Introducción al R

Objetos, Apply y aggregate

true

19 de marzo de 2018

Contents

Objetos

Ya hemos visto la definición de un objeto, además de nuestro primer objeto: un vector. Ahora veremos cuatro de los objetos más usados en los primetos pasos en R: factores, listas, matrices y data.frames.

Factor

- Un tipo de vector para datos categóricos

```
z <- factor(LETTERS[1:3], ordered = TRUE)
x <- factor(c("a", "b", "b", "a"))
x
```

```
## [1] a b b a
## Levels: a b
```

Los factores son útiles cuando se conocen los valores posibles de una variable puede tomar, incluso si no se ve todos los valores en un determinado conjunto de datos. El uso de un factor en lugar de un vector de caracteres hace evidente cuando algunos grupos no contienen observaciones:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
```

```
## sex_char
## m
## 3
```

```
table(sex_factor)
```

```
## sex_factor
## m f
## 3 0
```

Listas

Es *vector generalizado*. Cada lista está formada por componentes (que pueden ser otras listas), y cada componente puede ser de un tipo distinto. Son unos "contenedores generales".

```
n <- c(2, 3, 5)
s <- c("aa", "bb", "cc", "dd", "ee")
b <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
x <- list(n, s, b, 3)
```

A las listas a veces se les llama *vectores recursivos*, porque pueden contener otras listas.

```
x <- list(list(list(list())))
str(x)
```

```
## List of 1
##  $ :List of 1
##   ..$ :List of 1
##   .. ..$ : list()
```

```
is.recursive(x)
```

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

`c()` combinará varias listas en una sola. Si se tiene una combinación de vectores y listas, `c()` coerciona a los vectores como listas antes de combinarlos. Compara los resultados de `list()` y `c()`:

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
```

```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ : num [1:2] 3 4
```

```r
str(y)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

Missing Values

Los valores perdidos se denotan por `NA` o `NaN` para operaciones matemáticas no definidas.

- `is.na()` se usa para comprobar si un objeto es `NA`
- `is.nan()` se usa para comprobar si un objeto es `NaN`
- `NA` también pertenecen a una clase como numeric `NA`, existe character `NA`, etc.
- Un `NaN` también es un `NA` pero al revés no es cierto

```r
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

```r
is.nan(x)
```

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

```r
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```r
is.nan(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

Matrices y Dataframes

En esta clase, vamos a cubrir las *matrices* y *data frames*. Ambos representan los tipos de datos "rectangulares", lo que significa que se utilizan para almacenar datos tabulares, con filas y columnas.

La principal diferencia, como se verán, es que las matrices sólo pueden contener una sola clase de datos, mientras que las data frames pueden consistir en muchas clases diferentes de datos.

Matrices

Es un tipo de objeto que contiene elementos del mismo tipo. A diferencia de los vectores, este tiene el atributo `dim`, veamos:

- Vamos a crear un vector que contiene los números del 1 al 20 con el operador `:` . Almacenar el resultado en una variable llamada my_vector.
- Escribe `dim(my_vector)`. Resulta que no tiene este atributo.
- Sin embargo, la función `dim` se usa para pedir o asignar este atributo. Escribe `dim(my_vector) <- c(4, 5)`
- Ahora, mira cuál es la dimensión de `my_vector`

  - Al igual que en la clase de matemáticas, cuando se trata de un objeto de 2 dimensiones (piense mesa rectangular), el primer número es el número de filas y el segundo es el número de columnas. Por lo tanto, `my_vector` ahora tiene 4 filas y 5 columnas.
  - ¡Pero espera! Eso no suena como un vector más. Bueno, no lo es. Ahora es una matriz. Ver el contenido de `my_vector` ahora para ver lo que parece. Imprime el contenido de `my_vector`

- Ves, ahora tenemos una matriz, confirmemos esto usando a función `class()`, así: `class(my_vector)`.
- Efectivamente, `my_vector` es ahora una matriz. Deberíamos almacenarlo en una nueva variable que nos ayuda a recordar lo que es. Almacena el valor de `my_vector` en una nueva variable llamada `my_matrix`.

El código del ejemplo anterior sería:

```
my_vector <- 1:20
dim(my_vector)
```

```
## NULL
```

```
dim(my_vector) <- c(4, 5)
my_vector
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```
class(my_vector)
```

```
## [1] "matrix"
```

```
my_matrix <- my_vector
```

El ejemplo que hemos utilizado hasta ahora estaba destinado a ilustrar el punto de que una matriz es simplemente un vector con un atributo de dimensión. Un método más directo de la creación de la misma matriz utiliza la función de `matrix()`.

- Mira la ayuda de `matrix()`. Encuentra la manera de crear una matriz que contiene los mismos números (1-20) y dimensiones (4 filas, 5 columnas) usando a la función de `matrix()`. Almacenar el resultado en una variable llamada `my_matrix2`.
- Ahora veamos si `my_matrix` y `my_matrix2` son idénticas. Usamos la función `identical()`

El código sería:

```
my_matrix2 <- matrix(1:20, nrow = 4, ncol = 5)
identical(my_matrix , my_matrix2)
```

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

Ahora, imagina que los números en la mesa representan algunas medidas de un experimento clínico, donde cada fila representa un paciente y cada columna representa una variable para la que se tomaron mediciones.

Podemos querer etiquetar las filas, para que sepamos qué números pertenecen a cada paciente en el experimento. Una forma de hacer esto es agregar una columna a la matriz, que contiene los nombres de las cuatro personas.

- Vamos a empezar por la creación de un vector de caracteres que contiene los nombres de nuestros pacientes - Josefa, Gina, Jose, y Julio Recuerda que las comillas dobles dicen R que algo es una cadena de caracteres. Almacena el resultado en la variable llamada `patients`.
- Ahora vamos a utilizar la función `cbind()` para *combinar columnas*. No te preocupes por guardar el resultado en una nueva variable. Sólo tienes que usar `cbind()` con dos argumentos - el vector de los pacientes y `my_matrix`.

- Algo está raro en el resultado! Parece que la combinación del vector *character* con nuestra matriz de números hizo que todo esté entre comillas dobles. Esto significa que nos quedamos con una matriz de caracteres, lo que no es bueno.
- Si recuerdas, dijimos que las matrices sólo pueden contener un tipo de datos. Por lo tanto, cuando tratamos de combinar un vector de caracteres con una matriz numérica, `R` se vio obligado a *coeccionar* los números en caracteres, de ahí las comillas dobles.

Data.frame

- Por lo tanto, estamos todavía con la cuestión de cómo incluir los nombres de nuestros pacientes en la tabla sin dañar de nuestros datos numéricos. Prueba lo siguiente - `my_data <- data.frame(patients, my_matrix)`

Parece que la función `data.frame()` nos permitió guardar nuestro vector de caracteres de los nombres justo al lado de nuestra matriz de números. Eso es exactamente lo que esperábamos!

- Chequea el tipo de objeto que hemos creado con `class(my_data)`

También es posible asignar nombres a las filas y columnas de un data frame, lo cual es otra posible forma de determinar qué fila de valores en nuestra tabla pertenece a cada paciente.

- Ya que tenemos seis columnas (incluyendo nombres de los pacientes), tendremos que crear primero un vector que contiene un elemento para cada columna. Crea un vector de caracteres llamado `cnames` que contiene los valores siguientes (en orden) - *patient*, *age*, *weight*, *bp*, *rating*, *test*.
- Ahora, utilice los `colnames()` para establecer el atributo `colnames` para nuestro data frame. Es similar a la función `dim()` que usamos antes. Imprime `my_data`.

El código sería:

```
patients <- c("Josefa", "Gina", "Jose", "Julio")
cbind(patients,my_matrix)
```

```
##      patients
## [1,] "Josefa" "1" "5" "9"  "13" "17"
## [2,] "Gina"   "2" "6" "10" "14" "18"
## [3,] "Jose"   "3" "7" "11" "15" "19"
## [4,] "Julio"  "4" "8" "12" "16" "20"
```

```
my_data <- data.frame(patients, my_matrix)
cnames <- c("patient", "age", "weight", "bp", "rating", "test")
colnames(my_data) <- cnames
```

Desde luego, podemos crear un data frame directamente, por ejemplo

```
my.data.frame <- data.frame(
    ID = c("Carla", "Pedro",    "Laura"),
    Edad = c(10, 25, 33),
    Ingreso = c(NA, 34, 15),
    Sexo = c(TRUE, FALSE, TRUE),
  Etnia = c("Mestizo","Afroecuatoriana","Indígena")
)
```

Operaciones con matrices

- R posee facilidades para manipular y hacer operaciones con matrices. Las funciones `rbind()` y `cbind()` unen matrices con respecto a sus filas o columnas respectivamente:

```
m1 <- matrix(1, nr = 2, nc = 2)
m2 <- matrix(2, nr = 2, nc = 2)

rbind(m1, m2)
cbind(m1,m2)
```

- El operador para el producto de dos matrices es `%*%`. Por ejemplo, considerando las dos matrices m1 y m2:

```
ma <- rbind(m1, m2) %*% cbind(m1, m2)
```

- La transpuesta de una matriz se realiza con la función `t`; esta función también funciona con data frames.

```
t(ma)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    2    4    4
## [2,]    2    2    4    4
## [3,]    4    4    8    8
## [4,]    4    4    8    8
```

- Para el cálculo de la inversa se usa `solve`

```
solve(ma)
```

Otro uso de la función `solve()` es la solución de sistemas de ecuaciones, por ejemplo:

$$
\begin{aligned}
3x + 2y + z &= 1 \quad (1)\\
5x + 3y + 4z &= 2 \quad (2)\\
x + y - z &= 1 \quad (3)\\
&\quad (4)
\end{aligned}
$$

Cuya solución en `R` sería:

```
A <- matrix(c(3,5,1,2,3,1,1,4,-1),ncol=3)
b <- c(1,2,1)
solve(A,b)
```

```
## [1] -4  6  1
```

Arrays

- Generalización multidimensional de vector. Elementos del mismo tipo.

```
x <- array(1:20, dim=c(4,5))
```

Bucles

- Una ventaja de R comparado con otros programas estadísticos con "menus y botones" es la posibilidad de programar de una manera muy sencilla una serie de análisis que se puedan ejecutar de manera sucesiva.

- Por ejemplo, definamos un vector con 50.000 componentes y calculemos el cuadrado de cada componente primero usando las propiedades de R de realizar cálculos componente a componente y luego usando un ciclo.

```
x <- 1:50000
y <- x^2
```

- Con bucles:

```
z <- 0
for (i in 1:50000) z[i] <- x[i]^2
```

FOR

La sintaxis de la intrucción es:

for (i in valores) { instrucciones }

Ejemplo:

```
for (i in 1:5)
{
  print (i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Ejemplos

- ivalores: numérico

```
for (i in c(3,2,9,6)){
    print(i^2)
}
```

```
## [1] 9
## [1] 4
## [1] 81
## [1] 36
```

- i caractér e ivalores vector:

```
medios.transporte <- c("carro", "camion","metro", "moto")
    for (vehiculo in medios.transporte)
    {print (vehiculo)}
```

```
## [1] "carro"
## [1] "camion"
## [1] "metro"
## [1] "moto"
```

While

- Ejemplo 1:

```
i <- 1
while (i<=10) i <- i+4
i
```

```
## [1] 13
```

- Ejemplo 2:

```
i <- 1
while (TRUE){ # Loop similar al anterior
  i <- i+4
  if(i>10) break
}
i
```

```
## [1] 13
```

If - Else

- Empecemos con un ejemplo simple:

```
r <- 5
if(r==4){
  x <-1
}else{
  x <- 3
  y <- 4
}
```

Funciones

Las funciones son declaradas con `function(x,y,...)` seguido de llaves `{}`. Los valores dentro de la función son los parámetros o valores de entrada. Dentro de las llaves se ubican las operaciones a realizar con dichos parámetros.

Combinando lo aprendido

- Realicemos una función que cuenta el número de elementos impares en un vector:

```
oddcount <- function(x)
{
  k <- 0 # se asigna 0 a k
  for( n in x)
    {
    if(n%%2 ==1) k <- k+1
    }
  return(k)
}
```

- Probemos la función

```
x <- seq(1:3)
oddcount(x)
```

```
## [1] 2
```

Oasis: Algo de cálculo

- Derivada de $f(x) = e^{2x}$

```
D(expression(exp(x^2)),"x")
```

```
## exp(x^2) * (2 * x)
```

- Integral de $\int_0^1 x^2$

```
integrate(function(x) x^2,0,1)
```

```
## 0.3333333 with absolute error < 3.7e-15
```

- Chequear el paquete `ryacas` para mas cálculo simbólico

VAN

Su expresión es $VAN = \sum_{i=0}^{n} \frac{Vi}{(1+K)^i} - I_0$

```
VAN <- function(I0,n,K,V)
  {
  for (i in 1:n)
    {
    y[i] <- V/(1+K)^i
    }
  sum(y) - I0
  }
```

La Familia Apply

- Existen algunas funciones que nos facilitan la vida en lugar de usar *loops*:
  - lapply: Itera sobre una lista y evalúa una función en cada elemento.
  - sapply: Lo mismo que laaply pero trata de simplifica el resultado.
  - apply: Aplica una función sobre las dimensiones de un array.
  - tapply: Aplica una función sobre subconjuntos de un vector
  - mapply: Versión multivariada de lapply

lapply

- *lapply* siempre retorna una lista, independientemente de la clase del objeto de entrada

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.4674775
```

```
x <- list(a = 1:4, b = rnorm(10),
c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.1655865
##
## $c
## [1] 0.7557949
##
## $d
## [1] 5.016356
```

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.542622
##
## [[2]]
## [1] 0.7077897 0.3247444
##
## [[3]]
## [1] 0.7144785 0.2315559 0.8254674
##
## [[4]]
## [1] 0.7692807 0.5451750 0.4482910 0.5086720
```

sapply

- *sapply* tratará de simplificar el resultado de lapply de ser posible
- Si el resultado es una lista donde cada elemento es de longitud 1, entonces retorna un vector
- Si el resultado es una lista donde cada elemento es un vector de la misma longitud (>1), retorna una matriz.
- Si lo puede descifrar las cosas, retorna una lista

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.3795073
##
## $c
## [1] 0.9316183
##
## $d
## [1] 5.018747
```

```
sapply(x, mean)
```

```
##          a          b          c          d
##  2.5000000 -0.3795073  0.9316183  5.0187468
```

```
mean(x)
```

```
## Warning in mean.default(x): argument is not numeric or logical: returning
## NA
```

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

apply

- *apply* se use para evaluar una función sobre las dimensiones de un array
- Es más usado para evaluar una función sobre las filas o columnas de una matriz
- En general no es más rápido que un loop, pero cabe en una sola línea (:

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)
```

```
##  [1]  0.259963497  0.031126171 -0.002406084 -0.005862196 -0.077456612
##  [6] -0.497980714 -0.142377414  0.647944974  0.596511948 -0.372475272
```

```
apply(x, 1, sum)
```

```
##  [1] -2.3728267  7.9651428 -1.3573880  1.4547592  2.7945101 -3.6334982
##  [7]  2.4950707  1.0856998 -2.6765449 -1.1502157 -0.1683383  0.5011588
## [13]  3.0756787 -0.2690655  1.2060850 -2.2089367 -1.4753120  3.4834508
## [19] -2.2124545  2.2027908
```

- Para sumas y medias de matrices tenemos algunos *shortcuts*:
  - `rowSums = apply(x, 1, sum)`
  - `rowMeans = apply(x, 1, mean)`
  - `colSums = apply(x, 2, sum)`
  - `colMeans = apply(x, 2, mean)`
- Las funciones cortas son más rápidas, pero no se nota menos que se use matrices grades.

tapply

- *tapply* Se usa para aplicar funciones sobre subconjuntos de un vector.
- Tomamos medias por grupo:

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```
tapply(x, f, mean)
```

```
##         1         2         3
## 0.3631614 0.5073675 1.0666630
```

- Para encontrar rangos por grupo:

```
tapply(x, f, range)
```

```
## $`1`
## [1] -1.200163  1.872138
```

```
##
## $`2`
## [1] 0.06448187 0.97431980
##
## $`3`
## [1] -0.3640164  1.9786321
```

Ejercicio

Al evaluar la relación estadística de dos variables, hay muchas alternativas a la medida de correlación estándar (correlación producto-momento de Pearson). Algunos pueden haber oído hablar de la correlación de rangos de Spearman, por ejemplo. Estas medidas alternativas tienen varias motivaciones, como la solidez de valores atípicos, que son elementos de datos extremos y posiblemente erróneos.

Aquí, propongamos una nueva medida de este tipo (en realidad se relaciona con uno de amplio uso, τ *de Kendall*), pero para ilustrar algunas de las técnicas de programación R introducidas hasta el momento, especialmente `ifelse ()`.

Ayuda: Considere los vectores `x` e `y`, que son series temporales, por ejemplo, para las mediciones de las acciones de dos empresas recogidas una vez por hora. Definiremos nuestra medida de asociación entre ellos como la *fracción del tiempo* `x` *y* `y` *que aumentan o disminuyen juntos*, es decir, la proporción de `i` para la cual `y[i + 1] -y [i]` tiene el mismo signo que `x[i + 1] -x [i]`.

```
x <- c(5,12,13,3,6,0,1,15,16,8,88)
y <- c(4,2,3,23,6,10,11,12,6,3,2)
udcorr(x,y)
```

```
## [1] 0.4
```

Aggregate y By

By

- Para ejecutar esta función, usaremos la base de datos *InsectSprays*

```
data(InsectSprays)
InsectSprays$x <- rnorm(length(InsectSprays$count))
by(InsectSprays,InsectSprays$spray,summary)
```

```
## InsectSprays$spray: A
##      count       spray        x
##  Min.   : 7.00   A:12   Min.   :-0.8067
##  1st Qu.:11.50   B: 0   1st Qu.:-0.3502
##  Median :14.00   C: 0   Median : 0.3144
##  Mean   :14.50   D: 0   Mean   : 0.1896
##  3rd Qu.:17.75   E: 0   3rd Qu.: 0.6532
##  Max.   :23.00   F: 0   Max.   : 1.0266
## ------------------------------------------------------------
## InsectSprays$spray: B
##      count       spray        x
##  Min.   : 7.00   A: 0   Min.   :-1.6958
##  1st Qu.:12.50   B:12   1st Qu.:-0.6216
##  Median :16.50   C: 0   Median :-0.3418
##  Mean   :15.33   D: 0   Mean   :-0.1208
```

```
##  3rd Qu.:17.50   E: 0   3rd Qu.: 0.3436
##  Max.   :21.00   F: 0   Max.   : 1.8609
## ------------------------------------------------------------
## InsectSprays$spray: C
##      count       spray        x
##  Min.   :0.000   A: 0   Min.   :-2.47235
##  1st Qu.:1.000   B: 0   1st Qu.:-0.67126
##  Median :1.500   C:12   Median :-0.22920
##  Mean   :2.083   D: 0   Mean   :-0.47263
##  3rd Qu.:3.000   E: 0   3rd Qu.:-0.05488
##  Max.   :7.000   F: 0   Max.   : 0.57480
## ------------------------------------------------------------
## InsectSprays$spray: D
##      count        spray        x
##  Min.   : 2.000   A: 0   Min.   :-2.05973
##  1st Qu.: 3.750   B: 0   1st Qu.:-0.86631
##  Median : 5.000   C: 0   Median :-0.70794
##  Mean   : 4.917   D:12   Mean   :-0.59432
##  3rd Qu.: 5.000   E: 0   3rd Qu.: 0.07158
##  Max.   :12.000   F: 0   Max.   : 0.68736
## ------------------------------------------------------------
## InsectSprays$spray: E
##      count      spray        x
##  Min.   :1.00   A: 0   Min.   :-1.63215
##  1st Qu.:2.75   B: 0   1st Qu.:-0.09586
##  Median :3.00   C: 0   Median : 0.53005
##  Mean   :3.50   D: 0   Mean   : 0.34624
##  3rd Qu.:5.00   E:12   3rd Qu.: 0.97744
##  Max.   :6.00   F: 0   Max.   : 1.62139
## ------------------------------------------------------------
## InsectSprays$spray: F
##      count       spray        x
##  Min.   : 9.00   A: 0   Min.   :-0.9722
##  1st Qu.:12.50   B: 0   1st Qu.:-0.2328
##  Median :15.00   C: 0   Median : 0.4000
##  Mean   :16.67   D: 0   Mean   : 0.2885
##  3rd Qu.:22.50   E: 0   3rd Qu.: 0.7122
##  Max.   :26.00   F:12   Max.   : 1.2955
```

Aggregate

```
aggregate(InsectSprays[,-2],
list(InsectSprays$spray),median)
```

```
##   Group.1 count          x
## 1       A  14.0  0.3143804
## 2       B  16.5 -0.3418336
## 3       C   1.5 -0.2291996
## 4       D   5.0 -0.7079370
## 5       E   3.0  0.5300547
## 6       F  15.0  0.3999762
```