Estructura de datos en R

Daniel Eduardo Macias Estrada

4/1/2021

Estructura de datos

Un vector es una secuencia ordenada de datos, `R` dispone de muchos tipos de datos, por ejemplo

- `logical`: lógicos (`True` o `FALSE`)
- `integer`: números enteros, $\mathbb{Z}$
- `numeric`: números reales, $\mathbb{R}$
- `complex`: números complejos, $\mathbb{C}$
- `character`: palabras

En los vectores de R, todos sus objetos han de ser del mismo tipo: todos números, todos palabras, etc. Cuando queramos usar vectores formados por objetos de diferentes tipos, tendremos que usar listas generalizadas, `lists`

Básico

- `c()`: para definir un vector
- `scan()`: para definir un vector
- `fix(x)`: para modificar visualmente el vector x
- `rep(a,n)`: para definir un vector constante que contiene el dato a repetido n veces

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

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

```
rep("Mates", 5)
```

```
## [1] "Mates" "Mates" "Mates" "Mates" "Mates"
```

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

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

Ejemplo

Vamos a crear un vector que contenga 3 copias de 1 9 9 8 0 7 2 6 con la función scan:

```
> scan()
1: 1 9 9 8 0 7 2 6
9: 1 9 9 8 0 7 2 6
17: 1 9 9 8 0 7 2 6
25:
Read 24 items
 [1] 1 9 9 8 0 7 2 6 1 9 9 8 0 7 2 6 1 9 9 8 0 7 2 6
```

Ejercicio

1. Repite tu año de nacimiento 10 veces
2. Crea el vector que tenga como entradas 16, 0, 1, 20, 1, 7, 88, 5, 1, 9, llámalo vec y modifica la cuarta entrada con la función fix()

1.-

```
rep("02/06/02",10)
```

```
##  [1] "02/06/02" "02/06/02" "02/06/02" "02/06/02" "02/06/02" "02/06/02"
##  [7] "02/06/02" "02/06/02" "02/06/02" "02/06/02"
```

2.-

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

Progresiones y Secuencias

Una progresión aritmética es una sucesión de números tales que la diferencia, d, de cualquier par de términos sucesivos de la secuencia es constante.

$$a_n = a_1 + (n-1) \cdot d$$

- `seq(a,b,by=d)`: para generar una progresión aritmética de diferencia d que empieza en a hasta llegar a b
- `seq(a,b, length.out=n)`: define progresión aritmética de longitud n que va de a a b con diferencia d. Por tanto $d = (b-a)/(n-1)$
- `seq(a,by=d, length.out=n)`: define la progresión aritmética de longitud n y diferencia d que empieza en a
- `a:b`: define la secuencia de números enteros ($\mathbb{Z}$) consecutivos entre dos números a y b

Ejemplo

```
# Progresión creciente
seq(5,60, by=3.5)
```

```
##  [1]  5.0  8.5 12.0 15.5 19.0 22.5 26.0 29.5 33.0 36.5 40.0 43.5 47.0 50.5 54.0
## [16] 57.5
```

```
# Progresión decreciente
seq(20, -4, by = -2)
```

```
##  [1] 20 18 16 14 12 10  8  6  4  2  0 -2 -4
```

```
seq(4,25, length.out = 8)
```

```
## [1]  4  7 10 13 16 19 22 25
```

```
seq(4,length.out = 6, by = 6)
```

```
## [1]  4 10 16 22 28 34
```

```
1:10
```

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

```
23:-10
```

```
##  [1]  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5
## [20]   4   3   2   1   0  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

```
-1:3
```

```
## [1] -1  0  1  2  3
```

```
-(2:5)
```

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

Ejercicio

- Imprime los números del 1 al 20
- Imprime los 20 primeros números pares
- Imprime 30 números equidistantes entre el 17 y el 98, mostrando solo 4 cifras significativas

```
#1.-
1:20
```

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

```r
#2.-
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
```

```r
#3.-
round(seq(17,98, length.out = 30), digits = 4)
```

```
##  [1] 17.0000 19.7931 22.5862 25.3793 28.1724 30.9655 33.7586 36.5517 39.3448
## [10] 42.1379 44.9310 47.7241 50.5172 53.3103 56.1034 58.8966 61.6897 64.4828
## [19] 67.2759 70.0690 72.8621 75.6552 78.4483 81.2414 84.0345 86.8276 89.6207
## [28] 92.4138 95.2069 98.0000
```

Funciones

Cuando queremos aplicar una función a cada uno de los elementos de un vector de datos, la función `sapply` nos ahorra tener que programar con bucles en `R`:

- `sapply(nombre_de_vector, FUN=nombre_de_funcion)`: para aplicar dicha función a todos los elementos del vector
- `sqrt(x)`: calcula un nuevo vector con las raíces cuadradas de cada uno de los elementos del vector x

Existen algunas operaciones que se pueden realizar a un vector

```r
x = 1:10
x + pi
```

```
##  [1]  4.141593  5.141593  6.141593  7.141593  8.141593  9.141593 10.141593
##  [8] 11.141593 12.141593 13.141593
```

```r
pi*x
```

```
##  [1]  3.141593  6.283185  9.424778 12.566371 15.707963 18.849556 21.991149
##  [8] 25.132741 28.274334 31.415927
```

```r
sqrt(x)
```

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

```r
2^x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```r
x^2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Para hacer uso de la función *sapply*

```r
sapply(x, 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
```

```r
cd = function(x){x^pi}
sapply(x, FUN = cd)
```

```
##  [1]    1.000000    8.824978   31.544281   77.880234  156.992545  278.377578
##  [7]  451.807873  687.291335  995.041645 1385.455731
```

Otros ejemplos de operaciones de vectores

Suma de vectores

```r
1:10 + 1:10
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```r
(1:10) * (1:10)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```r
(1:10)^(1:10)
```

```
##  [1]           1           4          27         256        3125       46656
##  [7]      823543    16777216   387420489 10000000000
```

Funciones para vectores

Dado un vector de datos x podemos calcular muchas medidas estadísticas acerca del mismo:

- **lenght()**: calcula la longitud del vector x
- **max(x)**: calcula el máximo del vector x
- **min(x)**: calcula el mínimo del vector x
- **sum(x)**: calcula la suma de las entradas del vector x
- **prod(x)**: calcula el producto de las entradas del vector x
- **mean(x)**: calcula la media aritmética de las entradas del vector x
- **diff(x)**: calcula el vector formado por las diferencias sucesivas entre entradas del vector original x
- **cumsum(x)**: calcula el vector formado por las sumas acumuladas de las entradas del vector original x
  - Permite definir sucesiones descritas mediante sumatorios
  - Cada entrada de **cumsum(x)** es la suma de las entradas de x hasta su posición

```r
x = c(1,2,3,4,5,6)
mean(x)
```

```
## [1] 3.5
```

```r
cumsum(x)
```

```
## [1]  1  3  6 10 15 21
```

Orden

- **sort(x)**: ordena el vector en orden natural de los objetos que lo forman: orden numérico creciente, orden alfabético...
- **rev(x)**: invierte el orden de los elementos del vector x

```r
v = c(1,7,5,2,4,6,3)
sort(v)
```

```
## [1] 1 2 3 4 5 6 7
```

```r
rev(v)
```

```
## [1] 3 6 4 2 5 7 1
```

Ejercicio

- Combinar dos funciones anteriores **sort** y **rev** para crear una función que dado un vector x nos lo devuelva ordenado en orden descendiente
- Razonar si aplicar primero **sort** y luego **rev** a un vector x daría en general el mismo resultado que aplicar primero **rev** y luego **sort**
- Investigar la documentación de la función **sort** para leer si cambiando algún argumento de la misma se puede obtener el mismo resultado que haber programado en el primer ejercicio

```r
# 1.-
newSort = function(x){rev(sort(x))}
y = c(1,93,4,-6,29,3,-12)
newSort(y)
```

```
## [1]  93  29   4   3   1  -6 -12
```

```r
# 2.-
rev(sort(y))
```

```
## [1]  93  29   4   3   1  -6 -12
```

```r
sort(rev(y))
```

```
## [1] -12  -6   1   3   4  29  93
```

```r
#3.-
#Se puede hacer el mismo efecto del primer ejercicio, haciendo uso del argumento decreasing = TRUE
```

Subvectores

- **vector[i]**: de la i-ésima entrada del vector
- **vector[length(vector)]**: nos da la última entrada del vector
- **vector[a:b]**: si a y b son dos números naturales, nos da el subvector con las entradas del vector original tal que van de la posición a-ésima hasta la b-ésima
- **vector[-i]**: si i es un número, este subvector está formado por todas las entradas del vector original menos la entrada i-ésima. Si i resulta ser un vector, entonces es un vector de índices y crea un nuevo vector con las entradas del vector original cuyos índices pertenecen a i
- **vector[-x]**: si x es un vector (de índices), entonces este es el complementario de vector$[x]$

También podemos utilizar operadores lógicos

- ==: $=$
- !=: $\neq$
- >=: $\geq$
- <=: $\leq$
- <: $<$
- >: $>$
- !: NO lógico
- &: Y lógico
- |: O lógico

Ejemplo

```r
v = c(12,3,5,25,6,3,46)
v[2]
```

```
## [1] 3
```

```r
v[-c(3,5)]
```

```
## [1] 12  3 25  3 46
```

```r
v[v != 19 & v>15]
```

```
## [1] 25 46
```

```r
#obtener números pares
v[v%%2 == 0]
```

```
## [1] 12  6 46
```

```r
x = seq(3,24, by=2.4)
x
```

```
## [1]  3.0  5.4  7.8 10.2 12.6 15.0 17.4 19.8 22.2
```

```r
#penúltimo elemento
x[length(x)-1]
```

```
## [1] 19.8
```

```r
#subvector de x
x[3:7]
```

```
## [1]  7.8 10.2 12.6 15.0 17.4
```

```r
#subvector de x de reversa
x[7:3]
```

```
## [1] 17.4 15.0 12.6 10.2  7.8
```

```r
# obtener subvector de posiciones par
x[seq(2,length(x), by=2)]
```

```
## [1]  5.4 10.2 15.0 19.8
```

Condicionales

- **which(x cumple condicion)**: para obtener los índices de las entradas del vector x que satisfacen la condición dada.
- **which.min(x)**: nos da la primera posición en la que el vector x toma su valor mínimo
- **which(x==min(x))**: da todas las posiciones en las que el vector x toma sus valores mínimos
- **which.max(x)**: nos da la primera posición en la que el vector x toma su valor máximo
- **which(x==max(x))**: da todas las posiciones en las que el vector x toma sus valores máximos

```r
z = c(32,2,6,12,8,7,7,1,9)
z[z>5]
```

```
## [1] 32  6 12  8  7  7  9
```

```r
z[which(z>5)]
```

```
## [1] 32  6 12  8  7  7  9
```

```r
which(z > 2 & z < 9)
```

```
## [1] 3 5 6 7
```

```r
z[which(z>2 & z<6)]
```

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

Los valores NA

Significa "Not available"

Se puede dar de la siguiente manera

```r
z = c(32,2,6,12,8,7,7,1,9)
z[length(z)+5] = 4
z
```

```
##  [1] 32  2  6 12  8  7  7  1  9 NA NA NA NA  4
```

Si se hacen las funciones conocidas nos da como resultado NA

```r
sum(z)
```

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

```r
prod(z)
```

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

Sin embargo, estas funciones cuentan con el argumento **na.rm = TRUE**, el cual elimina los NA de un vector

```r
sum(z, na.rm = TRUE)
```

```
## [1] 88
```

```r
prod(z, na.rm = TRUE)
```

```
## [1] 65028096
```

Para conocer que elemenots son NA o no se puede hacer uso de la función **is.na()**

```r
is.na(z)
```

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

Para poder reemplazar dichos valores dentro del vector, se usa la media aritmética obtenida

```r
z[is.na(z)] = mean(z, na.rm = TRUE)
z
```

```
##  [1] 32.0  2.0  6.0 12.0  8.0  7.0  7.0  1.0  9.0  8.8  8.8  8.8  8.8  4.0
```

Para poder hacer uso de funciones como **cumsum()**, en donde el argumento **na.rm** no existe, podemos hacer lo siguiente

```r
z = c(32,2,6,12,8,7,7,1,9)
z[length(z)+5] = 4
cumsum(z[!is.na(z)])
```

```
##  [1] 32 34 40 52 60 67 74 75 84 88
```