Estructura de datos en R

Martin Malo

30/6/2021

```
rep(1997,10)
```

```
##  [1] 1997 1997 1997 1997 1997 1997 1997 1997 1997 1997
```

```
vec <- c(16,0,1,20,1,7,88,5,1,9)
fix(vec)
```

Para saber si dos numeros son divisibles entre ellos se debe evaluar el resto donde este debe ser igual a 0

```
13251 %% 7
```

```
## [1] 0
```

Progresiones y secuencias

```
seq(4, 35, length.out = 7)
```

```
## [1]  4.000000  9.166667 14.333333 19.500000 24.666667 29.833333 35.000000
```

Calcula 7 numeros que puedan satisfacer una secuencia del 4 al 35

```
seq(4, length.out = 7, by = 3)
```

```
## [1]  4  7 10 13 16 19 22
```

Muestra 7 valores que comiencen en 4 aumenten de 3 en 3

```
1:20
```

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

```
seq(2, length.out = 20, by = 2)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

```
print(seq(17, 98, length.out = 30), 4)
```

```
##  [1] 17.00 19.79 22.59 25.38 28.17 30.97 33.76 36.55 39.34 42.14 44.93 47.72
## [13] 50.52 53.31 56.10 58.90 61.69 64.48 67.28 70.07 72.86 75.66 78.45 81.24
## [25] 84.03 86.83 89.62 92.41 95.21 98.00
```

```
x <- c(rep(pi, 5), 5:10, seq(1, 5, length.out = 8))
print(c(1, 33, x, seq(1, length.out = 14, by = 30)), 4)
```

```
##  [1]   1.000  33.000   3.142   3.142   3.142   3.142   3.142   5.000   6.000
## [10]   7.000   8.000   9.000  10.000   1.000   1.571   2.143   2.714   3.286
## [19]   3.857   4.429   5.000   1.000  31.000  61.000  91.000 121.000 151.000
## [28] 181.000 211.000 241.000 271.000 301.000 331.000 361.000 391.000
```

Funciones y orden de vectores

```
x <- 1:10
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

```
sapply(x, FUN = function(elemento){sqrt(elemento)})
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

sapply ejecuta funciones propias a vectores. En este ejemplo sapply esta haciendo lo mismo que sqrt pero es util cuando R no tiene cargadas funciones que apliquen a vectores automaticamente o para aplicar funciones propias

```
cuadrado <- function(x){x^2}
v <- c(1:6)
sapply(v, FUN = cuadrado)

mean(v)
prod(v)
cumsum(v)
cummax(v)
cummin(v)
cumprod(v)

sort(v)
rev(v)

x <- c(4,3,2,6,5,7,8,9,1)
rev(sort(x))
sort(v, decreasing = TRUE)
```

Si yo quisiera elevar al cuadrado el vector v, no podria realizar $cuadrado^v$. Pero si aplico sapply si se me es posible indicando que vector voy a usar y cual es la funcion.

`prod()` va a multiplicar todos los valores del vector `cumsum()` lo que hace es calcular un nuevo vector de un vector dado su acumulado

`sort()` ordena el vector por default de orden ascendente numerico y alfabetico si son palabras

`rev()` invierte el orden del vector dado

`rev(sort())` va a causar que se cree un orden descendente

```
seq(1, 20, length.out = 10)
```

```
##  [1]  1.000000  3.111111  5.222222  7.333333  9.444444 11.555556 13.666667
##  [8] 15.777778 17.888889 20.000000
```

```
diff(seq(1, 20, length.out = 33))
```

```
##  [1] 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375
## [10] 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375
## [19] 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375 0.59375
## [28] 0.59375 0.59375 0.59375 0.59375 0.59375
```

`diff()` crea un vector de las distancias de valor a valor de un vector o secuencias o pregresion dada

Subvectores

Los subvectores son una clasificacion especificada a la que queremos llegar o ver de un vector dado. Para acceder a ellos se utilizan los [ ].

Vamos a especificar algunas de las siguientes sintaxis:

```
x <- seq(3, 50, by = 3.5)
x[3]
x[8]
x[length(x)]
x[length(x)-1]
x[length(x)-2]
x[-3]
x[4:8]
x[8:4]
x[seq(2, length(x), by = 2)]
x[seq(1, length(x), by = 2)]
x[-seq(2, length(x), by = 2)]
x[(length(x)-3):length(x)]
x[c(1,4,6)]
x[x > 30]
x[x > 20 & x < 40]
x[x != 3 & x != 17]
x[x < 10 | x > 40]
x[x %% 2 == 0]
x[x %% 2 == 1]
```

```
x <- c(4,2,5,6,8,9,7,9,0,2,3,0)
y <- c(1,5,-3,6,-8,-7,-9,3,-5,0)
x[y > 0]

which(x > 4)
x[which(x > 4)]
which(x > 2 & x < 8)
which(x < 5 | x %% 2 == 0)
x[which(x < 5 | x %% 2 == 0)]
which(x %% 2 == 0)
which.min(x)
which(x == min(x))
which.max(x)
which(x == max(x))
```

`x[3]` va a mostrar solo el tercer valor del vector.

`x[-3]` nos va a mostrar todo el vector menos el que haya estado en la 3ra posicion.

`x[4:8]` no va a mostrar los vectores desde la posicion 4 hasta la 8.

`x[8:4]` mostrará los vectores desde la poscion 4 hasta la 8 invertido.

`x[length(x)]` va a mostrar el ultimo valor del vector.

`x[length(x)-1]` va a mostrar el penultimo

`x[length(x)-2]` va a mostrar el antepenultimo valor y asi sucesivamente.

`x[seq(2, length(x), by = 2)]` nos va a mostrar los valores que esten en una posicion par del vector.

`x[seq(1, length(x), by = 2)]` nos va a mostrar los valores que esten en una posicion impar del vector.

`x[-seq(2, length(x), by = 2)]` va a ocultar los valores que esten en posicion par del vector

---

`x[(length(x)-3):length(x)]` va a mostrar los valores que esten desde la posicion ultima -3 hasta la ultima posicion

`x[c(1,4,6)]` nos va a mostrar solamente los valores que esten en las posiciones indicadas

---

`x[x > 20 & x < 40]` va a mostrar los valores que sean mayores a 20 y menores a 40

`x[x != 3 & x != 17]` va a mostrar todos los valores menos los que sean iguales a 3 y 17

`x[x < 10 | x > 40]` va a mostrar los valores menores a 10 y mayores a 40

`x[x %% 2 == 0]` va a mostrar solamente los valores par. Lo hace mediante el resto donde si yo divido un numero par para 2, el resto siempre deberá ser 0

`x[x %% 2 == 1]` va a mostrar solo los valores impar. Si yo divido un numero impar par 2, este siempre tendrá un resto igual a 1

La funcion `which` devuelven la posicion del valor que cumpla cierta condicion. Recordemos que los [ ] devuelven el valor y no la posicion

`which(x > 4)` nos va a mostrar las posiciones donde existen valores mayores a 4

`x[which(x > 4)]` devuelve los valores que cumplen la condicion which

`which(x > 2 & x < 8)` nos muestra las posiciones de los valores que se menores a 2 y mayores a 8

`which(x %% 2 == 0)` muestra las posiciones de los valores par

`which.min(x)` muestra la posicion del valor minimo de todo el vector

`which(x == min(x))` muestra todas las posiciones donde se encuentran los valores minimos

`which.max(x)` muestra la posicion del valor maximo de todo el vector

`which(x == max(x))` muestra todas las posiciones donde se encuentran los valores maximos

```
x[x > 0 & x <2]
```

Cuando no existe un valor en el vector que cumpla una condicion, R devuelve numeric(0) indicando

```
which(x > 0 & x < 2)
```

Asi mismo, si no existen valores para cumplir una condicion, no habran posiciones, por lo que al usar la funcion `which()` R devolverá integer(0).

Los NAs

```
x[3] <- 99
x[3:6] <- x[3:6] + 10
x[(length(x) - 3):length(x)] <- 0
x[(length(x) + 9)] <- 9
sum(x)
sum(x, na.rm = TRUE)
mean(x)
mean(x, na.rm = T)
which(x == NA)
is.na(x)
which(is.na(x))
x[which(is.na(x))]

y <- x
y[is.na(y)]
y[is.na(y)] <- mean(y, na.rm = T)

cumsum(y, na.rm = T)
cumsum(y[!is.na(y)])

x_clean <- na.omit(x)
attr(x_clean, "na.action") <- NULL
```

`x[3] <- 99` La posicion numero 3 del vector fue editada y ahora vale 99

`x[3:6] <- x[3:6] + 10` Los valores en las posiciones del 3 al 6 fueron reemplazadas por su valor sumado en 10

`x[(length(x) - 3):length(x)] <- 0` El ultimo valor mas otros tres fueron reemplazados por 0

`x[(length(x) + 9)] <- 9` Desde la ultima posicion añade 8 nuevas posiciones y en la novena pon el 9. Aqui en esas 8 posiciones adicionales que no tienen valores, R pone NAs. Si un vector tiene por lo menos un NA, ya no se pueden realizar operaciones

`sum(x, na.rm = TRUE)` Con na.rm le decimos a la funcion que no tome en cuenta los NA por lo que es capaz de realizar la suma

Si queremos usar which para identificar a los NAs no podemos, ya que NA no es un valor, es como un concepto que which no es capaz de calcular.

`which(is.na(x))` Con is.na podemos identificar a los NAs del vector por lo que podemos combinar which(is.na()) para identificar las posiciones de los NAs

`y <- x, y[is.na(y)], y[is.na(y)] <- mean(y, na.rm = T)` Por lo general a los NAs se los reemplaza por un valor, que en la estadistica suele hacerse por la media

La funcion cumsum no admite el na.rm por lo que se utiliza asi: `cumsum(y[!is.na(y)])`

Con `na.omit()` podemos eliminar los NAs pero no es recomendable

Cuando usamos `na.omit()` en el resultado nos quedan unos atributos y para eliminarlos le indicamos a R que los convierta en NULL a esos atributos de la siguiente manera: `attr(x_clean, "na.action") <- NULL`