

Estructuras de datos en R

Carlos C.

2022-08-02

Vectores y tipos de datos en R

Un **vector** es una secuencia ordenada de datos. R dispone de muchos tipos de datos, por ejemplo:

- logical: valores 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, todas las entradas deben ser del mismo tipo: todas números, todas palabras, etc. Cuando queramos usar vectores formados por objetos de diferentes tipos de datos tendremos que usar *listas generalizadas* (lists).

R va a definir la *class* del vector de acuerdo a la siguiente jerarquía:

character complex numeric integer logical

(vgr) Supongamos que tenemos el siguiente vector que tiene diferentes tipos de datos

```
vector<-c(27, TRUE, 3.5, "Carlos")
class(vector)
```

```
[1] "character"
```

Básico

Para crear/manipular vectores podemos hacer uso de las siguientes líneas:

- **c()**: para definir un vector. Significa concatenar.

```
# (vgr)
c(1,2,3,4,5)
```

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

Es posible concatenar varios vectores con la misma sentencia.

```
# (vgr)
x<-c(1,3,5,7,9)
y<-c(0,2,4,6,8)

vector<-c(x,y)
vector
```

```
[1] 1 3 5 7 9 0 2 4 6 8
```

- **scan()**: para definir un vector, entrada por entrada, en la consola. Se introducen las entradas del vector separándolas por un espacio y para terminar se da un doble “Enter”. Con la función *scan()* podemos definir parámetros como *sep*=“ ” y *dec*=“.” para especificar si nuestros datos están separados por algún otro valor que no sea un espacio o si los decimales de los datos que estamos leyendo vienen representados por una coma. Esta función nos permite leer datos de páginas web o archivos locales.
- **fix(x)**: para modificar visualmente el vector *x* desde una ventana en RStudio. Se modifica de forma “manual” las entradas del vector.
- **rep(a,n)**: para definir un vector constante que contiene el dato *a* repetido *n* veces. Es una función muy útil para hacer simulaciones.

```
# (vgr)
rep("Act",7)
```

```
[1] "Act" "Act" "Act" "Act" "Act" "Act" "Act"
```

Progresiones y Secuencias

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

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

- **seq(a, b, by=d)**: para agregar una progresión aritmética de diferencia d que empieza en a hasta llegar a b . Es posible definir la $d < 0$ para indicar que es una progresión aritmética decreciente.

```
# (vgr)
seq(5,60,5)
```

```
[1] 5 10 15 20 25 30 35 40 45 50 55 60
```

```
# (vgr)
seq(5,60, 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
```

- **seq(a, b, length.out=n)**: define una progresión aritmética de longitud n que va de a a b con diferencia d . Por lo tanto, $d = (b - a)/(n - 1)$. En este caso estamos especificando que queremos n entradas en el vector.

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

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

- **seq(a, by=d, length.out=n)**: define la progresión aritmética de longitud n y diferencia d que empieza en a . Con esto logramos un vector con n entradas, la primera será a y cada entrada aumentará en a unidades.

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

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

- **a:b**: define la secuencia de números enteros consecutivos entre dos números a y b .

```
1:10
```

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

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_función)**: para aplicar dicha función a todos los elementos del vector.

```
x<-1:5
# Definimos la función dentro de sapply
sapply(x, FUN=function(elemento){elemento^2})
```

```
[1] 1 4 9 16 25
```

```
x<-1:5
# Definimos la función aparte
cuadrado<-function(elemento){elemento^2}

sapply(x,FUN=cuadrado)
```

```
[1] 1 4 9 16 25
```

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

- **length(x)**: 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 s
- **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

```
x<-c(1,3,7,12,20,30)
diff(x)
```

```
[1] 2 4 5 8 10
```

- **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 es la suma de las entradas de x hasta su posición.

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

```
[1] 1 3 6 10 15 21 28 36 45 55
```

Orden

- **sort(x)**: ordena el vector en orden natural de los objetos que lo forman: el orden numérico creciente, orden alfabético, etc. Podemos definir el parámetro *decrease=TRUE* para que los ordene de forma decreciente.

```
v<-c(1,7,5,2,4,6,3)
sort(v)
```

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

```
# De forma decreciente
sort(v, decreasing = TRUE)
```

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

- **rev(v)**: invierte el orden de los elementos del vector *v*

```
# v<-c(1,7,5,2,4,6,3)
rev(v)
```

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

NOTA: no es necesario volver a llamar al vector *v* para usarlo en otro chunk, se queda guardado en la memoria de R.

Ejercicio

Producto notable

La fórmula del producto notable es

$$(a + b)^2 = a^2 + 2ab + b^2$$

Función con R

```
binomioNewton2<-function(a,b){  
  a^2+2*a*b+b^2  
}  
binomioNewton2(2,1)
```

```
[1] 9
```

Binomio de Newton

$$(a + b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} \cdot b^k$$

Función con R

```
binomioNewton=function(a,b,n){  
  cumsum(choose(n,(0:n))*a^{n-(0:n)}*b^{(0:n)})[n+1]  
}  
binomioNewton(2,1,2)
```

```
[1] 9
```

La función *cumsum* regresa un vector, por lo que le estamos pidiendo que nos regresa la entrada $n + 1$.

Subvectores

`vector[i]` nos regresa la i -ésima entrada del vector.

```
x<-seq(1,50, by=3.5)
x
```

```
[1]  1.0  4.5  8.0 11.5 15.0 18.5 22.0 25.5 29.0 32.5 36.0 39.5 43.0 46.5 50.0
```

- Los índices en R empiezan en 1
- `x[length(x)]`: nos da la última entrada del vector

```
x[length(x)]
```

```
[1] 50
```

- `x[a:b]`: nos da el subvector con las entradas del vector original que van de la *posición a* hasta la *posición b*.

```
x[3:5]
```

```
[1]  8.0 11.5 15.0
```

- `x[-i]`: nos devuelve el subvector formado por todas las entradas del vector original menos la entrada i -ésima.

```
x[-c(1,15)]
```

```
[1]  4.5  8.0 11.5 15.0 18.5 22.0 25.5 29.0 32.5 36.0 39.5 43.0 46.5
```

- `x[-y]`: si y es un vector (de índices), entonces este es el complementario de vector $x[y]$.

```
y<-1:10
x[-y]
```

```
[1] 36.0 39.5 43.0 46.5 50.0
```

También podemos utilizar operadores lógicos:

- `==`: Igualdad
- `!=`: Desigualdad
- `>=`: Mayor o igual que
- `<=`: Menor o igual que
- `!`: NO lógico
- `&`: Y lógico
- `|`: O lógico

Condicionales

La clave para hacer buen uso de los *which()* es entender que regresan la **posición** en la que se ubica un elemento que cumple con cierta condición, si quisieramos saber qué elemento es, debemos hacer *x[which()]*.

```
x<-c(3,8,10,-2,0,5,12,-3,2,0,-3,12,1)
x
```

```
[1]  3  8 10 -2  0  5 12 -3  2  0 -3 12  1
```

- **which(x cumple condición)**: para obtener los *índices* (posiciones) de las entradas del vector *x* que satisfacen la condición dada.

```
which(x<0)
```

```
[1]  4  8 11
```

- **which.min(x)**: nos devuelve la *primera posición* en la que el vector *x* toma su valor mínimo.

```
which.min(x)
```

```
[1]  8
```

- **which.max(x)**: nos da la *primera posición* en la que el vector *x* toma su valor máximo.

```
which.max(x)
```

```
[1]  7
```

- **which(x==min(x))**: devuelve todas las posiciones en la que el vector *x* toma su valor mínimo (en caso de que el valor mínimo aparezca en más de una entrada).

```
which(x==min(x))
```

```
[1]  8 11
```

- **which(x==max(x))**: da todas las posiciones en las que el vector *x* toma su valor máximo.

```
which(x==max(x))
```

```
[1]  7 12
```


Los valores NA

Una vez que hayamos creado un vector, podemos cambiar sus entradas o agregar nuevas.

```
# Cambiar valores
x<-1:10
x[3:5]=32
x
```

```
[1] 1 2 32 32 32 6 7 8 9 10
```

```
# Agregar nuevos valores
x[11]=11
x[12:13]=c(12,13)
x[18]=18
x
```

```
[1] 1 2 32 32 32 6 7 8 9 10 11 12 13 NA NA NA NA 18
```

Cuando tenemos un vector con valores NA no es posible hacer cálculos como *mean*, *std* y otras. Una primera solución podría ser usar el argumento **na.rm=TRUE** (na remove)

```
# Mean
mean(x,na.rm = T)
```

```
[1] 13.78571
```

Por otro lado, podemos conocer qué entradas son las que poseen valores NA, es decir, las posiciones del vector en las que se encuentran los valores NA. Esto se puede hacer con la función **is.na()**

```
# Devuelve las posiciones donde se encuentran los valores nulos
is.na(x)
```

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

Para saber en qué posiciones del vector *x* se encuentran estos valores NA podemos hacer uso de la función *which()* (recordar que esta función regresa *posiciones*).

```
which(is.na(x))
```

```
[1] 14 15 16 17
```

Ahora que conocemos las posiciones en donde se tienen valores NA podemos hacer varias técnicas para tratarlos. Una técnica simple es **sustituir los valores NA por su media**.

```
x[which(is.na(x))]=mean(x, na.rm=T)
x
```

```
## [1] 1.00000 2.00000 32.00000 32.00000 32.00000 6.00000 7.00000 8.00000
## [9] 9.00000 10.00000 11.00000 12.00000 13.00000 13.78571 13.78571 13.78571
## [17] 13.78571 18.00000
```

Una alternativa, no recomendable, podría ser omitir los registros que tengan valores NA. Esto se logra con la función **na.omit()**

```
v<-1:10  
v[14]=14  
na.omit(v)
```

```
[1] 1 2 3 4 5 6 7 8 9 10 14  
attr("na.action")  
[1] 11 12 13  
attr("class")  
[1] "omit"
```

Las leyendas que aparecen debajo del vector pueden ser ignoradas, no afectan en la realización de cálculos ni a las propiedades. Se indican qué posiciones contenían valores nulos.

(Buscar más técnicas para tratar valores nulos)

Factor

Poner el ejemplo de las notas

Los factores pueden ayudarnos a agrupar (min 16 ejemplo)

Factor ordenado