Creando Selecciones de Datos

Luciano Selzer

28 June, 2018

Subconjuntos

R tiene muchas herramientas para crear subconjuntos, y dominarlos te va a permitir realizar tareas complejas de manera sencilla.

Hay seis formas distintas en la que podemos realizar subconjuntos de cualquier objeto y tres diferentes operadores de subconjuntos para las distintas estructuras de datos.

Empecemos con el caballo de batalla de R: los vectores atómicos.

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x
```

```
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

Ahora que hemos creado un vector de ejemplo para experimentar, como obtenemos sus contenidos.

Accediendo a los elementos por sus índices

Podemos extraer los elementos de un vector usando el indice correpondiente, empezando por 1:

```
x[1]
```

```
  a
5.4
```

```
x[4]
```

```
  d
4.8
```

El corchete es una función.
Para matrices y vectores, significa “dame el elemento n°”

Podemos pedir varios elementos a la vez:

```
x[c(1, 3)]
```

```
  a   c
5.4 7.1
```

O rebanadas de un vector:

```
x[1:4]
```

```
  a   b   c   d
5.4 6.2 7.1 4.8
```

¿Qué hace `1:4`?

El operador `:` crea una secuencia de número desde el número a la izquierda hasta el número a la derecha.

```
1:4
```

```
[1] 1 2 3 4
```

```
c(1, 2, 3, 4)
```

```
[1] 1 2 3 4
```

Podemos pedir el mismo elemento muchas veces:

```
x[c(1,1,3)]
```

```
  a   a   c
5.4 5.4 7.1
```

Si le pedimos un número por fuera del indice, R devuelve un valor perdido:

```
x[6]
```

```
<NA>
  NA
```

Si le pedimos el elemento 0, nos devuelve un vector vacío:

```
x[0]
```

```
named numeric(0)
```

La numeración de los vectores empieza en 1

> En muchos lenguajes de programación (C, Python) el primer elemento del vector tiene indice 0.
> En R, el primer elemento es 1.

Eliminando elementos

Si usamos un número negativo como indice del vector, R va a devolver todos los elementos **excepto** el especificado.

```
x[-2]
```

```
  a   c   d   e
5.4 7.1 4.8 7.5
```

Podemos eliminar varios elementos:

```
x[c(-1, -5)]  # o x[-c(1,5)]
```

```
  b   c   d
6.2 7.1 4.8
```

Para eliminar los elementos de un vector, tenemos que asignar los resultados de nuevo a la variable:

```
x <- x[-4]
x
```

```
  a   b   c   e
5.4 6.2 7.1 7.5
```

Ejercicio 1

Dado el siguiente código:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

1. Crea tres comandos diferentes que produzcan la siguiente salida:

```
  b   c   d
6.2 7.1 4.8
```

2. Compara con tu compañero ¿Usaron estrategias diferentes?

Subconjuntos por nombre

Podemos extrar los elementos usando su nombre, en vez de un índice:

```
x[c("a", "c")]
```

```
  a   c
5.4 7.1
```

En general es mucho más confiable, el número se puede referir a otro elemento pero el nombre es raro que cambie.

Pero es más complicado usarlo para eliminar elementos:

```
x[-which(names(x) == "a")]
```

```
  b   c   d   e
6.2 7.1 4.8 7.5
```

Para eliminar muchos elementos por nombre hay que usar otro operador

```
x[-which(names(x) %in% c("a", "c"))]
```

```
  b   d   e
6.2 4.8 7.5
```

Ejercicio 2

Ejecuta el siguiente código para definir el vector `x` como arriba:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

Dado el vector `x` ¿Qué esperas que haga el siguiente código?

```
x[-which(names(x) == "g")]
```

Ejercicio 2

Prueba el comando para ver que obtenés ¿Se comprobaron tus espectativas?

¿Por qué obtuviste este resultado? (Tip: prueba este comando por partes - está es una buena estrategia de debugging)

¿Cuál de las siguientes opciones son verdad?

A) Si no hay ningún valor `TRUE` pasado a `which`, devuelve un vector vacío.
B) Si no hay ningún valor `TRUE` pasado a `which`, muestra un mensaje de error. C)`integer()` es un vector vacío.
C) Al hacer un vector negativo vacío se produce un vector de "todo".
D) `x[]` da el mismo resultado que `x[integer()]`

Tip: Nombres no únicos

Es posible que multiples elementos de un vector pueden tener el mismo nombre. Para data.frames, las columnas pueden tener el mismo nombre pero los `row.names` deben ser únicos.

Tip: Nombres no únicos

Considera estos ejemplos:

```
x <- 1:3
x
```

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

```
names(x) <- c('a', 'a', 'a')
x
```

```
a a a
1 2 3
```

Tip: Nombres no únicos

```
x['a']  # only returns first value
```

```
a
1
```

```
x[which(names(x) == 'a')]  # returns all three values
```

```
a a a
1 2 3
```

Tip: Obteniendo ayudo sobre operadores

Recuerda que puedes buscar ayuda sobre operadores envolviendolos en comillas:
`help("%in%")` o `?"%in%"`

Entonces ¿Por qué no podemos usar el operador `==` como antes?

```
names(x) == c('a', 'c')
```

```
Warning in names(x) == c("a", "c"): longer object length is not a multiple
of shorter object length
```

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

`==` funciona de manera ligeramente diferente que `%in%`. Va a comparar cada elemento de su izquierda con el correspodiente a la derecha.

Aquí hay una ilustración:

```
c("a", "b", "c", "e")  # names of x
   |    |    |    |    # The elements == is comparing
c("a", "c")
```

Cuando uno de los argumentos es más corto que el otro, el más corto es *reciclado*

```
c("a", "b", "c", "e")  # names of x
   |    |    |    |    # The elements == is comparing
c("a", "c", "a", "c")
```

Si la longitud del vector más corto no es múltiplo del mayor entonces se produce un mensaje de aviso (*warning*)
¡La diferencia entre `==` e `%in%` es importante!

Subconjuntos mediantes otras operaciones lógicas

También se puede hacer otros subconjuntos usando vectores lógicos:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
```

```
a a
1 2
```

Las reglas de reciclaje aplican

```
x[c(TRUE, FALSE)]
```

```
a a
1 3
```

Dado que las operaciones de comparacion devuelven vectores lógicos se pueden usar.

```
x[x > 7]
```

```
named integer(0)
```

Combinando condiciones lógicas

Muchas veces es necesario combinar varias condiciones lógicas. Por ejemplo:
Países de Asia **o** Europa **y** que tengan expectativas de vida dentro de cierto rango.

- `&`, operador "Y lógico": devuelve `TRUE` si ambos son `TRUE`
- `|`, operador "O lógico": devuelve `TRUE` si al menos uno es `TRUE`

Las reglas de reciclado aplican para estos operadores

`&&` y `||` son similares pero solo operan en el primer elemento.

- `!`, operador "NO lógico": convierte `TRUE` en `FALSE` y viceversa

Adicionalmente:

- `all` devuelve `TRUE` cuando todos los valores son `TRUE`
- `any` devuelve `TRUE` si alguno de los valores es `TRUE`

Ejercicio 3

Dado el siguiente código:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

1. Escribe un comando que cree un subconjunto de todos los valores que sean mayores que 4 y menores que 7.

Manejando valores especiales

En algún momento vas a encontrar con funciones que no pueden manejar valor perdidos, infinitos o no definidos.

- `is.na` devuelve un vector lógico donde hay `NA`.
- `is.nan`, e `is.infinite` hacen lo mismo con valores `NaN` e `Inf`.
- `is.finite` devuelve todas las posiciones en un vector que no contienen `NA`, `NaN` o `Inf`.
- `na.omit` filtra todos los valores perdidos de un vector.

Subconjuntos de Factores

Funciona igual que con vectores:

```
f <- factor(c("a", "a", "b", "c", "c", "d"))
f[f == "a"]
```

```
[1] a a
Levels: a b c d
```

```
f[f %in% c("b", "c")]
```

```
[1] b c c
Levels: a b c d
```

```
f[1:3]
```

```
[1] a a b
Levels: a b c d
```

Pero no elimina los niveles:

```
f[-3]
```

```
[1] a a c c d
Levels: a b c d
```

Subconjuntos de matrices

Con la matrices también se usa la función `[`. Pero se necesitan dos argumentos, el primero es la fila y el segundo la columna:

```
set.seed(1)
m <- matrix(rnorm( 6 * 4), ncol = 4, nrow = 6)
m[3:4, c(3, 1)]
```

```
           [,1]       [,2]
[1,]  1.12493092 -0.8356286
[2,] -0.04493361  1.5952808
```

Podemos dejar alguno de los argumentos en blanco para otener todas las filas o columnas respectivamente:

```
m[, c(3,4)]
```

```
           [,1]        [,2]
[1,] -0.62124058  0.82122120
[2,] -2.21469989  0.59390132
[3,]  1.12493092  0.91897737
[4,] -0.04493361  0.78213630
[5,] -0.01619026  0.07456498
[6,]  0.94383621 -1.98935170
```

Si solo accedemos a una fila o una columna, R la convierte automaticamente a un vector:

```
m[3,]
```

```
[1] -0.8356286  0.5757814  1.1249309  0.9189774
```

Para evitarlo hay que usar el argumento `drop = FALSE`

```
m[3, , drop = FALSE]
```

```
           [,1]      [,2]     [,3]      [,4]
[1,] -0.8356286 0.5757814 1.124931 0.9189774
```

A diferencia de los vectores, da un error si se trata de acceder a un elemento inexistente:

```
m[, c(3,6)]
```

```
Error in m[, c(3, 6)]: subscript out of bounds
```

Tip: Arreglos multidimensionales

Cuando trabajamos con arreglos multidimensionales, cada argumento de `[` corresponde a una dimensión. Por ejemplo, en un arreglo 3D, los primeros 3 argumentos corresponden a las filas, columnas, y profundidad.

Como las matrices son vectores, podemos hacer subconjuntos usando solo un argumento.

```
m[5]
```

```
[1] 0.3295078
```

Es confuso y poco útil en general. Pero es útil notar que las matrices están construidas usando un *formato columnar*.

```
matrix(1:6, nrow = 2, ncol = 3)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Si deseas poblar la matriz por filas, usa el argumento `byrow = TRUE`:

```
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

También es posibles usar nombres para realizar subconjuntos de matrices.

Ejercicio 4

Dado el siguiente código:

```
m <- matrix(1:18, nrow = 3, ncol = 6)
print(m)
```

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

1. ¿Cuál de los siguientes comandos va a extraer los valores 11 y 14?

A. `m[2,4,2,5]` B. `m[2:5]` C. `m[4:5,2]` D. `m[2,c(4,5)]`

Subconjuntos de Listas

Hay tres operadores para realizar subconjuntos de listas.
`[` que ya vimos y `[[` y `$`

`[` siempre devuelve una lista.

```
xlist <- list(a = "UNTDF", b = 1:10, data = head(iris))
xlist[1]
```

```
$a
[1] "UNTDF"
```

Se puede usar de igual forma que con vectores pero las comparaciones no funcionan.

```
xlist[1:2]
```

```
$a
[1] "UNTDF"

$b
 [1]  1  2  3  4  5  6  7  8  9 10
```

Para extraer elementos individuales de una lista hay que usar `[[`

```
xlist[[1]]
```

```
[1] "UNTDF"
```

El resultado es un vector no una lista

Pero no se puede extraer más de un elemento por vez.

```
xlist[[1:2]]
```

```
Error in xlist[[1:2]]: subscript out of bounds
```

Ni usarlo para eliminar elementos

```
xlist[[-1]]
```

```
Error in xlist[[-1]]: attempt to select more than one element in get1index <real>
```

Pero se pueden usar los nombres para hacer subconjuntos y extraer elementos:

```
xlist[["a"]]
```

```
[1] "UNTDF"
```

La función `$` es un atajo para extraer elementos por nombre.

```
xlist$data
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Hay que tener cuidado con la concordancia parcial y no da error si el elemento no existe.

Ejercicio 5

Dada la siguiente lista

```
xlist <- list(a = "UNTDF", b = 1:10, data = head(iris))
xlist[1]
```

```
$a
[1] "UNTDF"
```

Usa el conocimiento de subconjuntos de listas y vectores para extraer el número 2 de `xlist`.

Ejercicio 6

Dado el siguiente modelo lineal:

```
mod <- aov(pop ~ lifeExp, data=gapminder)
```

Extrae los grados de libertad residuales (*residual degrees of freedom*) (pista: `attributes()` te va a ayudar)

Data frames

Las data.frames son listas disfrazadas, por lo que las mismas reglas de las listas se aplican. Pero también son objectos bidimensioanales.

Con un argumento `[` se comporta como en una lista y devuelve una columna. El resultado es un data.frame.

```
head(gapminder[3], 5)
```

```
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
```

De igual manera, `[[` va a actuar para extraer una *columna individual*:

```
head(gapminder[["lifeExp"]])
```

```
[1] 28.801 30.332 31.997 34.020 36.088 38.438
```

Y `$` provee una forma conveniente de extraer una columna por nombre:

```
head(gapminder$year)
```

```
[1] 1952 1957 1962 1967 1972 1977
```

Y con dos argumentos, `[` se comporta como si fuese una matriz.

```
gapminder[1:3,]
```

```
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
```

Si solo elegimos una fila el resultado es un data.frame:

Pero para una columna devuelve una vector a menos que especifiquemos `drop = FALSE`

Ejercicio 7

Arregla cada uno de los siguientes errores comunes de subconjuntos de data.frames.

1. Extrae todas las observaciones del año 1957

```
gapminder[gapminder$year = 1957,]
```

2. Extrae todas las columnas excepto desde la 1 a la 4

```
gapminder[,-1:4]
```

Ejercicio 7

3. Extrae todas las filas donde la expectativa de vida es mayor de 80 años.

```
gapminder[gapminder$lifeExp 80]
```

4. Extrae la primer fila, y la cuarta y quinta columna (`lifeExp` y `gdpPercap`).

```
gapminder[1, 4, 5]
```

5. Avanzado: extrae las filas que contienen informacion de los años 2002 y 2007.

```
gapminder[gapminder$year == 2002 | 2007,]
```

Ejercicio 8

1. ¿Por qué `gapminder[1:20]` devuelve un error? ¿Cómo difiere de `gapminder[1:20, ]`?
2. Crea un nuevo `data.frame` llamado `gapminder_small` que solo contenga las filas 1 a 9 y 19 a 23. Puedes hacerlo en uno o dos pasos.