Programación en R

Entornos de Análisis de Datos: R

Alberto Torres Barrán

2021-01-07

Vectores

- Dos tipos de vectores:
  1. vectores **atomic**, 6 tipos distintos: `logical`, `integer`, `double`, `character`, `complex` y `raw`
  2. listas, que son vectores recursivos (pueden contener otras listas)
- Dos propiedades principales:
  - tipo, función `typeof()`
  - longitud, función `length()`
- Los elementos de un vector pueden tener nombre

```
c(a = 1, b = 2, c = 3)
## a b c
## 1 2 3
```

Vectores atómicos

- Comprobar el tipo: `is.logical()`, `is.integer()`, `is.double()`, `is.character()`
- Convertir de un tipo a otro: `as.logical()`, `as.integer()`, `as.double()`, `as.character()`
- Cuando combinamos elementos de distinto tipo, existe una conversión implícita al tipo más genérico

```
5 + TRUE
## [1] 6
c(4.5, "hola")
## [1] "4.5"  "hola"
```

Listas

Pueden contener elementos de distinto tipo, incluido otras listas

```
l <- list(a = "a", b = 10.2, c = TRUE, d = 1:10, e = list(1, 2))
str(l)
## List of 5
##  $ a: chr "a"
##  $ b: num 10.2
##  $ c: logi TRUE
##  $ d: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ e:List of 2
##   ..$ : num 1
##   ..$ : num 2
```

Indexado

Distintas formas de indexar elementos:

1. `[` extrae una sub-lista

```
l[1:3]
## $a
## [1] "a"
##
## $b
## [1] 10.2
##
## $c
## [1] TRUE
```

2. `[[` extrae un elemento

```
l[[4]]
##  [1]  1  2  3  4  5  6  7  8  9 10

l[["d"]]
##  [1]  1  2  3  4  5  6  7  8  9 10
```

1. `$` similar a `[[` pero solo se puede usar con la etiqueta del elemento (no posicion)

```
l$d
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Vectores aumentados

- Los vectores pueden contener atributos arbitrarios (metadatos)
- Usando estos atributos se construyen vectores aumentados:
  - *Factors*, a partir de vectores de enteros
  - *Dates* y *Date-times*, a partir de vectores numéricos
  - *Tibbles* y *data.frames*, a partir de listas
- Podemos comprobar estos tipos aumentados con la función `class()`

```
class(mpg)
## [1] "tbl_df"     "tbl"        "data.frame"

typeof(mpg)
## [1] "list"
```

Funciones

- Escribir funciones evita la repetición de código
- Al igual que las funciones incluidas en R, tienen argumentos de entrada y un valor de retorno
- Útiles cuando copiamos y pegamos el **mismo** código para usarlo con distintas variables
- Ejemplo: función que cuenta el número de valores NA en un vector

```
count_na <- function(x) {
  sum(is.na(x))
}

count_na(c(4, 6, NA, 3))
## [1] 1
```

Ejecución condicional

- La sentencia `if` permite ejecutar código dependiendo de una condición (la parte del `else` es opcional)

```
if (condicion) {
  # codigo que se ejecuta si condicion es TRUE
} else {
  # codigo que se ejecuta si condicion es FALSE
}
```

- Es común usar los operadores lógicos:

Operador	Descripción
`<`	menor que
`<=`	menor o igual
`>`	mayor que
`>=`	mayor o igual
`==`	igual
`!=`	distinto
`!x`	negación (no `x`)

Múltiples condiciones

- Primero se comprueba la primera condición, y solo si es falsa se comprueba la segunda

```
if (condicion1) {
  # condigo a ejecutar si condicion1 es TRUE
} else if (condicion2) {
  # codigo a ejecutar si condicion1 es FALSE pero condicion2 es TRUE
} else {
  # si ambas son FALSE
}
```

- También podemos usar `&&` (AND lógico) y `||` (OR lógico) para combinar múltiples condiciones en una

Sentencia if_else()

Sentencia condicional vectorizada (librería `dplyr`)

```
mpg %>%
  mutate(consumo = if_else(cty < 20, "bajo", "alto")) %>%
  select(cty, consumo)
## # A tibble: 234 x 2
##       cty consumo
##     <int> <chr>
##  1    18 bajo
##  2    21 alto
##  3    20 alto
##  4    21 alto
##  5    16 bajo
##  6    18 bajo
##  7    18 bajo
##  8    18 bajo
##  9    16 bajo
## 10    20 alto
## # ... with 224 more rows
```

Argumentos opcionales

- Podemos añadir argumentos opcionales asignándoles un valor por defecto

```
count_na <- function(x, normalize = FALSE) {
  if (normalize) {
    mean(is.na(x))
  } else {
    sum(is.na(x))
  }
}
```

- Ejemplo:

```
count_na(c(NA, NA, 3, 5, NA, 2), normalize = TRUE)
## [1] 0.5
```

- Si falta un argumento no opcional, R devuelve un error

```
count_na()
## Error in count_na(): el argumento "x" está ausente, sin valor por omisión
```

Valores de retorno

- Por defecto, las funciones devuelven el resultado de la última linea de código
- También se puede usar la sentencia `return()`

```
# Función que cuenta el número de valores NA en un vector
count_na <- function(x, normalize = FALSE) {
  if (!is.vector(x)) {
    return(NA)
  }

  if (normalize) {
    mean(is.na(x))
  } else {
    sum(is.na(x))
  }
}
```

```
count_na(mpg)
## [1] NA
```

```
count_na(list(1, 2, 3, NA))
## [1] 1
```

Iteración

- Repetir el mismo código con varios elementos de un vector o lista
- Existen dos tipos principales de bucles en R:
  - `for` (número predeterminado de repeticiones)
  - `while` (número variable de repeticiones)
- Hemos visto otra forma "oculta" de iteración, la función `across` de la librería `dplyr`

Bucle for

Los bucles for tienen 3 partes:

1. **Salida**: el número de repeticiones es conocido, con lo que podemos crear de antemano el vector que almacena los resultados

2. **Secuencia**: determina el número de repeticiones y el valor de `i` en cada una de ellas

3. **Cuerpo**: entre corchetes (`{}`), es el código que se va a repetir para cada uno de los valores de `i`

```
df <- select(mpg, is.numeric)

# Salida
output <- vector("double", ncol(df))
# Secuencia
for (i in seq_along(df)) {
  # Cuerpo
  output[[i]] <- median(df[[i]])
}

output
## [1]    3.3 2003.5    6.0   17.0   24.0
```

Patrones de iteración

1. Iterar sobre índices: en cada iteración el valor de `i` es un número entero, se usa la función `seq_along`

2. Iterar sobre los elementos: en cada iteración el valor de `x` es cada uno de los elementos de la secuencia:

```
for(x in df) {
  print(mean(x))
}
## [1] 3.471795
## [1] 2003.5
## [1] 5.888889
## [1] 16.85897
## [1] 23.44017
```

- En general podemos iterar sobre:

    1. vectores

    2. listas

    3. dataframes, en cuyo caso iteramos sobre las **columnas**

Bucle while

- Es más general, todo bucle for se puede reescribir como un bucle `while`

```
i <- 1
while (i <= ncol(df)) {
  i <- i + 1
}
```

- Se utiliza cuando no se conoce de antemano el número de repeticiones:

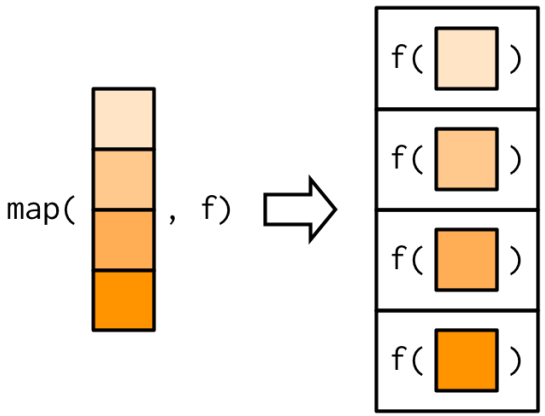
```
set.seed(1)

n_caras <- 0
while(n_caras < 3) {
  moneda <- sample(c("cara", "cruz"), size = 1)
  print(paste("Lanzando moneda...", moneda))

  if(moneda == "cara") {
    n_caras <- n_caras + 1
  }
}
## [1] "Lanzando moneda... cara"
## [1] "Lanzando moneda... cruz"
## [1] "Lanzando moneda... cara"
## [1] "Lanzando moneda... cara"
```

Familia de funciones map

- Iteran sobre un vector realizando una operación sobre cada uno de sus elementos:
  - `map()`, crea una lista
  - `map_lgl()`, crea un vector lógico
  - `map_int()`, crea una vector de enteros
  - `map_dbl()`, crea un vector de dobles
  - `map_chr()`, crea un vector de cadenas de caracteres
  - `map_df()`, crea un data frame
- Sustituyen a los bucles en los casos de uso más comunes



Ejemplo

- El bucle que hemos visto anteriormente:

```
df <- select(mpg, is.numeric)

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

output
## [1]    3.471795 2003.500000    5.888889   16.858974   23.440171
```

- Se puede reemplazar por una llamada a la función `map_dbl`:

```
map_dbl(df, mean)
##       displ        year         cyl         cty         hwy
##    3.471795 2003.500000    5.888889   16.858974   23.440171
```

- Aunque el resultado es el mismo, en general preferimos la versión con `map` porque el código es más conciso y fácil de leer

Funciones propias

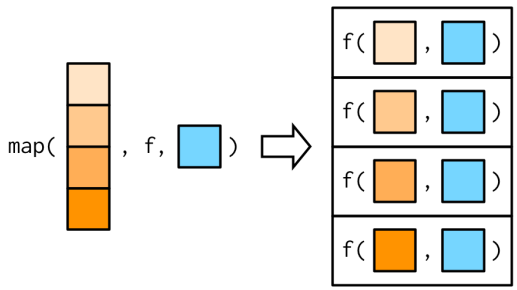
- `map_*` acepta functiones definidas por el usuario

```
count_na <- function(x, normalize = FALSE) {
  if (normalize) {
    mean(is.na(x))
  } else {
    sum(is.na(x))
  }
}

map_dbl(airquality, count_na)
##   Ozone Solar.R    Wind    Temp   Month     Day
##      37       7       0       0       0       0
```

- Si las funciones tienen parámetros opcionales, se los podemos pasar como argumentos a map

```
map_dbl(airquality, count_na,
        normalize=TRUE)
##      Ozone    Solar.R       Wind
## 0.24183007 0.04575163 0.00000000 0.0000
```



Funciones anónimas

- `purrr` incluye una sintáxis especial para crear funciones anónimas

```
map_df(df, ~(. - min(.)) / ((max(.) - min(.))))
## # A tibble: 234 x 5
##     displ  year   cyl   cty   hwy
##     <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 0.0370     0   0   0.346 0.531
##  2 0.0370     0   0   0.462 0.531
##  3 0.0741     1   0   0.423 0.594
##  4 0.0741     1   0   0.462 0.562
##  5 0.222      0   0.5 0.269 0.438
##  6 0.222      0   0.5 0.346 0.438
##  7 0.278      1   0.5 0.346 0.469
##  8 0.0370     0   0   0.346 0.438
##  9 0.0370     0   0   0.269 0.406
## 10 0.0741     1   0   0.423 0.5
## # ... with 224 more rows
```

- Más información: `map`

R base vs purrr

- Existen funciones en R base similares a la familia `map`
- En general, se recomienda el uso de las funciones de `purrr` porque sus argumentos y nombres son más consistentes
- Por tanto, intentar evitar el uso de funciones como:
  - `lapply` (equivalente a `map`)
  - `vapply` (equivalente a `map_dbl`, `map_chr`, `map_lgl`, etc.)
  - `sapply` (no tenemos control sobre el tipo del vector de salida)
  - `apply` (devuelve un array, no un dataframe)
- Diferencias entre R base y purrr
- Tutorial de purrr