BSM

Blockchain School
for Management

Tema 2 - Estructuras de Datos y Tipos

Módulo II - Programación en R

Nicolás Forteza

2022-11-07

R se puede usar como una calculadora.

```
4 + 3
```

```
[1] 7
```

Cualquier operación aritmética, R la puede realizar:

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Elevar: ^ ó **

No olvidemos otras operaciones matemáticas, como `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`... (éstas son funciones).

```
log(7)
```

```
[1] 1.94591
```

Pero en realidad, ¿qué hay por debajo?

```
class(7)
```

```
[1] "numeric"
```

La función `class()` devuelve el tipo de *clase* o *objeto* de R que estamos evaluando. Sabemos que 7 es un número, pero para R, 7 es una *clase* de tipo `"numeric"`.

Objetos

Un objeto de R es una estructura de datos* (pizza) que tiene atributos (queso, tomate, masa) y métodos (cocinar, preparar, etc).

Cualquier entidad en R es un objeto.

Para este curso, no es relevante saber sobre objetos, pero sí tener presente qué son.

R tiene dos tipos de objetos numéricos: `"double"` y `"integer"`

- `"double"`: números decimales.
- `"integer"` numeros enteros.

```
typeof(7L)
```

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

La `L` indica que es un número `large` y lo convierte en `"integer"`. En la práctica, no distinguimos entre estos tipos, y simplemente tipamos tal que así:

```
7; typeof(7)
```

```
[1] 7
```

```
[1] "double"
```

Atributos de un número

Para ver qué atributos tiene un objeto, simplemente hacemos:

```
attributes(7)
```

```
NULL
```

`NULL` = nulo, nada.

Inf y Nan

En muchos lenguajes de programación, como R, tenemos representaciones matemáticas de indeterminaciones, tales como:

```
paste("Esto es un número infinito positivo:", 1 / 0)
```

```
[1] "Esto es un número infinito positivo: Inf"
```

Y "not a number" , o *NaN*:

```
paste("Esto no es un número!:", Inf / Inf)
```

```
[1] "Esto no es un número!: NaN"
```

Los `NaN` aparecerán siempre en muchos datasets, pues representan la *no existencia* de un valor para una variable y observación.

Variables

El operador <- o = sirve para guardar variables

Una variable guarda en la memoria de la sesión el objeto con el cual evalúas el operador.

```
a <- 7
```

En R, podemos interactuar entre diferentes variables (siempre y cuando los objetos que éstas contengan, sean compatibles entre sí para su uso).

```
b <- a - 3
print(b)
```

```
[1] 4
```

Ya veremos que no todo se puede usar para realizar operaciones, pero con objetos numéricos sí que se puede realizar todo (casi).

```
c <- b + 3L
print(c)
```

```
[1] 7
```

```
typeof(c)
```

```
[1] "double"
```

Tenemos, además, funciones que nos pueden ayudar a verificar el tipo de número:

```
is.double(c) # es de tipo double?
```

```
[1] TRUE
```

```
is.numeric(c)  # más de lo mismo
```

```
[1] TRUE
```

```
is.integer(c)  # es integer?
```

```
[1] FALSE
```

Reflexión: ¿qué puede ser lo que devuelven estas funciones?

Ejercicios

- Calcula la raíz cuadrada de tu año de nacimiento y guarda el resultado en una variable.
- Multiplica el resultado por el número *pi* (en R, el comando `pi` devuelve dicho número).

En R, tenemos objetos *booleanos*, que indican simplemente **Verdadero** o **Falso**.

```
print(TRUE); print(FALSE); print(T); print(F)
```

```
[1] TRUE

[1] FALSE

[1] TRUE

[1] FALSE
```

Operadores lógicos

Además, tenemos una serie de operadores con los que comparar objetos, y que junto con los objetos *booleanos*, podemos empezar a crear programas con algo de complejidad. A continuación se listan los principales operadores lógicos:

```
< # menor que
<= # menor o igual que
> # mayor que
>= # mayor o igual que
== # igual que
!= # no igual que
!x # no x
x|y # x O y
x&y # x Y y
```

Usando los operadores lógicos y *booleanos* podemos comparar objetos:

```
7 > 4
```

```
[1] TRUE
```

```
7 != 7
```

```
[1] FALSE
```

También tenemos funciones que evalúan el output de evaluar operaciones lógicas:

```
isTRUE(7 == 7); isFALSE(7 == 7)
```

```
[1] TRUE
```

```
[1] FALSE
```

En R, podemos crear secuencias de números con la función `seq`:

```
# del 1 al 13, cada 4
seq(1, 13, by=4)
```

```
[1]  1  5  9 13
```

```
# 5 números repartidos uniformemente entre el 0 y el 100
seq(0, 100, length.out=5)
```

```
[1]   0  25  50  75 100
```

Ejercicio

- Crear una secuencia de números entre el 0 y el 1, con una longitud de 10, y aplicar la función `exp` al resultado.
- Guardar el resultado en una variable llamada `v`.
- Inspeccionar la variable `v` el visor de archivos y variables en RStudio. ¿Qué vemos?

Un vector en R es una estructura de datos que contiene otros objetos de R. Para crear un vector, utilizamos la función `c()`.

```
vectorTest <- c(2, 4, 6, 8)
```

Con el resultado del ejercicio anterior, hemos creado un vector. De hecho, la función `seq()` devuelve un vector de objetos tipo número!.

Una forma más fácil de crear vectores es:

```
vectorTest2 <- c(30:35)
```

Lo anterior es lo mismo qu ejecutar:

```
seq(30, 35, 1)
```

Podemos también añadir elementos al final o al principio de la creación del vector:

```
vectorTest3 <- c(vectorTest, vectorTest2)
print(vectorTest3)
```

```
 [1]  2  4  6  8 30 31 32 33 34 35
```

Por defecto, los vectores no se ordenan de forma automática.

Funciones de los vectores

Podemos calcular diversas características de los vectores, tales como la longitud, el máximo, el mínimo, el rango ... y podemos realizar operaciones también, tales como la suma y el producto.

```
sum(vectorTest)
```

```
[1] 20
```

```
range(vectorTest)
```

```
[1] 2 8
```

```
length(vectorTest)
```

```
[1] 4
```

¿Qué devuelve la función `range()`?

Operaciones con vectores

Claramente podemos también realizar operaciones con vectores, como ya hiimos en el ejercicio anterior.

```
(vectorTest ** 2) + 1
```

```
[1]  5 17 37 65
```

```
(vectorTest / 10)
```

```
[1] 0.2 0.4 0.6 0.8
```

Ordenando vectores

Con la función `sort` podemos ordenar vectores. O también con la función `order`

```
# decreasing indica si lo queremos decreciente o no.
sort(vectorTest, decreasing = T)
```

```
[1] 8 6 4 2
```

La función `order` toma más parámetros.

Reflexión: ¿Qué otros parámetros toma?

También podemos crear vectores con objetos de tipo *booleanos*.

```
vectorTest + 2 < 5
```

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

Primero se computan las operaciones (en este caso la suma), y después se realiza la operación lógica de comparación.

Otras operaciones:

```
is.infinite(vectorTest)
```

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

Vectores con objetos de tipo caracter

Hasta ahora no hemos visto este tipo de objetos, pero las cadenas de caracter son otro tipo de dato.

```
class("Hola!")
```

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

Vectores con objetos de tipo caracter

No vamos a ver más sobre caracteres hasta pasados unos temas. Simplemente remarcar que si creamos un vector en el que mezclamos objetos de diferentes tipos, R fusiona la clase de los objetos a uno mismo.

```
# fijaros cómo 5 es un número
# y "manzana" una cadena de caracteres
c(5, "manzana")
```

```
[1] "5"       "manzana"
```

R devuelve `"5"` y `"manzana"`, pasando el primer elemento de dicho vector a ser una cadena de caracter en vez de un número.

Lo mismo pasa si incluimos *booleanos*.

```
c(vectorTest, vectorTest < 5)
```

```
[1] 2 4 6 8 1 1 0 0
```

En R, el 1 significa `TRUE` y el 0 `FALSE`.

¿Cómo podemos hacer acceder a una parte del vector? ¿Y a un elemento en concreto?

Para coger el primer elemento:

```
vectorTest[1]
```

```
[1] 2
```

Para coger el segundo elemento:

```
vectorTest[2]
```

```
[1] 4
```

Para coger del primer al tercer elemento:

```
vectorTest[1:3]
```

```
[1] 2 4 6
```

Para coger desde el primero hasta el último, cada dos elementos:

```
vectorTest[seq(1, length(vectorTest), 2)]
```

```
[1] 2 6
```

La función `length` devuelve la longitud de un vector. Lo utilizaremos mucho.

```
length(vectorTest)
```

```
[1] 4
```

Usando *booleanos* y operadores lógicos, podremos filtrar los elementos según la lógica que indiquemos:

```
# todos los elementos cuyo valor es mayor a 5
vectorTest[vectorTest > 5]
```

```
[1] 6 8
```

Ejercicio

- Filtrar el vector `v` del anterior ejercicio. Sólo queremos los elementos cuyo valor es menor o igual que 2.
- Guardar el resultado en otra variable llamada `v1`
- Coger el último elemento de `v1` e imprimirlo en consola. No se permite hacer `v1[7]`.

Ejercicio

- Crear un nuevo vector cuyos valores sean el doble que `v1`. Guardarlo como `v2`.
- Crear un nuevo vector denominado `v3` que incluya a los vectores `v1` y `v2`.
- Crear un nuevo vector `v4` cuyos valores sean igual a `v3` menos la media de `v3`
- Calcular el logaritmo de `v4` y filtrar dicho resultado para quedarnos con valores que sean positivos y distintos de `NaN`. Guardarlo en `v5`.
- Si la suma de `v5` es `0.8525288` ... ¡buen trabajo!

Podemos usar el filtrado para cambiar los valores de los elementos que hemos seleccionado con el filtro.

```
a <- rep(c(1, Inf), 4)
a[is.infinite(a)] <- 2
```

Una lista es un objeto que contiene una colección de componentes **ordenados**. No tienen por qué ser del mismo tipo, al contrario que como en los vectores.

```
coche1 <- list(modelo='Mustang', fabricante='Ford',
               modelos.cilindros=c(4, 6, 8))
```

Para acceder a un componente, simplemente hacemos:

```
coche1$modelo
```

```
[1] "Mustang"
```

El operador $ accede al componente por su nombre

Aquí el orden importan más aún! Podemos acceder mediante indexación:

```
coche1[1]
```

```
$modelo
[1] "Mustang"
```

```
coche1[[1]]
```

```
[1] "Mustang"
```

Ojo! No devuelven lo mismo.

Acceder mediante indexación sólamente con 1 corchete, devuelve una lista. Accediendo con 2 corchetes, devuelve el *nth* elemento.

Podemos acceder con vectores a la indexación de 1 sólo corchete.

```
coche1[c(1, 2)]
```

```
$modelo
[1] "Mustang"

$fabricante
[1] "Ford"
```

Una matriz es un vector que cuenta con un atributo llamado `dim` que indica el número de filas y columnas de la matriz. Es decir, es bidimensional. Todos los elementos de una matriz deben de ser del mismo tipo.

```
x <- c(1:20)
matrix(x,nrow = 4)
```

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

Podemos indicar el número de filas en la creación de la matriz.

Podemos indicar el número de columnas también:

```
matrix(x,ncol = 4)
```

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

Podemos indicar ambos:

```
matrix(x,ncol = 6, nrow=4)
```

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

R siempre va a crear una matriz de `nrow` por `ncol`.

```
m <- matrix(x,ncol = 6, nrow=4)
nrow(m)
```

```
[1] 4
```

```
ncol(m)
```

```
[1] 6
```

```
dim(m)
```

```
[1] 4 6
```

Con el argumento `byrow` la matriz se crea en otra dirección.

```
matrix(x, ncol=6, nrow=4, byrow = T)
```

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

Como siempre, podemos saber si una variable es una matriz con la función apropiada:

```
is.matrix(m)
```

```
[1] TRUE
```

Para acceder a los elementos, seguimos esta sintaxis:

```
# fila 1, columna 4
m[1, 4]
```

```
[1] 13
```

```
# filas de la 1 a la 3, columna 3
m[1:3, 3]
```

```
[1]  9 10 11
```

Podemos seleccionar elementos de una matriz con otra matriz.

```
sel <- rbind(c(1, 2), c(3, 4), c(3, 5))
m[sel]
```

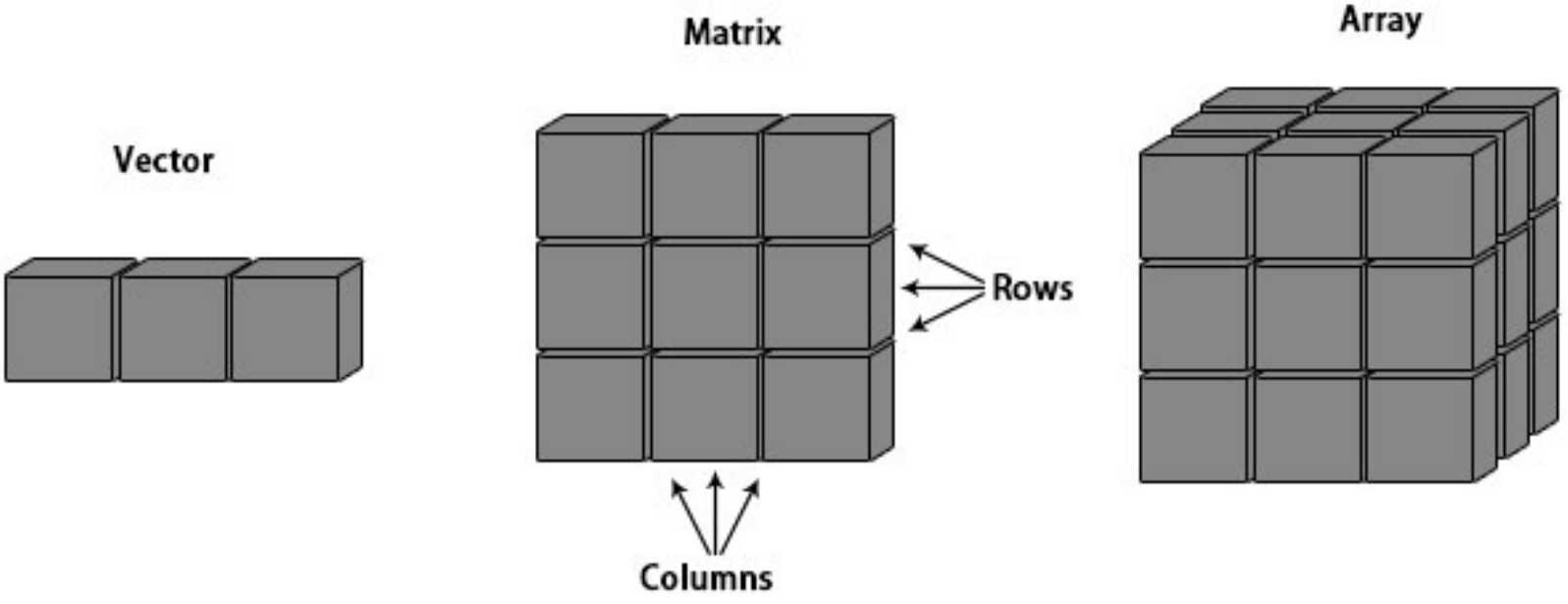
```
[1]  5 15 19
```

`rbind` concatena vectores como si fueran filas. `cbind` como si fueran columnas.

Un array es prácticamente lo mismo que una matriz, pero puede ser **tridimensional** (pensad en un cubo).

```
a <- array(data=c(1: 27), dim = c(3, 3, 3))
```

Arrays



Podemos acceder de la misma manera que a las matrices, pero tenemos que ser conscientes de que tenemos una dimensión más.

```
#  primera fila de cada capa del cubo
a[1,,]
```

```
     [,1] [,2] [,3]
[1,]    1   10   19
[2,]    4   13   22
[3,]    7   16   25
```

```
# segunda fila, columnas 1 a la 3, de todas las capas
a[2, 1:3, ]
```

```
     [,1] [,2] [,3]
[1,]    2   11   20
[2,]    5   14   23
[3,]    8   17   26
```

```
# filas de 1 a la 2, columnas de la 2 a la 3,
# la terecra capa del cubo
a[1:2, 2:3, 3]
```

```
     [,1] [,2]
[1,]   22   25
[2,]   23   26
```