Vectorización, la familia apply y otros

Subconjuntos

Esta sección está basada en **wickham2014advanced** disponible en línea.

Aprender a extraer subconjuntos de los datos es importante y permite realizar operaciones complejas con los mismos. De los conceptos importantes que se deben aprender son

- Los operadores para extraer subconjuntos (subsetting operators)
- Los 6 tipos de extracciones de subconjuntos
- Las diferencias a la hora de extraer subconjuntos de las diferentes estructuras de datos (factores, listas, matrices, dataframes)
- El uso de la extracción de subconjuntos junto a asignar variables.

Cuando tenemos que extraer pedazos de los datos (o analizar solamente parte de éstos), necesitamos complementar `str()` con `[[`, es decir, la estructura nos dirá cómo utilizar el operador subconjunto de manera que de hecho extraigamos lo que queremos.

Operadores para extraer subconjuntos

Dependiendo la estructura de datos que tenemos, será la forma en la que extraemos elementos de ella. Hay dos operadores de subconjunto: `[[` y `$`. `[[` se parece a `[` pero regresa un solo valor y te permite sacar pedazos de una lista. `$` es un atajo útil para `[[`.

Vectores atómicos

¿De qué formas puedo extraer elementos de un vector? Hay varias maneras **sin importar** la *clase* del vector.

- **Enteros positivos** regresan los elementos en las posiciones especificadas en el orden que especificamos.

```
x <- c(5.6, 7.8, 4.5, 3.3)

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

```
## [1] 4.5 5.6
```

```
## Si duplicamos posiciones, nos regresa resultados duplicados
x[c(1, 1, 1)]
```

```
## [1] 5.6 5.6 5.6
```

```
## Si usamos valores reales, se coerciona a entero
x[c(1.1, 2.4)]
```

```
## [1] 5.6 7.8
```

```r
x[order(x)]
```

```
## [1] 3.3 4.5 5.6 7.8
```

```r
x[order(x, decreasing = T)]
```

```
## [1] 7.8 5.6 4.5 3.3
```

- **Enteros negativos** omiten los valores en las posiciones que se especifican.

```r
x
```

```
## [1] 5.6 7.8 4.5 3.3
```

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

```
## [1] 7.8 3.3
```

Mezclar no funciona.

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

- **Vectores lógicos** selecciona los elementos cuyo valor correspondiente es `TRUE`. Esta es una de los tipos más útiles.

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

```
## [1] 5.6 7.8
```

```r
x[c(TRUE, FALSE)] # Autocompleta el vector lógico al tamaño de x
```

```
## [1] 5.6 4.5
```

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

```
## [1] 5.6 7.8  NA
```

- **Nada** si no especifico nada, me regresa el vector original

```r
x[]
```

```
## [1] 5.6 7.8 4.5 3.3
```

- **Cero** el índice cero no aplica en R, te regresa el vector vacio

```r
x[0]
```

```
## numeric(0)
```

- Si el vector tiene **nombres** también los puedo usar.

```r
names(x) <- c("a", "ab", "b", "c")
x["ab"]
```

```
##  ab
## 7.8
```

```r
x["d"]
```

```
## <NA>
##   NA
```

```r
x[grep("a", names(x))]
```

```
##   a  ab
## 5.6 7.8
```

Las **listas** operan básicamente igual a vectores recordando que si usamos `[` regresa una lista y tanto `[[` y `$` extrae componentes de la lista.

Matrices y arreglos

Para estructuras de mayor dimensión se pueden extraer de tres maneras:

- Con vectores múltiples
- Con un solo vector
- Con una matriz

```r
m <- matrix(1:12, nrow = 3)
colnames(m) <- LETTERS[1:4]
m[1:2, ]
```

```
##      A B C  D
## [1,] 1 4 7 10
## [2,] 2 5 8 11
```

```r
m[c(T, F, F), c("B", "C")]
```

```
## B C
## 4 7
```

```r
m[1, 4]
```

```
##  D
## 10
```

Como ven, es solamente generalizar lo que se hace en vectores replicándolo al número de dimensiones que se tiene.

```r
m[c(T, F, F)]
```

```
## [1]  1  4  7 10
```

```r
class(m[c(T, F, F)])
```

```
## [1] "integer"
```

[simplifica al objeto. En matriz, me quita la dimensionalidad, en listas me da lo que esta dentro de esa celda.

Dataframes

```r
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])

df[c(1, 2), ]
```

```
##   x y z
## 1 1 3 a
## 2 2 2 b
```

```r
df[, c(1, 2)]
```

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1
```

```r
df[, c("z", "x")]
```

```
##   z x
## 1 a 1
## 2 b 2
## 3 c 3
```

```r
df[c("z", "x")]
```

```
##   z x
## 1 a 1
## 2 b 2
## 3 c 3
```

```r
class(df[, c("z", "x")])
```

```
## [1] "data.frame"
```

```r
class(df[c("z", "x")])
```

```
## [1] "data.frame"
```

```r
str(df["x"])
```

```
## 'data.frame':    3 obs. of  1 variable:
##  $ x: int  1 2 3
```

```r
str(df[, "x"])
```

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

```r
str(df$x)
```

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

Ejercicios

1. Utiliza la base mtcars
2. Arregla los errores al extraer subconjuntos en dataframes

```r
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

3. ¿Por qué al correr *x <- 1:5; x[NA]* obtengo valores perdidos?
4. Genera una matriz cuadrada tamaño 5 llamada m. ¿Qué te da correr m[upper.tri(m)]?
5. ¿Por qué al realizar mtcars[1:20] me da un error? ¿Por qué mtcars[1:2] no me lo da? ¿Por qué mtcars[1:20,] es distinto?
6. Haz una función que extraiga la diagonal de la matriz m que creaste antes. Debe dar el mismo resultado que ejecutar diag(m)
7. ¿Qué hace df[is.na(df)] <- 0?

Operadores lógicos

Ejemplo: Supongamos que queremos saber qué elementos de x son menores que 5 y mayores que 8.

Operador	Descripción
<	menor que
<=	menor o igual que
>	mayor que
==	exactamente igual que
!=	diferente de
!x	no x
x | y	x O y
x & y	x Y y
isTRUE(x)	checa si x es verdadero

```
x <- c(1:10)
x[(x>8) | (x<5)]
```

```
## [1]  1  2  3  4  9 10
```

```
# ¿Cómo funciona?
x
```

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

```
x > 8
```

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

```
x < 5
```

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

```
x > 8 | x < 5
```

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

```
# x > 8 || x < 5
x[c(T,T,T,T,F,F,F,F,T,T)]
```

```
## [1]  1  2  3  4  9 10
```

Ejercicio

- Todo lo que existe es un objeto.
- Todo lo que sucede es una llamada a una función.

Vectorización