamientos <- lanzamientos + 1
}
lanzamientos
```

```
[1] 43
```

Ejercicio

Imagina que tienes un directorio lleno de archivos CSV que quieres importar. Tienes sus ubicaciones en un vector, y ahora quieres leer cada uno con `read_csv()`. Escribe un bucle for que los cargue en un solo data frame.

```
#Voy a generar tibbles:
tibble(hora = Sys.time(), Var1 = rnorm(10))

# A tibble: 10 x 2
   hora                  Var1
   <dttm>               <dbl>
 1 2021-06-09 13:30:56 -0.796
 2 2021-06-09 13:30:56 -2.65
 3 2021-06-09 13:30:56 -0.115
 4 2021-06-09 13:30:56  1.31
 5 2021-06-09 13:30:56 -0.453
 6 2021-06-09 13:30:56  0.693
 7 2021-06-09 13:30:56 -0.330
 8 2021-06-09 13:30:56 -0.567
 9 2021-06-09 13:30:56  0.595
10 2021-06-09 13:30:56  1.20
```

Ejercicio

```
replicate(8, tibble(hora = Sys.time(), x1 = rnorm(10))) # List of 16

     [,1]         [,2]         [,3]         [,4]         [,5]         [,6]
hora POSIXct,10 POSIXct,10 POSIXct,10 POSIXct,10 POSIXct,10 POSIXct,10
x1   numeric,10 numeric,10 numeric,10 numeric,10 numeric,10 numeric,10
     [,7]         [,8]
hora POSIXct,10 POSIXct,10
x1   numeric,10 numeric,10

replicate(8, {Sys.sleep(1); tibble(hora = Sys.time(), x1 = rnorm(10))}, simplify = F) # List of 8

[[1]]
# A tibble: 10 x 2
   hora                     x1
   <dttm>                <dbl>
 1 2021-06-09 13:18:23 -0.519
 2 2021-06-09 13:18:23 -0.252
 3 2021-06-09 13:18:23 -0.293
 4 2021-06-09 13:18:23 -0.0755
 5 2021-06-09 13:18:23 -0.642
 6 2021-06-09 13:18:23  0.514
 7 2021-06-09 13:18:23 -0.779
 8 2021-06-09 13:18:23  0.905
 9 2021-06-09 13:18:23 -0.451
10 2021-06-09 13:18:23 -1.13

[[2]]
# A tibble: 10 x 2
   hora                     x1
   <dttm>                <dbl>
 1 2021-06-09 13:18:24  0.186
 2 2021-06-09 13:18:24  0.442
 3 2021-06-09 13:18:24 -0.867
 4 2021-06-09 13:18:24  0.558
 5 2021-06-09 13:18:24  0.120
```

Ejercicio

```
system('mkdir data') # creo una carpeta

set.seed(1234)
replicate(8, {
  Sys.sleep(1) # espero un momento
  tabla <- tibble(hora = Sys.time(), # genero un tibble
                  x1 = rnorm(10))
  nombre <- sample(100000, size = 1) # genero un nombre
  write_csv(tabla, file = str_c('data/', nombre, '.csv')) # guardo la tabla con el nombre en un .csv
  print(dir('data/'))
  },
  simplify = F) %>%
  invisible()

[1] "85374.csv"
[1] "85374.csv" "98435.csv"
[1] "85374.csv" "98435.csv" "98865.csv"
[1] "59011.csv" "85374.csv" "98435.csv" "98865.csv"
[1] "59011.csv" "73520.csv" "85374.csv" "98435.csv" "98865.csv"
[1] "30658.csv" "59011.csv" "73520.csv" "85374.csv" "98435.csv" "98865.csv"
[1] "30658.csv" "37741.csv" "59011.csv" "73520.csv" "85374.csv" "98435.csv"
[7] "98865.csv"
[1] "30658.csv" "37741.csv" "4987.csv"  "59011.csv" "73520.csv" "85374.csv"
[7] "98435.csv" "98865.csv"
```

Ejercicio

```
archivos <- dir('data', pattern = "\\.csv", full.names = T)
datos <- vector('list', length(archivos))
for (file in archivos){
  read_csv(file)
}


-- Column specification
--------------------------------------------------------
cols(
 hora = col_datetime(format = ""),
 x1 = col_double()
)
-- Column specification
--------------------------------------------------------
cols(
 hora = col_datetime(format = ""),
 x1 = col_double()
)
-- Column specification
--------------------------------------------------------
cols(
```

Ejercicio

```
for (i in seq_along(archivos)){
  datos[[i]] <- read_csv(archivos[i])
}
```

```
bind_rows(datos) %>% dim

[1] 80  2

bind_rows(datos)

# A tibble: 80 x 2
   hora                   x1
   <dttm>              <dbl>
 1 2021-06-09 16:19:19 -0.773
 2 2021-06-09 16:19:19  1.61
 3 2021-06-09 16:19:19 -1.16
 4 2021-06-09 16:19:19  0.657
 5 2021-06-09 16:19:19  2.55
 6 2021-06-09 16:19:19 -0.0348
 7 2021-06-09 16:19:19 -0.670
 8 2021-06-09 16:19:19 -0.00760
 9 2021-06-09 16:19:19  1.78
10 2021-06-09 16:19:19 -1.14
```

Bucles *for* vs. funcionales

Queremos hacer algo muchas veces, podemos empaquetar ese bucle en una función y *aplicarlo* a lo que queramos.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
```

```
[1] 0.26297388 0.10206332 0.03101509 0.20086114
```

Bucles *for* vs. funcionales

```
col_media <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
col_resumen <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}

col_resumen(df, mean)

[1] 0.26297388 0.10206332 0.03101509 0.20086114

col_resumen(df, median)

[1] -0.1300696 -0.2606265  0.1035090  0.0576949
```

Funciones *map*

Es muy común iterar sobre vectores, tablas de datos, listas, arreglos. Entonces nos podemos ahorrar escribir toda la cuestión del `for`.

En R base: `apply`, `lapply`, `sapply`, ...

```
df %>% apply(2, mean) # promedia columnas. igual que colMeans()

         a          b          c          d 
0.26297388 0.10206332 0.03101509 0.20086114 

df %>% apply(1, mean) # promedia columnas. igual que colMeans()

 [1] -0.68158357  0.24506245  0.67546482  0.10400059  0.26377208  0.30151427
 [7]  0.26102628  0.42170812 -0.14348828  0.04480678

lapply(df, mean) # lee el tibble como lista, y es como apply(2, mean) pero sin simplify

$a
[1] 0.2629739

$b
[1] 0.1020633

$c
[1] 0.03101509

$d
[1] 0.2008611
```

Funciones *map*

`purrr` tiene las funciones `map`:

```
map_dbl(df, mean)

         a          b          c          d
0.26297388 0.10206332 0.03101509 0.20086114

map_int(df, mean)

Error: Can't coerce element 1 from a double to a integer

map(df, mean)

$a
[1] 0.2629739

$b
[1] 0.1020633

$c
[1] 0.03101509

$d
[1] 0.2008611
```

Funciones *map*

```
map_lgl(df, function(x) mean(x) > 0)

   a    b    c    d
TRUE TRUE TRUE TRUE

df %>% map_lgl(~ mean(.x) > 0)

   a    b    c    d
TRUE TRUE TRUE TRUE

modelos <- mtautos %>%
  split(.$cilindros) %>%
  map(~lm(millas ~ peso, data = .x))
```

Manejando los errores

```
log_seguro <- safely(log)
log_seguro(10)
```

```
$result
[1] 2.302585

$error
NULL
```

```
log_seguro("a")
```

```
$result
NULL

$error
<simpleError in .Primitive("log")(x, base): non-numeric argument to mat
```

```
str(log_seguro("a"))
```

```
List of 2
 $ result: NULL
 $ error :List of 2
  ..$ message: chr "non-numeric argument to mathematical function"
  ..$ call   : language .Primitive("log")(x, base)
```

Manejando los errores

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
y
```

```
[[1]]
[[1]]$result
[1] 0

[[1]]$error
NULL


[[2]]
[[2]]$result
[1] 2.302585

[[2]]$error
NULL


[[3]]
[[3]]$result
```

Map sobre múltiples argumentos

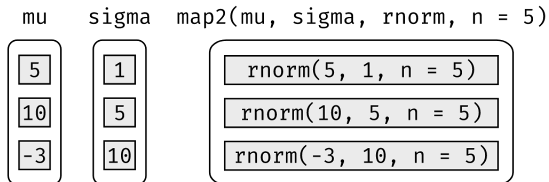
```
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
seq_along(mu) %>%
  map(~rnorm(5, mu[[.x]], sigma[[.x]])) %>%
  str()
```

```
List of 3
 $ : num [1:5] 3.72 5.41 4.18 3.76 3.8
 $ : num [1:5] 5.67 18.08 13.22 11.96 17.13
 $ : num [1:5] -2.08 -6.41 -5.11 4.58 -11.83
```

```
map2(mu, sigma, rnorm, n = 5) %>% str
```

```
List of 3
 $ : num [1:5] 3.68 4.79 5.87 4.21 4.79
 $ : num [1:5] 13.78 8.89 16.63 3.74 5.79
 $ : num [1:5] -4.87 -8.96 4.85 11.78 1.1
```

Map sobre múltiples argumentos



Se generaliza con pmap():

```
n <- list(1, 3, 5) # tambien vale con c(1, 3, 5)
args1 <- list(n, mu, sigma)
pmap(args1, rnorm) %>% str()
```

```
List of 3
 $ : num 6.52
 $ : num [1:3] 11.1 15.9 10
 $ : num [1:5] -1.61 0.33 -21.3 4.59 14.52
```