

### Tipos de datos simples

- Clase y tipo de un dato
- Almacenamiento de valores en variables
- Comprobar el tipo de una variable antes de usarla
- Objetos en el espacio de trabajo

### Vectores

- Creación de vectores
- Acceso a los elementos de un vector
- Generación de vectores aleatorios
- Operar sobre vectores

### Matrices

- Creación de una matriz
- Acceso a los elementos de una matriz
- Columnas y filas con nombre

### Factors

## 2. Tipos de datos (I)


En R prácticamente todos los datos pueden ser tratados como objetos, incluidos los tipos de datos más simples como son los números o los caracteres. Entre los tipos de datos disponibles tenemos vectores, matrices, *factors*, **data frames** y listas.

El objetivo del presente capítulo es el de introducir todos esos tipos de datos y familiarizarnos con su uso a través de algunos ejercicios.

### 2.1 Tipos de datos simples

Los tipos de datos simples o fundamentales en R son los siguientes:

- **numeric**: Todos los tipos numéricos, tanto enteros como en coma flotante y los expresados en notación exponencial, son de esta clase. También pertenecen a ellas las constantes **Inf** y **NaN**. La primera representa un valor infinito y la segunda un valor que no es numérico.

 Aunque todos los tipos numéricos comparten una misma clase, que es **numeric**, el tipo de dato (que determina la estructura interna del almacenamiento del valor) es **double**. Distinguiremos entre clase y tipo más adelante.

- **integer**: En R por defecto todos los tipos numéricos se tratan como **double**. El tipo **integer** se genera explícitamente mediante la función **as.integer()**. El objetivo es facilitar el envío y recepción de datos entre código R y código escrito en C.
- **complex**: Cualquier valor que cuente con una parte imaginaria, denotada por el sufijo **i**, será tratado como un número complejo.
- **character**: Los caracteres individuales y cadenas de caracteres tienen esta clase. Se delimitan mediante comillas simples o dobles.
- **logical**: Esta es la clase de los valores booleanos, representados en R por las constantes **TRUE** y **FALSE**.

#### 2.1.1 Clase y tipo de un dato

Para los tipos de datos simples, en general la clase y el tipo coinciden salvo en el caso de datos numéricos no enteros, cuya clase es **numeric** siendo su tipo **double**.

Podemos obtener la clase y tipo de cualquier dato, ya sea constante o una variable, mediante las dos siguientes funciones:

**Sintaxis 2.1** `class(objeto)`

Devuelve<sup>a</sup> un vector con los nombres de las clases a las que pertenece el objeto.

<sup>a</sup>También es posible asignar un valor, según la sintaxis `class(objeto) <- 'clase'`, lo cual permite cambiar la clase de un objeto

**Sintaxis 2.2** `typeof(objeto)`

Devuelve una cadena indicando el tipo de dato interno usado por el objeto.

El siguiente ejercicio utiliza las dos funciones citadas para comprobar cuál es la clase y el tipo de varios datos:

**Ejercicio 2.1** Comprobación de la clase y tipo de distintos datos simples

```
> class(45)

[1] "numeric"

> class(34.5)

[1] "numeric"

> class("R")

[1] "character"

> class(TRUE)

[1] "logical"

> class(Inf)

[1] "numeric"

> class(1+2i)

[1] "complex"

> class(NaN)

[1] "numeric"

> typeof(45)

[1] "double"

> typeof(34.5)

[1] "double"

> typeof("R")
```

```
[1] "character"

> typeof(TRUE)

[1] "logical"

> typeof(Inf)

[1] "double"

> typeof(1+2i)

[1] "complex"

> typeof(NaN)

[1] "double"
```

### 2.1.2 Almacenamiento de valores en variables

Las variables en R no se definen ni declaran previamente a su uso, creándose en el mismo momento en que se les asigna un valor. El operador de asignación habitual en R es `<-`, pero también puede utilizarse `=` y `->` tal y como se aprecia en el siguiente ejercicio. Introduciendo en cualquier expresión el nombre de una variable se obtendrá su contenido. Si la expresión se compone únicamente del nombre de la variable, ese contenido aparecerá por la consola.

#### Ejercicio 2.2 Asignación de valores a una variable

```
> a <- 45
> a

[1] 45

> a = 3.1416
> a

[1] 3.1416

> "Hola" -> a
> a

[1] "Hola"
```

### 2.1.3 Comprobar el tipo de una variable antes de usarla

Aunque no es una necesidad habitual mientras se trabaja interactivamente con R, al escribir funciones y paquetes sí que es necesario comprobar el tipo de un dato antes de proceder a utilizarlo. De esta manera se evitan errores en caso de recibir un

parámetro de tipo inadecuado. La comprobación la llevaremos a cabo con alguna de las funciones `is.tipo()`:

### Sintaxis 2.3 `is.TIPO(objeto)`

Las funciones `is.numeric()`, `is.character()`, `is.integer()`, `is.infinite()` e `is.na()` comprueban si el objeto entregado como parámetro es del tipo correspondiente, devolviendo `TRUE` en caso afirmativo o `FALSE` en caso contrario.

## 2.1.4 Objetos en el espacio de trabajo

A medida que vayamos almacenando valores en variables, estas quedarán vivas en nuestro espacio de trabajo hasta en tanto no cerremos la consola. En RStudio el panel **Environment** facilita una lista de todos los objetos existentes. Desde la consola, podemos recurrir a la función `ls()` para obtener esa misma lista:

### Sintaxis 2.4 `ls([entorno,pattern=])`

Facilita un vector de cadenas de caracteres conteniendo los nombres de los objetos existentes en el entorno. Esta función puede opcionalmente tomar varios parámetros, cuyo objeto es establecer el entorno cuyo contenido se quiere obtener y establecer filtros para solamente recuperar objetos cuyos nombres se ajustan a un cierto patrón.

Tal y como se explicó en el capítulo previo, es posible almacenar el contenido de los objetos en un archivo y recuperarlo con posterioridad. También podemos eliminar cualquier objeto existente en el entorno, usando para ello la función `rm()`:

### Sintaxis 2.5 `rm(objetos)`

Elimina del entorno los objetos cuyos nombres se facilitan como parámetros, liberando la memoria ocupada por estos.

En ejercicios previos ya hemos creado algunas variables. El propuesto a continuación enumera esas variables y elimina una de ellas:

### Ejercicio 2.3 Enumeración de objetos en el entorno y eliminación

```
> ls()

[1] "a"                "is.installed" "rutaPrevia"

> rm(a)
> ls()

[1] "is.installed" "rutaPrevia"
```



La función `is.installed()`, que definíamos en un ejercicio anterior, también es una variable, concretamente de tipo `function`. Por esa razón aparece en la lista facilitada por `ls()`. Como variable que es, puede ser eliminada mediante la función `rm()`.

## 2.2 Vectores

Los vectores en R contienen elementos todos del mismo tipo, accesibles mediante un índice y sin una estructura implícita: por defecto no hay dimensiones, nombres asignados a los elementos, etc. A partir de un vector es posible crear otros tipos de datos que sí tienen estructura, como las matrices o los *data frames*.

### 2.2.1 Creación de vectores

Existe un gran abanico de funciones para generar vectores con distintos contenidos. La más usada es la función `c()`:

#### Sintaxis 2.6 `c(par1, ..., parN)`

Crea un vector introduciendo en él todos los parámetros recibidos y lo devuelve como resultado.

En el siguiente ejercicio puede comprobarse cómo se utiliza esta función. Aunque en este caso el resultado se almacena en sendas variables, podría en su lugar mostrarse directamente por la consola o ser enviado a una función.

#### Ejercicio 2.4 Creación de vectores facilitando una enumeración de los elementos

```
> diasMes <- c(31,29,31,30,31,30,31,31,30,31,30,31)
> dias <- c('Lun','Mar','Mié','Jue','Vie','Sáb','Dom')
> diasMes

[1] 31 29 31 30 31 30 31 31 30 31 30 31

> dias

[1] "Lun" "Mar" "Mié" "Jue" "Vie" "Sáb" "Dom"
```

Si los valores a introducir en el vector forman una secuencia de valores consecutivos, podemos generarla utilizando el operador `:` en lugar de enumerarlos todos individualmente. También podemos recurrir a las funciones `seq()` y `rep()` para producir secuencias y vectores en los que se repite un contenido.

#### Sintaxis 2.7 `seq(from=inicio, to=fin[, by=incremento] [,length.out=longitud])`

Crea un vector con valores partiendo de inicio y llegando como máximo a fin. Si se facilita el parámetro `by`, este será aplicado como incremento en cada paso de la secuencia. Con el parámetro `length.out` es posible crear vectores de una longitud concreta, ajustando automáticamente el incremento de cada paso.

#### Sintaxis 2.8 `rep(valor, veces)`

Crea un vector repitiendo el objeto entregado como primer parámetro tantas veces como indique el segundo.

El siguiente ejercicio muestra algunos ejemplos de uso de las anteriores funciones, generando varios vectores con distintos contenidos:

**Ejercicio 2.5** Creación de vectores facilitando una enumeración de los elementos

```

> quincena <- 16:30
> quincena

[1] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

> semanas <- seq(1,365,7)
> semanas

[1] 1 8 15 22 29 36 43 50 57 64 71 78 85 92 99
[16] 106 113 120 127 134 141 148 155 162 169 176 183 190 197 204
[31] 211 218 225 232 239 246 253 260 267 274 281 288 295 302 309
[46] 316 323 330 337 344 351 358 365

> rep(T,5)

[1] TRUE TRUE TRUE TRUE TRUE

> c(rep(T,5),rep(F,5))

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

> rep(c(T,F), 5)

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

```

**2.2.2 Acceso a los elementos de un vector**

Podemos saber cuántos elementos contiene un vector mediante la función `length()`. Esta información nos será útil a la hora de acceder a cualquiera de los elementos del vector, usando para ello la notación `vector[elemento]`.

**Sintaxis 2.9** `length(vector)`

```
length(vector) <- longitud
```

Devuelve un entero indicando el número de elementos contenidos en el vector. También permite establecer el tamaño de un vector.

Al acceder al contenido de un vector se pueden obtener uno o más elementos, facilitando entre corchetes uno o más índices. Para usar más de un índice es necesario facilitar un vector entre los corchetes. También pueden utilizarse números negativos que, en este contexto, indican qué elementos no desean obtenerse. Algunos ejemplos:

**Ejercicio 2.6** Acceso al contenido de un vector

```

> length(dias)

[1] 7

> length(semanas)

```

```
[1] 53

> dias[2]          # Solo el segundo elemento


[1] "Mar"

> dias[-2]         # Todos los elementos menos el segundo

[1] "Lun" "Mié" "Jue" "Vie" "Sáb" "Dom"

> dias[c(3,7)]     # Los elementos 3 y 7

[1] "Mié" "Dom"
```

 Los valores simples, como la variable **a** que creábamos en un ejercicio previo, también son vectores, concretamente vectores de un solo elemento. A pesar de ello, podemos utilizar con estos datos la función `length()` y el operador `[]` para acceder a su contenido, incluso con valores constantes:

#### **Ejercicio 2.7** Datos simples como vectores

```
> length(5)

[1] 1

> 5[1]

[1] 5
```

### **2.2.3 Generación de vectores aleatorios**

En ocasiones es necesario crear vectores con valores que no son secuencias ni valores repetidos, sino a partir de datos aleatorios que siguen una cierta distribución. Es usual cuando se diseñan experimentos y no se dispone de datos reales. R cuenta con un conjunto de funciones capaces de generar valores aleatorios siguiendo una distribución concreta. Las dos más usuales son `rnorm()`, asociada a la distribución normal, y `runif()`, que corresponde a la distribución uniforme:

#### **Sintáxis 2.10** `rnorm(longitud[,mean=media][,sd=desviación])`

Genera un vector de valores aleatorios con el número de elementos indicado por `longitud`. Por defecto se asume que la media de esos valores será 0 y la desviación 1, pero podemos usar los parámetros `mean` y `sd` para ajustar la distribución.

#### **Sintáxis 2.11** `runif(longitud[,min=mínimo][,max=máximo])`

Genera un vector de valores aleatorios con el número de elementos indicado por `longitud`. Por defecto se asume que el valor mínimo será 0 y el máximo 1, pero podemos usar los parámetros `min` y `max` para ajustar la distribución.

Antes de utilizar estas funciones, podemos establecer la semilla del generador de valores aleatorios a fin de que el experimento sea reproducible. Para ello recurriremos a la función `set.seed()`:

**Sintaxis 2.12** `set.seed(semilla)`

Inicializa el algoritmo de generación de valores aleatorios con la semilla facilitada como parámetro.

La diferencia entre los valores generados por las dos anteriores funciones puede apreciarse en la Figura 2.1. En ella aparecen dos histogramas representando un conjunto de 1000 valores producidos por `rnorm()` (arriba) y `runif()`.

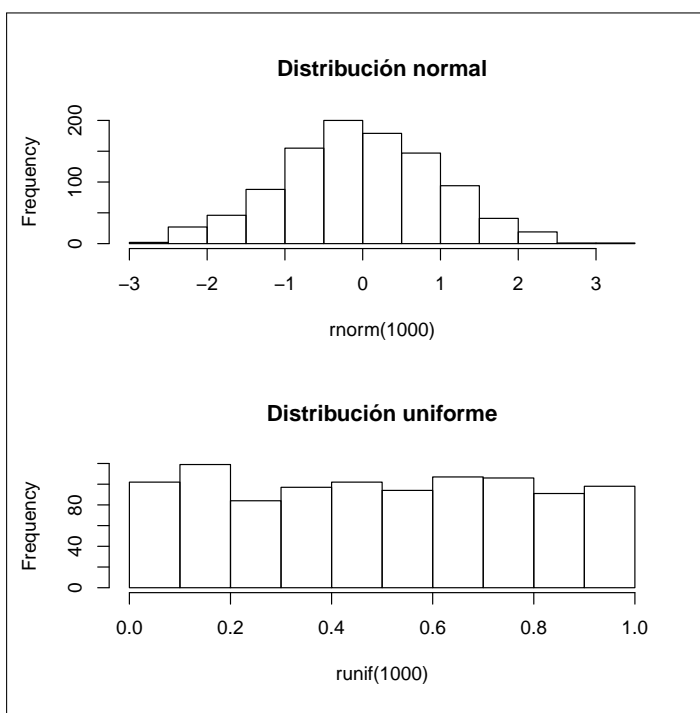


Figura 2.1: DISTRIBUCIÓN DE LOS VALORES GENERADOS POR `RNORM()` Y `RUNIF()`

En el siguiente ejercicio se muestra cómo establecer la semilla para el generador de números aleatorios y cómo obtener sendos vectores de valores generados por las dos funciones citadas:

**Ejercicio 2.8** Generación de números aleatorios

```
> set.seed(4242)
> rnorm(100, mean=10, sd=3)
```

```
[1] 16.601503  7.782490 11.597301 12.759985 11.251900  7.049153
[7]  4.087217 12.464106  7.505729  9.416750  5.900797  6.930702
[13] 11.462176 13.634359 11.915339  8.320269 12.219524  4.553933
[19] 14.626155  9.829156 10.867099  7.073053  9.469537  9.398100
[25]  9.629391 12.712182  5.536700 16.427711  4.279690 12.181423
[31]  4.775551  6.310720 14.329029 11.314088  5.968193 13.372781
[37]  5.319390  5.886031 19.273538  7.377150  9.222643  8.400000
[43] 16.307809  8.762114  6.517521  7.350390 15.326032  9.000971
```



```
[49] 10.128321 10.220721 12.423237 9.295735 7.661076 10.664515
[55] 15.588475 15.646877 9.293694 9.181816 14.327371 9.470363
[61] 6.989486 11.248542 11.961460 12.153842 6.076185 9.772697
[67] 10.623449 10.217452 12.734511 15.803807 13.701893 7.093037
[73] 16.272279 11.458514 8.253390 10.349666 10.063023 9.581695
[79] 10.499814 9.124369 8.715328 11.906492 6.666753 15.347326
[85] 12.292075 6.786419 13.456128 5.749136 10.983657 2.637713
[91] 9.446325 9.363261 9.336171 15.940844 9.236966 6.066087
[97] 10.272637 12.942529 8.637694 12.164843

> loto <- as.integer(runif(6, min=1, max=49))
> loto

[1] 15 9 22 43 16 15
```

### 2.2.4 Operar sobre vectores

La mayor parte de funciones y operadores con que cuenta R están preparados para actuar sobre vectores de datos. Esto significa que no necesitamos codificar un bucle a fin de aplicar una operación a cada elemento de forma individual. Si bien esto resulta totalmente posible, en general el rendimiento es considerablemente peor.

Supongamos que tenemos dos vectores con un gran volumen de datos y que precisamos operar a fin de obtener un nuevo vector, en el cada elemento contenga por cada elemento de los originales el cuadrado del primero más el segundo. Es la operación que se efectúa en el siguiente ejercicio, primero recorriendo los elementos y efectuando la operación de manera individual y después operando sobre el vector completo como un todo:

#### Ejercicio 2.9 Operar sobre todos los elementos de un vector

```
> vect1 <- rnorm(100000)
> vect2 <- rnorm(100000)
> vect3 <- c() # El vector de resultados está inicialmente vacío
> system.time(for(idx in 1:length(vect1))
+   vect3[idx] <- vect1[idx] * vect1[idx] + vect2[idx]
+ )

   user  system elapsed 
15.45    0.17   15.63 


> system.time(vect4 <- vect1 * vect1 + vect2)

   user  system elapsed 
0      0      0 

> stopifnot(vect3 == vect4) # Resultados deben ser idénticos
```

La sentencia `for` es similar al bucle del mismo nombre existente en multitud de lenguajes de programación. En este caso la variable `idx` tomará los valores en el intervalo que va de 1 al número de elementos del primer vector. A cada ciclo se hace el cálculo para un elemento del vector. Mediante la función `system.time()` obtenemos el tiempo que ha tardado en ejecutarse la expresión facilitada como parámetro, en este caso todo el bucle.

En la segunda llamada a `system.time()` podemos comprobar cómo se opera sobre vectores completos. La sintaxis es sencilla, solamente hemos de tener en cuenta que los operadores, en este caso `*` y `+`, actúan sobre cada elemento. La sentencia final verifica que los resultados obtenidos con ambos métodos son idénticos, de no ser así la ejecución se detendría con un error.

 En R el operador para comprobar la igualdad entre dos operandos es `==` no `=`, como en la mayoría de lenguaje derivados de C.

**Sintaxis 2.13** `for(variable in secuencia) sentencia`

Recorre todos los valores en la secuencia. A cada ciclo la `variable` tomará un valor y se ejecutará la `sentencia`. Si necesitamos ejecutar varias sentencias podemos encerrarlas entre llaves.

**Sintaxis 2.14** `system.time(expresión)`

Ejecuta la expresión y devuelve el tiempo que se ha empleado en ello.

**Sintaxis 2.15** `stopifnot(expresion1, ..., expresionN)`

Verifica que todas las expresiones indicadas devuelven `TRUE`, en caso contrario se genera un error y la ejecución se detiene.

## 2.3 Matrices

Una matriz R no es más que un vector que cuenta con un atributo llamado `dim` indicando el número de filas y columnas de la matriz. Se trata, por tanto, de una estructura de datos en la que todos los elementos han de ser del mismo tipo. Podemos crear una matriz a partir de un vector, así como indicando el número de filas y columnas dejando todos sus elementos vacíos. También cabe la posibilidad de agregar manualmente el citado atributo `dim` a un vector, convirtiéndolo en una matriz.

### 2.3.1 Creación de una matriz

La función encargada de crear una nueva matriz es `matrix()`:

**Sintaxis 2.16** `matrix(vector[, nrow=numFilas, ncol=numCols, byrow=TRUE|FALSE], dimnames=nombres)`

Crea una matriz a partir del vector de datos entregado como primer parámetro. Si no es posible deducir a partir de la longitud del vector el número de filas y columnas, estos son parámetros que también habrá que establecer. El argumento `byrow` determina si los elementos del vector se irán asignando por filas o por columnas. El uso de los nombres para las dimensiones se detalla más adelante.

Al igual que ocurre con los vectores, la función `length()` devuelve el número de elementos de una matriz. Para conocer el número de filas y columnas podemos usar las función `nrow()`, `ncol()` y `dim()`.

**Sintaxis 2.17** `nrow(matriz)`

Devuelve un entero indicando el número de filas que tiene la matriz.

**Sintaxis 2.18** `ncol(matriz)`

Devuelve un entero indicando el número de columnas que tiene la matriz.

**Sintaxis 2.19** `dim(matriz)`

Devuelve un vector con el número de filas y columnas que tiene la matriz.

También es posible asignar un vector a fin de modificar la estructura de la matriz.

A continuación se muestran varias formas distintas de crear una matriz:

**Ejercicio 2.10** Creación de matrices y obtención de su estructura

```
> mes <- matrix(1:35,ncol=7) # Dos formas de generar exactamente  
> mes <- matrix(1:35,nrow=5) # la misma matriz  
> mes
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]  
[1,]    1    6   11   16   21   26   31  
[2,]    2    7   12   17   22   27   32  
[3,]    3    8   13   18   23   28   33  
[4,]    4    9   14   19   24   29   34  
[5,]    5   10   15   20   25   30   35
```

```
> mes <- matrix(1:35,nrow=5,ncol=7,byrow=T)  
> mes
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]  
[1,]    1    2    3    4    5    6    7  
[2,]    8    9   10   11   12   13   14  
[3,]   15   16   17   18   19   20   21  
[4,]   22   23   24   25   26   27   28  
[5,]   29   30   31   32   33   34   35
```

```
> length(mes)
```

```
[1] 35
```

```
> nrow(mes)
```

```
[1] 5
```

```
> ncol(mes)
```

```
[1] 7
```

```
> dim(mes)

[1] 5 7
```

Podemos determinar si un cierto objeto es o no una matriz mediante la función `is.matrix()`. También tenemos a nuestra disposición una función para convertir objetos de otros tipos en matrices: `as.matrix()`.

**Sintaxis 2.20** `is.matrix(objeto)`

Devuelve TRUE si el objeto es una matriz o FALSE en caso contrario.

**Sintaxis 2.21** `as.matrix(objeto)`

Intenta convertir el objeto facilitado como parámetro en una matriz.

En el siguiente ejemplo se muestra cómo es posible generar una matriz a partir de un vector, sencillamente estableciendo sus dimensiones mediante la función `dim()`.

**Ejercicio 2.11** Conversión de un vector en una matriz

```
> is.matrix(vect4)

[1] FALSE

> dim(vect4)

NULL

> dim(vect4) <- c(1000, 100)
> is.matrix(vect4)

[1] TRUE

> dim(vect4)

[1] 1000 100
```

### 2.3.2 Acceso a los elementos de una matriz

El acceso a los elementos de una matriz es similar al caso de los vectores. Se usa el mismo operador `[]`, pero en este caso se esperan dos parámetros separados por una coma: la fila y la columna. Cualquiera de ellos puede obviarse, recuperando filas y columnas completas. También es posible facilitar vectores de índices.

**Ejercicio 2.12** Acceso a los elementos de una matriz

```
> mes[2,5] # Quinto elemento de la segunda fila

[1] 12

> mes[2,]  # Segunda fila completa
```

```
[1] 8 9 10 11 12 13 14

> mes[,2]    # Segunda columna completa

[1] 2 9 16 23 30
```

### 2.3.3 Columnas y filas con nombre

Por defecto las filas y columnas de una matriz tienen asignado únicamente un índice, pero es posible establecer nombres en ambas dimensiones. Esa es la finalidad del parámetro `dimnames` de la función `matrix()`. También podemos usar las funciones `rownames()` y `colnames()` tanto para obtener como para establecer los nombres asociados a filas y columnas, respectivamente.

#### Sintaxis 2.22 `rownames(matriz)`

```
rownames(matriz) <- nombres
```

Devuelve o establece los nombres asociados a las filas de la matriz. Estos se facilitan como un vector de cadenas de caracteres.

#### Sintaxis 2.23 `colnames(matriz)`

```
colnames(matriz) <- nombres
```

Devuelve o establece los nombres asociados a las columnas de la matriz. Estos se facilitan como un vector de cadenas de caracteres.

Los nombres asignados a filas y columnas no solamente sirven como descripción de la estructura de la matriz, apareciendo cuando esta se muestra en la consola, sino que también pueden ser utilizados para acceder a los elementos de la matriz. Por ejemplo:

#### Ejercicio 2.13 Asignación de nombres a columnas y filas

```
> rownames(mes) <- c('Semana1', 'Semana2', 'Semana3',
+                    'Semana4', 'Semana5')
> colnames(mes) <- dias
> mes

      Lun Mar Mié Jue Vie Sáb Dom
Semana1  1  2  3  4  5  6  7
Semana2  8  9 10 11 12 13 14
Semana3 15 16 17 18 19 20 21
Semana4 22 23 24 25 26 27 28
Semana5 29 30 31 32 33 34 35

> attributes(mes)

$dim
[1] 5 7

$dimnames
$dimnames[[1]]
[1] "Semana1" "Semana2" "Semana3" "Semana4" "Semana5"
```

```
$dimnames[[2]]
[1] "Lun" "Mar" "Mié" "Jue" "Vie" "Sáb" "Dom"

> mes[, 'Jue']


Semana1 Semana2 Semana3 Semana4 Semana5
      4      11      18      25      32

> mes['Semana4',]

Lun Mar Mié Jue Vie Sáb Dom
  22  23  24  25  26  27  28


> mes['Semana2', 'Vie']

[1] 12
```

 Es posible editar interactivamente, en un sencillo editor, el contenido de una matriz utilizando la función `fix()`. Esta precisa el nombre de la matriz como único parámetro.

## 2.4 Factors

Al trabajar con bases de datos es habitual que tengamos que operar con datos de tipo categórico. Estos se caracterizan por tener asociada una descripción, una cadena de caracteres, y al mismo tiempo contar con un limitado número de valores posibles. Almacenar estos datos directamente como cadenas de caracteres implica un uso de memoria innecesario, ya que cada uno de las apariciones en la base de datos puede asociarse con un índice numérico sobre el conjunto total de valores posibles, obteniendo una representación mucho más compacta. Esta es la finalidad de los *factors* en R.

 Desde una perspectiva conceptual un *factor* de R sería equivalente a un tipo enumerado de C++/Java y otros lenguajes de programación.

Por tanto, un *factor* internamente se almacena como un número. Las etiquetas asociadas a cada valor se denomina niveles. Podemos crear un *factor* usando dos funciones distintas `factor()` y `ordered()`. El número de niveles de un factor se obtiene mediante la función `nlevels()`. El conjunto de niveles es devuelto por la función `level()`.

**Sintaxis 2.24** `factor(vectorDatos[,levels=niveles])`

Genera un vector a partir del vector de valores facilitado como entrada. Opcionalmente puede establecerse el conjunto de niveles, entregándolo como un vector de cadenas de caracteres con el argumento `levels`.

**Sintaxis 2.25** `ordered(vectorDatos)`

Actúa como la anterior función, pero estableciendo una relación de orden entre los valores del *factor*. Esto permitirá, por ejemplo, usar operadores de comparación entre los valores.

**Sintaxis 2.26** `nlevels(factor)`

Devuelve un entero indicando el número de niveles con que cuenta el *factor*.

**Sintaxis 2.27** `levels(factor)`

Devuelve el conjunto de niveles asociado al *factor*.

En el siguiente ejercicio se crea un vector de 1000 elementos, cada uno de los cuales contendrá un valor que representa un día de la semana. Como puede apreciarse, la ocupación de memoria cuando se utiliza un *factor* es muy importante.

**Ejercicio 2.14** Definición y uso de *factors*

```
> mdias <- c(dias[as.integer(runif(1000,0,7)+1)])
> mdias[1:10]

[1] "Jue" "Mar" "Mar" "Jue" "Jue" "Mar" "Mar" "Lun" "Jue" "Sáb"

> object.size(mdias)

8376 bytes

> fdias <- factor(mdias)
> fdias[1:10]

[1] Jue Mar Mar Jue Jue Mar Mar Lun Jue Sáb
Levels: Dom Jue Lun Mar Mié Sáb Vie

> object.size(fdias)

4800 bytes

> nlevels(fdias)

[1] 7

> levels(fdias)

[1] "Dom" "Jue" "Lun" "Mar" "Mié" "Sáb" "Vie"

> levels(fdias)[1] <- 'Sun'
```

Los niveles de un *factor* no pueden compararse entre si a menos que al definirlos se establezca una relación de orden entre ellos. Con este objetivo usaríamos la función `ordered()` antes indicada. El siguiente ejercicio muestra un ejemplo de uso:

**Ejercicio 2.15** Definición y uso de *factors* ordenados

```

> peso <- ordered(c('Ligero', 'Medio', 'Pesado'))
> tam <- peso[c(sample(peso, 25, replace=T))]
> tam

[1] Pesado Pesado Medio  Ligero Medio  Medio  Ligero Medio
[9] Ligero Ligero Medio  Medio  Pesado Ligero Pesado Pesado
[17] Pesado Medio  Medio  Medio  Medio  Pesado Pesado Ligero
[25] Medio
Levels: Ligero < Medio < Pesado

> tam[2] < tam[1]

[1] FALSE

```

Otra ventana del uso de *factors* es que sus elementos pueden ser utilizados como valores numéricos, algo que no es posible hacer con una cadena de caracteres tal y como se aprecia en el siguiente ejemplo:

**Ejercicio 2.16** Conversión de elementos de un *factor* a valores numéricos

```

> dias[3]

[1] "Mié"

> fdias[3]

[1] Mar
Levels: Sun Jue Lun Mar Mié Sáb Vie

> as.numeric(fdias[3])

[1] 4

> as.numeric(dias[3])

[1] NA

```