purrr

Entornos de Análisis de Datos: R

Alberto Torres Barrán

2020-01-14

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
```

1. `[[` 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

- Las funciones evitan la repetición de código
- Al igual que las funciones de R, tienen argumentos de entrada y un valor de retorno
- Útiles cuando copiamos y pegamos el **mismo** código para usarlo con distintas variables

```r
# 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

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

Se puede usar `&&` (AND lógico) y `||` (OR lógico) para combinar múltiples expresiones

Múltiples condiciones

```
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
}
```

Sentencia ifelse()

Es una sentencia condicional vectorizada

```
mpg %>%
  mutate(consumo = ifelse(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
```

Valores por defecto

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

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

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.atomic(x)) {
    return(NA)
  }

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

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

Iteración

- Se utiliza para aplicar el mismo código a varias entradas
- La forma más conocida son los bucles:
  - for
  - while

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

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

- Hemos visto otra forma "oculta" de iteración, las funciones `summarize_all()` y `summarize_if()`

purrr

- La librería proporciona funciones que sustituyen a los bucles en la mayoría de los casos más comunes:
  - `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

```r
df <- select_if(mpg, is.numeric)
map_dbl(df, mean, na.rm = TRUE)
##       displ        year         cyl         cty         hwy
##    3.471795 2003.500000    5.888889   16.858974   23.440171
```

`map_*()` acepta functiones definidas por el usuario y tiene una sintáxis especial para declarar funciones anónimas

```
map_df(df, ~(. - min(.) / (max(.) - min(.))))
## # A tibble: 234 x 5
##    displ  year   cyl   cty   hwy
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  1.50 1777.     3  17.7  28.6
##  2  1.50 1777.     3  20.7  28.6
##  3  1.70 1786.     3  19.7  30.6
##  4  1.70 1786.     3  20.7  29.6
##  5  2.50 1777.     5  15.7  25.6
##  6  2.50 1777.     5  17.7  25.6
##  7  2.80 1786.     5  17.7  26.6
##  8  1.50 1777.     3  17.7  25.6
##  9  1.50 1777.     3  15.7  24.6
## 10  1.70 1786.     3  19.7  27.6
## # ... with 224 more rows
```