Índice general

Tema 2

El entorno de programación y análisis estadístico R: Objetos de datos

2.5. Objetos de datos

Todo lenguaje de programación proporciona medios para acceder a los datos almacenados en la memoria. R lo hace a través de estructuras de datos específicas a las que nos hemos referido antes como objetos de datos.

Los objetos de datos se pueden clasificar según su dimensión, o si son homogéneos (todos los datos del mismo tipo) o heterogéneos (datos de distinta naturaleza). Esto permite identificar los siguientes objetos de datos en R:

	Homogéneos	Heterogéneos
Dim-1	vector	list
Dim-2	matrix	data frame
Dim-n	array	

R no tiene estructuras de datos 0-dimensionales (escalares), lo que serían los números individuales o una cadena de caracteres, sino que un escalar es un vector un sólo elemento.

A continuación vamos a describir cada uno de estos tipos de objetos, así como las funciones básicas para operar con ellos.

2.5.1. Vectores

En R un vector es un conjunto de valores todos del mismo tipo (numérico, carácter, lógico, etc.). Los vectores en R son los objetos "atómicos" y por tanto los primeros a estudiar. Como comentábamos antes, no existe un clase de objetos de tipo escalar sino que consisten en objetos de tipo vector con longitud 1.

Crear vectores

Una forma sencilla de crear un vector en R es mediante la concatenación de valores través de la función `c()`. Veamos algunos ejemplos y su resultado:

```
> x<-c(1,2,3,4,5)
> x

[1] 1 2 3 4 5

> y<-c(-1,0,x,6)
> y

[1] -1  0  1  2  3  4  5  6

> z<-c("uno","dos")
> z

[1] "uno" "dos"

> k<-c(x,z)
> k

[1] "1"   "2"   "3"   "4"   "5"   "uno" "dos"
```

Observa que para especificar una cadena de caracteres (como en el vector `z`) tenemos que usar comillas[1]. Por otro lado, en el último caso, el objeto `k` se crea concatenando un vector numérico con uno de tipo carácter, el resultado es un vector de tipo carácter donde los números se consideran como cadena de caracteres (como indican las comillas que R usa para imprimirlos).

Además del nombre del propio vector (que le damos al crearlo y asignarlo) es posible darle nombre a cada uno de sus elementos. Eso se puede hacer usando la función `names`, un ejemplo con el vector `x` anterior sería:

```
> names(x)<-c("x1","x2","x3","x4","x5")
> x

x1 x2 x3 x4 x5
 1  2  3  4  5
```

Ten en cuenta que los nombres asignados debe constituir un vector de caracteres con la misma longitud del vector `x`.

Para crear vectores de tipo secuencia (como es el caso antes de `x` e `y`) una forma más simple es usar el operador : . Para obtener dichos vectores podríamos escribir:

```
> x<-1:5
> x
```

[1] Se pueden usar comillas dobles (") o simples (') indistintamente.

```
[1] 1 2 3 4 5

> y<--1:6
> y

[1] -1  0  1  2  3  4  5  6
```

También en orden inverso por ejemplo `5:1` . El operador : constituye un caso particular de la función `seq`, la cual permite crear secuencias equiespaciadas más generales. La función admite varios argumentos[2] que permiten definir el inicio (`from`) y el final (`to`) de la secuencia, así como su longitud (`length.out`) o el incremento de la secuencia (`by`)[3]. Por defecto estos argumentos toman todos el valor 1 de modo que si escribimos `seq()` obtenemos el valor programado por defecto para la función que en este caso sería un vector con un 1 como único elemento. Veamos algunos ejemplos de uso:

```
> seq(1,10)

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

> seq(10)

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

> seq(1,10,by=2)

[1] 1 3 5 7 9

> seq(1,10,length.out=15)

 [1]  1.000000  1.642857  2.285714  2.928571  3.571429  4.214286  4.857143  5.500000
 [9]  6.142857  6.785714  7.428571  8.071429  8.714286  9.357143 10.000000

> seq(-5,5)

 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

Otra función útil es `rep` que permite crear un vector repitiendo valores (numéricos o carácter). Variando sus argumentos[4] podremos crear distintos patrones de repetición como muestran los siguientes ejemplos:

[2] Consulta la ayuda, `help(seq)`, para más detalles.

[3] Fijando dicho incremento también fijamos la longitud de la secuencia por tanto sólo definiremos uno de los dos argumentos.

[4] Como antes te sugiero que mires la ayuda.

```
> rep(1,3)

[1] 1 1 1

> x<-10:12
> rep(x,each=2)

[1] 10 10 11 11 12 12

> rep(x,length.out=7)

[1] 10 11 12 10 11 12 10

> rep(1:4, each = 2, times = 3)

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

Finalmente es posible crear vectores con la función `vector` especificando (argumento `mode`) el tipo de valores que contendrán (`numeric`, `logical`, `complex`, `character`, etc.[5]), y su longitud (`length`).

```
> vector("numeric",2)

[1] 0 0

> vector("logical",2)

[1] FALSE FALSE

> vector("character",2)

[1] "" ""
```

El modo `numeric` engloba los dos tipos de números correspondientes a números reales (enteros, `integer` [6] y de precisión doble[7], `double`).

Un vector de tipo lógico (`logical`) tiene como elementos `TRUE` o `FALSE`. además de posibles valores perdidos, `NA`, como definiremos más abajo. Los vectores lógicos se pueden generar mediante expresiones lógicas como muestran los siguientes ejemplos:

[5] Consulta todos los tipos escribiendo `help(typeof)`

[6] El tipo `integer` permite representar de una forma más compacta el rango de enteros entre $-2e+09$ y $2e+09$.

[7] En R los números reales se almacena en formato de precisión doble (*double precision floating point numbers*). Además convierte de forma automática a este tipo cuando los cálculos lo requieren.

```
> x<--5:5 ; x

##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

> temp1<-x>0 ; temp1

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

> temp2<- x>0 & x<=3 ; temp2

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

En este caso creamos dos vectores lógicos, `temp1` y `temp2` como resultado de evaluar dos condiciones. Dichos vectores lógicos tienen la misma longitud del vector `x` (sobre el que define la expresión lógica), con elementos `TRUE` en las posiciones donde se cumple la expresión y `FALSE` en caso contrario.

Los vectores de tipo `character` son aquellos cuyos elementos son cadenas de caracteres[8], y como tales están delimitadas por comillas. Como vimos en ejemplos anteriores, este tipo de vectores se puede crear usando la función de concatenación `c()`. Otra función útil para la creación de este tipo de vectores es la función `paste` (y `paste0`), que permite concatenar varias cadenas de caracteres en una sola. Veamos algunos ejemplos que muestran su utilidad (consulta la ayuda de la función para más información):

```
> nombres <- paste(c("X","Y"), 1:10, sep="")
> nombres

 [1] "X1"  "Y2"  "X3"  "Y4"  "X5"  "Y6"  "X7"  "Y8"  "X9"  "Y10"

> nth <- paste0(1:10, c("st", "nd", "rd", rep("th", 7)))
> nth

 [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th"
```

Dos funciones útiles que crean vectores de tipo carácter son `letters` (vectores de letras) y `month.name` (vectores con los meses del año)[9]:

```
> letters[1:10]

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

[8] También se pueden utilizar caracteres especiales en ellas como por ejemplo saltos de línea, tabulaciones etc. Puedes consultar la ayuda escribiendo `help(Quotes)`

[9] `month.name` proporciona los meses en inglés, se puede escribir por ejemplo `format(ISOdate(2000, 1:12, 1), "%B")` para obtener los meses en el idioma local.

```
> LETTERS[1:10]

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

> month.name[1:4]

[1] "January"  "February" "March"    "April"
```

Acceder a los elementos de un vector

Una vez creado un vector `x` podremos acceder a sus elementos usando el operador `[`, escribiendo `x[...]` donde entre los corchetes indicaremos la posición (o posiciones) a las que queramos acceder. Por ejemplo:

```
> x<-seq(0,1,length.out=10)
> # el segundo elemento de x
> x[2]

[1] 0.1111111

> # los elementos segundo y cuarto
> x[c(2,4)]

[1] 0.1111111 0.3333333

> # los dos primeros elementos
> x[1:2]

[1] 0.0000000 0.1111111

> # los elementos en posiciones impares
> x[seq(1,10,by=2)]

[1] 0.0000000 0.2222222 0.4444444 0.6666667 0.8888889

> # repetir elementos
> x[c(2,2)]

[1] 0.1111111 0.1111111
```

En los ejemplos anteriores las posiciones que escribimos entre corchetes[10] consisten a

[10]Si no escribimos nada entre los corchetes, esto es `x[]`, devolvería el vector original.

su vez en vectores[11], en este caso vectores de enteros positivos. También es posible escribir entre los corchetes vectores de enteros negativos. Con ello lo que haríamos es eliminar del vector los elementos que ocupan las posiciones indicadas. El siguiente ejemplo ilustra este uso para el mismo vector `x` definido antes:

```
> x # vector completo

 [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667 0.7777778
 [9] 0.8888889 1.0000000

> x[-c(2,5)] # vector sin tres de sus elementos

[1] 0.0000000 0.2222222 0.3333333 0.5555556 0.6666667 0.7777778 0.8888889 1.0000000
```

Otra posibilidad consiste en especificar entre los corchetes una expresión lógica, lo cual nos permitiría acceder a los elementos del vector para los que dicha expresión se verifique (`TRUE`). Por ejemplo:

```
> x<--5:5
> x[x>0] # elementos positivos

[1] 1 2 3 4 5

> x[x>0 & x<=3] # elementos en el intervalo (0,3]

[1] 1 2 3

> peso<-c(60,75,56,70)
> sexo<-c("F","M","F","M")
> peso[sexo=="F"]

[1] 60 56
```

El último ejemplo muestra que la expresión lógica que ponemos entre corchetes para acceder a los elementos de un vector (en el ejemplo es `peso`) se puede referir a un vector diferente (en el ejemplo la condición es sobre el vector `sexo`).

Finalmente, si hemos definido nombres para los elementos de un vector (con la función `names`) podemos usarlos para acceder a ellos. Esto puede ser útil en algunas aplicaciones donde los datos corresponden a individuos que tenemos identificados mediante sus nombres, y puede resultar más sencillo recordar dichos nombres que la posición que ocupan

[11] En general hemos aplicado el operador como `x[y]`, donde tanto `x` como `y` son vectores. Cuando `y` tiene una longitud menor que `x` lo que obtenemos es un subvector de `x`. Pero qué ocurriría si `y` tiene una longitud mayor que `y`. En este caso se aplica las denominadas reglas de reclaje, a las que nos hemos referido anteriormente, y que significa que el más corto se extiende hasta la longitud del más largo. Prueba algún ejemplo para ver el resultado.

en el vector. Veamos un ejemplo sencillo de uso con el vector `peso` definido en el ejemplo anterior:

```
> names(peso)<-c("Marta","Jose","Paula","Paco")
> peso[c("Marta","Paula")]

Marta Paula
   60    56
```

Valores perdidos

Desde el punto de vista estadístico es habitual que los datos no estén completos, conteniendo lo que se denominan datos perdidos o faltantes (*missing*). La codificación de este tipo de datos en R se hace mediante el valor especial `NA` que presentamos anteriormente. Este valor puede por ejemplo constituir alguno de los elementos de un vector. Una función útil para localizar elementos faltantes en un vector es la función `is.na`. Aplicada a un vector, esta función devuelve un vector lógico (con la misma longitud que el argumento) indicando con el valor `TRUE` los elementos que son faltantes. Veamos algunos ejemplos de uso:

```
> x<-c(1:8,NA,10) # x es un vector con un dato faltante
> is.na(x)

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

> # calculamos la media del vector x
> mean(x) # operaciones con NA devuelven NA (por defecto)

[1] NA

> mean(x,na.rm=TRUE)  # el argumento na.rm permite ignorar el dato faltante

[1] 5.111111

> # otro uso de is.na:
> is.na(x)<-c(2,3)
> x # la sentencia anterior define como NA los elementos segundo y tercero

 [1]  1 NA NA  4  5  6  7  8 NA 10

> x[is.na(x)]<-0
> x # la sentencia anterior sustituye los NA del vector por 0's

 [1]  1  0  0  4  5  6  7  8  0 10
```

Los valores `NA` también pueden estar presentes en vectores de tipo carácter y los escribiremos sin comillas (de otro modo no lo reconocería como elementos faltante):

```
> x<-c('a','b',NA,'c','NA')
> is.na(x)

[1] FALSE FALSE  TRUE FALSE FALSE
```

Un valor relacionado con `NA` es `NaN` (*Not a Number*) que también describimos antes. Para gestionar este tipo de valores disponemos de la función `is.nan`, cuyo uso es similar a `is.na` como muestran los siguientes ejemplos:

```
> 0/0

[1] NaN

> Inf/Inf

[1] NaN

> x<-c(NaN,1:4,NA)
> is.na(x)  # TRUE para los NA y  los NaN

[1]  TRUE FALSE FALSE FALSE FALSE  TRUE

> is.nan(x) # solo TRUE para los NaN

[1]  TRUE FALSE FALSE FALSE FALSE FALSE

> x[is.nan(x)]<-NA
> x

[1] NA  1  2  3  4 NA
```

Operaciones con vectores

Con los vectores podemos realizar operaciones aritméticas básicas (`+`,`-`,`*`,`\`, etc.), en cuyo caso se realizarán elemento a elemento[12]. Observa los siguientes ejemplos:

[12]Hay que tener en cuenta que por defecto, cualquier operación que hagamos con valores `NA` resultará de nuevo `NA`.

```
> x<-1:5 ; y<-seq(0,1,length.out=5)
> x+y

[1] 1.00 2.25 3.50 4.75 6.00

> x*y

[1] 0.0 0.5 1.5 3.0 5.0

> exp(x)+log(1+y)

[1]   2.718282   7.612200  20.491002  55.157766 149.106306

> x^2

[1]  1  4  9 16 25

> z<-c(-2,-1)
> x+z

[1] -1  1  1  3  3
```

Observa que en el último caso los vectores `x` y `z` tienen distinta longitud (5 y 3, respectivamente). En este caso R no da ningún error[13] y realiza la operación completando el vector de longitud menor repitiendo sus elementos, en concreto usara `z` como si fuera `c(-2,-1,-2,-1,-2)`. Esto es lo que se conoce como la regla de reciclaje (*recycling rule*) en R, y que implica que el vector más corto se extiende hasta la longitud del más largo para poder realizar la operación.

Los vectores lógicos también se pueden utilizar en expresiones aritméticas, en cuyo caso son tratados[14]) como vectores numéricos con el valor 0 para `FALSE` y 1 para `TRUE`. Por ejemplo podemos evaluar una función a trozos como esta:

$$f(x) = \begin{cases} 2x & x < 0.3 \\ x^2 & 0.3 \leq x < 0.7 \\ 2 & x \geq 0.7 \end{cases}$$

en la rejilla de 10 valores equiespaciados $x = (0.1, 0.2, \ldots, 1)$, como sigue:

```
> x<-(1:10)/10
> (x<0.3)*(2*x)+(0.3<=x & x<0.7)*x^2+(x>=0.7)*2

 [1] 0.20 0.40 0.09 0.16 0.25 0.36 2.00 2.00 2.00 2.00
```

[13] Pero sí nos advierte del problema con un `warning` que ha sido omitido en estas notas.

[14] Es habitual en R que a través de los cálculos algunos objetos cambien su tipo o clase por exigencia (*coercion*) de los mismos. Veremos en la sección que también es posible hacer esto directamente usando el conjunto de funciones `as.tipo` donde `tipo` será por ejemplo `integer`, `logical`, `character` etc.

Por otro lado, R disponen de varias funciones de cálculo básicas que se pueden aplicar a vectores (funciones vectoriales[15]). Por ejemplo `sum` (suma de los elementos del vector), `prod` (producto), `cumsum` (sumas acumuladas), `cumprod` (productos acumulados), `min` y `max` (mínimo y máximo, respectivamente), `which.min` y `which.max` (posición que ocupan el mínimo y máximo, respectivamente), etc. [16]. Por otro lado, para la ordenación de vectores R dispone de las funciones `sort` y `order`. La primera aplicada a un vector permitiría su ordenación (por defecto de menor a mayor aunque se puede controlar con el argumento `decreasing`) y la segunda devuelve las posiciones de cada uno de los elementos del vector original en el vector ordenado. A continuación mostramos algunos ejemplos de uso de las funciones mencionadas en este párrafo.

```
> x<-c(2,3,5,10,6,1)
> cumsum(x)
[1]  2  5 10 20 26 27
> prod(x)
[1] 1800
> cumprod(x)
[1]    2    6   30  300 1800 1800
> max(x)
[1] 10
> which.max(x)
[1] 4
> x[which.min(x)] # equivale a min(x)
[1] 1
> sort(x)
[1]  1  2  3  5  6 10
> order(x)
[1] 6 1 2 3 5 4
> x[order(x)] # devuelve el mismo vector que order(x)
[1]  1  2  3  5  6 10
```

[15] Una de las mejores propiedades de R es su capacidad para evaluar funciones sobre un vector completo. Esto evita en muchos casos la necesidad de definir bucles, permitiendo una programación más eficiente y compacta.

[16] Consulta la ayuda por ejemplo `help(Math)` o `help(Summary)` para ver el grupo genérico de funciones de este tipo.

Con un uso similar a las funciones `which.min` y `which.max`, la función `which` nos permite encontrar qué elementos (en concreto qué índices) dentro de un vector verifican alguna condición lógica dada. Por ejemplo podemos usarla para encontrar el valor más cercano a uno dado en un vector numérico. La siguientes sentencias ilustran esto a partir de un vector de 100 valores aleatorios generados desde una distribución uniforme en el intervalo $(0, 1)$, en los que queremos buscar en qué posición está el elemento más próximo a su media:

```
> set.seed(1)
> x<-runif(100) # 100 valores aleatorios de una Uniforme(0,1)
> mx<-mean(x)
> which(abs(x-mx)==min(abs(x-mx)))

[1] 58
```

Observa que en este caso se podría haber usado también la función `which.min`. Compruébalo como ejercicio. También resuelve una tarea similar pero localizando ahora el valor más próximo por encima de la media.

Atributos básicos de un vector

Las propiedades básicas de un vector son su tipo (numérico, lógico, carácter, etc.) y su longitud[17]. Estas pueden obtenerse usando las funciones `mode` (o `typeof`) y `length`, respectivamente. Veremos que además son propiedades que tienen todos los objetos de R.

```
> x<-1:5
> mode(x)

[1] "numeric"

> typeof(x)

[1] "integer"

> length(x)

[1] 5

> y<- x>2
> mode(y)
```

[17] La longitud máxima de un vector es $2^{31} - 1$, teniendo en cuenta que además no podemos excedernos del límite que impone el sistema operativo del ordenador (consulta `help.search('memory-limits')` para más detalles). Si excedemos dicho límite R mostrará un mensaje de error del tipo `R cannot allocate vector of length...`.

```
[1] "logical"

> typeof(y)

[1] "logical"

> z<-letters[5]
> mode(z)

[1] "character"

> typeof(z)

[1] "character"
```

En R es posible cambiar la longitud de un vector ya creado de varias formas. Veamos esto con algunos ejemplos:

```
> x<-numeric()
> length(x)

[1] 0

> x[3]<-3
> x

[1] NA NA  3
```

La primera sentencia anterior crea un vector numérico vacío de longitud "arbitraria". Esto permite añadir después elementos al mismo y con ello actualizar su longitud. Observa que al añadir un elemento en la posición tercera R actualiza la longitud del vector a 3 y completa el resto de elementos hasta dicha posición con valores `NA`. Este ajuste automático también ocurre cuando truncamos un vector y re-asignamos sus valores, por ejemplo:

```
> x<-1:10
> x<-x[3*(1:3)]
> x

[1] 3 6 9
```

El reajuste de la longitud de un vector también se pueden conseguir usando la función `length`, por ejemplo:

```
> x<-numeric() # crea un vector de longitud 0
> length(x)<-3 # reajusta su longitud a 3
> x

[1] NA NA NA

> y<-1:5 # vector de longitud 5
> length(y)<-10 # incrementa su longitud a 10
> y

 [1]  1  2  3  4  5 NA NA NA NA NA

> length(y)<-3  # reduce su longitud a 3
> y

[1] 1 2 3
```

2.5.2. Factores

Un tipo especial de vectores son los objetos de tipo `factor`, de gran utilidad para la clasificación y agrupación de datos. Podemos pensar en factores como variables nominales (u ordinales), con un número fijo de posibles valores (niveles, *levels*). Algunos ejemplos serían la variable género (con dos posibles valores, "hombre" o "mujer"); la variable estado civil ("soltero/a", "viudo/a", "casado/a", "divorciado/a"), etc.

Crear factores

Para definir un factor en R podemos utilizar la función `factor`. Si consultamos la ayuda podemos ver que la función admite varios argumentos (`x`, `levels`, `labels`, etc.). De ellos sólo el primero (`x`) es obligatorio especificar, ya que constituye el vector de datos a partir de los cuales se generará el factor, y el resto son opcionales con determinados valores por defecto. Vamos a ilustrar esta función con los siguientes ejemplos de uso:

```
> civil<-c('soltero/a','viudo/a', 'casado/a', 'soltero/a','viudo/a', 'divorciado/a',
+      'soltero/a', 'casado/a', 'soltero/a', 'divorciado/a')
> civil

 [1] "soltero/a"    "viudo/a"      "casado/a"     "soltero/a"    "viudo/a"
 [6] "divorciado/a" "soltero/a"    "casado/a"     "soltero/a"    "divorciado/a"

> civil.f<-factor(civil)
> civil.f
```

```
 [1] soltero/a    viudo/a      casado/a     soltero/a    viudo/a      divorciado/a
 [7] soltero/a    casado/a     soltero/a    divorciado/a
Levels: casado/a divorciado/a soltero/a viudo/a
```

En este caso la función `factor` crea el factor `civil.f` tomando los niveles (`levels`) de los posibles (distintos) valores que contiene el vector de datos `x`. Además le asigna a cada nivel del factor una etiqueta (`labels`) consistente en la misma cadena de caracteres que el propio nivel. Podemos observar que aunque la información que hay en los dos objetos `civil` y `civil.f` es la misma, la impresión de los mismos es diferente. De hecho ambos corresponden a dos clases de objetos distintos y con distintos atributos. Para comprobar la clase de objeto podemos usar la función `class`, y para ver sus atributos la función `attributes`:

```
> class(civil)

[1] "character"

> attributes(civil)

NULL

> class(civil.f)

[1] "factor"

> attributes(civil.f)

$levels
[1] "casado/a"     "divorciado/a" "soltero/a"    "viudo/a"

$class
[1] "factor"
```

Por defecto los niveles del factor se tratan en orden alfabético, no obstante esto se puede alterar si lo queremos especificando en el argumento `levels` el orden deseado:

```
> factor(civil,levels=c('soltero/a', 'casado/a', 'divorciado/a','viudo/a'))

 [1] soltero/a    viudo/a      casado/a     soltero/a    viudo/a      divorciado/a
 [7] soltero/a    casado/a     soltero/a    divorciado/a
Levels: soltero/a casado/a divorciado/a viudo/a
```

En algunos casos los datos con los que se trabaja contienen este tipo de variables pero se recogen codificadas mediante valores numéricos. En este caso la función `factor` nos

permite asignarle etiquetas a cada uno de dichos valores. Por ejemplo consideremos el siguiente vector de valores 1-2 codificando el género de los individuos (hombre-mujer):

```
> sexo.f<-factor(c(1,1,2,1,1,2,2,1,2,1),labels=c('hombre','mujer'))
> sexo.f

 [1] hombre hombre mujer  hombre hombre mujer  mujer  hombre mujer  hombre
Levels: hombre mujer
```

Observa que por defecto el factor se imprime usando las etiquetas para los niveles. Si preferimos números podemos cambiarlo escribiendo

```
> unclass(sexo.f)

 [1] 1 1 2 1 1 2 2 1 2 1
attr(,"levels")
[1] "hombre" "mujer"
```

Esta función además se puede usar para cualquier factor para el cual queramos convertir los niveles en valores numéricos. Por ejemplo para el factor `civil.f` sería:

```
> unclass(civil.f)

 [1] 3 4 1 3 4 2 3 1 3 2
attr(,"levels")
[1] "casado/a"     "divorciado/a" "soltero/a"    "viudo/a"
```

Acceder a los elementos de un factor

Para acceder a los elementos de un vector se utiliza el operador `[` de forma análoga al caso de los vectores. Por ejemplo las siguientes sentencia muestran elementos del factor `civil.f` creado antes:

```
> civil.f[1] # primer elemento del factor

[1] soltero/a
Levels: casado/a divorciado/a soltero/a viudo/a

> civil.f[-(1:5)] # todos menos los 5 primeros elementos

[1] divorciado/a soltero/a    casado/a     soltero/a    divorciado/a
Levels: casado/a divorciado/a soltero/a viudo/a
```

Si queremos acceder a los niveles del factor, así como modificarlos podemos usar la función `levels` como muestran los siguientes ejemplos:

```
> levels(sexo.f) # todos los niveles

[1] "hombre" "mujer"

> levels(sexo.f)[2] # el segundo nivel

[1] "mujer"

> levels(sexo.f)<-c('masculino','femenino')
> sexo.f

 [1] masculino masculino femenino  masculino masculino femenino  femenino  masculino
 [9] femenino  masculino
Levels: masculino femenino
```

Aplicaciones

Los factores son habitualmente usados en la práctica para realizar tareas de agrupación. Por ejemplo nos permiten realizar resúmenes estadísticos para los distintos grupos. Para ello una función muy útil es `tapply` como ilustra el siguiente ejemplo. Supongamos que tenemos una muestra de 10 individuos de los que se ha anotado la edad y su género. El factor `sexo.f` que hemos creado antes sería el género y la edad se almacena a continuación en el vector `edad`. Para obtener la media de la edad separadamente para hombres y mujeres podemos escribir:

```
> edad<-c(23,25,20,19,20,22,24,20,23,19)
> tapply(edad,sexo.f,mean)

masculino  femenino
    21.00     22.25
```

O bien un resumen numérico básico usando la función `summary`:

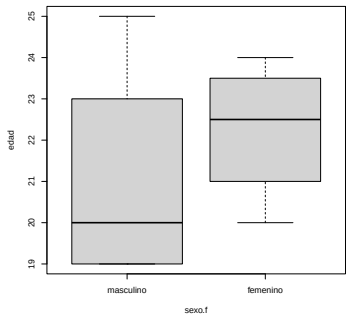
```
> tapply(edad,sexo.f,summary)

$masculino
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.00   19.25   20.00   21.00   22.25   25.00

$femenino
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  20.00   21.50   22.50   22.25   23.25   24.00
```

La agrupación definida por un factor también se puede usar para construir por ejemplo un diagrama de cajas (*boxplot*) múltiple. Por ejemplo con los datos anteriores:

```
> boxplot(edad~sexo.f)
```



2.5.3. Matrices y arrays

Las matrices y los *arrays*[18] en R permiten almacenar datos numéricos en dos o más dimensiones, respectivamente. Desde el punto de vista del lenguaje estos objetos son vectores con un atributo adicional, su dimensión (`dim`), y opcionalmente nombres para cada una de las dimensiones (`dimnames`).

Crear matrices

Una matriz se puede crear a partir de un vector añadiéndole el atributo dimensión. Por ejemplo:

```
> x<-1:10
> dim(x)<-c(2,5)
> x

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

En este ejemplo al asignarle la dimensión (2 filas y 5 columnas) al vector `x` lo convertimos en una matriz con dicha dimensión, disponiendo los elementos del vector por columnas. Una forma más flexible de crear matrices es a través de la función `matrix`. Dicha función dispone del argumento (lógico) `byrow` que permite controlar si la matriz se rellena por columnas (`byrow=FALSE`, valor por defecto) o por filas (`byrow=TRUE`) como en el siguiente ejemplo:

[18] También denominados arreglos.

```
> matrix(1:10,nrow=2,ncol=5,byrow=TRUE)

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

Por otro lado, una matriz se puede pensar como un conjunto de vectores filas o columnas. Con esta idea una matriz se puede construir en R usando las funciones `rbind` y `cbind`, como ilustran los siguientes ejemplos:

```
> cbind(1:3,4:6,7:9)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> rbind(1:3,4:6,7:9)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> cbind(matrix(1:9,3,3),c(1,0,1))

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

> cbind(matrix(1:9,3,3),0)

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

Una vez creada una matriz es posible asignarle nombres a sus filas y/o columnas usando las funciones `rownames` y/o `colnames` [19]. El uso es similar al asignar nombres a los elementos de un vector que vimos anteriormente:

[19] También es posible usar la función `dimnames` para asignar nombres a cada una de las dimensiones en una sola sentencia. No obstante el valor a asignar en este caso es un objeto de tipo `list` que veremos después.

```
> A<-matrix(1:9,3,3)
> rownames(A)<-c('fila.1','fila.2','fila.3')
> A

       [,1] [,2] [,3]
fila.1    1    4    7
fila.2    2    5    8
fila.3    3    6    9
```

Las matrices pertenecen a la clase de objetos `matrix`, cuyos atributos son sus dimensiones. Esto podemos verlo utilizando las funciones `class` y `attributes` que mencionamos anteriormente y que son aplicables a cualquier objeto de R. Por ejemplo:

```
> A<-matrix(1:4,2,2)
> class(A)

[1] "matrix" "array"

> attributes(A)

$dim
[1] 2 2
```

Acceder a los elementos de una matriz

Al igual que con los vectores se utiliza el operador `[` para seleccionar y referirnos a los elementos de una matriz, así `A[n,m]` es el elemento de la matriz `A` situado en la `n`-ésima fila y `n`-ésima la columna. También es posible referirnos a una porción de la matriz (una fila, una columna, una submatriz) como si fuera un objeto único. Para ello se aplica la siguiente regla general de R: cuando un índice se deja en blanco se entiende "todos". Así, `[,1]` se refiere a todos los elementos de la primera columna, y `[1,]` a todos los elementos de la primera fila. Veamos algunos ejemplos:

```
> A<-matrix(1:9,3,3)
> A[1,1] # elemento en la fila 1 y columna 1

[1] 1

> A[,1]  # columna 1 completa

[1] 1 2 3

> A[1,]  # fila 1 completa
```

```
[1] 1 4 7

> A[1:2,1:2] # submatriz

     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

Además podemos usar índices negativos para eliminar filas o columnas completas, así como expresiones lógicas. Observa los siguiente ejemplos con la matriz `A` creada antes:

```
> A[,-1]

     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    6    9

> A[-1,-1]

     [,1] [,2]
[1,]    5    8
[2,]    6    9

> A>3

      [,1] [,2] [,3]
[1,] FALSE TRUE TRUE
[2,] FALSE TRUE TRUE
[3,] FALSE TRUE TRUE

> B<-A; B[A>3]<-NA
> B

     [,1] [,2] [,3]
[1,]    1   NA   NA
[2,]    2   NA   NA
[3,]    3   NA   NA
```

Por último, dado que las matrices consisten en vectores con atributos adicionales, es posible seleccionar parte de sus elementos utilizando un único vector, en cuyo caso es resultado sería otro vector. Por ejemplo con la matriz `A` anterior:

```
> A[1:5]

[1] 1 2 3 4 5
```

Observa que para seleccionar los elementos de la matriz en este ejemplo, esta es tratada como si fuera un vector resultado de la concatenación de sus columnas.

Arrays

Así como las matrices extienden los vectores añadiéndoles una dimensión más, los *arrays* son la extensión de las matrices a más de dos dimensiones. Entenderemos por tanto un *array* como una variable multi-indexada, esto es, una colección de números con múltiples índices.

Como describíamos con las matrices, podemos crear un *array* a partir de un vector usando la función `dim`:

```
> x<-1:24
> dim(x)<-c(3,4,2)
> x

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

> class(x)

[1] "array"

> attributes(x)

$dim
[1] 3 4 2
```

Otra forma de crear este tipo de objeto es usando la función `array`. Por ejemplo podemos obtener el mismo resultado anterior escribiendo:

```
> x<-array(1:24,dim=c(3,4,2))
```

El acceso a los elementos de un *array* sigue las mismas reglas que con las matrices, por ejemplo para el *array* `x` creado antes podemos escribir:

```
> x[1,1,1]

[1] 1

> x[,1,1]

[1] 1 2 3

> x[,,1]

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> x[,,-1]

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Operaciones con matrices

Las operaciones aritméticas válidas para vectores lo son también para matrices, y por lo general se aplican elemento a elemento (como si fueran vectores). Además R ofrece funciones y operadores específicos para el cálculo matricial. La siguiente tabla describe los más habituales:

Función	Descripción
`*`	multiplicación elemento a elemento
`%*%`	multiplicación matricial
`t()`	transpuesta
`diag()`	diagonal
`solve(A)`	inversa de `A`
`solve(A,b)`	solución del sistema $Ax = b$
`qr()`	descomposición de Cholesky
`eigen()`	autovalores y autovectores
`svd()`	descomposición singular
`crossprod(A1,A2)`	calcula `t(A1)%*%A2`
`rowSums(A)`, `colSums(A)`	sumas por filas y columnas, respectivamente
`rowMeans(A)`, `colMeans(A)`	medias por filas y columnas, respectivamente

Vamos a ilustrar el uso de estas funciones a través de ejemplos. Comenzamos con cálculos sencillos:

```
> A<-matrix(c(0,1,-1,2),2,2)
> A*A

     [,1] [,2]
[1,]    0    1
[2,]    1    4

> A%*%A

     [,1] [,2]
[1,]   -1   -2
[2,]    2    3

> crossprod(A,A)

     [,1] [,2]
[1,]    1    2
[2,]    2    5

> rowSums(A)

[1] -1  3

> colSums(A)

[1] 1 1
```

Ahora, utilizando la función `solve` resolvemos el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} 3x + 4y - z &= 8 \\ 5x - 2y + z &= 4 \\ 2x - 2y + z &= 1 \end{aligned}$$

```
> A<-matrix(c(3,5,2,4,-2,-2,-1,1,1),3,3)
> b<-c(8,4,1)
> solve(A,b)

[1] 1 2 3
```

La forma estándar de calcular la inversa de una matriz en R es usando la misma función `solve` que nos permitió antes resolver el sistema de ecuaciones lineales. Otra opción es usar la descomposición de Choleski, si la matriz es definida positiva y simétrica. Para ello la función `chol` permitiría calcular dicha descomposición y con su resultado `cho2inv` calcularía la inversa. Veamos como ejemplo el cálculo de la inversa de $A'A$ para la matriz del ejemplo anterior:

```
> AA<-crossprod(A,A)
> solve(AA)

           [,1]       [,2]      [,3]
[1,]  0.2222222 -0.7222222 -2.222222
[2,] -0.7222222  2.7222222  8.222222
[3,] -2.2222222  8.2222222 25.222222

> chol2inv(chol(AA))

           [,1]       [,2]      [,3]
[1,]  0.2222222 -0.7222222 -2.222222
[2,] -0.7222222  2.7222222  8.222222
[3,] -2.2222222  8.2222222 25.222222
```

Otra función útil es `outer`. Dadas dos matrices `A` y `B` podemos escribir `outer(A,B,*)` (o equivalentemente `A%o%B`) y nos dará su producto exterior. En la función, la operación multiplicación (`*`) se puede sustituir por cualquier otra función. Esto nos permite por ejemplo evaluar una función en dos dimensiones sobre una rejilla de valores, como se muestra a continuación:

```
> f <- function(x, y) 2*y/(1 + x^2)
> xx<-seq(0,1,length.out=5); yy<-xx+1
> outer(xx,yy,f)

         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 2.000000 2.500000 3.000000 3.500000 4.000000
[2,] 1.882353 2.352941 2.823529 3.294118 3.764706
[3,] 1.600000 2.000000 2.400000 2.800000 3.200000
[4,] 1.280000 1.600000 1.920000 2.240000 2.560000
[5,] 1.000000 1.250000 1.500000 1.750000 2.000000
```

La primera de la sentencia anteriores define un objeto de tipo `function`. De momento no prestamos mucha atención a la sintaxis de dicha sentencia ya que lo estudiaremos detenidamente en el tema siguiente.

Otra operación de interés a realizar con una matriz de datos consiste en su ordenación por los valores de una determinada columna o fila. Para esto podemos usar la función `order` que vimos anteriormente. Por ejemplo, consideramos la siguiente matriz:

```
> x <- matrix(c(17,10,11,12,5,14,13,10,8),ncol=3); x

     [,1] [,2] [,3]
[1,]   17   12   13
[2,]   10    5   10
[3,]   11   14    8
```

La matriz ordenada por los valores de la primera columna sería:

```
> x[order(x[,1]),]

     [,1] [,2] [,3]
[1,]   10    5   10
[2,]   11   14    8
[3,]   17   12   13
```

Y ordenada por los valores de la primera fila:

```
> x[,order(x[1,])]

     [,1] [,2] [,3]
[1,]   12   13   17
[2,]    5   10   10
[3,]   14    8   11
```

Si queremos el orden inverso entonces podemos usar la función `rev` delante de `order` como sigue:

```
> x[,rev(order(x[1,]))]

     [,1] [,2] [,3]
[1,]   17   13   12
[2,]   10   10    5
[3,]   11    8   14
```

2.5.4. Listas

Vectores, matrices y factores permiten almacenar datos todos del mismo tipo (`numeric`, `character`, `logic`, `complex`). Es posible por ejemplo combinar varios vectores numéricos (incluso matrices) para crear una matriz, sin embargo no es posible crear matrices con distintos tipos de datos. En R las listas proporcionan una manera flexible de almacenar información de diversos tipos.

Una lista en R es un objeto de la clase `list` que consiste a su vez en una colección de objetos (posiblemente de distinto tipo) que constituyen sus componentes.

Crear una lista

Pensemos por ejemplo en crear un objeto de datos consistente en una matriz numérica (`A`), sus medias por filas (`m1`) y columnas (`m2`), los máximos y mínimos globales (`minA`, `maxA`), los nombres de las columnas (`nomb`), y el resultado de la evaluación de una condición lógica (`cond`). La función `list` nos permite combinar todas estas componentes en una lista de R como sigue:

```
> A<-matrix(1:6,3,2)
> m1<-rowMeans(A)
> m2<-colMeans(A)
> minA<-min(A)
> maxA<-max(A)
> nomb<-c('col1','col2','col3')
> cond<-A > mean(A)+sd(A)
> lista<-list(A,m1,m2,minA,maxA,nomb,cond)
> lista

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

```
[[2]]
[1] 2.5 3.5 4.5

[[3]]
[1] 2 5

[[4]]
[1] 1

[[5]]
[1] 6

[[6]]
[1] "col1" "col2" "col3"

[[7]]
      [,1]  [,2]
[1,] FALSE FALSE
[2,] FALSE FALSE
[3,] FALSE  TRUE
```

Es posible también dar nombre a (algunas o todas) las componentes de la lista. Esto se puede hacer en el momento en que la creamos, usando la función `list` con la sintaxis `list(nombre1=objeto1, nombre2=objeto2, ...)` (comprueba la diferencia de resultado en el ejemplo anterior), o bien usando la función `names` con la sintaxis `names(lista)<-c('nombre1','nombre1',....)`.

Por otro lado, la función `length` aplicada a una lista nos da el número de componentes de la misma.

También es posible crear una lista vacía, escribiendo `vector(mode='list')` (con un número indefinido de componentes) o `vector(mode='list', length=num)` (con `num` componentes), y posteriormente añadir componentes. Veamos a continuación cómo acceder a las componentes y en su caso modificarlas.

Acceder a las componentes de una lista

Para acceder a las componentes de una lista podemos usar los operadores `[`, `[[` ó `$`. El primero de ellos permite seleccionar varios elementos, devolviendo una lista, mientras que los dos últimos se refieren a una única componente. Veamos algunos ejemplos de uso con la lista que hemos creado antes, dándole nombre a cada componente:

```
> lista<-list(matriz=A,med_fil=m1,med_col=m2,minimo=minA,maximo=maxA,
+             nombres=nomb,cond=cond)
> lista[[1]] # accede a la primera componente (la matriz A)
```

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

> lista[['matriz']] # equivale a la anterior

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> lista$matriz      # equivale a la anterior

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

Los usos anteriores nos permite acceder a la primera componente de la lista. Como en este caso dicha componente es una matriz el objeto devuelto es una matriz. Así por ejemplo podremos escribir `lista[[1]][1,2]` para acceder al elemento de la matriz situado en la primera fila y segunda columna (o equivalentemente `lista[['matriz']][1,2]` o `lista$matriz[1,2]`).

Por otro lado el operador `[` aplicado a una lista permite acceder a uno o varias componentes de la lista, devolviendo otra lista. Observa en el ejemplo siguiente la diferencia entre `lista[[1]]` y `lista[1]`, así como otros usos:

```
> lista[[1]]

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> lista[1] #parecido al anterior pero distinto tipo de objeto

$matriz
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> lista[2:3]
```

```
$med_fil
[1] 2.5 3.5 4.5

$med_col
[1] 2 5
```

En el último caso, el operador : nos permite devolver una "sub-lista" con las componentes segunda y tercera de la lista original. Observa además que los nombres se transfieren a las sub-listas.

Una vez que sabemos acceder a las componentes de una lista podemos, si queremos, modificarlas o incluso añadir nuevas componentes. Por ejemplo, en la lista anterior podemos cambiar la componente séptima que es una matriz lógica por su contraria, y añadir una nueva componente consistente en una nueva lista:

```
> lista[[7]]<-!cond
> lista[[8]]<-list(1:10,factor(c(1,2,2,1,1,1,2)))
> lista

$matriz
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

$med_fil
[1] 2.5 3.5 4.5

$med_col
[1] 2 5

$minimo
[1] 1

$maximo
[1] 6

$nombres
[1] "col1" "col2" "col3"

$cond
     [,1]  [,2]
[1,] TRUE  TRUE
[2,] TRUE  TRUE
```

```
[3,] TRUE FALSE

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

[[8]][[2]]
[1] 1 2 2 1 1 1 2
Levels: 1 2
```

La función de concatenación `c` nos permite también concatenar listas por ejemplo:

```
> lista1<-lista[2:3]
> lista2<-lista[8]
> c(lista1,lista2)

$med_fil
[1] 2.5 3.5 4.5

$med_col
[1] 2 5

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

[[3]][[2]]
[1] 1 2 2 1 1 1 2
Levels: 1 2
```

Aplicaciones

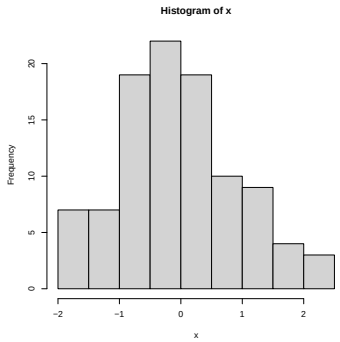
Los objetos de tipo **`list`** son requeridos como argumentos de varias funciones en el sistema base de R. Un ejemplo es la función **`matrix`** que veíamos anteriormente para crear una matriz. El argumento **`dimnames`**, si se especifica (veíamos que era opcional), debe hacerse indicando una lista con dos componentes que serán los nombres de la filas y las columnas, respectivamente. Un ejemplo sería:

```
> matrix(1:4,2,2,dimnames=list(c('fil1','fil2'),c('col1','col2')))

     col1 col2
fil1    1    3
fil2    2    4
```

Los objetos de tipo `list` también son devueltos por muchas funciones habituales de R. Por ejemplo la función `hist`, que permite construir un histograma a partir de un vector de datos, también devuelve una lista con varias componentes. Veamos un ejemplo usando dicha función para representar un histograma de unos datos generados desde una distribución normal estándar:

```
> x<-rnorm(100) # genera 100 valores aleatorios de la normal
> hist(x)
```



Observa que el resultado de la función sólo muestra el gráfico, sin embargo para construirlo, R ha realizado varios cálculos y especificaciones intermedias, estas se recogen en una lista que podemos guardar y visualizar como sigue:

```
> resultado<-hist(x)
> resultado

$breaks
 [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5

$counts
[1]  7  7 19 22 19 10  9  4  3

$density
[1] 0.14 0.14 0.38 0.44 0.38 0.20 0.18 0.08 0.06

$mids
[1] -1.75 -1.25 -0.75 -0.25  0.25  0.75  1.25  1.75  2.25

$xname
[1] "x"

$equidist
[1] TRUE
```

```
attr(,"class")
[1] "histogram"
```

Observa que entre las componentes de la lista se encuentran los intervalos usados para representar el histograma y las frecuencias asociadas, además de la densidad de frecuencia. Para continuar el ejemplo vamos a acceder a las componentes de la lista que hemos creado antes con la función `hist` para comprobar que en efecto la suma de las densidades de frecuencia multiplicadas por las amplitudes de los intervalos es igual a 1:

```
> sum(diff(resultado$breaks)*resultado$density)

[1] 1
```

Observa que en este caso todas las amplitudes son iguales por lo que podríamos haber simplificado la expresión.

2.5.5. Data frames

Otro objeto de datos básico es el denominado *data frame*, traducido en algunos casos como "hoja de datos", si bien es más común utilizar el término anglosajón. Se trata de la estructura de datos fundamental para la modelización estadística en R.

Habitualmente se usa este tipo de objeto para almacenar datos de una muestra de individuos donde cada registro corresponde a un individuo, y consisten en observaciones de distintas variables. Así podemos pensar en un data frame como una matriz cuyas columnas pueden corresponder a datos de distinto tipo, por ejemplo numérico y carácter, pero todas de la misma longitud. En este sentido un data frame guarda cierta similitud con las listas que veíamos antes, salvo que es menos flexible que estas.

Crear un data frame

Para crear un data frame podemos usar la función `data.frame` cuya sintaxis básica vamos a ilustrar con un ejemplo. Consideremos la siguiente tabla que corresponde a una pequeña muestra de datos de cinco individuos ficticios y cuatro variables:

DNI	Edad	Sexo	Estudios	Salario
22456715A	45	Hombre	Superior	2500
22456716B	35	Mujer	Superior	1500
22456717C	52	Hombre	Profesional	2000
22456718D	60	Mujer	Medio	1200
22456719E	25	Hombre	Profesional	1800

Podemos crear el data frame a partir de 5 vectores (dos numéricos para la edad y el salario, dos factores para sexo y estudios y uno de tipo carácter para el DNI) con los datos de las columnas como sigue:

```
> dni<-c('22456715A','22456716B','22456717C','22456718D','22456719E')
> edad<-c(45,35,52,60,25)
> sexo<-factor(c('Hombre','Mujer','Hombre','Mujer','Hombre'))
> estudios<-factor(c('superior','superior','profesional','medio','profesional'))
> salario<-c(2500,1500,2000,1200,1800)
> datos<-data.frame(dni,edad,sexo,estudios,salario)
> datos

        dni edad   sexo    estudios salario
1 22456715A   45 Hombre    superior    2500
2 22456716B   35  Mujer    superior    1500
3 22456717C   52 Hombre profesional    2000
4 22456718D   60  Mujer       medio    1200
5 22456719E   25 Hombre profesional    1800
```

La impresión que hace R del data frame es similar al de una tabla de datos. Observamos que en el ejemplo anterior R ha tomado por defecto los nombres de los vectores como nombres de las columnas en la tabla. Esto lo podemos cambiar si queremos de varias formas, por ejemplo usando la sintaxis `data.frame(nombre1=vector1, nombre2=vector2,...)`, y asignando los nombres que queramos o bien a posteriori, una vez creado el data frame usando la función `names` o también `colnames`[20]. Como ejemplo vamos a cambiar los nombres de las columnas primera y quinta como sigue:

```
> names(datos)  # observa los nombres actuales de las columnas

[1] "dni"      "edad"     "sexo"     "estudios" "salario"

> names(datos)[c(1,5)]<-c('NIF','sueldo')
> names(datos) # observa el resultado

[1] "NIF"      "edad"     "sexo"     "estudios" "sueldo"
```

La estructura de un data frame puede verse utilizando la función `str`:

```
> str(datos)

'data.frame': 5 obs. of  5 variables:
 $ NIF     : chr  "22456715A" "22456716B" "22456717C" "22456718D" ...
 $ edad    : num  45 35 52 60 25
 $ sexo    : Factor w/ 2 levels "Hombre","Mujer": 1 2 1 2 1
 $ estudios: Factor w/ 3 levels "medio","profesional",..: 3 3 2 1 2
 $ sueldo  : num  2500 1500 2000 1200 1800
```

[20] También se puede añadir nombres a las filas con la función `rownames`, observa escribiendo `rownames(datos)` que por defecto R ha asignado como nombres a las filas `"1", "2"`, etc.

Una función útil para realizar un resumen estadístico elemental de los datos en un data frame es la función `summary`. En el ejemplo anterior:

```
> summary(datos)

     NIF                 edad          sexo          estudios     sueldo
 Length:5           Min.   :25.0   Hombre:3   medio      :1   Min.   :1200
 Class :character   1st Qu.:35.0   Mujer :2   profesional:2   1st Qu.:1500
 Mode  :character   Median :45.0              superior   :2   Median :1800
                    Mean   :43.4                              Mean   :1800
                    3rd Qu.:52.0                              3rd Qu.:2000
                    Max.   :60.0                              Max.   :2500
```

Observa que el resumen que hace R se ajusta al tipo de datos que representa cada columna en el data frame. Por ejemplo para los factores hace un recuento de los valores en los distintos niveles, para los datos de tipo carácter proporciona el tamaño muestral y finalmente, para los de tipo numérico ofrece el mínimo, el máximo, los tres cuartiles y la media.

Acceder a los elementos de un data frame

Los data frames comparten características con las listas y las matrices, en este sentido, para acceder o seleccionar parte de sus elementos, se utilizan los operadores `[`, `[[` y `$` con reglas similares a las descritas para dichos objetos. Así en general, dado un data frame `D`:

- Escribiendo `D[v]` y dentro un solo índice `v`, se comportaría como una lista y seleccionaría las correspondientes columnas del data frame. Por ejemplo `D[1:3]` seleccionaría las tres primeras columnas.
- Escribiendo `D[v1,v2]` y dentro dos índices (`v1` y `v2`), se comportaría como una matriz y seleccionaría los correspondientes elementos. Por ejemplo `D[1:3,1]` seleccionaría el primer elemento de las tres primeras columnas.

Veamos algunos ejemplos de uso con el data frame `datos` que hemos creado antes:

```
> datos[2]

  edad
1   45
2   35
3   52
4   60
5   25

> datos[,2] # equivale a datos$edad
```

```
[1] 45 35 52 60 25

> datos[1:2,c('edad','estudios')]

  edad estudios
1   45 superior
2   35 superior

> datos[datos$sexo=='Hombre',]

        NIF edad   sexo    estudios sueldo
1 22456715A   45 Hombre    superior   2500
3 22456717C   52 Hombre profesional   2000
5 22456719E   25 Hombre profesional   1800
```

Empezando por la última sentencia vemos que podemos seleccionar individuos (filas del data frame) que cumplan una condición lógica, para ello usamos el operador `[` con dos índices, en el primero ponemos la condición y el segundo lo dejamos vacío. Por otro lado, observamos que las dos primeras sentencia producen resultados que en principio parecen distintos salvo el formato de escritura. Esto se debe al tipo de objeto que devuelven los operadores usados en cada caso. Seleccionado una columna del data frame con un único índice como ocurre con `datos[2]` el objeto resultante es otro data frame, mientras que usando dos índices `datos[2,]` el objeto resultante es una matriz. Para comprobar las diferentes estructuras de datos podemos por ejemplo usar la función `str` y escribir:

```
> str(datos[2])

'data.frame': 5 obs. of  1 variable:
 $ edad: num  45 35 52 60 25

> str(datos[,2])

 num [1:5] 45 35 52 60 25
```

Observa que en el segundo caso la selección simplifica la dimensión del objeto a devolver. Este comportamiento (por defecto) se podría cambiar de modo que se preservara siempre la dimensión original, para ello incluiríamos `drop=FALSE` en los corchetes[21] escribiendo:

```
> str(datos[,2,drop=FALSE])

'data.frame': 5 obs. of  1 variable:
 $ edad: num  45 35 52 60 25
```

[21] Consulta la ayuda de este operador `help('[')` para más detalles.

Esto también se podría aplicar a matrices. Dada una matriz `A`, la sentencia `A[,1]` devuelve un vector (con la primera columna de la matriz), sin embargo si escribimos `A[,1,drop=FALSE]`, obtenemos una matriz con una única columna.

Gestión y manipulación de data frames

Con los data frames es posible realizar diversas tareas de selección o transformación, así como gestionar varios de ellos por ejemplo uniéndolos o buscando la intersección. A continuación nos ocupamos de este tipo de operaciones. En las ilustraciones en este caso vamos a utilizar el data frame `mtcars` que está disponible en el paquete *datasets*[22]. Comenzamos explorando dicho data frame.

Consultando la ayuda podemos ver que el data frame `mtcars` consiste en 32 observaciones de 11 variables numéricas. Las variables se refieren a diversos aspectos del funcionamiento y diseño de los automóviles y se observan en 32 automóviles de los años 1973-73. La función `head` muestra la cabecera y primeras filas del data frame[23]:

```
> head(mtcars)

                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Además de los operadores `[` y `$` que describimos con anterioridad, es posible seleccionar partes de un data frame que cumplen alguna condición lógica usando la función `subset`[24] Por ejemplo vamos a usarla par seleccionar los datos correspondientes a automóviles con cambio automático y con potencia de más de 90 caballos:

```
> subset(mtcars,subset= vs==0 & hp>90)

                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

[22]Este paquete se carga por defecto como parte del sistema base de R por lo que podemos usar todos los objetos que contiene directamente. Puedes consultar la ayuda del paquete para ver sus contenido, y en particular la ayuda del data frame `mtcars` que usamos en este ejemplo.

[23]Para acceder a su contenido solo tenemos que escribir su nombre, también podemos cargarlo en el espacio de trabajo escribiendo `data(mtcars)`, aunque no es necesario para estos ejemplos.

[24]Esta función se aplica también a vectores y matrices, y en cada caso devuelve un objeto del mismo tipo.

```
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
....

> subset(mtcars,subset= vs==0 & hp>90, select=c(-vs))

                     mpg cyl  disp  hp drat    wt  qsec am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  1    4    4
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0    3    2
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0    3    4
....
```

En el primer caso obtenemos como resultado una parte del data frame, la que cumple la condición impuesta en el argumento `subset`. Con la segunda sentencia seleccionamos todas salvo la columna `vs` después de la selección.

En algunos casos nos puede interesar realizar transformaciones de los datos en un data frame. Entre otras opciones podemos usar para ello la función `transform`. La sintaxis general es `transform(df,expresion1,expresion2,...)` donde `df` es el data frame que queremos transformar y `expresion1, 2, ...` son expresiones que definen transformaciones de la forma `columna=tranformación`. Por ejemplo podemos transformar el peso (`wt`) que está dado en libras a kilogramos:

```
> transform(mtcars,wt=wt/2.2046)

                     mpg cyl  disp  hp drat        wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 1.1884242 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 1.3040914 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 1.0523451 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 1.4583144 19.44  1  0    3    1
....

> transform(mtcars,wt2=wt/2.2046)

                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb       wt2
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4 1.1884242
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4 1.3040914
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1 1.0523451
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1 1.4583144
....
```

Observa que en el segundo caso la transformación realizada da lugar a una nueva columna en el data frame. No obstante en este caso el resultado de las transformaciones solo se ha impreso, si queremos guardarlo tendríamos que asignar el resultado a un nuevo data frame, o al mismo, reemplazando en tal caso su contenido por el transformado.

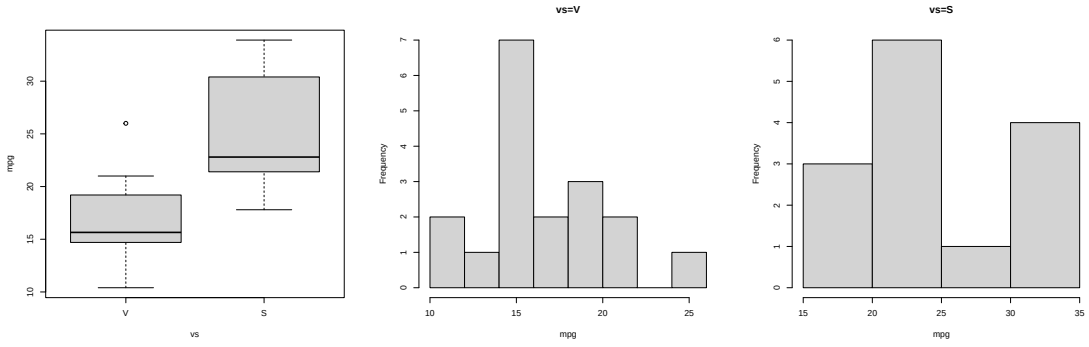
La función `within` resulta una alternativa más flexible y versátil para realizar transformaciones o seleccionar en un data frame. Para ilustrar su uso vamos a realizar la siguiente tarea con el data frame anterior: Observamos que hay varias variables del data frame que se podrían tratar como factores en R. Dos de ellas son variables nominales con dos categorías, `vs` y `am`, y tres son variables ordinales, `cyl`, `gear` y `carb`. Para transformar dichas variables (originalmente numéricas) y crear un nuevo data frame con los datos transformados en factores adecuados, podemos escribir:

```
> mtcars2 <- within(mtcars, {
+     vs <- factor(vs, labels = c("V", "S"))
+     am <- factor(am, labels = c("automatic", "manual"))
+     cyl  <- ordered(cyl)
+     gear <- ordered(gear)
+     carb <- ordered(carb)
+ })
> mtcars2

                   mpg cyl  disp  hp drat    wt  qsec vs        am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  V    manual    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  V    manual    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  S    manual    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  S automatic    3    1
....
```

La sintaxis de la función nos pide escribir las transformaciones a realizar en el segundo argumento. Dado que se trata de varias expresiones de asignación, las agrupamos usando los delimitadores {...}. Otra función similar es la función `with` con la que podemos además evaluar funciones sobre el data frame. Por ejemplo:

```
> with(mtcars2, boxplot(mpg~vs))
> with(subset(mtcars2,vs=='V'),hist(mpg,main='vs=V'))
> with(subset(mtcars2,vs=='S'),hist(mpg,main='vs=S'))
```



Hay que tener en cuenta que, mientras que la función `within` devolvía el data frame modificado, la función `with` devuelve el valor de la expresión a evaluar.

Hay varias funciones más que son útiles para la gestión y manejo de data frames y que no incluimos en estas notas. Por ejemplo, `merge` (para unir data frames), `na.omit` y `complete.cases` (para gestionar posibles datos perdidos), `unique` (borrar columnas duplicadas), `match` (comparación de data frames), etc. Una buena referencia que incluye ejemplos ilustrativos de uso es el libro de M.J. Crawley (2013)[25].

2.6. Comprobación y cambio del tipo de objeto

En una sesión de trabajo con R podemos necesitar comprobar el tipo o clase a la que pertenece un objeto (*membership*), así como cambiar a un tipo diferente (*coercion*). Esto por ejemplo es importante en la programación de funciones, donde los argumentos pueden ser requeridos de un tipo particular.

En R existe una familia de funciones que comienza por `is.` para la comprobación y otra que comienza por `as.` para el cambio.

Por ejemplo consideremos el siguiente vector numérico:

```
> x<-c(1,2,2,1,2,1,1,1)
> is.numeric(x)

[1] TRUE
```

Por su construcción es un objeto del tipo `numeric` y no es por ejemplo un factor:

```
> is.factor(x)

[1] FALSE
```

Sin embargo podemos convertir dicho objeto en un factor de forma sencilla como:

```
> x<-as.factor(x)
> x

[1] 1 2 2 1 2 1 1 1
Levels: 1 2

> class(x)

[1] "factor"
```

En general una expresión del tipo `is.xxx(objeto)` permite comprobar si `objeto` es de la clase `xxx`. Mientras que una expresión del tipo `as.xxx(objeto)` permite convertir `objeto` a uno de la clase `xxx`. La tabla 2.1 muestra las expresiones de este tipo más

[25] Crawley, M.J. (2013). *The R book*. John Wiley & Sons.

habituales[26]. Algunas de estas funciones han de usarse con cuidado. Por ejemplo mientras que `is.logical`, `is.integer`, `is.double` y `is.character`, funcionan como cabe esperar, las funciones `is.vector` y `is.numeric` no comprueban que el objeto es un vector, leyendo la documentación puede ayudarte a entender por qué.

Tipo	Comprobación	Cambio
Array	is.array	as.array
Carácter	is.character	as.character
Complejo	is.complex	as.complex
Data frame	is.data.frame	as.data.frame
Double	is.double	as.double
Factor	is.factor	as.factor
Lista	is.list	as.list
Lógico	is.logical	as.logical
Matriz	is.matrix	as.matrix
Numérico	is.numeric	as.numeric
Vector	is.vector	as.vector

Tabla 2.1: Funciones para la comprobación el cambio (*coercion*) del tipo de objeto.

El cambio de clase o tipo es algo que habitualmente ocurre en R sin que lo especifiquemos directamente, como requisito para poder procesar algunos cálculos o evaluar funciones. Por ejemplo esto ocurre con variables de tipo lógico cuyos valores `TRUE` o `FALSE` se convierte en 1 y 0, respectivamente, en expresiones aritméticas.

Los siguientes ejemplos nos muestran algunos usos más de estas funciones.

```
> as.matrix(1:2)

     [,1]
[1,]    1
[2,]    2

> as.complex(1:2)

[1] 1+0i 2+0i

> A<-matrix(1:4,2,2)
> is.numeric(A)

[1] TRUE

> as.numeric(A)
```

[26]También se puede consultar todas las funciones que comienzan por `is.` o `as.` escribiendo `apropos('is.')` o `apropos('ss.')`, respectivamente.

```
[1] 1 2 3 4

> D<-as.data.frame(A)
> is.data.frame(D[1])

[1] TRUE

> is.data.frame(D[1,])

[1] TRUE

> is.data.frame(D[,1])

[1] FALSE

> is.vector(D[,1])

[1] TRUE

> as.numeric(factor(c('H','M')))

[1] 1 2

> hh<-hist(runif(100),plot=FALSE)
> is.list(hh)

[1] TRUE
```

Existe un tipo especial de objeto llamado `NULL`. Se utiliza para indicar que un objeto está ausente. No debemos confundirlo con valores perdidos `NA` o por ejemplo con vectores de longitud cero. El objeto `NULL` no tiene tipo y sus propiedades no se pueden modificar. Además es único. La función `is.null` se puede usar para comprobar si un objeto es de este tipo.

2.7. Leer datos de ficheros

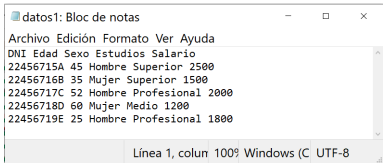
Habitualmente los datos para un análisis estadístico se toman de ficheros externos. Hay varias funciones disponibles en R para leer datos de estos ficheros y almacenarlos en objetos de datos de R como los que hemos visto anteriormente. Cuando los datos se van almacenar en un data frame la forma más cómoda de leerlos, y quizá la más recomendable, es con la función `read.table`. Otra función básica es la función `scan`. A continuación describiremos ambas posibilidades centrándonos principalmente en la lectura de datos de tipo texto. No obstante R permite la importación de datos en muchos otros

formatos (Excel, SPSS, bases de datos MySQL, Oracle, etc.). Para ampliar este tema la referencia más adecuada es el documento de ayuda *Data Import/Export* en CRAN.

2.7.1. Función `read.table`

Cuando es posible preparar los datos en un fichero de tipo texto con formato de tabla (data frame) la función principal y la más sencilla para importarlos en R es `read.table`. Esta función está diseñada para leer ficheros de datos donde cada fila representa una observación de varias variables, dadas a su vez por columnas. Los datos podrán estar separados por comas, espacios en blanco, tabulaciones etc., además la primera fila podrá contener los nombres de las variables. Con tal formato la función almacenará los datos en un data frame con la misma estructura.

Para ilustrar su uso básico antes de entrar en más detalles crearemos un fichero de tipo texto (con nombre `datos.txt`) con una tabla de datos como por ejemplo el que muestra la siguiente figura:



El fichero en este caso tiene una primera fila especificando los nombres de las variables y los valores están separados por un espacio en blanco. Para importar los datos escribiremos:

```
> datos<-read.table(file="datos.txt",header=TRUE)
> datos

        DNI Edad   Sexo    Estudios Salario
1 22456715A   45 Hombre    Superior    2500
2 22456716B   35  Mujer    Superior    1500
3 22456717C   52 Hombre Profesional    2000
4 22456718D   60  Mujer       Medio    1200
....
```

El objeto `datos` es un data frame[27] que contiene los datos. En la función hemos especificado únicamente dos argumentos, el primero indica el nombre del fichero cuyo valor debe ser una cadena de caracteres, y el segundo corresponde a un valor lógico que indica si la primera contiene los nombres de las variables. En este caso no se ha especificado el camino completo donde está alojado el fichero por lo que se espera que esté en el directorio de trabajo. Si no es así habría que escribir el camino completo, por ejemplo

[27] Puedes utilizar ahora las funciones de comprobación para confirmar el tipo de objeto, así como el de cada una de las columnas.

`file="C:/clase/datos1.txt"` leería el fichero en el directorio[28] `clase` de la unidad `C` del ordenador.

El directorio de trabajo se puede gestionar usando las funciones `getwd` y `setwd`. La primera devolvería el actual directorio de trabajo escribiendo `getwd()`, y la segunda permite cambiarlo. Por ejemplo:

```
> setwd("C:/clase")

Error in setwd("C:/clase"): no es posible cambiar el directorio de trabajo

> getwd()

[1] "C:/Users/Usuario/Dropbox/Clases2021_22/EstadisticaComputacional_Mates/Apuntes"
```

Uno de los errores más comunes en el uso de esta función se produce cuando la primera fila contiene los nombres de las variables y alguno de ellos contiene espacios en blanco. En este caso se generaría un error dado que el sistema interpretaría que hay más columnas de las que aparecen a continuación. Esto se puede evitar de dos formas: eliminar los espacios en blanco en los nombres de las variables (usando por ejemplo `.` o `_`), o bien utilizando como separador la coma. En este último caso sería necesario especificar el argumento `sep=","` en la función `read.table`, ya que por defecto asume uno o varios espacios en blanco, o utilizar directamente la función `read.csv` [29] que está diseñada para leer este tipo de ficheros.

Si consultamos la ayuda de la función `read.table` podemos ver los valores por defecto para otros argumentos que no hemos indicado en el ejemplo anterior. Además de `sep` al que nos hemos referido antes, destacamos los siguientes:

- `dec` para indicar el separador de decimales (por defecto `dec=","`).
- `row.names` y `col.names` para indicar nombres de filas y columnas, respectivamente.
- `na.strings` permite especificar un vector de caracteres que serán interpretados como valores perdidos (`NA`). Por defecto los campos en blanco se interpretan así para datos de tipo lógico y numérico (o complejo).
- `as.is` para indicar si las columnas de caracteres se tratan como factores.

A continuación usamos el argumento `as.is` con los datos del ejemplo anterior. Observa que por defecto todas las columnas no numéricas se han almacenado en el data frame como variables de tipo `character` (compruébalo), no obstante dos de ellas (la tercera y

[28] El separador entre directorios es /.

[29] `csv` viene de *comma separated values*. Existe otra versión de la función `read.csv2` que se usa en los casos donde la coma es el separador decimal, en cuyo caso el separador de valores es el punto y coma (;). Además existen otras variantes como `read.delim` que puedes ver consultando la ayuda de la misma función `read.table`.

la cuarta) no lo son. Para leer el fichero indicando que queremos tratarlas como factores podemos escribir:

```
> datos<-read.table(file="datos.txt",header=TRUE, as.is=c(1))
> str(datos)

'data.frame': 5 obs. of  5 variables:
 $ DNI     : chr  "22456715A" "22456716B" "22456717C" "22456718D" ...
 $ Edad    : int  45 35 52 60 25
 $ Sexo    : Factor w/ 2 levels "Hombre","Mujer": 1 2 1 2 1
 $ Estudios: Factor w/ 3 levels "Medio","Profesional",..: 3 3 2 1 2
....
```

2.7.2. Función `scan`

Hemos visto cómo la función `read.table` es muy útil para leer datos que tienen el formato de un data frame. Para otros tipos de estructuras más complicadas disponemos funciones más flexibles (y también más complejas como se puede ver consultando la ayuda de la función) como la función `scan` (o la función `readLines`). Por defecto la función `scan` asume que los datos que leemos son numéricos de doble precisión, y para otro tipo de datos (`character`, `logical`, `integer` o `complex`) debemos especificarlo usando el argumento `what`. Para combinar datos de distinto tipo este argumento podrá ser una lista en cuyo caso las líneas del fichero se asumen registros cada uno conteniendo el número de campos dado por la longitud de la lista especificada.

La función `scan` asume por defecto datos separados por espacios en blanco (`' '`) o tabuladores (`'\t'`), salvo que se especifique otro separador usando el argumento `sep`. A diferencia de `read.table` no es posible tomar los nombres de las variables de la primera fila con `header`, y si estos estuvieran presentes podemos usar el argumento `skip=1` para saltar esa línea.

Como ejemplo vamos a leer el mismo fichero `datos.txt` que leímos antes con la función `read.table` para comprobar las ventajas que nos daba esta última función. Primero debemos saltar la primera línea de cabecera con `skip=1` y luego, dado que cada fila contiene datos de distinto tipo (numérico y alfanumérico) debemos usar el argumento `what` especificando una lista (con '' para el tipo alfanumérico y `0` para el numérico) como sigue:

```
> datos<-scan(file="datos.txt",skip=1,what=list(DNI='',edad=0,sexo='',
+                                               estudios='',salario=0))
> str(datos)

List of 5
 $ DNI     : chr [1:5] "22456715A" "22456716B" "22456717C" "22456718D" ...
```

```
 $ edad    : num [1:5] 45 35 52 60 25
 $ sexo    : chr [1:5] "Hombre" "Mujer" "Hombre" "Mujer" ...
 $ estudios: chr [1:5] "Superior" "Superior" "Profesional" "Medio" ...
....
```

Con esto nuestros datos ahora están almacenados en el objeto de tipo `list` con nombre `datos`, donde cada uno de sus elementos corresponde a una columna de datos. Podemos ahora transformar el objeto en un data frame con la estructura deseada escribiendo:

```
> datos<-as.data.frame(datos)
> str(datos)

'data.frame': 5 obs. of  5 variables:
 $ DNI     : chr  "22456715A" "22456716B" "22456717C" "22456718D" ...
 $ edad    : num  45 35 52 60 25
 $ sexo    : chr  "Hombre" "Mujer" "Hombre" "Mujer" ...
 $ estudios: chr  "Superior" "Superior" "Profesional" "Medio" ...
....
```

Usando la función `scan` es posible además leer datos introducidos de forma interactiva desde el teclado. Para ello escribiríamos `scan()` y a continuación los valores que queramos leer. Esos valores se irán almacenando en un vector hasta que dejemos una línea en blanco. Prueba por ejemplo a crear el siguiente vector de datos de ocho valores:

```
datos2<-scan()
1
2 3 4
5 6 7
8
```

Este uso puede ser una alternativa a la función `c()` para introducir datos que hemos previamente copiado desde algún editor de textos, pdf o una hoja de cálculo.

2.7.3. Datos de paquetes de R

Como parte del sistema base de R podemos acceder a gran variedad de datos, disponibles habitualmente como objetos de tipo data frame. Estos están almacenados en el paquete *datasets*, así como en otros paquetes recomendados. Podemos ver todos los datos disponibles en el paquete *datasets* escribiendo `data()`. Para acceder a estos conjuntos de datos del paquete *datasets* basta con escribir su nombre, o bien usar la sentencia `data(nombre-datos)` para cargar los datos en el espacio de trabajo. Como ejemplo podemos acceder el conjunto de datos `cars`, ver su estructura y cargarlos en el espacio de trabajo:

```
> str(cars)

'data.frame': 50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...

> data(cars)
```

Además del paquete *datasets*, tenemos datos disponibles en la mayoría de los paquetes recomendados, así como en los que tengamos instalados. Para ver todos los disponibles en dichos paquetes podemos escribir `data(package = "nombre-paquete")`, para ver los datos disponibles en un paquete específico, o bien buscar en todos los paquetes instalados escribiendo `data(package= .packages(all.available = TRUE))`. Para acceder a estos datos es podemos, o bien cargar primero el paquete (usando la función `library`), en cuyo caso todos sus datos estarán accesibles, o bien utilizar la sentencia `data(nombre-datos, package="nombre-paquete")`. Por ejemplo podemos acceder a los datos `coal` del paquete *boot* escribiendo:

```
> data(coal,package="boot")
> str(coal)

'data.frame': 191 obs. of  1 variable:
 $ date: num  1851 1852 1852 1852 1852 ...
```

Funciones `attach` y `detach`

Hemos visto anteriormente como acceder a las componentes de un data frame. Por ejemplo podemos acceder a la primera columna del data frame `cars`, escribiendo `cars$speed`. La referencia al data frame antes del operador `$` puede resultar algo tediosa en algunos casos. Alternativamente podemos usar la función `attach` para hacer accesibles directamente las columnas del data frame.Por ejemplo:

```
> attach(cars)
```

Con esto la función `attach` añade el objeto `cars` a la lista de búsqueda, en la segunda posición (compruébalo escribiendo `search()`). Y a partir de ahí podemos referirnos a sus componentes sin necesidad de indicar el nombre del data frame. Por ejemplo podemos escribir:

```
> mean(speed)

[1] 15.4
```

La función `detach` permite revertir el proceso anterior. Observa su efecto en el ejemplo anterior:

```
> detach(cars)
> mean(speed)

Error in mean(speed): objeto 'speed' no encontrado
```

Anteriormente mencionamos que la función `detach` también permitía eliminar paquetes de la lista de búsqueda. En la ayuda puedes ver otros usos además de más detalles sobre dichas funciones.

2.7.4. Escribir datos en ficheros externos

Una forma sencilla de exportar datos de una sesión de trabajo (por ejemplo para que sean leídos con otros programas) es escribiéndolos en un fichero de texto. Para ello disponemos entre otras de las funciones `write.table` y `write`. La primera funciona en sentido inverso a `read.table` y tiene especificaciones similares. La segunda permite escribir una matriz en un fichero. La sintaxis completa de las funciones puede consultarse en la ayuda. A continuación ilustramos su uso con algunos ejemplos.

```
> write.table(cars[1:10,],file='datos1.txt')
> datos1<-read.table('datos1.txt',header=TRUE)
> datos1

   speed dist
1      4    2
2      4   10
3      7    4
4      7   22
....
```

Observa que por defecto la función `write.table` imprime los nombres de las columnas y filas en el fichero. Esto es posible modificarlo usando los argumentos `col.names` y `row.names`.

En el ejemplo siguiente la función `write` nos permite escribir un vector de 10 valores aleatorios de una distribución uniforme en un fichero:

```
> write(runif(10),file='datos2.txt',ncolumns=1)
```

Si queremos escribir una matriz de datos con esta función tenemos que usar el argumento `ncol` para especificar el número de columnas, y que tener en cuenta que R traspone la matriz para escribirla. Por tanto para conseguir una impresión en el fichero similar al de la matriz debemos previamente trasponer la matriz a escribir. Por ejemplo para escribir el contenido del data frame `datos1` que hemos creado antes escribiríamos:

```
> write(t(as.matrix(datos1)),file='datos3.txt',ncolumns=2)
```

Observa por tanto que para este último uso es más apropiada la función `write.table`.

Observa que cuando imprimimos en un fichero con `write.table` o `write`, si este no existe se crea. Y si existe se reemplaza su contenido por defecto a no ser que especifiquemos el argumento `append=TRUE`. También es posible desde R borrar ficheros con la función `unlink`. Por ejemplo podemos borrar el último fichero creado escribiendo `unlink('datos3.txt)`.

2.8. Listas de búsqueda

Anteriormente hemos utilizado las funciones `ls` (también `objects`) para ver todos los objetos creados durante una sesión y guardados en el espacio de trabajo. También la función `search` para ver qué paquetes están cargados, así como los data frame *attached*. Otra función relacionada es `searchpaths` que devuelve un listado con los caminos (directorios) donde se han almacenado todos ellos. Esto constituyen la lista de búsqueda de R. Entre los objetos mostrados está siempre primero `.GlobalEnv` y al final el paquete *base*. El primero es el *global environment*, lo que hemos denominado anteriormente espacio de trabajo (*workspace*). Cada vez que R tenga que buscar un objeto lo hará comenzando por el `.GlobalEnv` y si no lo encuentra continuará buscando en los siguientes lugares de la lista de búsqueda. Es posible que existan dos objetos con el mismo nombre almacenados en posiciones distintas en la lista de búsqueda. En ese caso cuando pidamos a R que busque el objeto siempre se quedará con el que está en la posición más adelantada.

La lista de búsqueda se puede ampliar por ejemplo añadiendo paquetes con la función `library` o data frames (también objetos de tipo lista) con `attach`. Consultando la ayuda de esta última función podemos ver que es posible especificar la posición que queremos que ocupe el objeto en la lista de búsqueda (argumento `pos`). Por defecto R lo pondrá en la segunda posición, no obstante podemos indicar otro valor pero nunca la primera posición puesto que esa está reservada para `.GlobalEnv`. También es posible reducir la lista de búsqueda con la función `detach`.