

# Una breve introducción a R

DATA SCIENCE

M.José Ramírez-Quintana

*(con modificaciones de José Hernández-Orallo)*

MITSS

Universitat Politècnica de València

28 de noviembre de 2017

---

## Índice

1. El Entorno R	2
2. Creación de objetos. Asignación. Atributos.	3
3. Vectores	4
4. Factores y Tablas	6
5. Secuencias	9
6. Matrices y Arrays	10
7. Listas	12
8. Evitando bucles: apply y otras facilidades	14
9. Data frame	16
10. Cambiando nombres y valores de los factores (levels)	18
11. Valores Faltantes	20
12. Ordenar datos	22
13. Cargando y salvando datos en R	22
14. Gráficas en R	24
15. Exportando gráficos en R	27
16. Estructuras de Control	28
17. Creando funciones	30
18. Paquetes	31

## 1. El Entorno R

R es un software de código abierto con licencia GNU para el análisis de datos. Una de las principales ventajas de R es que nos permite conocer cómo son los datos, qué podemos hacer con ellos o cómo podemos transformarlos. R es un lenguaje orientado a objetos (aunque también posee características de otros lenguajes funcionales, imperativos, ...) muy potente, flexible y especialmente “extensible”, con una interfaz por línea de comandos, aunque existen interfaces gráficas disponibles para usar con R. Es también un lenguaje interpretado, lo cual significa que los comandos escritos en el teclado son ejecutados directamente sin necesidad de construir ejecutables. Adicionalmente, R es un lenguaje matricial con una sintaxis muy simple e intuitiva.

Este entorno está disponible para múltiples plataformas ( consulta <http://cran.r-project.org/>). R es un lenguaje muy extendido por lo que existen muchos libros, videos y tutoriales que os pueden ayudar tanto en la instalación de R como en el aprendizaje de este lenguaje. Por ejemplo, el tutorial on-line que podéis encontrar en <http://www.r-tutor.com/r-introduction>.

En la figura siguiente se muestra la consola de R. En la línea de comandos aparece el símbolo “>” indicando que R está preparado para recibir comandos. R usa diferentes colores para distinguir los comando introducidos por el usuario (en azul) de las respuestas proporcionadas a estos comandos. Observa la respuesta de R al comando  $2 * 3$ . Antes del resultado, 6, hay un 1 entre corchetes. Este número indica el número de línea de la respuesta y es muy útil cuando la respuesta consta de muchas líneas (como cuando visualizamos un dataset) ya que nos permite ir hacia atrás.

Al ser R orientado a objetos, las variables, datos, funciones, resultados, etc., se guardan en la memoria activa del computador en forma de objetos con un nombre específico. El nombre de un objeto debe comenzar con una letra (A-Z and a-z) y puede incluir letras, dígitos (0-9), y puntos (.). R discrimina entre letras mayúsculas y minúsculas para el nombre de un objeto, de tal manera que x y X se refiere a objetos diferentes. El punto (.) no sirve para denotar subobjetos. Por ejemplo, `a.b` es el nombre de un objeto que no tiene ninguna relación con el objeto `a`. Las subpropiedades se acceden generalmente con el símbolo dolar \$, como veremos más adelante.

El símbolo # comenta una línea hasta el final de la misma. Podemos acceder a la documentación o ayuda de cualquier función usando el símbolo de interrogación (“?”), o la función “help”, como en “`help(sort)`”. Un documento .html se envía al navegador mostrando la ayuda sobre la función o término. Si queremos buscar toda la documentación de R podemos usar el doble símbolo de interrogación “??”.  
“`??sort`”.

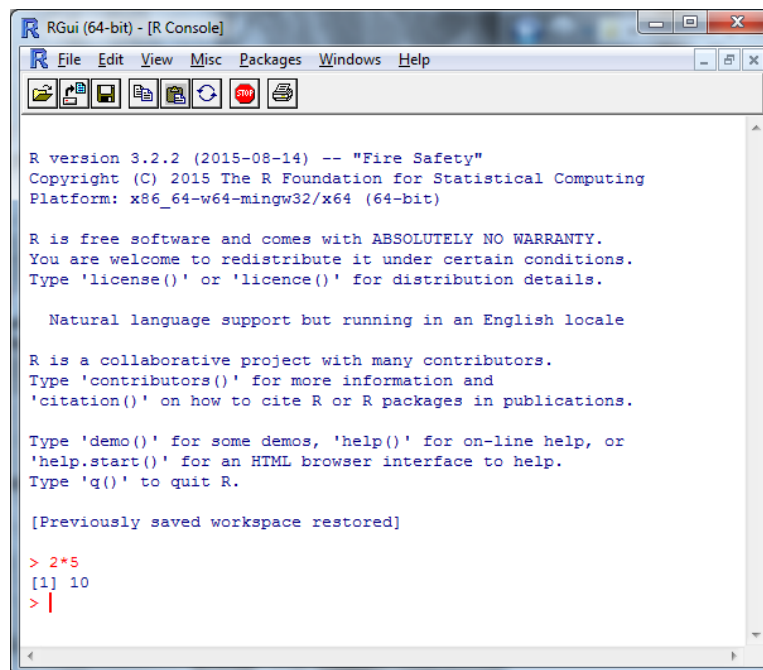


Figura 1: R Interfaz del entorno.

## 2. Creación de objetos. Asignación. Atributos.

Un objeto puede ser creado con el operador “ $\leftarrow$ ” o “ $\leftarrow$ ” dependiendo de la dirección en que asigna el objeto:

```
> x<-5 #ejemplo de asignación
> x
[1] 5
> y<-3*3
> y
[1] 9
```

Si el objeto ya existe, su valor anterior es borrado después de la asignación (la modificación afecta solo objetos en memoria, no a los datos en el disco). Alternativamente podemos usar “ $=$ ” o el comando `assign`.

```
> assign("X",3.4)
> X
[1] 3.4
> y=2
> y
[1] 2
```

Los objetos en R , además de nombre y contenido, también tiene atributos que especifican el tipo de datos representados por el objeto. Todo objeto tiene dos atributos intrínsecos: tipo y longitud. El tipo se refiere a la clase básica de los elementos en el objeto (numérico, carácter, complejo y lógico). La longitud es simplemente el número de elementos en el objeto. Para ver el tipo y la longitud de un objeto se pueden usar las funciones `mode` (o alternativamente `class()`) y `length`, respectivamente:

```
> A<-"Data Science"; h=17; l<-TRUE
> mode(A); mode(h); mode(l)
[1] "character"
[1] "numeric"
[1] "logical"
```

### 3. Vectores

El tipo de objeto más básico en R es el vector. Incluso cuando ponemos `x<-3` estamos creando un vector con un único elemento. Los vectores se usan para almacenar valores pertenecientes a un mismo tipo atómico (carácter, lógico, numérico y complejo). En R los vectores se crean usando la función `c()`:

```
> v<-c("AVS","DAS","DIM","TVD")
> v
[1] "AVS" "DAS" "DIM" "TVD"
> length(v)
[1] 4
> mode(v)
[1] "character"
```

Si los tipos son diferentes se fuerza la coerción.

```
> u<-c(1,2,4,6.0)
> u
[1] 1 2 4 6
> w<-c(1.3,2.5,3.9,5)
> w
[1] 1.3 2.5 3.9 5.0
```

Se puede hacer coerción explícita usando las funciones `as.*` tal y como indica la siguiente tabla:

Función	Conversión
as.numeric	FALSE $\rightarrow$ 0
	TRUE $\rightarrow$ 1
	"1", "2", ... $\rightarrow$ 1,2,...
	"A", ... $\rightarrow$ NA
as.logical	0 $\rightarrow$ FALSE
	otros números $\rightarrow$ TRUE
	"FALSE", "F" $\rightarrow$ FALSE
	"TRUE", "T" $\rightarrow$ TRUE
	otros caracteres $\rightarrow$ NA
as.character	1,2, ... $\rightarrow$ "1", "2", ...
	FALSE $\rightarrow$ "FALSE"
	TRUE $\rightarrow$ "TRUE"

```

> u<-c(1,2,3,4)
> class(u)
[1] "numeric"
> as.numeric(u)
[1] 1 2 3 4
> as.logical(u)
[1] TRUE TRUE TRUE TRUE
> as.character(u)
[1] "1" "2" "3" "4"
> x<-c("a","b","c")
> mode(x)
[1] "character"
> as.numeric(x)
[1] NA NA NA
Mensajes de aviso perdidos
NAs introducidos por coerción

```

También podemos crear un vector con la función `vector()` indicando su modo y longitud. El vector se creará inicializado a un valor por defecto (por ejemplo, si es numérico con 0, si es carácter con "", si es lógico con FALSE, ...).

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Para acceder a un elemento del vector, ponemos su índice entre corchetes simples:

```
> v[2]
[1] "DAS"
> v[5] <- "MITSS"
> v
[1] "AVS" "DAS" "DIM" "TVD" "MITSS"
> t <- c(v[2], v[4])
> t
[1] "DAS" "TVD"
```

R dispone de varias funciones que trabajan con vectores:

Operadores usuales	$+, -, *, /, ^$
Funciones aritméticas	log, exp, sin, cos, tan, sqrt, etc
Máximo, mínimo y rango	max, min, range
Longitud	length
Producto y suma	prod, sum
Media y varianza	mean, var
Ordenación	sort

## 4. Factores y Tablas

Un factor es un vector utilizado para especificar valores discretos de los elementos de otro vector de igual longitud. En R existen factores nominales y factores ordinales. Los factores tienen *levels* que son los posibles valores que pueden tomar aunque R almacena estos valores como códigos numéricos (más eficiente).

Supongamos que disponemos de una muestra de 8 personas de las que conocemos qué mascota tienen

```

> pet<-c("cat","dog","dog","cat","cat","snake",
+ "parrot","cat")
> pet
[1] "cat" "dog" "dog" "cat" "cat" "snake" "parrot" "cat"
> Fpet<-factor(pet)
> Fpet
[1] cat  dog  dog  cat  cat  snake  parrot cat
Levels: cat dog parrot snake
> levels(Fpet)
[1] "cat" "dog" "parrot" "snake"
> mode(Fpet)
[1] "numeric"

```

Podemos usar los factores para contar las ocurrencias de cada valor (level). Para ello usamos la función `table()`:

```

> table(Fpet)
Fpet
   cat    dog  parrot  snake
   4      2      1      1

```

Esta función también puede cruzar varios factores (dando lugar a una tabla de contingencia). Por ejemplo, podemos crear otro factor con la información de si el propietario de la mascota es un niño o una niña.

```

> own<-factor(c("girl","boy","boy","girl","girl",
+ "boy","boy","boy"))
> own
[1] girl boy  boy  girl girl boy  boy  boy
Levels: boy girl
> table(own)
own
 boy girl
   5    3

```

```
> table(Fpet,own)
      own
Fpet    boy girl
cat      1    3
dog      2    0
parrot   1    0
snake    1    0
```

Si solo estamos interesados en las frecuencias marginales se usa la función `margin.table(table,dimension):`

```
> t<-table(Fpet,own)
> margin.table(t,1)
Fpet
  cat    dog parrot  snake
   4      2      1      1
> margin.table(t,2)
own
 boy girl
   5    3
```

y si estamos interesados en las frecuencias relativas usamos `prop.table(table,dimension):`

```
> prop.table(t,1)
      own
Fpet    boy girl
cat    0.25 0.75
dog    1.00 0.00
parrot 1.00 0.00
snake  1.00 0.00
> prop.table(t,2)
      own
Fpet    boy girl
cat    0.2  1.0
dog    0.4  0.0
parrot 0.2  0.0
snake  0.2  0.0
```

El orden de los niveles puede establecerse usando el argumento `levels` de la función `factor():`

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+ levels = c("yes", "no"))
> x
[1] yes yes no  yes no
```



```
Levels: yes no
```

## 5. Secuencias

R proporciona diferentes funciones para generar secuencias. Por ejemplo, podemos generar secuencias numéricas de la siguiente forma:

```
> x<-1:15
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> y<-5:1
> y
[1] 5 4 3 2 1
```

La función `seq()` puede generar secuencias de números reales:

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

donde el primer argumento indica el principio de la secuencia, el segundo el final y el tercero el incremento que se debe usar para generar la secuencia. También se puede usar:

```
> seq(length=9, from=1, to=5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Otra función útil para generar secuencias con un cierto patrón es `rep()`:

```
> rep(5,5)
[1] 5 5 5 5 5
```

La función `gl()` permite generar secuencias que involucran factores. `gl(k,n)` genera una secuencia donde `k` es el número de niveles del factor y `n` el número de repeticiones de cada nivel:

```
> gl(2,4,labels=c("boy","girl"))
[1] boy boy boy boy girl girl girl girl
Levels: boy girl
```

También podemos generar secuencias aleatorias. Para ello R proporciona diferentes funciones con la forma general `rfunc(n, p1, p2,...)` donde `r` puede tomar los valores `d`, `p`, `q` para obtener la densidad de probabilidad, la densidad de probabilidad acumulada, el valor del cuartil, respectivamente; `func` indica la distribución (`norm` para normal, `binom` para binomial, `beta` para beta, `unif` para uniforme, ...); `n` es el número de datos generado, y `p1`, `p2`, ... son valores que toman los parámetros de la distribución.

Distribution	rfunc
Gaussian (normal)	<code>rnorm(n, mean=0, sd=1)</code>
exponential	<code>rexp(n, rate=1)</code>
gamma	<code>rgamma(n, shape, scale=1)</code>
Poisson	<code>rpois(n, lambda)</code>
beta	<code>rbeta(n, shape1, shape2)</code>
binomial	<code>rbinom(n, size, prob)</code>
geom	<code>rgeom(n, prob)</code>
uniform	<code>runif(n, min=0, max=1)</code>

```
> runif(5)
[1] 0.9111758 0.4921438 0.3238787 0.4402546 0.1792024
> rnorm(10, mean=5, sd=1)
[1] 4.481318 4.319482 4.105529 7.498646 5.668317
4.046477 6.629298 4.627495 6.276711 4.497717
```

## 6. Matrices y Arrays

En matemáticas una matriz es una estructura bidimensional. Sin embargo, en R una matriz es realmente un vector con un atributo adicional (`dim`), el cual a su vez es un vector numérico de longitud 2, que define el número de filas y columnas de la matriz, es decir su “estructura”. Las matrices se definen como sigue:.

```
%matrix(data, nrow, ncol, byrow, dimnames)
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

donde `byrow` indica si los valores deben llenar la matriz por columnas (`TRUE`) o filas (`FALSE`), y `dimnames` permite poner nombre a las filas y columnas. Veamos un ejemplo,

```
> matrix(1:6, 2, 3, byrow=TRUE)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Las matrices pueden crearse a partir de vectores dando los valores apropiados al atributo `dim`:

```
> x<-1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> dim(x) <- c(2, 5)
> x
      [,1] [,2] [,3] [,4] [,5]
```

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

Podemos trasponer una matriz con `t`:

```
> t(x)
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

También podemos construir matrices uniendo vectores o matrices por filas (`rbind`) o columnas (`cbind`):

```
> x<-1:3
> y<-10:12
> cbind(x,y)
      x  y
[1,]  1 10
[2,]  2 11
[3,]  3 12
> rbind(x,y)
      [,1] [,2] [,3]
x       1    2    3
y      10   11   12
```

Observa que en las matrices se usa la notación de índices usual `[i,j]`. Podemos especificar ambos índices, uno o ninguno:

```
x<-matrix(1:4,2,2)
> x[1,2]
[1] 3
> x[1,]
[1] 1 3
> x[,2]
[1] 3 4
> x[,]
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Finalmente, los arrays son generalizaciones multidimensionales de las matrices, con más de dos dimensiones. Si los datos no son suficientes para llenar todas las dimensiones, éstos se replican:

```

> a<-array(1:10,c(3,2,2))
> a
, , 1
      [,1] [,2]
[1,]     1     4
[2,]     2     5
[3,]     3     6

, , 2
      [,1] [,2]
[1,]     7    10
[2,]     8     1
[3,]     9     2

```

## 7. Listas

Las listas son un tipo especial de vectores que contienen elementos de diferentes clases. Los valores pueden indexarse por nombres.

```

> animal <-list(orden='carnivoros',familia='felinos',
+ nombre='lince')
> animal
$orden
[1] "carnivoros"
$familia
[1] "felinos"
$nombre
[1] "lince"

```

Podemos extraer los elementos de una lista usando el siguiente esquema de indexado:

```

> animal[[1]]
[1] "carnivoros"

```

Observa que hemos usado dobles corchetes para acceder al elemento. Si usamos corchetes simples obtenemos un resultado diferente:

```

> animal[1]
$orden
[1] "carnivoros"

```

`animal[1]` extrae una sublista formada por la primera componente de `animal`, mientras que `animal[[1]]` extrae el valor de esa primera componente. También podemos acceder a los valores usando el nombre de la componente:

**nombre.lista\$nombre.componente**

```
> animal$orden  
[1] "carnivoros"
```

Podemos añadir a una lista elementos dinámicamente usando nombres. Podemos empezar con la lista vacía, como sigue:

```
> l <- list()  
> l$a <- 3  
> l$b <- 5  
> l$c <- 8  
> l  
$a  
[1] 3  
$b  
[1] 5  
$c  
[1] 8
```

Las listas pueden no tener nombres y estar compuestas por varios tipos de elementos, como sigue:

```
> mylist <- list(3,'a')  
> mylist[[1]]  
[1] 3  
> mylist[[2]]  
[1] "a"  
> mylist[[3]] <- 8.5  
[[1]]  
[1] 3  
[[2]]  
[1] "a"  
[[3]]  
[1] 8.5
```

Podemos tener vectores como elementos de una lista:

```
> animal$clase<-c('vertebrado','mamifero')  
> animal  
$orden  
[1] "carnivoros"  
$familia  
[1] "felinos"  
$nombre  
[1] "lince"
```

```
$clase
[1] "vertebrado" "mamifero"
```

Si las clases de todos los elementos son compatibles (o coercionables) podemos convertir una lista en un vector con la función `unlist(x, recursive=TRUE, use.names=TRUE)`

```
> unlist(animal)
orden      familia  nombre  clase1  clase2
"carnivoros" "felinos" "lince" "vertebrado" "mamifero"
```

## 8. Evitando bucles: apply y otras facilidades

Para vectores, matrices arrays y listas, disponemos de la potencia de aplicar una función a todos los elementos sin necesidad de un bucle.

La función principal es `lapply`, pero hay otras variantes que facilitan su uso: `sapply`, `vapply`, y otras. La más sencilla es `sapply(x, fun, ...)`, donde `x` es una lista (o algo que puede coercionarse a una lista si no lo es, como un vector), `fun` es el nombre de la función y `...` otros argumentos. El resultado es generalmente mostrado como el tipo de datos más simple compatible con el dato resultante de la aplicación de la función. Por ejemplo, podemos redondear todos los elementos de un vector de números reales y devolver como resultado un vector de números enteros.

```
> myvector <- c(3.5, 7.8, 4.2, 2.5)
> sapply(myvector, round)
[1] 4 8 4 2
```

Con listas es preferible usar la función `lapply(x, fun, ...)`, porque `lapply` siempre devuelve una lista.

```
> lapply(animal, length)
$order
[1] 1
$familia
[1] 1
$nombre
[1] 1
$clase
[1] 2
```

A continuación vemos más ejemplos donde podemos ver el tipo del resultado dependiendo de que usemos `lapply` o `sapply`. El segundo ejemplo hace uso de una lista de vectores:

```
> sapply(animals, length)
orden familia nombre clase
1 1 1 1 2
```

```
> x <- list(a = 1:4, b = rnorm(10),
+ c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] -0.06742888
$c
[1] 1.182268
$d
[1] 5.039349

> sapply(x, mean)
      a      b      c      d
2.50000000 -0.06742888 1.18226849 5.03934889
```

De forma similar podemos aplicar funciones a estructuras. Por ejemplo, tenemos varias formas alternativas (sin usar bucles) de contar cuántos elementos hay en un vector que cumplan una cierta condición:

```
> myvector
[1] 3.5 7.8 4.2 2.5
> myvector + 1
[1] 4.5 8.8 5.2 3.5
> myvector < 5
[1] TRUE FALSE TRUE TRUE
> sum(myvector < 5)
[1] 3
> myvector[myvector < 5]
[1] 3.5 4.2 2.5
```

También existe una función especial **which** que devuelve los índices de los elementos que satisfacen cierta condición:

```
> which(myvector < 5)
[1] 1 3 4
```

Y podemos anidar varias cosas en expresiones muy compactas. Por ejemplo, la siguiente expresión añade 1 a los elementos que son menores que 5:

```
> myvec <- myvector
```

```
> myvec[which(myvec < 5)] <- myvec[myvec < 5] + 1
> myvec
[1] 4.5 7.8 5.2 3.5
```

Hay incluso otras formas de hacer la expresión anterior más corta. Por ejemplo, usando la función `ifelse`

```
> myvec <- myvector
> ifelse(myvec < 5, myvec +1, myvec)
> myvec
[1] 4.5 7.8 5.2 3.5
```

En cualquier caso, suele ser mucho más rápido que usar bucles.

## 9. Data frame

Un *data frame* es la estructura de datos más adecuada para almacenar tablas de datos. Son similares a las matrices (ya que son bidimensionales) pero a diferencia de éstas pueden almacenar datos de diferentes tipos. Cada fila en un data frame representa un caso, observación o ejemplo, y las columnas, los atributos. Por ejemplo, podemos crear un data frame como sigue:

```
> d<-data.frame(nombre=c('Ana','Pepe','Manolo','Rosa','Maria'),
+ edad=c(21,34,54,27,41))
```

```
> d
  name age
1  Ana  21
2  Pepe 34
3 Manolo 54
4  Rosa 27
5  Maria 41
```

Vamos a añadir una columna “ciudad” a la tabla. La única restricción es que la columna tenga el mismo número de filas que la tabla a la que se le añade:

```
> ciudad<-c('Valencia','Barcelona','Madrid','Valencia',
+ 'Valencia')
> trabajo<-c('estudiante','comerciante','ingeniero',
+ ''medico','periodista')
> d2<-cbind(ciudad,trabajo,d)
> d2
  ciudad      trabajo nombre edad
1 Valencia estudiante   Ana   21
```



2	Barcelona	comerciante	Pepe	34
3	Madrid	ingeniero	Manolo	54
4	Valencia	medico	Rosa	27
5	Valencia	perodista	Maria	41

Podemos acceder a los valores almacenados en filas o columnas, e incluso seleccionar datos que satisfacen ciertas condiciones. Por ejemplo, podemos consultar los valores del atributo “ciudad”:

```
> d2$ciudad
[1] Valencia Barcelona Madrid Valencia Valencia
Levels: Barcelona Madrid Valencia
```

Observa que al ser valores nominales, R lo ha convertido en un factor. Como en las listas, los dobles corchetes sirven para extraer los valores de un data frame mientras que los corchetes simples sirven para extraer elementos:

```
> d2[[1]]
[1] Valencia Barcelona Madrid Valencia Valencia
Levels: Barcelona Madrid Valencia
> d2[1]
  ciudad
1 Valencia
2 Barcelona
3 Madrid
4 Valencia
5 Valencia
```

Adicionalmente, también podemos seleccionar los nombres y edad de aquellas personas que viven en “Valencia”

```
> d2[d2$ciudad=='Valencia',c("nombre","edad")]
  nombre edad
1 Ana 21
4 Rosa 27
5 Maria 41
```

o seleccionar aquellas personas cuya edad es mayor de 35 años:

```
> d2[d2$edad>35,]
  ciudad nombre edad
3 Madrid Manolo 54
5 Valencia Maria 41
```

Para seleccionar un subconjunto de elementos de la tabla podemos usar la función `subset()`:

```
> subset(d2,ciudad=='Valencia',trabajo:edad)
```

	trabajo	nombre	edad
1	estudiante	Ana	21
4	medico	Rosa	27
5	perodista	Maria	41

Las funciones `ncol()`, `nrow()` nos indican las filas y columnas de un data frame (o un array o matriz):

```
> nrow(d2)
[1] 5
> ncol(d2)
[1] 4
```

Para transformar unas determinadas columnas o crear nuevas variables, podemos proceder redundantemente como:

```
> d3 <- d2
> d3[, 'age'] <- -d3[, 'age']
> d3
```

	city	job	name	age
1	Valencia	student	Anne	-21
2	Barcelona	dealer	Joe	-34
3	Madrid	engineer	Mario	-54
4	Valencia	physician	Rose	-27
5	Valencia	journalist	Mary	-41

O podemos usar la función `transform()` con la que creamos otro data.frame con las características deseadas (de forma mas compacta):

```
> d3<-transform(d, edad=-edad)
> d3
```

	nombre	edad
1	Ana	-21
2	Pepe	-34
3	Manolo	-54
4	Rosa	-27
5	Maria	-41

## 10. Cambiando nombres y valores de los factores (levels)

Podemos asignar o cambiar los nombres de las componentes de un vector, lista, matriz o data frame con la función `names()`:

```
> x <- 1:3
```

```

> names(x)
NULL
> names(x) <- c("first", "second", "third")
> x
  first second  third
    1      2      3
> names(x)
[1] "first" "second" "third"

```

Es también importante saber cambiar valores de los factores (levels), ya que sin ello algunos valores no se pueden modificar, por ejemplo:

```

> d
  name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5  Mary  41
> d[5,1]
[1] Mary
Levels: Anne Joe Mario Mary Rose
> d[5,1] <- "Susan"
Warning message:
In '[<-.factor'('*tmp*', iseq, value = "Susan") :
  invalid factor level, NA generated
> levels(d[,1])[levels(d[,1])=="Mary"] <- "Susan"
> d
  name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5 <NA>  41
> d[5,1] <- "Susan"
> d
  name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5 Susan  41

```

Tenemos que ser muy cuidadosos con estas cosas ya que si un level se usa dos veces y es renombrado todos los valores que usan ese level se pierden.

## 11. Valores Faltantes

R tiene valores infinitos, indefinidos (no son números), faltantes y vacíos. Es importante distinguirlos. Los valores infinitos son completamente aceptables y podemos operar con ellos:

```
> a <- 1/0
> a
[1] Inf
> a + 3
[1] Inf
> -a
[1] -Inf
```

El símbolo NaN denota un valor numérico que *no es un número*. Por ejemplo,

```
> b <- 0/0
> b
[1] NaN
> b + 3
[1] NaN
> b*0
[1] NaN
```

Esto es diferente de un valor faltante, el cual se denota como NA (*no disponible*), el cual es independiente del tipo de datos.

```
> t <- c(T, F, NA, FALSE)
> t
```

Finalmente, el valor NULL denota un objeto vacío y no debe confundirse con NA ni con NaN. Observa que en SQL, NULL representa un valor faltante, pero no es así en R. Los objetos vacíos son útiles para definir variables a las que daremos valor posteriormente.

```
> a <- NULL
> a
> NULL
> a <- c(a, 3) # works because 'a' has been previously defined
> a
[1] 3
> a <- c(a, 5)
> a
[1] 3 5
```

Hay que tener cuidado ya que los valores NULL suelen ser eliminados de los vectores, ya que NULL significa nada.

```
> s<-c(T,F,NULL,FALSE)
> s
[1] TRUE FALSE FALSE
```

En cualquier momento podemos comprobar el valor de un objeto usando `is.infinite()`, `is.finite()`, `is.na()`, `is.nan()` or `is.null()`:

```
> t<-c(T,F,NA,FALSE)
> is.na(t)
[1] FALSE FALSE TRUE FALSE
> is.nan(t)
[1] FALSE FALSE FALSE FALSE
> is.nan(0/0)
[1] TRUE
> is.null(0/0)
[1] FALSE
> is.infinite(0/0)
[1] FALSE
```

Una tarea común en algunos conjuntos de datos es eliminar valores faltantes:

```
> x<- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

En el siguiente ejemplo, usamos `complete.cases()` para seleccionar aquellos pares (or filas en un data frame) que no tienen valores faltantes:

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a",NA,NA,"d","e","f")
> good <- complete.cases(x, y)
> good
[1] TRUE FALSE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 4 5
> y[good]
[1] "a" "d" "f"
> mydf <- data.frame(x,y)
> good2 <- complete.cases(mydf)
> good2
[1] TRUE FALSE FALSE TRUE FALSE TRUE
```

## 12. Ordenar datos

La función `sort()` es la encargada de ordenar, de forma ascendente, los números o los caracteres alfanuméricos. La ordenación utiliza el orden ASCII, por lo tanto los números van antes que las letras mayúsculas y por último las minúsculas:

```
> x<-c("auu","1uu","Auu")
> sort(x)
[1] "1uu" "Auu" "auu"
```

Otras funciones de ordenación son:

- `order()`: devuelve los índices del vector ordenado.
- `unique()`: devuelve los valores diferentes en un vector (eliminando duplicados).
- `duplicated()`: determina qué valores están duplicados en un vector y devuelve un vector lógico.

Por ejemplo, podemos usar estas funciones para obtener las tres personas más jóvenes en un conjunto de datos pequeño:

```
> d
  name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5  Mary  41
> d[order(d[2])[1:3],1]
[1] Anne Rose Joe
Levels: Anne Joe Mario Mary Rose
```

## 13. Cargando y salvando datos en R

Desde un fichero de texto podemos cargar datos tabulados usando las funciones `read.table()`, `read.csv()` o `read.xls()` (este último depende de la versión del fichero Excel, por lo que se recomienda exportar los ficheros excel a un formato estándar, por ejemplo csv):

- Si el archivo de texto cumple las siguientes condiciones, podemos cargarlo directamente con la función `read.table()` (usando los valores de sus atributos por defecto):

- La primera línea del archivo debe contener el nombre de cada atributo.
- En cada una de las líneas siguientes, el primer elemento es la etiqueta de la fila, y a continuación deben aparecer los valores de cada atributo.
- Los valores están separados por el caracter “ ”.

Si el archivo tiene un elemento menos en la primera línea que en las restantes, obligatoriamente será el esquema anterior el que se utilice. Predeterminadamente, los elementos numéricos (excepto las etiquetas de las filas) se almacenan como variables numéricas; y los no numéricos como factores. El caracter usado para indicar comentarios se asume que es el # y que las columnas están separadas por un espacio. Todo ello se puede cambiar usando los siguientes atributos:

- **file**: el nombre del fichero conteniendo los datos.
  - **header**: valor lógico que indica si existe la primera fila de cabecera.
  - **sep**: una cadena que indica cómo están separadas las columnas.
  - **colClasses**: un vector caracter que indica la clase de cada columna.
  - **nrows**: número de filas en el dataset.
  - **comment.char**: una cadena que indica el caracter usado para los comentarios (poner igual a “” si el fichero no contiene líneas comentadas).
  - **skip**: número de líneas a contar desde el principio del fichero que se deben saltar.
  - **stringsAsFactors**: TRUE si las variables de tipo caracter deben codificarse como factores.
- **read.csv()** es como la función anterior salvo que las columnas están separadas por comas.

Ten en cuenta que en algunos países europeos (como España o Francia) y de Latinoaméricaa el punto decimal se representa con una coma y las columnas se suelen indicar con punto y coma. En consecuencia, algunos ficheros generados desde hojas de cálculo pueden estar en este formato. En este caso, **dec** y **sep** deben especificarse.

Al igual que con **read.X()**, hay varias funciones **write.Y()** para escribir ficheros. Puedes acudir a la ayuda de estas funciones o al manual de R para más funciones de carga y escritura de ficheros.

Finalmente, es importante definir el directorio de trabajo desde donde vamos a leer y escribir los ficheros usando:

```
> WORKDIR <- "C:/MyFiles/"
> setwd(WORKDIR)
```

Podemos saber dónde estamos con `getwd()`.

## 14. Gráficas en R

R proporciona una extensa batería de utilidades gráficas estándar así como para que el usuario se construya otras a su medida. Para ver una demostración de gráficos en colores ejecutar `> demo(graphics)`.

La siguiente tabla resume algunas de las funciones gráficas de R.

<code>plot(x)</code>	muestra los valores de <code>x</code> (en el eje <code>y</code> ) ordenados en el eje <code>x</code>
<code>pie(x)</code>	muestra los valores de <code>x</code> en un gráfico circular
<code>boxplot(x)</code>	muestra los valores de <code>x</code> en un gráfico tipo <i>box-and-whiskers</i>
<code>barplot(x)</code>	muestra un histograma de los valores de <code>x</code>
<code>hist(x)</code>	muestra un histograma de las frecuencias de <code>x</code>

Algunos atributos (presentes en varias de las funciones gráficas de la tabla anterior) son:

Atributo	Descripción
<code>add=</code>	si es <code>TRUE</code> superpone el gráfico en el ya existente. Valor por defecto <code>FALSE</code> .
<code>axes=</code>	si es <code>FALSE</code> no dibuja los ejes ni la caja del gráfico. Valor por defecto <code>TRUE</code> .
<code>type=</code>	tipo de gráfico: <code>''p''</code> puntos (valor por defecto). <code>''l''</code> líneas. <code>''b''</code> puntos conectados por líneas. <code>''o''</code> puntos conectados por líneas cubriéndolos. <code>''h''</code> líneas verticales. <code>''s''</code> escaleras, los datos en la parte superior de las líneas verticales. <code>''S''</code> escaleras, los datos en la parte inferior de las líneas verticales.
<code>xlim=, ylim=</code>	límites inferiores y superiores de los ejes.
<code>xlab=, ylab=</code>	títulos en los ejes; deben ser variables de tipo carácter.
<code>main=</code>	título principal; debe ser de tipo carácter.
<code>sub=</code>	sub-título (escrito en una letra más pequeña).

Además de los modificadores anteriores, uno puede siempre añadir a un `plot` elementos gráficos de bajo nivel, como `lines`, `points`, `text`, etc. Los más habituales son:

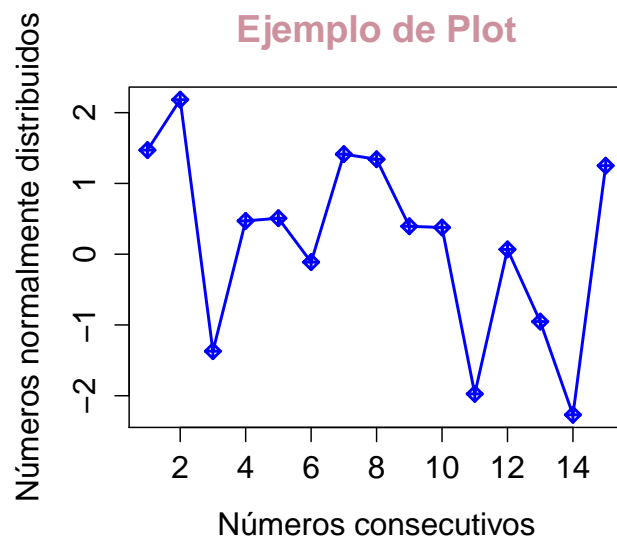


function	description
<b>lines</b>	añade líneas a un plot conectando los puntos por líneas.
<b>points</b>	añade puntos.
<b>text</b>	añade un texto en las coordenadas indicadas.
<b>title</b>	añade título a los ejes, o a todo el gráfico (incluyendo subtítulo).
<b>mtext</b>	añade texto en los márgenes de la gráfica.
<b>axis</b>	añade ejes a un dibujo. Los ejes se disponen usando como argumento: 1=below, 2=left, 3=above y 4=right.

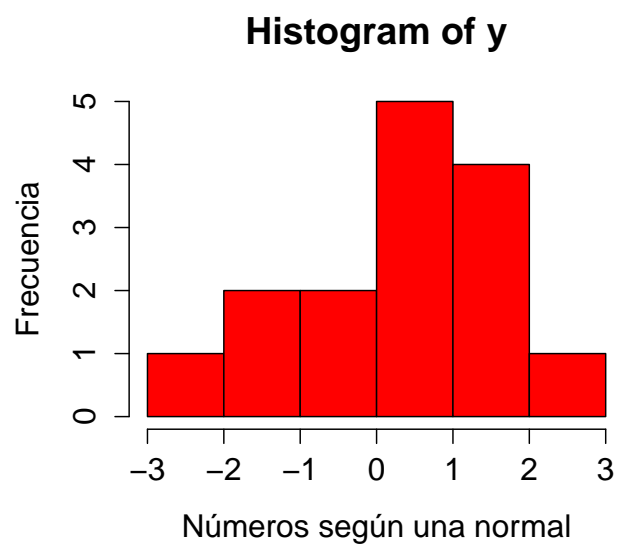
La función **par()** se usa para establecer algunos parámetros gráficos globales que afectarán a todos los gráficos que hagamos durante una sesión de R. Muchos de estos parámetros también pueden especificarse como argumentos de elementos gráficos de alto nivel, en cuyo caso solo afectarán al gráfico en cuestión. Los parámetros más habituales son:

parameter	description
<b>pch</b>	Carácter empleado para dibujar los puntos.
<b>lty</b>	Tipo de línea. Puede especificarse: por números: 1=sólida, 2=guiones, 3=puntos, 4=punto-guión por carácter: "solid", "dashed", "dotted", "dotdashed"
<b>lwd</b>	Grosor de las líneas (numérico, por defecto 1).
<b>col</b>	Color en líneas de ejes, <b>lines()</b> y <b>text()</b> . Se puede especificar como número, string o código hex (ver <b>colors()</b> ).
<b>bg</b>	Color del fondo del gráfico.

A continuación mostramos ejemplos de tres tipos de gráficos usando diferentes parámetros. Se incluye el código, así como el gráfico generado.

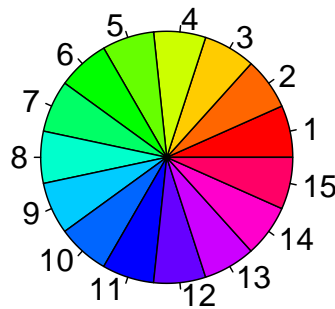


```
>x<-1:15
> y<-rnorm(15)
> plot(x,y,type="o",pch=9,col="blue",lwd=2,xlab=
+"Números consecutivos",ylab="Números normalmente
+ distribuidos")
> title("Ejemplo de Plot",col.main="pink3")
```



```
> hist(y,col="red",xlab="Números según una normal",  
+ ylab="Frecuencia")
```

### Ejemplo gráfico tipo pie



```
> z<-rep(1,15)  
> pie(z,col=rainbow(15))  
> title("Ejemplo gráfico tipo pie",col.main="purple3")
```

Hay muchas otras funciones para hacer gráficos y también muchos paquetes gráficos externos.

## 15. Exportando gráficos en R

Si en lugar de mostrar el gráfico en la pantalla queremos salvarlo en un fichero, disponemos de las siguientes alternativas. La opción más sencilla es desde la consola de R. Desde allí, pulsando el botón derecho del ratón, podemos salvar el gráfico como un metafichero, como postscript o como bitmap. Otra opción también sencilla es desde Rstudio. En la ventana “Plots” podemos salvar el gráfico como una imagen o como un fichero PDF.

Si queremos tener más control sobre cómo hacerlo, podemos abrir un fichero antes de ejecutar las funciones para generar el gráfico, usando las funciones `postscript()`, `pdf()`, `png()`, .... Otra opción es usar cualquier dispositivo de ventanas `x11()` (para Unix/Linux/Windows), `quartz()` (para MAC OS X) o el dispositivo por defecto (la función `dev.cur()` muestra el dispositivo activo).

Con cualquiera de las opciones anteriores, tenemos que ejecutar `dev.off()` al final. El siguiente ejemplo muestra cómo redirigir la salida de un gráfico a un fichero:

```
> WORKDIR <- "C:/MyFiles/"
> setwd(WORKDIR)
> pdf("myfile.pdf", width=7, height=7)
> plot(1:5)
> dev.off()
```

## 16. Estructuras de Control

En R se usan las llaves `{}` para agrupar expresiones. Las estructuras de control más comunes son:

- `if, else`: condicional

```
if(<condition>) {
    ## do something
} else {
    ## do something else
}
if(<condition1>) {
    ## do something
} else if(<condition2>) {
    ## do something different
} else {
    ## do something different
}
```

La parte `else` es opcional.

- Bucles

- `for`: ejecuta un bucle un número determinado de veces. Para ello usa una variable (el *iterador*) y le va asignando valores sucesivos desde una secuencia o un vector con la siguiente sintaxis:  
`for (name in values) expresión`

```
> for(i in 1:10) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
```

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

- **while**: ejecuta un bucle mientras una condición es cierta. La condición se comprueba primero y si es cierta entonces se ejecuta el cuerpo del bucle, se vuelve a comprobar la condición y así sucesivamente.

```
> count <- 0
> while(count < 10) {
+   print(count)
+   count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

- **repeat**: ejecuta un bucle infinitamente. La única forma de salir del bucle es llamando a **break**.
- **break**: termina la ejecución de un bucle.
- **next**: salta una iteración de un bucle.
- **return**: sale de una función.

```
> count <- 0
> repeat{
+   count <- count + 1
+   if (count %% 2 == 0){
+       next
+   } else {
+       print(count)
+   }
+   if (count >= 9) {break}
+ }
[1] 1
[1] 3
[1] 5
[1] 7
```

```
[1] 9
```

Muchas de estas estructuras de control no se usan en las sesiones interactivas sino al escribir funciones.

## 17. Creando funciones

Las funciones en R se crean con la siguiente directiva:

```
function(<arguments>) {## Do something interesting}.
```

En R las funciones pueden pasarse como argumentos de otras funciones y pueden ser anidadas (es decir, son funciones de orden superior), podemos aplicarlas parcialmente, se evalúan de forma perezosa,... Como las funciones son objetos, podemos ver su definición en cualquier momento:

```
> myfun <- function(x, y)
+ { if (x>y) return(-x) else return(y)}
> myfun(5,3)
[1] -5
> myfun
function(x, y) { if (x>y) return(-x) else return(y)}
```

También podemos ver los argumentos de una función con `args(name)`:

```
> args(myfun)
function (x, y)
NULL
```

Algunas funciones tienen un número variable de argumentos. Por ejemplo, la función `paste`:

```
> args(paste)
function (... , sep = " ", collapse = NULL)
NULL
```

Observa el argumento `...`. Este argumento se usa en R para denotar un número variable de argumentos (por ejemplo, indicando que no queremos modificar sus valores por defecto, o cuando extendemos una función). En este caso, `paste` es una función que permite al usuario concatenar cualquier número de strings, o incluso combinar vectores de strings, como se muestra en el siguiente ejemplo:

```
> paste(c('a','b','c'),c(1,2),sep=":")
[1] "a:1" "b:2" "c:1"
>
> paste(c('a','b','c'),c(1,2))
[1] "a 1" "b 2" "c 1"
```

Podemos definir funciones con un número variable de argumentos. Supongamos que queremos concatenar siempre solo dos vectores y con el caracter separador ":". Para ello podríamos definirnos una nueva función a partir de la función predefinida `paste()`:

```
> myfunction<-function(x,y){paste(x,y,sep=":",...)}  
myfunction(c('a','b','c'),c(1,2))  
[1] "a:1" "b:2" "c:1"
```

Y si quisiéramos hacer lo mismo pero para un número variable de vectores:

```
> otherfunction<-function(...){paste(...,sep=":")}  
> otherfunction(c('a','b','c'),c(1,2),rep('AB',4))  
[1] "a:1:AB" "b:2:AB" "c:1:AB" "a:2:AB"
```

Podemos escribir una función directamente en la consola o usando el editor. En este último caso, podemos copiarla y pegarla en la consola o, si la hemos guardado en un fichero, podemos cargar el fichero con `source()`, como cualquier otro programa.

## 18. Paquetes

Finalmente, una de las características más potentes de R es el enorme número de librerías disponibles. Estas librerías están estandarizadas por el sistema CRAN. Hay muchos espejos desde los que descargar los paquetes oficiales. La descarga puede hacerse desde el menú de la consola de R o desde código, como vemos a continuación.

En primer lugar, si queremos usar una librería (paquete) que no tenemos descargado se produce un error:

```
> library(rocc)  
Error in library(rocc) :  
  there is no package called 'rocc'
```

El error se produce porque el paquete no está instalado. Podemos instalarlo manualmente desde el menú de R ("Package" -> "Install Packages") o podemos hacerlo automáticamente usando la función `install.packages()`. Por ejemplo:

```
> install.packages("rocc")  
Installing package into 'C:/Users/....'  
also installing the dependencies 'bitops', ...  
  
trying URL 'https://cran.rstudio.com/....'  
Content type 'application/zip' length 36009 ...  
downloaded 35 KB
```

```
...  
  
package 'bitops' successfully unpacked and MD5 ...  
...  
  
The downloaded binary packages are in  
  C:\Users\...
```

Ahora ya podemos usarlo:

```
> library(rocc)  
Loading required package: ROCR  
Loading required package: gplots  
  
Attaching package: 'gplots'  
The following object is masked from 'package:stats':  
  lowess
```