



**Afi**

Escuela  
de Finanzas

# Programación en R

Curso Introducción al Lenguaje

**Manuel Rueda Álvaro**

[mrueda@afi.es](mailto:mrueda@afi.es)

Abril 2021

# Índice

---

1. Introducción
2. R basics
3. Estructuras de datos en R
4. Funciones
5. Familia *apply*
6. Funciones de utilidad
7. Tratamiento de datos
8. Traps & tips en R
9. Materiales

# 1 | Introducción

# ¿Qué es R?

- R es un lenguaje estadístico *Open Source*.
- Es muy potente, flexible y extensible.
- Utilizado en muchas empresas (Google, Microsoft, Facebook, BBVA, etc...) y universidades.
- Empleado por estadísticos y *data miners* en el desarrollo de *software*.
- A diferencia de las hojas de cálculo tradicionales, en R se escriben sentencias de programación en lugar de las clásicas fórmulas. Es necesario conocer la estructura de los datos.
- Se pueden hacer prototipos con pocas líneas de código.
- R es un lenguaje oportunista, combina programación funcional con elementos de orientación a objetos (clases S3 y S4).

# Historia de R

- R es una implementación del lenguaje estadístico **S** (combinado con el lenguaje de programación **Scheme**).
- S fue desarrollado en los laboratorios de AT&T por **John Chambers** a finales de los 70.
- Las dos principales implementaciones de S son:
  - R
  - S+ (S-PLUS)
- Suele haber varias *releases* al año (normalmente la más importante en Abril):
  - 3.5.3 (Great Truth) 11/03/2019
  - ...
  - 3.6.3 (Holding the Windsock) 29/02/2020
  - ...
  - 4.0.0 (Arbor Day) 24/04/2020

# Ventajas de R

- R es un gran *software* para resolver problemas de análisis de datos. Existen gran cantidad de paquetes para tratamiento de datos, modelización estadística, *data mining* y gráficos.
- Existe una comunidad de usuarios creando paquetes (extendiendo R). Gran parte de estos paquetes están implementados en el propio R. Para obtener un mejor rendimiento computacional, se pueden realizar implementaciones en C, C++ y Fortran.
- R es muy útil para hacer gráficos, analizar datos y obtener modelos estadísticos con datos que caben en la memoria RAM del PC. Existen limitaciones, desde el punto de vista de la memoria, con grandes volúmenes de datos.
- Es muy común utilizar otro lenguaje de programación para preparar los datos:
  - Volúmenes pequeños o medianos: Python, Perl...
  - Volúmenes grandes: Hadoop, Pig, Hive, Spark...

# ¿A qué nos referimos por R?

- Por R nos solemos referir a:
  - El lenguaje de programación.
  - El interprete que ejecuta el código escrito en R.
  - El sistema de generación de gráficos de R.
  - Al IDE de programación en R, o también conocido como **RStudio** (incluye el interprete de R, sistema de gráficos, gestor de paquetes e interfaz de usuario).

# Instalación de R

- Proyecto R: <https://www.r-project.org/>
- Existen instaladores binarios para Windows, Linux y Mac: <http://cran.es.r-project.org/>
- En Linux, lo más cómodo es utilizar el gestor de paquetes:
  - `sudo apt-get install r-base`
  - `sudo yum install r-base`



## *Console mode (I)*

- Para abrir la consola de R, ejecutamos desde la línea de comandos o desde Terminal:
  - `$>R`
- Se abre la consola, la cual permite escribir comandos de forma interactiva. Cada uno de estos comandos recibe el nombre de **expresiones**.
- El interprete de R lee estas **expresiones** y responde con el resultado o con un mensaje de error.
- La interfaz de comandos almacenará los pasos seguidos al analizar los datos.
- Puede ser una forma conveniente de afrontar un nuevo problema.

## Console mode (II)

- Con el comando **history()** se muestra el historial de los comandos que se han introducido durante la sesión de R.
- Para obtener la documentación y ayuda sobre un comando/función: **?comando**
  - Por ejemplo: **?history**
- Los nombres de variables, paquetes, directorios, etc. se autocompletan usando tabulador.
- Si se escribe en la consola el nombre de una función se muestra su código.
  - Por ejemplo: **history**

## Console mode (III)

- Algunos ejemplos para comenzar:

```
> # Creación de un vector de enteros
> c(43, 42, 12, 8, 5)
[1] 43 42 12 8 5
>
> # Creación y asignación de una variable
> edades <- c(43, 42, 12, 8, 5)
>
> # Para ver el contenido de una variable
> edades
[1] 43 42 12 8 5
>
> # Algunas funciones
> sum(edades)
[1] 110
>
> mean(edades)
[1] 22
>
> range(edades)
[1] 5 43
```

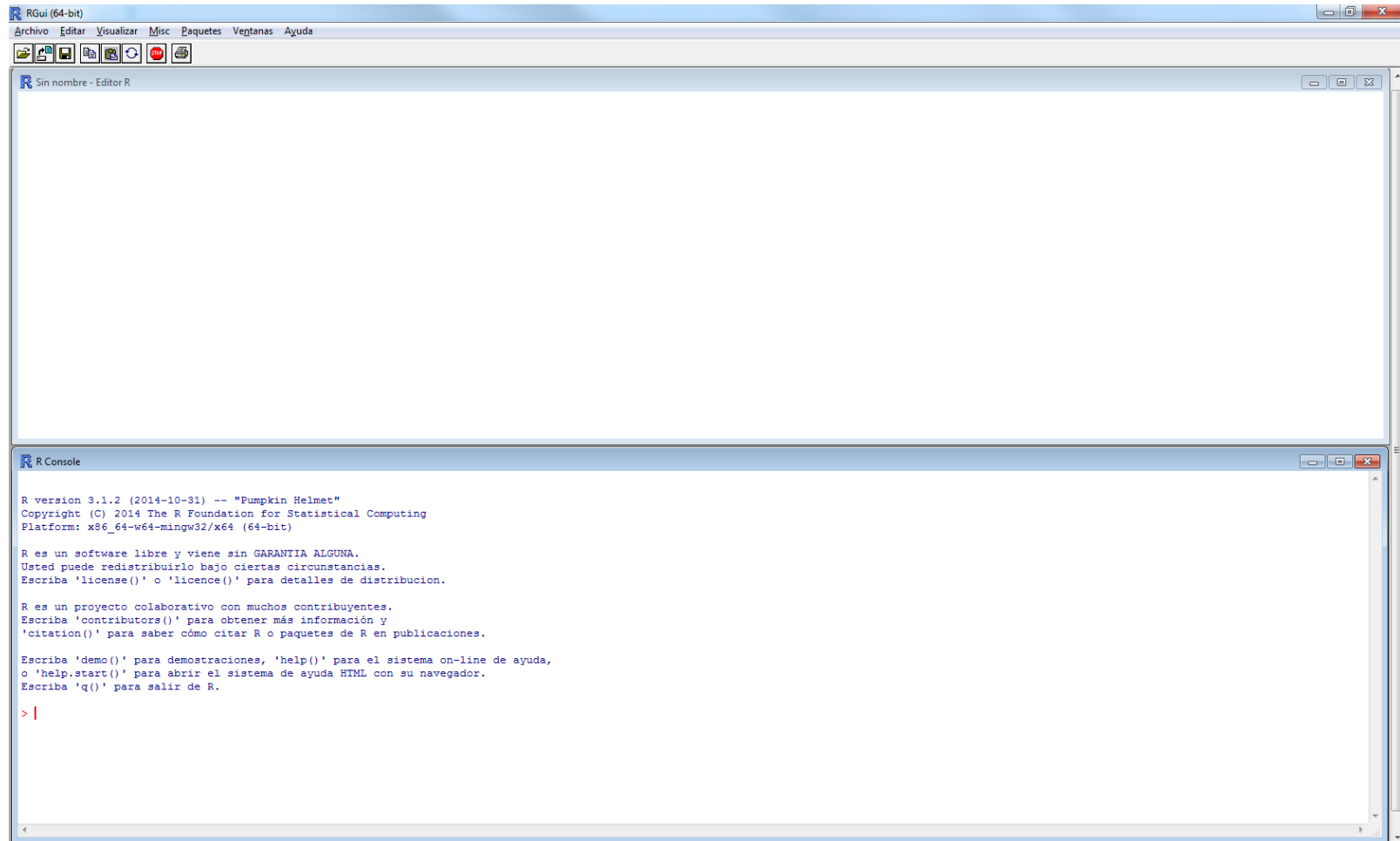
## Console mode (IV)

- Operaciones básicas:

```
> 1+2+3
[1] 6
>
> 1+2*3
[1] 7
>
> (1+2)*5
[1] 15
>
> c(0, 1, 1, 2, 3, 5, 8)
[1] 0 1 1 2 3 5 8
>
> 1:20
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

## Introducción

# Console mode (V)



## Batch mode

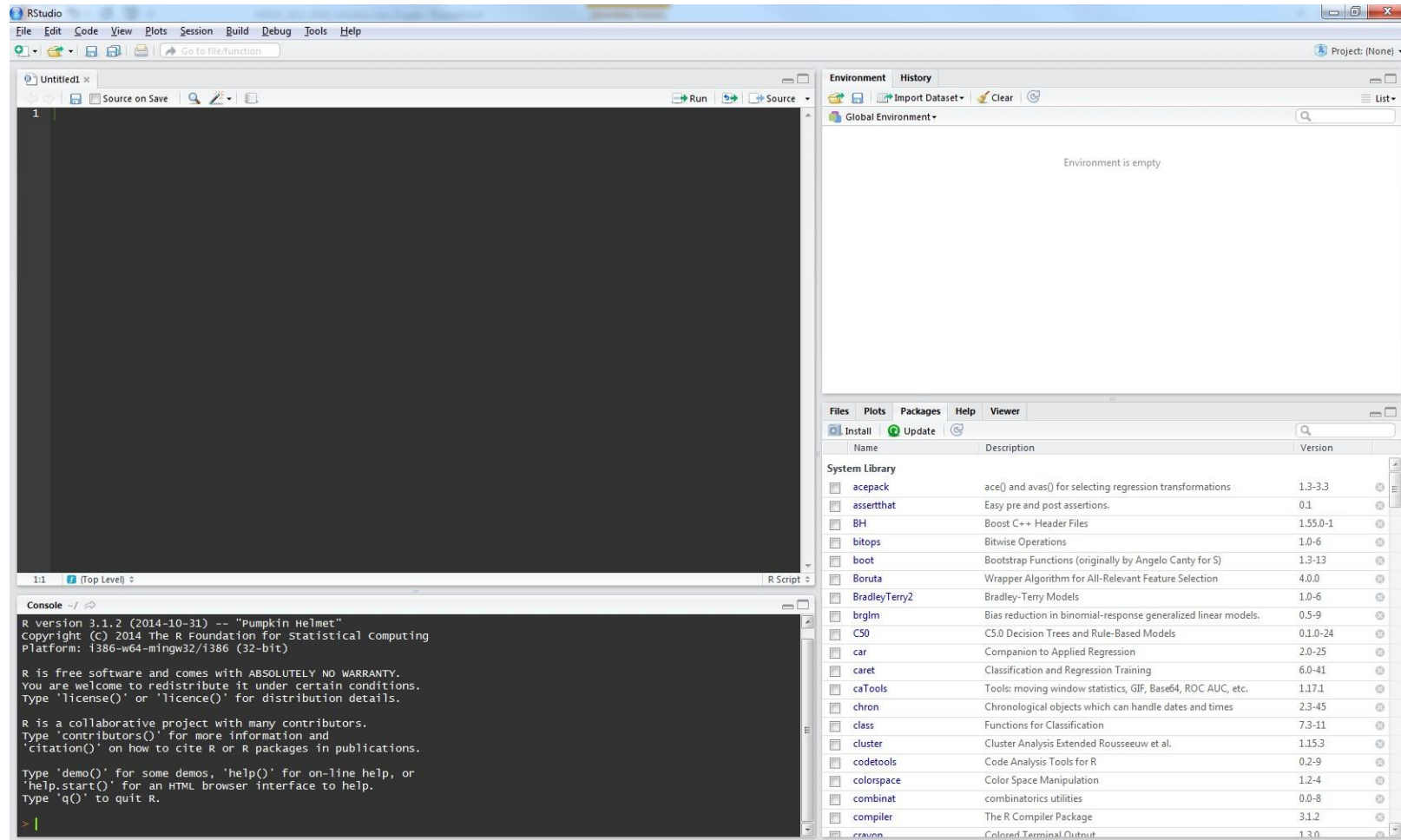
- Los comandos interactivos ejecutados en la consola son muy útiles para una exploración rápida o un análisis *ad-hoc*.
- Si hay que repetir los cálculos *n* veces, la consola no es práctica.
- Para ejecutar en modo *batch*:
  - Crear un fichero R con el código. Por ejemplo: `calcula_resultados.R`
  - Invocar el fichero con el comando `Rscript`. Por ejemplo:
    - `Rscript calcula_resultados.R 2016`
  - Puede recibir parámetros. Por ejemplo: 2016
- Mediante el comando `source(input_path_file)` cargamos el código fuente de otro fichero R.

# RStudio (I)

- IDE de programación para desarrollar proyectos en R: <https://www.rstudio.com/>
- Existen dos versiones:
  - RStudio Desktop
  - RStudio Server (interfaz de RStudio Desktop en versión web)
- Ambas versiones tienen versión *open source* (gratuita) y comercial (con soporte incluido).
- Permite la gestión completa de un proyecto de software:
  - Consola R
  - Gestión de ficheros
  - Ayuda
  - Gestión de paquetes (instalación, actualización, etc.)
  - Revisión del historial de comandos
  - ...

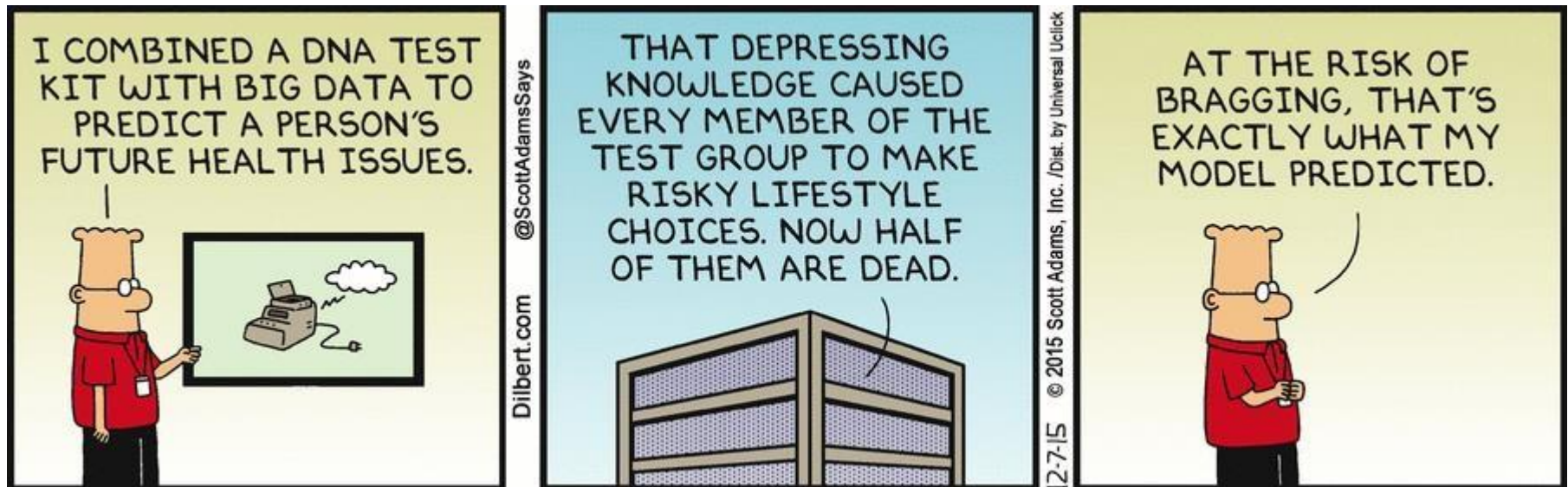
## Introducción

# RStudio (II)





# Machine learning



# 2 | R basics

# Expresiones, objetos y símbolos (I)

- El código R está compuesto de **expresiones (*expressions*)**. Algunos ejemplos de expresiones:
  - Asignaciones
  - Sentencias condicionales
  - Operaciones aritméticas
  - ...
- Las expresiones se componen de objetos y funciones. Cada expresión se separa de otra mediante una nueva línea o un punto y coma (;).
- El código R manipula **objetos (*objects*)**. Algunos ejemplos de objetos:
  - Vectores
  - Listas
  - Funciones
  - ...

# Expresiones, objetos y símbolos (II)

- Formalmente los nombres de las variables en R se llaman **símbolos (*symbols*)**. Es decir, asignamos el objeto a un símbolo del entorno actual.
- El entorno está formado por el conjunto de símbolos presentes en un cierto contexto.
- Los objetos en R tienen modo, clase, y tipo:
  - **Modo (*mode*)**: indica el modo de un objeto por R (como es almacenado según Becker, Chambers & Wilks, 1988).
  - **Tipo (*typeof*)**: indica el tipo “interno” de un objeto R (como es almacenado). Es una versión actualizada del modo. Tipo y modo suelen coincidir pero no siempre es así.
  - **Clase (*class*)**: indica la clase de un objeto R.
- Cuando se cambia el modo/tipo de un objeto se denomina **coerción**. Puede cambiar el modo/tipo, pero no necesariamente la clase. Es lo que usualmente se denomina *casting* en otros lenguajes de programación.

# Tipos de datos básicos en R

Tipo básico	Descripción
<i>logical</i>	Valores booleanos TRUE/FALSE (pueden abreviarse a T/F).
<i>integer</i>	Números enteros (32 bit con signo).
<i>double</i>	Números reales (de doble precisión).
<i>character</i>	Cadenas de caracteres (strings). Los valores van entrecomillados.
<i>complex</i>	Números complejos.

- Tanto el tipo *integer* como *double* tienen siempre modo *numeric*.

# Objetos comunes en R

Tipo objeto	Descripción
<i>vector</i>	Array de una dimensión. Todos los elementos son del mismo tipo básico. Cada elemento puede tener nombre.
<i>matrix</i>	Array numérico de dos dimensiones. Las filas y las columnas pueden tener nombre.
<i>factor</i>	Array de datos ordinales (ordenados) o categóricos (sin orden).
<i>data.frame</i>	Array de dos dimensiones. Cada columna es un vector o factor. Las filas y las columnas pueden tener nombre.
<i>list</i>	Array de objetos. Los elementos de la lista pueden ser de diferentes tipos y tener nombre.

# ¿Cómo determinar la naturaleza de un objeto?

Función	Descripción
<code>typeof(x)</code>	El tipo del objeto x.
<code>mode(x)</code>	El modo del objeto x.
<code>class(x)</code>	La clase del objeto x.
<code>storage.mode(x)</code>	El modo de almacenamiento del objeto x.
<code>attributes(x)</code>	Los atributos de x. Por ejemplo: class y dim.
<code>str(x)</code>	Imprime un resumen de la estructura de x.
<code>dput(x)</code>	Escribe una representación ASCII del objeto x.

# Valores no numéricos

- Existen ciertos valores especiales para representar situaciones determinadas:
  - **NULL**: se utiliza para indicar que una variable no contiene ningún objeto.
  - **NA**: valor especial para indicar que no hay datos en un determinado lugar (vector, *data frame*...)
  - **Inf**: infinito positivo.
  - **-Inf**: infinito negativo.
  - **NaN**: *Not a Number*.
- **Tip**: se puede tener una lista de NULLs pero no un vector de NULLs. En cambio sí se puede tener un vector de NAs.

```
> x <- NULL
> is.null(x)
[1] TRUE
> length(x)
[1] 0
```

```
> y <- NA
> is.na(y)
[1] TRUE
> length(y)
[1] 1
```



# Operadores básicos (I)

Operador aritmético	Descripción
+	Suma.
-	Resta.
*	Multiplicación.
/	División.
$\wedge$ o **	Potencia.
%%	Módulo.
%/%	División entera.

Otros operadores	Descripción
n:m	Generación de secuencias entre dos números.
%in%	Pertenencia.

## Operadores básicos (II)

Operadores relacionales	Descripción
<	Menor que.
<=	Menor o igual que.
==	Igualdad.
>	Mayor que.
>=	Mayor o igual que.
!=	Desigualdad.

## Operadores básicos (III)

Operadores lógicos	Descripción
&	AND para vectores.
&&	AND para no vectores.
	OR para vectores.
	OR para no vectores.
!	NOT.

# Sentencias condicionales

```
# Sentencias condicionales
x <- -3
if (x < 0) {
  print("x es un número negativo")
}

x <- 5
if (x < 0) {
  print("x es un número negativo")
} else {
  print("x es un número positivo o cero")
}

x <- 5
if (x < 0) {
  print("x es un número negativo")
} else if (x == 0) {
  print("x es cero")
} else {
  print("x es un número positivo o cero")
}

ifelse(x > 0, "x es número positivo", "x es un número negativo")

x <- "size"
switch(x, red = "cloth", size = 5, name = "table")
```

# Bucles

```
# Bucles
ctr <- 1
while (ctr <= 7) {
  print(paste("ctr vale", ctr))
  ctr <- ctr + 1
}

ctr <- 1
while (ctr <= 7) {
  if (ctr %% 5 == 0)
    break
  print(paste("ctr vale", ctr))
  ctr <- ctr + 1
}

cities <- c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro", "Cape Town")
for (city in cities) {
  print(city)
}

cities <- c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro", "Cape Town")
for (city in cities) {
  if (nchar(city) == 6)
    next
  print(city)
}

cities <- c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro", "Cape Town")
for (i in 1:length(cities)) {
  print(paste(cities[i], "está en la posición", i, "del vector de ciudades."))
}
```

# Control de errores

```
#####  
# Control de errores #  
#####  
inputs = list(1, 2, 4, -5, "oops", 0, 10)  
  
for(input in inputs) {  
  print(paste("log of", input, "=", log(input)))  
}  
  
for(input in inputs) {  
  try(print(paste("log of", input, "=", log(input))))  
}  
  
for(input in inputs) {  
  tryCatch(print(paste("log of", input, "=", log(input))),  
    warning = function(w) { print(paste("negative argument", input)); log(-input) },  
    error = function(e) { print(paste("non-numeric argument", input)); NaN })  
}
```

# Constantes *built-in*

Constante	Descripción
LETTERS	26 letras del alfabeto en mayúsculas.
letters	26 letras del alfabeto en minúsculas.
month.abb	Abreviatura (3 letras) de los meses en inglés.
month.name	Meses en inglés.
pi	Ratio entre la circunferencia de un círculo y su diámetro.

# Inspección de objetos (I)

Función	Descripción
<code>mode(x)</code>	El modo del objeto x.
<code>class(x)</code>	La clase del objeto x.
<code>typeof(x)</code>	El tipo del objeto x.
<code>dput(x)</code>	Escribe una representación ASCII del objeto x.
<code>str(x)</code>	Imprime un resumen de la estructura de x.
<code>summary(x)</code>	Muestra un resumen de los estadísticos básicos de x.
<code>head(x)</code>	Muestra los primeros elementos del objeto x.
<code>tail(x)</code>	Muestra los últimos elementos del objeto x.
<code>attributes(x)</code>	Los atributos de x. Por ejemplo: class y dim.
<code>is.null(x)</code>	Comprueba si el objeto es NULL.



## Inspección de objetos (II)

Función	Descripción
<code>names(x)</code>	Nombre de los elementos (para vectores y listas).
<code>dimnames(x)</code>	Devuelve una lista con los nombres de las filas y las columnas.
<code>colnames(x)</code>	Nombre de las columnas (para matrices y <i>data frames</i> ).
<code>rownames(x)</code>	Nombre de las filas (para matrices y <i>data frames</i> ).
<code>dim(x)</code>	Devuelve un vector con el número de filas y columnas del objeto.
<code>nrow(x)</code>	Número de filas del objeto.
<code>ncol(x)</code>	Número de columnas del objeto.

## Inspección de objetos (III)

Función	Descripción
is.list(x)	TRUE o FALSE dependiendo de si es una lista o no.
is.factor(x)	TRUE o FALSE dependiendo de si es un factor o no.
is.complex(x)	TRUE o FALSE dependiendo de si es tiene el tipo básico <i>complex</i> .
is.character(x)	TRUE o FALSE dependiendo de si es tiene el tipo básico <i>character</i> .
is.matrix(x)	TRUE o FALSE dependiendo de si es un matriz o no.
is.numeric(x)	TRUE o FALSE dependiendo de si es tiene el tipo básico <i>numeric</i> .
is.integer(x)	TRUE o FALSE dependiendo de si es tiene el tipo básico <i>integer</i> .
is.vector(x)	TRUE o FALSE dependiendo de si es un vector o no.
is.data.frame(x)	TRUE o FALSE dependiendo de si es un <i>data frame</i> o no.
is.ordered(x)	TRUE o FALSE dependiendo de si está ordenado o no.

# Entorno de trabajo

Función	Descripción
ls()	Devuelve la lista de variables del entorno.
rm(x)	Elimina una variable del entorno de trabajo.
help(x)/?x	Muestra la ayuda sobre una determinada función y/o paquete.
q()/quit()	Termina la sesión de trabajo.

# Ejercicio 1

- Realiza el ejercicio del fichero *Ejercicio01\_basics.R*



# 3 | Estructuras de datos en R

# Vectores (*vector*)

- Los vectores son *arrays* de una única dimensión. Los índices del *array* van desde 1 hasta la longitud del vector, **length(v)**. Los vectores son también conocidos como vectores atómicos.
- Todos los elementos del vector son del mismo tipo básico:
  - *logical*
  - *integer*
  - *double*
  - *character*
  - *complex*
- Tiene un tamaño fijo que es fijado en su creación.
- La manera más simple de crear un vector es utilizando al función de combinación **c(v1, v2, ...)**.
- Para dar nombre a los elementos de un vector con la función **names(v)**.

# Creación de vectores

```
# Creación de vectores de longitud fija
v1 <- vector(mode = 'logical', length = 4)
v1
v2 <- vector(mode = 'integer', length = 4)
v2
numeric(4)
character(4)

# Usando el operador de secuencia
v3 <- 1:5
v3
v4 <- 1.4:5.4
v4
v5 <- seq(from = 0, to = 1, by = 0.1)
v5

# Usando la función de combinación
v6 <- c(TRUE, FALSE)
v6
v7 <- c(1.3, 7, 7/20)
v7
v8
v9 <- c('black', 'white')
v9
v10 <- c(v1, v3)
v10

# Creación de vector nombres
v11 <- c(a = 1, b = 2, c = 3)
v11
```

# Operaciones sobre vectores

- Cuando se realizan operaciones aritméticas entre dos vectores, R devuelve otro vector con los resultados de la operación elemento a elemento. También es posible realizar operaciones booleanas. La mayoría de las funciones y operaciones son “*vectorizadas*”.

```
a_vector <- c(1, 2, 3)
b_vector <- c(4, 5, 6)

total_vector <- a_vector + b_vector
total_vector
```

- Si los vectores no tienen el mismo tamaño, R repite el menor de ellos tantas veces como sea necesario.

```
total_vector <- a_vector + 1
total_vector
```

- Podemos aplicar funciones a un vector. Por ejemplo: `sum(v)`, `max(v)`, `mean(v)`...

```
sum(total_vector)
max(total_vector)
mean(total_vector)
```



# Indexación de vectores (I)

- Llamamos indexación a la selección de los elementos de un vector. Para hacer dicha selección empleamos los **corchetes [ ]**.
- Existen varias maneras de hacer selección:
  - Indexar con números positivos

```
# Indexando con números positivos
v[1] # Seleccionamos el primer elemento
v[c(1, 1, 4, 5)] # Seleccionamos el primero dos veces, el cuarto y el quinto
v[20:30] # Obtenemos los elementos entre el índice 20 y 30
v[70:100] <- 0 # Asignamos el valor cero a los elementos entre los índices 70 y 100
v[which(v > 30)] # Seleccionamos los índices de los elementos > 30
```

- Indexar con números negativos

```
# Indexando con números negativos
v[-1] # Seleccionamos todos menos el primer elemento
v[-c(1, 3, 4, 5)] # Seleccionamos todos menos el primero, el tercero, el cuarto y el quinto
v[-length(v)] # Todos menos el último
```

# Indexación de vectores (II)

- Indexar con vectores lógicos o expresiones booleanas

```
# Indexando con vectores lógicos o expresiones booleanas
v0 <- v[1:5]
v0[c(TRUE, FALSE, TRUE, FALSE, FALSE)] # Seleccionamos el primero y el tercero
v[v > 30] # Todos los > 30
v[v > 30 & v <= 50] # Todos los > 30 y <= 50
v[v == 0] # Todos los 0
v[v %in% c(10, 20, 30)] # Seleccionamos el 10, 20 y 30
```

- Indexar por nombre

```
# Indexando por nombre
names(v0) <- c("alpha", "beta", "gamma", "delta", "omega")
v0["alpha"]
v0["beta"] <- 500
v0[c("delta", "omega")]
v0[!(names(v0) %in% c("alpha", "beta"))]
```

# Información sobre vectores

Función	Retorno
<code>dim(df)</code>	NULL.
<code>is.atomic(v)</code>	TRUE (si solo contiene elementos del mismo tipo).
<code>is.vector(v)</code>	TRUE (si es un vector).
<code>is.list(v)</code>	FALSE (si es una lista).
<code>is.factor(v)</code>	FALSE (si es un factor).
<code>is.recursive(v)</code>	FALSE (si contiene una estructura recursiva).
<code>length(v)</code>	Número de elementos en el vector.
<code>names(v)</code>	NULL o un vector de <i>characters</i> con los nombres de cada elemento.

# Traps sobre vectores

- Coerciones de tipos automáticas

```
x <- c(5, 'a') # Convierte 5 a '5'
x <- 1:3
x[3] <- 'a' # Convierte a '1', '2' y 'a'
typeof(1:2) == typeof(c(1, 2))
```

- Operaciones booleanas “vectorizadas” y no “vectorizadas”

```
c(T, F, T) && c(T, F, F) # TRUE !vectorizada
c(T, F, T) & c(T, F, F) # TRUE, FALSE, FALSE
```

## Ejercicio 2

- Realiza el ejercicio del fichero *Ejercicio02\_vectors.R*



# Matrices (*matrix*)

- Las matrices son *arrays* de dos dimensiones, los cuales contienen elementos del mismo tipo. Se organizan en filas y columnas.
- La manera más simple de crear una matriz es utilizando la función **matrix(data, nrow, ncol, byrow)**.
  - **data**: vector con los valores que tendrá la matriz.
  - **nrow**: número de filas deseadas.
  - **ncol**: número de columnas deseadas.
  - **byrow**: define si la matriz se llena por filas o por columnas.
- Igual que en los vectores, podremos dar nombres tanto a las filas como a las columnas con las funciones **rownames(m)** y **colnames(m)**, respectivamente.

# Creación de matrices

```
m1 <- matrix(1:9, byrow = TRUE, nrow = 3)
m1

m2 <- matrix(c(0, -1, 4)) # Crea una matriz con una columna
m2

d1 <- diag(3) # Crea una matriz diagonal 3x3
d1

d2 <- diag(c(1, 2, 3)) # Crea una matriz diagonal y asigna el vector como diagonal
d2

t_m1 <- t(m1) # Traspuesta de m1
e <- eigen(m1) # Lista con autovalores y autovectores
d <- det(m1) # Determinante de la matriz
```

# Operaciones sobre matrices

- Podemos realizar operaciones aritméticas con matrices (sumas, restas, multiplicaciones...) de manera similar a como hemos hecho con los vectores.

```
a_matrix <- matrix(1:9, byrow = TRUE, nrow = 3)
b_matrix <- matrix(11:19, byrow = TRUE, nrow = 3)

total_matrix <- a_matrix + b_matrix
total_matrix
```

- Igual que hicimos con los vectores, podemos realizar la misma operación sobre todos los elementos de la matriz. En este caso, R creará una matriz auxiliar con las mismas dimensiones que la matriz original.

```
total_matrix <- a_matrix + 2
total_matrix
```

- También aplicaremos funciones a una matriz. Por ejemplo: rowSums(m), colSums(m), rowMeans(m), colMeans(m)...

```
rowSums(total_matrix)
colMeans(total_matrix)
max(total_matrix)
```



# Manipulación de matrices

- Para añadir columnas a una matriz se utiliza la función **cbind(m1, m2, ...)**, la cual une matrices y/o vectores por columna.

```
# Unión de matrices por columnas
big_matrix_2 <- cbind(a_matrix, b_matrix)
big_matrix_2

# Unión de matriz y vector por columnas
big_matrix_2 <- cbind(big_matrix_2, c(1, 5, 6))
big_matrix_2
```

- Para añadir filas a una matriz se utiliza la función **rbind(m1, m2, ...)**, la cual une matrices y/o vectores por fila.

```
# Unión de matrices por filas
big_matrix_1 <- rbind(a_matrix, b_matrix)
big_matrix_1

# Unión de matriz y vector por filas
big_matrix_1 <- rbind(big_matrix_1, c(1, 5, 6))
big_matrix_1
```

# Indexación de matrices (I)

- Al igual que en los vectores, utilizaremos los **corchetes [ ]** para indexar matrices. En el caso particular de las matrices usaremos dos números enteros: uno para la fila y otro para la columna **[row, column]**.
- Para seleccionar todos los elementos de una fila o una columna, basta con no incluir ningún número antes o después de la coma, respectivamente. Por ejemplo: `matrix[row, ]`, `matrix[, col]`.
- Se puede hacer selección de varias maneras:
  - Indexar con números positivos

```
# Indexando con números positivos
m[1, ] # Seleccionamos la primera fila
m[1:2, ] # Seleccionamos las dos primeras filas
m[, 3] # Seleccionamos la última columna
m[, c(1, 3)] # Seleccionamos la primera y la última columna
m[1, ] <- 0 # Asigna un vector de ceros a la primera fila
```

# Indexación de matrices (II)

- Indexar con números negativos

```
# Indexando con números negativos  
m[-1, ] # Seleccionamos todas las filas menos la primera  
m[-nrow(m), -ncol(m)] # Quitamos la última fila y la última columna
```

- Indexar con vectores lógicos o expresiones booleanas

```
# Indexando con vectores lógicos o expresiones booleanas  
m_selection <- matrix(c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, FALSE, TRUE), byrow = TRUE, nrow = 3)  
m[m_selection] # Seleccionamos en función de una matriz de booleanos  
m[m > 7] # Todos los > 7  
m[m == 0] # Todos los 0
```

- Indexar por nombre

```
# Indexando por nombre  
colnames(m) <- c("c1", "c2", "c3")  
rownames(m) <- c("r1", "r2", "r3")  
m[, c("c1", "c3")] # Selección de columnas por nombre  
m[c("r2", "r3"), c("c1", "c2")] # Selección de filas y columnas por nombre
```

# Información sobre matrices

Función	Retorno
<code>dim(m)</code>	Dimensiones de la matriz (número de filas y columnas).
<code>is.atomic(m)</code>	TRUE (si solo contiene elementos del mismo tipo).
<code>is.vector(m)</code>	FALSE (si es un vector).
<code>is.list(m)</code>	FALSE (si es una lista).
<code>is.factor(m)</code>	FALSE (si es un factor).
<code>is.recursive(m)</code>	FALSE (si contiene una estructura recursiva).
<code>length(m)</code>	Número de elementos en la matriz.
<code>nrow(df)</code>	Número de filas.
<code>ncol(df)</code>	Número de columnas.
<code>colnames(m)</code>	NULL o un vector de <i>characters</i> con los nombres de las columnas.
<code>rownames(m)</code>	NULL o un vector de <i>characters</i> con los nombres de las filas.

## Ejercicio 3

- Realiza el ejercicio del fichero *Ejercicio03\_matrix.R*



# Arrays (*array*)

- Para crear un *array* de más de dos dimensiones utilizaremos la función **array(data, dim)**.
  - **data**: vector con los valores que tendrá el *array*.
  - **dim**: vector con las dimensiones del *array*.
- Los vectores y las matrices son casos especiales de *arrays*. Los vectores con una dimensión y las matrices con dos.

```
a <- array(1:8, dim=c(2, 2, 2))  
a  
  
m <- array(1:9, dim=c(3, 3))  
m
```

# Factores (*factor*)

- Los factores (*factors*) son un tipo de estructura de datos para almacenar variables categóricas.
- Recordatorio:
  - **Variable categórica:** aquella que puede tomar un número limitado de posibles valores (categorías). Ejemplos de variables categóricas: genero de una persona, nacionalidad...
    - Variable categórica nominal: aquella que no tiene un orden preestablecido.
    - Variable categórica ordinal: aquella que tiene un orden establecido.
  - **Variable continua:** aquella que puede tomar un número infinito de posibles valores. Ejemplos de variables continuas: peso de una persona, estatura de una persona...
- A los distintos valores que puede tomar la variable se les denomina niveles, ***factor levels***.
- ¿Por qué usar factores?
  - Permiten establecer un orden distinto al alfabético.
  - Muchos modelos/paquetes de R los emplean.

# Creación de factores

```
# Creación de factor sin orden
gender_vector <- c('M', 'F', 'F', 'M', 'M', 'F')
gender_factor <- factor(gender_vector)
gender_factor

# Creación de factor con orden (pero sin especificar un orden)
size_vector <- c('S', 'L', 'M', 'L', 'S', 'M')
size_factor <- factor(size_vector, ordered = TRUE) # L < M < S
size_factor

# Creación de factor con orden (especificando el orden)
size_vector_2 <- c('S', 'L', 'M', 'L', 'S', 'M')
size_factor_2 <- factor(size_vector_2, ordered = TRUE, levels = c("L", "M", "S")) # L < M < S
size_factor_2

# Creación de factor especificando etiquetas
gender_levels_2 <- c('M', 'F', '-') # Como se leen los datos a la entrada
gender_labels_2 <- c('Male', 'Female', NA) # Como se etiquetan
gender_vector_2 <- c('M', 'F', 'F', 'M', 'M', '-')
gender_factor_2 <- factor(gender_vector_2, levels = gender_levels, labels = gender_labels)
gender_factor_2
```



# Operaciones sobre factores (I)

- No es posible realizar operaciones aritméticas con factores.
- Pero sí es posible realizar operaciones booleanas (comparaciones).

```
# Comprobaciones en factors sin orden (solo =)  
gender_factor[1] == gender_factor[2]  
gender_factor[1] == size_factor[2] # ERROR: solo se pueden comparar si son del mismo tipo  
  
# Comprobaciones en factors con orden (se puede >, <...)  
size_factor[1] > size_factor[2]  
gender_factor[1] < gender_factor[2] # ERROR: solo se pueden comparar si son del mismo tipo
```

# Operaciones sobre factores (II)

```
# Obtener los niveles
levels(size_factor)
levels(size_factor)[1]

# Comprobar la existencia de niveles
any(levels(size_factor) %in% c('L', 'S'))

# Añadir nuevos niveles
levels(size_factor)[length(levels(size_factor)) + 1] <- 'XL'
levels(size_factor) <- c(levels(size_factor), 'XS')

# Reordenar niveles
levels(size_factor)
size_factor <- factor(size_factor, ordered = TRUE, levels(size_factor)[c(4, 1:3, 5)])

# Cambiar/re-nombrar niveles
levels(size_factor)[1] <- 'ExtraL'

# Eliminar niveles no utilizados
size_factor <- size_factor[drop = TRUE]
droplevels(size_factor)

# Unir factores
a <- factor(1:10)
b <- factor(letters[a])
union <- factor(c(as.character(a), as.character(b)))
cross <- interaction(a, b)
# ambos producen un conjunto no-ordenado de factors.
# levels: union: 20; cross: 100
# Items: union: 20; cross: 10
```

# Información sobre factores

Función	Retorno
<code>dim(f)</code>	NULL.
<code>is.atomic(f)</code>	TRUE (si solo contiene elementos del mismo tipo).
<code>is.vector(f)</code>	FALSE (si es un vector).
<code>is.list(f)</code>	FALSE (si es una lista).
<code>is.factor(f)</code>	TRUE (si es un factor).
<code>is.recursive(f)</code>	FALSE (si contiene una estructura recursiva).
<code>length(v)</code>	Número de elementos en el factor.
<code>is.ordered(f)</code>	TRUE o FALSE (dependiendo de si esta ordenado o no).
<code>mode(f)</code>	"numeric".
<code>class(f)</code>	"factor".
<code>typeof(f)</code>	"integer".

# Traps sobre factores

- Al leer ficheros con *datasets* las cadenas de caracteres se convierten automáticamente a factores. Al utilizar **read.table** y **read.csv** usar el parámetro **stringsAsFactors = FALSE**.
- Si los números de un fichero de factorizan se puede revertir con:  
`as.numeric(levels(f))[as.integer(f)]`
- Evitar NA's en factores, suelen causar problemas.
- La coerción de un objeto a factor suele ser una fuente de problemas.

## Ejercicio 4

- Realiza el ejercicio del fichero *Ejercicio04\_factors.R*



## *Data frame (data.frame)*

- Como vimos antes, una matriz sólo puede contener elementos del mismo tipo. Debido a esta limitación surgen los **data frames**.
- Normalmente un **data frame** contendrá un *dataset*, con las variables como columnas y las observaciones como filas.
  - Columnas: atributos, variables
  - Filas: observaciones, casos, instancias
- Internamente, R maneja los **data frames** como listas de vectores o factores, todas de la misma longitud.
- Podemos asimilarlos a una hoja de cálculo tradicional.
- Para crear *data frames* usaremos la función **data.frame(x, y, ...)**, la cual recibe como parámetros las columnas del *dataset*.

# Creación de *data frames*

- Es posible crear *data frames* “a mano”, pero no es la práctica habitual.

```
# Creación de data frame vacío  
empty <- data.frame()  
  
# A partir de dos vectores  
c1 <- 1:10 # vector de enteros  
c2 <- letters[1:10] # vector de strings  
df <- data.frame(col1 = c1, col2 = c2)
```

- Normalmente leeremos los *datasets* desde ficheros con las funciones `read.csv(filename)` o `read.table(filename)`.

```
# Lectura desde fichero  
df <- read.csv('filename.csv', header = T)
```

# Operaciones sobre *data frames*

- Análisis exploratorio de los datos de un *data frame*:
  - `head(df)`: devuelve las primeras observaciones.
  - `tail(df)`: devuelve las últimas observaciones.
  - `str(df)`: muestra de forma rápida la estructura de la información almacenada.
    - Número total de observaciones.
    - Número total de variables.
    - Lista con todos los nombres de las variables.
    - El tipo de cada variable.
    - Las primeras observaciones de cada variable.
  - `summary(df)`: muestra los estadísticos básicos de cada variable.

```
head(mtcars)
head(mtcars, 10)
head(mtcars, -10)

tail(mtcars)
tail(mtcars, 10)
tail(mtcars, -10)

str(mtcars)

summary(mtcars)
```



# Manipulación de *data frames*

- Para unir *data frames* la mejor manera es usar la función **rbind(df1, df2, ...)**, obteniendo un *data frame* resultante con más filas.

```
# Añadir filas
df <- rbind(mtcars, data.frame(mpg = 22, cyl = 5, disp = 202, hp = 100, drat = 2.56, wt = 3.1,
                              qsec = 15, vs = 1, am = 0, gear = 5, carb = 4, row.names=c("seat")))
```

- Para añadir columnas podemos emplear la función **cbind(df1, df2, ...)**, pasando como parámetros otro *data frame* o un vector. También existen otras maneras de añadir columnas.

```
# Añadir columnas
df$newcolumn <- rep(1, nrow(df))
df[, 'copyofhp'] <- df$hp
df$hp.gear <- df$hp / df$gear
v <- 1:nrow(df)
df <- cbind(df, v)
```

# Indexación de *data frames* (I)

- Al igual que sucedía con las matrices, utilizaremos los **corchetes [ ]** para indexar *data frames*. Emplearemos dos números enteros: uno para la fila y otro para la columna **[row, column]**.
- Podemos aplicar lo aprendido al indexar matrices para indexar *data frames*.
- Existe una manera rápida de seleccionar una columna, utilizando la expresión `df$column` ó `df["column"]` ó `df[1]`.
- Al indexar podemos obtener vectores o *data frames* dependiendo como lo hagamos.
  - Al seleccionar filas obtenemos siempre *data frames*.
  - Al seleccionar múltiples columnas obtenemos siempre *data frames*.
  - Al seleccionar columnas individuales podemos obtener *data frames* o vectores.

# Indexación de *data frames* (II)

- Indexación de celdas

```
# Indexando celdas
df <- data.frame(mtcars)
str(df)
df[5, 2] # obtiene una única celda
df[1:5, 1:2] # obtiene varias celdas
df[1:2, c('gear', 'am')]
df[1:2, c('gear', 'am')] <- 0 # Asignación de celdas
df[1:2, c('gear', 'am')]
```

- Indexación de filas

```
# Indexando filas (siempre devuelve data frames)
df[1, ]
df[-nrow(df), ]
df[1:5, ]
df[(df$hp > 150 & df$hp < 200), ]
subset(df, hp > 150 & hp < 200)

vrow <- as.numeric(as.vector(df[1, ])) # convertimos el resultados de la indexación en vector
```

# Indexación de *data frames* (III)

- Indexación de columnas

```
# Indexando columnas
df$hp # Devuelve un vector
df[, "hp"] # Devuelve un vector
df[, 4] # Devuelve un vector
df["hp"] # Devuelve un data frame con una columna
df[4] # Devuelve un data frame con una columna
df[["hp"]] # Devuelve un vector
df[, c(4, 6)] # Devuelve un data frame
df[, c("hp", "wt")] # Devuelve un data frame
```

# Unión de *data frames*

- Existe una función muy útil en R que nos permitirá unir dos *data frames*. La función **merge** recibe los siguientes parámetros:
  - *x, y*: *data frames* a combinar.
  - *by, by.x, by.y*: permiten especificar las columnas por las que se combinarán ambos *data frames*.
  - *all, all.x, all.y*: permite seleccionar si queremos obtener todas las filas de ambos *data frames* (FULL JOIN), todas las del *data frame x* (LEFT JOIN) o todas las del *data frame y* (RIGHT JOIN).
- Su funcionamiento es similar a los JOIN de SQL.

```
c1 <- 1:10
c2 <- letters[1:10]
c3 <- 5:20
c4 <- letters[5:20]
df.x <- data.frame(col1 = c1, col2 = c2)
df.y <- data.frame(col1 = c3, col2 = c4)

join <- merge(df.x, df.y, by = c("col1"))
left.join <- merge(df.x, df.y, by = c("col1"), all.x = T)
right.join <- merge(df.x, df.y, by = c("col1"), all.y = T)
full.join <- merge(df.x, df.y, by = c("col1"), all = T)
```

# Información sobre *data frames*

Función	Retorno
<code>dim(df)</code>	Dimensiones de la matriz (número de filas y columnas).
<code>is.atomic(df)</code>	FALSE (si solo contiene elementos del mismo tipo).
<code>is.vector(df)</code>	FALSE (si es un vector).
<code>is.list(df)</code>	TRUE (si es una lista).
<code>is.factor(df)</code>	FALSE (si es un factor).
<code>is.data.frame(df)</code>	TRUE (si es un <i>data frame</i> ).
<code>is.recursive(f)</code>	TRUE (si contiene una estructura recursiva).
<code>class(df)</code>	"data.frame".
<code>nrow(df)</code>	Número de filas.
<code>ncol(df)</code>	Número de columnas.
<code>colnames(m)</code>	NULL o un vector de <i>characters</i> con los nombres de las columnas.
<code>rownames(m)</code>	NULL o un vector de <i>characters</i> con los nombres de las filas.

## *Traps sobre data frames*

- Al leer *data frames* desde ficheros normalmente se emplea el argumento **stringsAsFactors = FALSE** para evitar la coerción a factores.
- Se suele evitar emplear nombres en filas y utilizarlos sólo en columnas.
- No utilizar `rbind(df1, df2, ...)` con factores, ya que algunas veces puede dar problemas (coerción).

## Ejercicio 5

- Realiza el ejercicio del fichero *Ejercicio05\_data\_frames.R*





# Listas (*list*)

- Una lista es una colección de diferentes tipos de objetos. Las listas también son conocidas como vectores recursivos.
- Estos objetos pueden ser:
  - Vectores atómicos.
  - Matrices.
  - *Data frames*.
  - Otras listas (cada una de las cuales puede tener una profundidad distinta).
  - ...
- Cada elemento de la lista puede ser de un tipo diferente, no tienen porque ser objetos de la misma clase.
- Al igual que en los vectores los índices de la lista van desde 1 hasta su longitud, **length(l)**.

# Creación de listas

```
my_vector <- 1:10
my_matrix <- matrix(1:9, ncol = 3)
my_df <- mtcars[1:10,]

# Creación de lista sin nombre
l1 <- list(my_vector, my_matrix, my_df)
l1

# Creación de lista con nombre
l2 <- list(vec = my_vector, mat = my_matrix, df = my_df)
l2

# utilizando la función de composición
l3 <- c(l1, l2)
l3
```

# Operaciones sobre listas

- Análisis exploratorio de los datos de una lista:
  - `head(l)`: devuelve los primeros elementos de la lista.
  - `tail(l)`: devuelve los últimos elementos de la lista.
  - `str(l)`: muestra de forma rápida la estructura de la lista. Funciona de manera similar a los *data frames*.

```
str(l2)
head(l2)
tail(t2)
```

- No es posible realizar operaciones aritméticas sobre los elementos de una lista.
- En algunas ocasiones puede resultar útil simplificar una lista, convirtiéndola en un vector atómico. Para ello utilizamos la función **`unlist(l)`**.

# Indexación de listas (I)

- Para indexar los elementos de una lista utilizaremos normalmente **dobles corchetes** `[[ ]]`. También es posible utilizar **corchetes simples** `[ ]` el igual que en los vectores atómicos. La diferencia entre las dos operaciones radica en el tipo de objeto que devuelve la selección:
  - Con **dobles corchetes** obtendremos un objeto de la clase del elemento ubicado en dicha posición.
  - Con **corchetes simples** R devolverá una lista con el objeto ubicado en dicha posición.
- También se puede utilizar el operador **\$** para obtener el mismo resultado que con los **dobles corchetes**.
- Cuando se trabaja con listas, la mayoría de las veces emplearemos **dobles corchetes** `[[ ]]` o **\$**, evitando utilizar **corchetes simples** `[ ]`.
- Por lo demás, la selección de elementos funciona como en el resto de objetos de R que hemos visto hasta ahora.

# Indexación de listas (II)

```
# Selección por índice
l2[2] #Devuelve una lista
l2[[2]] #Devuelve una matriz

class(l2[2])
class(l2[[2]])

# Selección por nombre
l2[["mat"]] #Devuelve una matriz
l2$"mat" #Devuelve una matriz
l2["mat"] # Devuelve una lista

l2[["mat"]][1, ] #Selecciona la primera fila de la matriz
l2[["vec"]][3] #Selecciona el tercer elemento del vector
```

# Información sobre listas

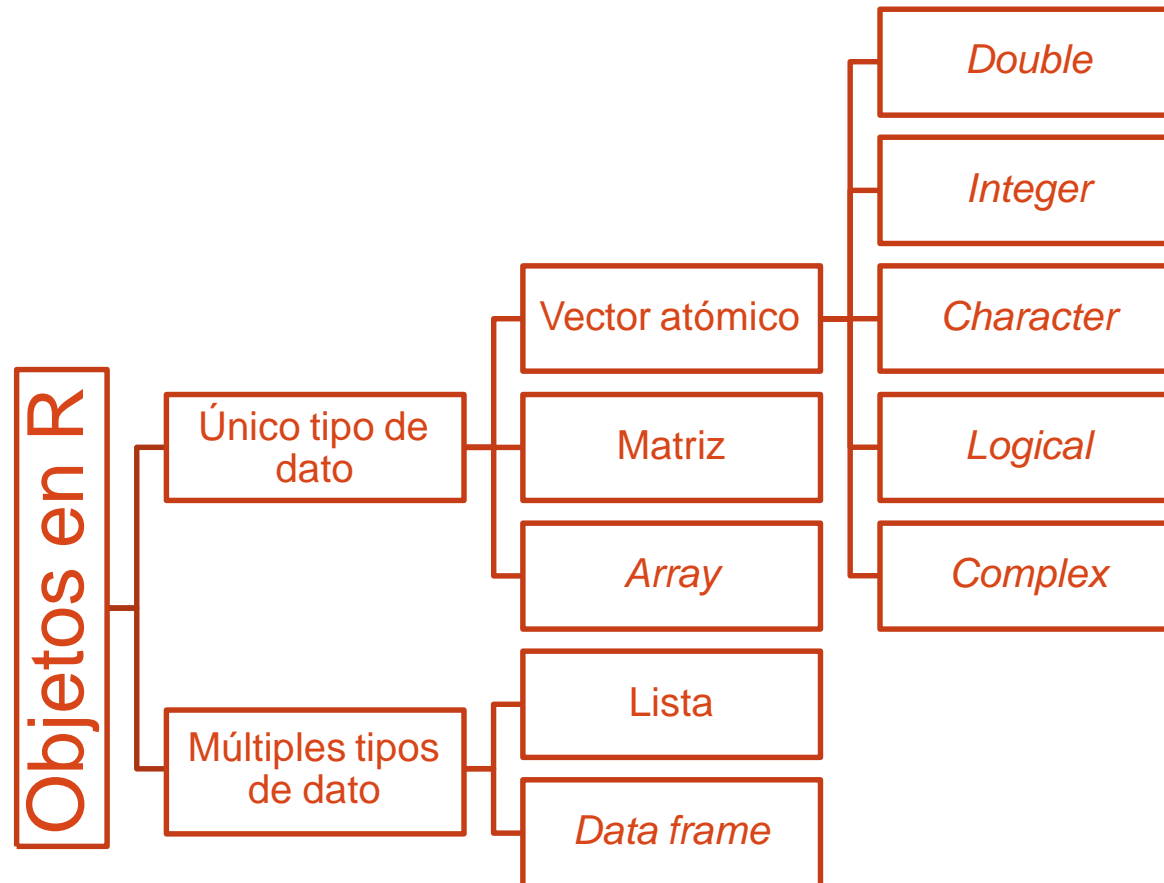
Función	Retorno
<code>dim(df)</code>	NULL.
<code>is.atomic(df)</code>	FALSE (si solo contiene elementos del mismo tipo).
<code>is.vector(df)</code>	TRUE (si es un vector).
<code>is.list(df)</code>	TRUE (si es una lista).
<code>is.factor(df)</code>	FALSE (si es un factor).
<code>is.recursive(f)</code>	TRUE (si contiene una estructura recursiva).
<code>class(df)</code>	"list".
<code>length(v)</code>	Número de elementos en la lista.
<code>names(v)</code>	NULL o un vector de <i>characters</i> con los nombres de cada elemento.

## Ejercicio 6

- Realiza el ejercicio del fichero *Ejercicio06\_listas.R*



# Resumen de objetos en R





# 4 | Funciones

# Funciones (I)

- Las funciones son objetos de R que evalúan un conjunto de argumentos de entrada y devuelven una salida. También son llamadas **closures**.

```
func <- function(x) {  
  #body  
}
```

- Los argumentos son un conjunto de nombres de variables que se emplearán dentro de la función.
- Los argumentos pueden ser:
  - Obligatorios.
  - Opcionales (tienen asignados un valor por defecto).
  - De longitud variable (se accede a cada uno de estos parámetros con ..1, ..2, etc.).
- El retorno de la función se especifica llamando a la función **return(x)**. En algunas ocasiones no es necesario, y se devolverá el resultado de la última expresión evaluada. Es recomendable utilizarlo por legibilidad.

## Funciones (II)

```
do_something <- function(a, b = 1) {  
  if (b == 0)  
    return(0)  
  a * b + a / b  
}  
  
do_something (4)  
do_something (4, 0)
```

- La función anterior tiene un parámetro obligatorio (a) y otro opcional (b).
- El parámetro b al ser opcional, si no es pasado en la llamada, tomará el valor 1.
- La función tiene dos retornos:
  - Cuando b es igual a cero (con return)
  - Cuando b no es igual a cero (sin return)

## Funciones (III)

- Para realizar el retorno de la función hay que utilizar **return(x)**.
  - OJO! En R *return* es una función
- Es posible definir funciones en una única línea. En estos casos las llaves resultan opcionales.

```
pow_two <- function(x) return(x^2)
```

- Dado que las funciones son objetos, se pueden pasar como argumentos a otras funciones (funciones de la familia *apply*). En estos casos es muy común utilizar **funciones anónimas**.
- Para ver los argumentos que recibe una función se puede utilizar la función **args(f)**.

# Ámbito de las variables en las funciones

- Cuando una función es invocada se crea un nuevo entorno (*frame*) para ella.
- Cada *frame* se encuentra dentro de la pila de llamadas, correspondiendo el primer *frame* al entorno global.
- Cada llamada a una función creará un *frame* local.
- Los nombres de las variables dentro de una función se resuelven en el siguiente orden:
  - Entorno local.
  - Entorno padre (funciones definidas dentro de funciones).
  - ...
  - Entorno global.
- De esta manera, una variable que está definida dentro de una función no está disponible fuera de la función (en los entornos superiores, aunque sí en los inferiores).

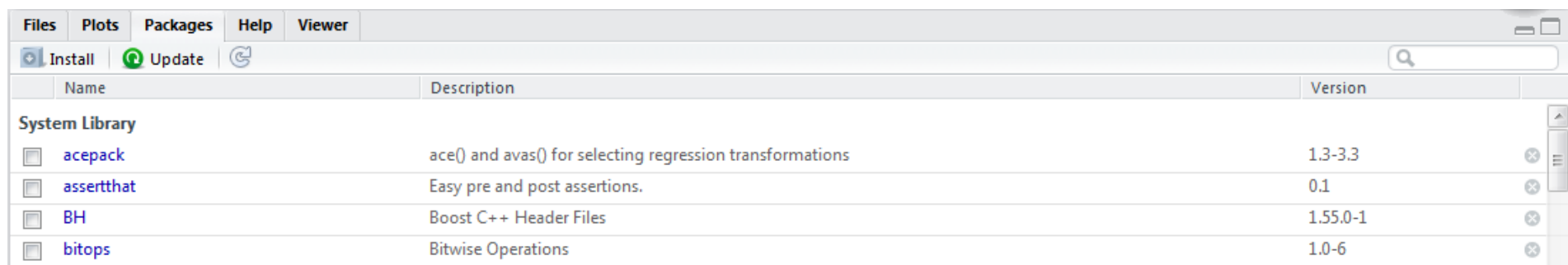
# Argumentos por valor

- R pasa los argumentos por valor. Es decir, una función no puede alterar el valor de la variable que se pasa como argumento en la llamada.
- Se generan copias locales de las variables en el ámbito de la función. Es posible modificar el valor de una variable en los entornos superiores (global) desde un entorno local con el operador de asignación `<-`.
- Los argumentos no tienen tipo, por lo que podemos pasar “cualquier cosa” en la llamada.
- En R no es necesario pasar los argumentos en orden, puesto que los podemos pasar por nombre.

```
my_function <- function(a, b, c) {  
  c(a, b, c)  
}  
  
my_function(1, 2, 3)  
  
my_function(c = 3, a = 1, b = 2)
```

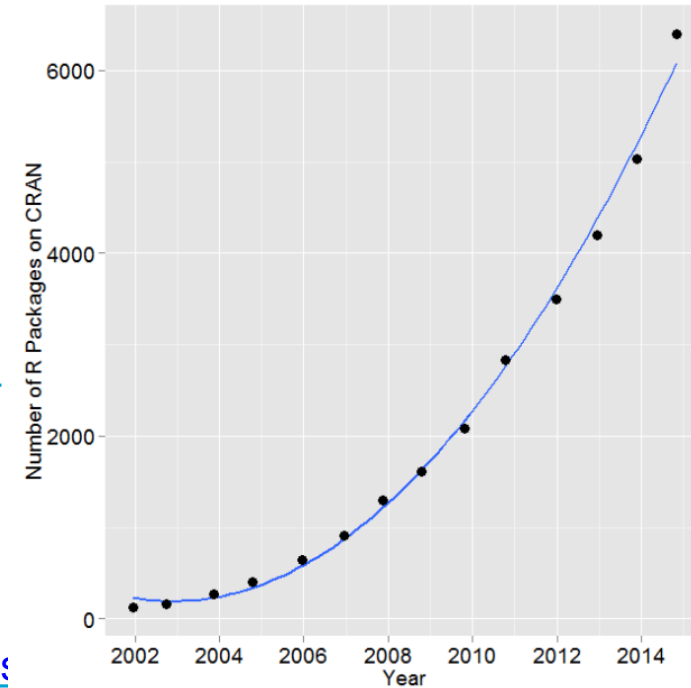
# Paquetes (Packages)

- Un paquete es una colección de funciones, datos y código compilado que han sido empaquetados juntos.
- Similares a las librerías de C/C++ o paquetes de clases de Java.
- Para utilizar un paquete hay que instalarlo y a continuación cargarlo:
  - `install.packages("packageName")`
  - `library(packageName)` o `require(packageName)`
- En RStudio se pueden gestionar los paquetes desde la pestaña Packages.



# CRAN

- Los paquetes de R se almacenan en un repositorio llamado **CRAN** (*Comprehensive R Archive Network*). Existen numerosas URLs para acceder al repositorio.
- Más de 6.000 paquetes disponibles.
- Paquetes por temática: <https://cran.r-project.org/web/views/>
- Paquetes por nombre: [https://cran.r-project.org/web/packages/available\\_packages\\_by\\_name.html](https://cran.r-project.org/web/packages/available_packages_by_name.html)
- Paquetes por fecha de publicación: [https://cran.r-project.org/web/packages/available\\_packages\\_by\\_date.html](https://cran.r-project.org/web/packages/available_packages_by_date.html)
- *Contributed Packages*: <https://cran.r-project.org/web/packages/>





# Comandos sobre paquetes

Comando	Descripción funcionalidad
<code>getOption("defaultPackages")</code>	Lista de paquetes por defecto.
<code>(.packages())</code>	Lista de paquetes cargados en la sesión.
<code>installed.packages()</code>	Devuelve la información de los paquetes instalados actualmente.
<code>available.packages()</code>	Devuelve una lista de los paquetes disponibles en el repositorio.
<code>old.packages()</code>	Devuelve una lista de los paquetes que tienen versiones nuevas disponibles.
<code>new.packages()</code>	Devuelve una lista de los paquetes no instalados disponibles en el repositorio.
<code>download.packages()</code>	Descarga una lista de paquetes.
<code>install.packages()</code>	Instala una lista de paquetes desde el repositorio.
<code>remove.packages()</code>	Desinstala una lista de paquetes.
<code>update.packages()</code>	Actualiza los paquetes instalados a su última versión.
<code>setRepositories()</code>	Cambia la lista actual de repositorios.

# Instalación de paquetes de otros repositorios

- No todos los paquetes están en CRAN.
- Con el paquete **devtools** se instalan paquetes desde otros repositorios populares (como por ejemplo: *Github*).
- Por ejemplo, **Hadley Wickman** utiliza *Github* para el desarrollo del *ggplot2*. Para instalar la última versión de desarrollo de *ggplot2*:
  - `library(devtools)`
  - `install_github("ggplot2")`

# 5 | Familia *apply*

# The *apply* family

- Familia de funciones que realizan una determinada operación a todos los elementos de un vector o una lista. La manera en que se ejecuta dicha operación varía dependiendo de la función que utilicemos:
  - *apply*
  - *lapply*
  - *sapply*
  - *vapply*
  - ...

## Familia *apply*

# `apply(X, MARGIN, FUN, ...)`

- La función **apply** toma la lista o vector `X` y aplica a todos sus elementos la función `FUN` por sus márgenes (`MARGIN`).
- ¿Qué quiere decir que aplica la función a los márgenes?
  - Por filas (1)
  - Por columnas (2)
  - Por filas y columnas (1:2)

```
m <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)
apply(m, 1, mean) # Por filas
apply(m, 2, mean) # Por columnas
apply(m, 1:2, function(x) x/2) # Por filas y por columnas m / 2
```

## lapply(X, FUN, ...)

- La función **lapply** toma la lista o vector X y aplica a todos sus elementos la función FUN. Es posible pasar argumentos (...) a la función FUN en la llamada a lapply.
- El retorno de la función es una lista con el resultado de la ejecución de la función a cada elemento.
- Ya que el resultado es una lista puede devolver objetos de diferente clase.
- Si queremos convertir la lista de retorno en vector podemos utilizar **unlist(l)**.

```
# lapply
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn",
                        "Queens", "Staten Island"),
            capital = FALSE)

# usando bucle
for (info in nyc) {
  print(class(info))
}

# usando lapply
lapply(nyc, class)
```

Familia *apply*

## Ejercicio 7

- Realiza el ejercicio del fichero *Ejercicio07\_lapply.R*



Familia *apply*

## sapply(X, FUN, ...)

- El funcionamiento de **sapply** es análogo a lapply.
- En este caso el retorno es un vector o matriz, si fuera posible. Evitándonos utilizar unlist(l).

```
# sapply
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")

sapply(cities, nchar) # obtenemos un vector
sapply(cities, nchar, USE.NAMES = FALSE)
```

- **Trap:** el retorno de sapply puede ser inesperado, dependiendo de si se puede simplificar el resultado o no.



Familia *apply*

## Ejercicio 8

- Realiza el ejercicio del fichero *Ejercicio08\_apply.R*



Familia *apply*

## vapply(X, FUN, FUN.VALUE, ...)

- El funcionamiento de **vapply** es análogo a `sapply`, pero en este caso podemos especificar el tipo de retorno de la función mediante el parámetro `FUN.VALUE`.
- Es una alternativa segura a `sapply`, ya que tenemos control sobre el retorno de la función.

```
# vapply
cities <- c("New York", "Paris", "London", "Tokyo",
           "Rio de Janeiro", "Cape Town")

vapply(cities, nchar, numeric(1))
```

- Puede decirse que `vapply` es una versión más robusta de `sapply`.

Familia *apply*

## Ejercicio 9

- Realiza el ejercicio del fichero *Ejercicio09\_vapply.R*



Familia *apply*

# Paquete PURR

- Paquete de programación funcional desarrollado por Lionel Henry y Hadley Wickham.
- PURR vs familia *apply*
  - Sintaxis muy concisa y consistente.
  - Optimización del tiempo de ejecución: ligeramente más lento que la familia *apply*.
  - PURR cuenta con funciones específicas para cada tipo de dato de retorno.

Familia *apply*

## map(X, FUN, FUN.VALUE, ...)

- El funcionamiento de map es análogo al de lapply.
- Formas de uso:
  - `map(vector, function(x) f(x, args))`
  - `map(vector, f, args)`
  - `map(vector, ~ f(.x, args))`

```
oil_prices <- list(2.37, 2.49, 2.18, 2.22, 2.47, 2.32)
multiply <- function(x, factor) {
  x * factor
}

unlist(map(oil_prices, function(x) multiply(x, factor = 3)))
unlist(map(oil_prices, multiply, factor = 3))
unlist(map(oil_prices, ~multiply(.x, factor = 3)))
```

Familia *apply*

## map\_xxx(X, FUN, FUN.VALUE, ...)

- map\_xxx funciona de forma análoga a vapply.
- Permite especificar el tipo de dato a devolver.

Función	Descripción
map_chr(x, func)	Devuelve una cadena de caracteres
map_lgl(x, func)	Devuelve un valor booleano
map_int(x, func)	Devuelve un número entero
map_dbl(x, func)	Devuelve un número decimal de precision doble.
map_dfr(x, func)	Devuelve un data.frame

- map\_dfr hace inferencia de tipos, pudiendo dar problemas.

Familia *apply*

## do.call (aunque no es de la familia)

- La función **do.call** permite llamar a cualquier función de R, pero en lugar de pasar todos los argumentos uno por uno (escribiéndolos), se le pasa una lista con los mismos.
- No es una función propia de la familia *apply*.

```
one <- data.frame(a = c("b", "b", "b", "c"),
                 b = c("B", "A", "D", "A"),
                 x = c(2.49778634403711, -0.594683138631719, 1.14857619580259, 1.14857619580259),
                 y = c(-0.351307445767206, 1.19629936975021, 0.653315728121014, 0.935608419299617))

two <- data.frame(a = c("b", "b", "b", "c"),
                 b = c("B", "A", "D", "A"),
                 x = c(2.49778634403711, -0.594683138631719, 1.14857619580259, 1.14857619580259),
                 y = c(-0.351307445767206, 1.19629936975021, 0.653315728121014, 0.935608419299617))

three <- data.frame(a = c("b", "b", "b", "c"),
                   b = c("B", "A", "D", "A"),
                   x = c(2.49778634403711, -0.594683138631719, 1.14857619580259, 1.14857619580259),
                   y = c(-0.351307445767206, 1.19629936975021, 0.653315728121014, 0.935608419299617))

res1 <- rbind(one, two, three)

allframes <- list(one, two, three)

res2 <- do.call(rbind, allframes)
```

# 6 | Funciones de utilidad



# Funciones para estructuras de datos (I)

Función	Descripción
<code>c(x, y)</code>	Concatenación/composición de vectores, listas...
<code>rep(value, n)</code>	Repite el valor n veces.
<code>append(x, value)</code>	Añade un elemento a un vector, lista...
<code>seq(from, to, by)</code>	Genera secuencias entre dos números con un incremento determinado. Otras: <code>seq_along</code> , <code>seq_len</code>
<code>sort(x)</code>	Ordena un vector o factor de manera ascendente o descendente.
<code>order(x)</code>	Devuelve las permutaciones a realizar para ordenar un vector.
<code>rank(x)</code>	Devuelve el ranking de cada elemento de un vector.
<code>rev(x)</code>	Invierte el orden del vector, lista... pasado por argumento.
<code>any(x)</code>	A partir de un vector de valores booleanos comprueba si uno es TRUE.
<code>all(x)</code>	A partir de un vector de valores booleanos comprueba si todos son TRUE.
<code>unique(x)</code>	Elimina duplicados y devuelve los valores únicos.

## Funciones para estructuras de datos (II)

Función	Descripción
<code>which(cond)</code>	Devuelve los índices del vector, lista... que cumplen una determinada condición.
<code>match(value, x)</code>	Devuelve los índices de <code>x</code> que coinciden en valor con <i>value</i> .
<code>as.list(x)</code>	Realiza la coerción de <code>x</code> a lista.
<code>as.factor(x)</code>	Realiza la coerción de <code>x</code> a factor.
<code>as.complex(x)</code>	Realiza la coerción de <code>x</code> a número complejo.
<code>as.character(x)</code>	Realiza la coerción de <code>x</code> a cadena de caracteres.
<code>as.matrix(x)</code>	Realiza la coerción de <code>x</code> a matriz.
<code>as.numeric(x)</code>	Realiza la coerción de <code>x</code> a número real.
<code>as.integer(x)</code>	Realiza la coerción de <code>x</code> a entero.
<code>as.vector(x)</code>	Realiza la coerción de <code>x</code> a vector.
<code>as.data.frame(x)</code>	Realiza la coerción de <code>x</code> a <i>data frame</i> .

# Funciones matemáticas (I)

Función	Descripción
is.finite(x)	Comprueba si no es infinito.
is.infinite(x)	Comprueba si es infinito.
abs(x)	Calcula el valor absoluto.
sqrt(x)	Calcula la raíz cuadrada.
log(x)	Calcula el logaritmo en base e.
log10(x)	Calcula el logaritmo en base 10.
exp(x)	Calcula la exponenciación.
ceiling(x)	Calcula el techo.
floor(x)	Calcula el suelo.
round(x)	Hace el redondeo de la parte decimal.
trunc()	Trunca eliminando la parte decimal.

## Funciones matemáticas (II)

Función	Descripción
sum(x)	Realiza la suma de todos los elemento de un vector.
prod(x)	Realiza el producto de todos los elemento de un vector.
cumsum(x)	Realiza la suma acumulativa de todos los elemento de un vector.
cumprod(x)	Realiza el producto acumulativo de todos los elemento de un vector.
mean(x)	Calcula la media aritmética de un vector.
sin(x)	Calcula el seno.
cos(x)	Calcula el coseno.
tan(x)	Calcula la tangente.
asin(x)	Calcula el arcoseno.
acos(x)	Calcula el arcocoseno.
atan(x)	Calcula el arcotangente.

# Funciones de cadenas de caracteres

Función	Descripción
toString(x)	Devuelve una representación del objeto.
as.character(x)	Realiza la coerción de x a cadena de caracteres.
nchar(x)	Devuelve el tamaño de la cadena.
toupper(x)	Convierte a mayúsculas.
tolower(x)	Convierte a minúsculas.
sub()/gsub()	Reemplaza una subcadena por otra dentro de la cadena de caracteres.
grep()/grepl()	Devuelve la posición de una subcadena en la cadena.
substr()	Devuelve una subcadena de la cadena.
paste()/paste0()	Concatena varias cadenas de caracteres.
format()	Formatea una cadena de caracteres (números, fechas...)
strsplit()	Separa una cadena de caracteres según un determinado carácter.

# Funciones de fechas

Función	Descripción
<code>Sys.Date()</code>	Devuelve un objeto de la clase <code>Date</code> con la fecha del sistema.
<code>Sys.time()</code>	Devuelve un objeto de la clase <code>POSIXct</code> con la fecha y hora del sistema.
<code>as.Date(x, f)</code>	Parsea una fecha según el formato establecido por parámetro.
<code>as.POSIXct(x, f)</code>	Parsea una fecha y hora según el formato establecido por parámetro.
<code>strptime(x, f)</code>	Funcionamiento igual que <code>as.POSIXct</code> .

- Paquetes dedicados a tratamiento de fechas:
  - `lubridate`
  - `zoo`

# 7 | Tratamiento de datos

# Importación

- En la mayoría de casos los datos los encontraremos en diferentes formatos y provenientes de diferentes fuentes de información. Las más habituales:
  - Ficheros de texto plano: csv, tsv,...
  - Excel
  - Otras herramientas estadísticas: SAS, STATA, SPSS
  - BD relacionales
  - ...
- Existen numerosas funciones y paquetes en R para cargar información. Algunos ejemplos:
  - Fichero de texto plano: `utils`, `readr`, `data.table`
  - Excel: `readxl`, `gdata`, `XLConnect`
  - SAS, STATA, SPSS: `haven`, `foreign`
  - BD relacionales: `RMySQL`, `RPostgreSQL`, `ROracle`



# Importando desde ficheros de texto plano (I)

- Dentro del paquete *utils* (el cual se carga por defecto al iniciar la sesión de R) existen varias funciones para leer ficheros de texto plano y cargar su contenido en un *data frame*.
- La función más importante del paquete es `read.table(fileName)`. Dicha función tiene una enorme cantidad de parámetros que nos permitirán configurar la carga según el fichero. Los más importantes:
  - `header`: para indicar si el fichero tiene cabeceras.
  - `sep`: para indicar el separador de nuestro fichero.
  - `stringsAsFactors`: para indicar si las cadenas de caracteres se convertirán en factores.
- La ruta al fichero es relativa al directorio de trabajo.

```
mun1_1 <- read.table("dat/municipios1.csv",  
                     header = TRUE,  
                     sep = ",",  
                     stringsAsFactors = FALSE)
```

# Importando desde ficheros de texto plano (II)

- Existen otras funciones para leer directamente ficheros csv (coma como separador) o tsv (tabulador como separador):
  - Con punto como separador decimal
    - csv: `read.csv`
    - tsv: `read.delim`

```
# Con read.csv
mun1_2 <- read.csv("dat/municipios1.csv", stringsAsFactors = FALSE)
```

- Con coma como separador decimal
  - csv: `read.csv2`
  - tsv: `read.delim2`

```
# Con read.delim2
mun2_2 <- read.delim2("dat/municipios2.tsv", stringsAsFactors = FALSE)
```

## Importando desde ficheros de texto plano (III)

	<i>utils</i>	<i>readr</i>	<i>data.table</i>
Genérico	<code>read.table()</code>	<code>read_delim()</code>	fread
csv	<code>read.csv()</code>	<code>read_csv()</code>	
tsv	<code>read.delim()</code>	<code>read_tsv()</code>	

- *readr* es un paquete desarrollado por **Hadley Wickham** (creador de varios paquetes famosos de R como *ggplot2*).
- Es más rápido cargando datos que el paquete *utils*.
- Por defecto, no convierte las cadenas de caracteres en factores.
- *data.table* es un paquete para manipulación de datos en R (no está creado exclusivamente para leer ficheros).
- *fread* infiere los tipos de las columnas y los separadores.
- Extremadamente rápido.

# Importando desde ficheros Excel

- El paquete más simple para cargar datos desde Excel es *readxl*. Lee ficheros con extensión xls y xlsx. Muy similar a *readr* (mismo creador **Hadley Wickham**).
- Para leer los archivos de Excel nos apoyaremos en dos funciones:
  - `excel_sheets()`: devuelve la lista de hojas disponibles en el archivo.
  - `read_excel()`: realiza la carga. Algunos parámetros:
    - `sheet`: para indicar el número de hoja a cargar.
    - `col_names`: para indicar si la tabla tiene cabeceras.
    - `col_types`: para indicar el tipo básico de las columnas (también se pueden omitir)
    - `skip`: para omitir las primeras filas del fichero.
- Otros paquetes: *gdata* y *XLConnect*.

# Importando desde ficheros SAS, STATA y SPSS

- Es posible cargar *datasets* en R provenientes de otras herramientas estadísticas y de tratamiento de datos.
- Los paquetes *haven* y *foreign* pueden trabajar con ficheros de SAS, STATA y SPSS.
- La siguiente tabla muestra las funciones para leer cada tipo de fichero en función del paquete.

	<i>haven</i>	<i>foreign</i>
SAS	read_sas()	
STATA	read_stata() read_dta()	read_dta()
SPSS	read_spss() read_por() read_sav()	read_spss()

# Importando desde bases de datos relacionales

- Existen diversos paquetes en R para conectar a base de datos. Estos paquetes son específicos para cada *software* de base de datos.
  - MySQL: RMySQL
  - PostgreSQL: RPostgreSQL
  - Oracle: ROracle
  - SQL Server: RODBC
  - Otros paquetes: DBI
- El procedimiento de trabajo es análogo al de otros lenguajes de programación
  - Establecer conexión.
  - Ejecutar la consulta para obtener los datos.
  - Traer los resultados (todo el *resultset* de una vez o fila a fila).
  - Cerrar conexión.
- Hacer consultas a una base de datos nos permite extraer la información de manera selectiva.

# Preparación y limpieza

- Muchos de los *datasets* a los que nos enfrentaremos estarán sucios de una manera o de otra. Por ejemplo:
  - Valores de una variable/atributo codificados como columnas.
  - Variable codificadas en filas y columnas.
  - Más de una variable contenida en la misma columna. Por ejemplo: trimestre y año.
  - Almacenar medidas en distintas unidades en la misma tabla.
  - Contienen *missing values*: valores desconocidos que habrá que eliminar o imputar. OJO→ Es importante saber por qué esos valores son desconocidos.
  - ...
- Al proceso, por el cual, transformamos un *dataset* en otro más conveniente para nuestro análisis, se le llama proceso de limpieza.
- Se suele decir que el proceso de limpieza y preparación de datos se lleva el **80% del tiempo**, mientras que el resto de tareas conllevan sólo el 20%. Es por este motivo, por el que el proceso de preparación es tan **importante**.

# Preparación y limpieza: Explorando

- El proceso de exploración tiene como objetivo crear una composición general de un *dataset*.
- Contiene tres fases:
  - Comprender y asimilar la estructura de los datos.
  - Ver los datos que lo componen.
  - Visualizar los datos que lo componen.
- Mediante estos tres sencillos pasos podremos identificar los primeros “problemas” en nuestros datos.
- Normalmente nuestro *dataset* estará contenido en un *data frame* de R, con las observaciones en las filas y las variables en las columnas.



# Preparación y limpieza: Explorando la estructura

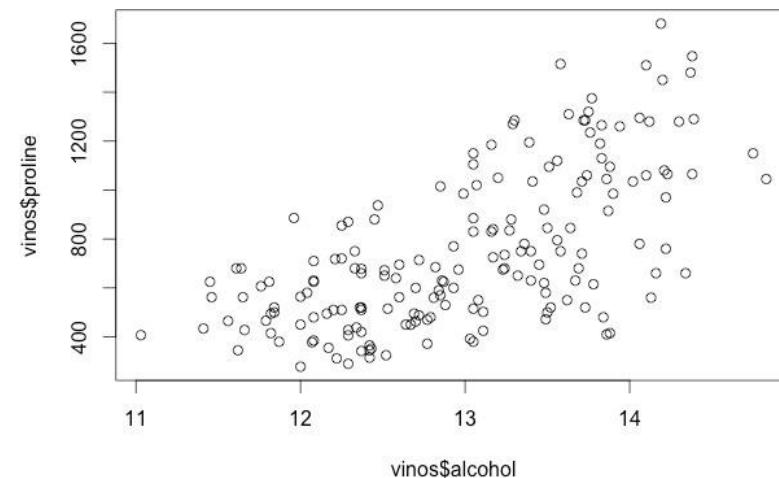
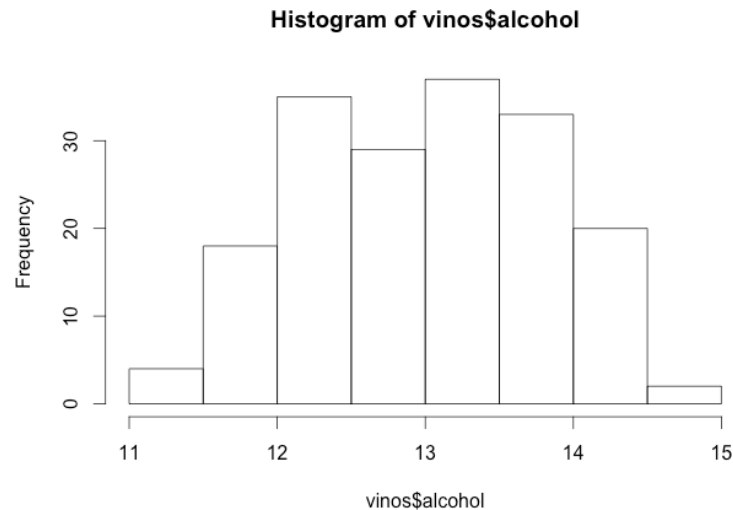
Función	Descripción
class(x)	Normalmente devolverá "data.frame".
dim(x)	Devuelve el número de filas y de columnas del <i>data frame</i> .
names(x)	Devuelve el nombre de las columnas del <i>data frame</i> .
str(x)	Devuelve información sobre la estructura del <i>data frame</i> : Número de filas u observaciones. Número de columnas o variables. Nombre de cada columna o variable. El tipo de cada variable o columna. Primeros valores u observaciones de cada columna.
summary(x)	Devuelve los estadísticos básicos de cada columna o variable: Mínimo. Máximo. Media. Cuartiles (mediana).

# Preparación y limpieza: Explorando los datos

Función	Descripción
head(x, n)	Muestra las primeras observaciones del <i>data frame</i> . Se puede establecer el número de observaciones devueltas con el parámetro n. Por defecto se muestran las seis primeras.
tail(x, n)	Muestra las últimas observaciones del <i>data frame</i> . Se puede establecer el número de observaciones devueltas con el parámetro n. Por defecto se muestran las seis últimas.
print(x)	Muestra el set de datos completo. OJO → Emplearlo sólo con <i>datasets</i> pequeños.

# Preparación y limpieza: Visualizando los datos

Función	Descripción
hist(x)	Muestra un histograma con la distribución de una determinada columna/variable.
plot(x, y)	Muestra un gráfico de puntos con la relación de dos variables.



# Preparación y limpieza: Visualizando los datos

```
# Cargamos el dataset
vinos <- read.csv("dat/wine.csv",
                  header = T)

# Exploración de la estructura
class(vinos)
dim(vinos)
names(vinos)
str(vinos)
summary(vinos)

# Exploración de los datos
head(vinos)
tail(vinos)
print(vinos)

# Visualizando los datos
hist(vinos$alcohol)
plot(vinos$alcohol, vinos$proline)
```

# Preparación y limpieza: Poniendo orden

- ¿Cuáles son los principios de unos datos “limpios”? [Artículo](#) de **Hadley Wickham** en **Journal of Statistical Software**.
- Estos principios son muy parecidos a los que aplicaríamos al diseñar una base de datos relacional.
- Los principios son:
  - Observaciones en filas.
  - Variables y atributos como columnas.
  - Cada fila de la tabla (*data frame*) contiene únicamente una unidad observacional. O lo que es lo mismo, no mezclar distintas entidades o cosas dentro de la misma tabla.
  - Todas las observaciones de una variable están medidas en la misma unidad.
- Los paquetes **tidyr** y **reshape2** de **Hadley Wickham**, contienen varias funciones que nos permitirán *poner orden* en nuestros datos.

# Preparación y limpieza: datos limpios y ordenados

name	age	eye_color	height
Jake	34	Other	6'1"
Alice	55	Blue	5'9"
Tim	76	Brown	5'7"
Denise	19	Other	5'1"

# Preparación y limpieza: valores como columnas

name	age	brown	blue	other	height
Jake	34	0	0	1	6'1"
Alice	55	0	1	0	5'9"
Tim	76	1	0	0	5'7"
Denise	19	0	0	1	5'1"

name	age	eye_color	height
Jake	34	Other	6'1"
Alice	55	Blue	5'9"
Tim	76	Brown	5'7"
Denise	19	Other	5'1"

# Preparación y limpieza: variables en filas y columnas

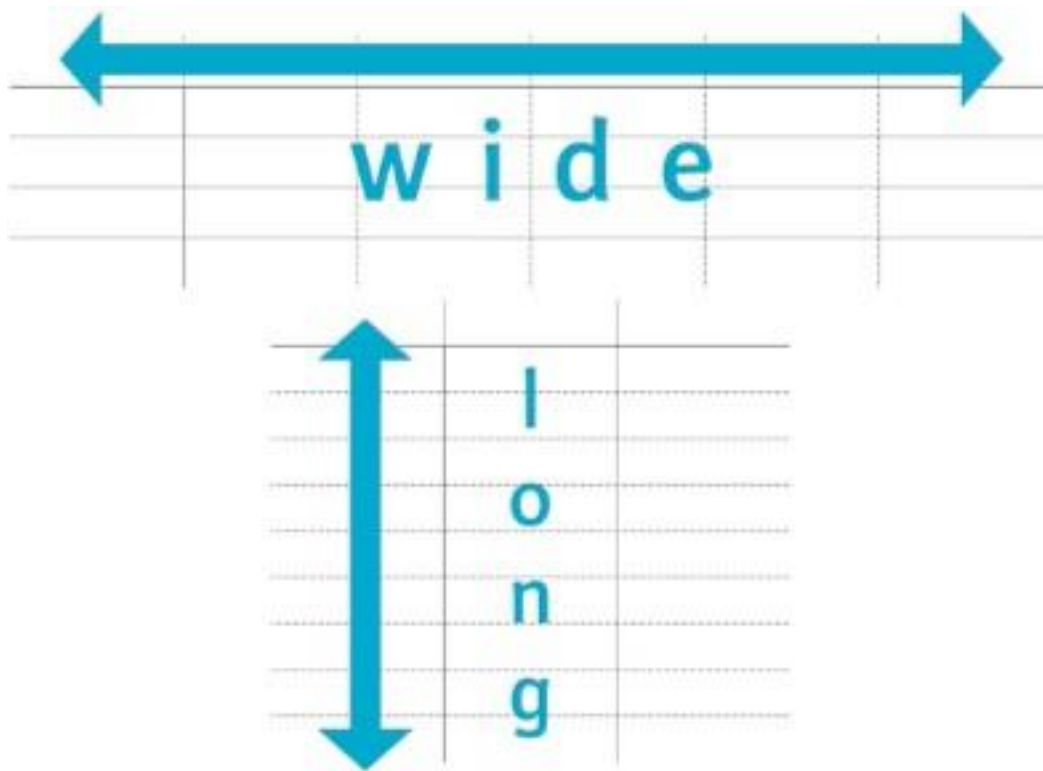
name	measurement	value
Jake	n_dogs	1
Jake	n_cats	0
Jake	n_birds	1
Alice	n_dogs	1
Alice	n_cats	2
Alice	n_birds	0

name	n_dogs	n_cats	n_birds
Jake	1	0	1
Alice	1	2	0



# Preparación y limpieza: *wide* vs *long*



- Hablamos de *wide datasets* cuando tienen más variables que observaciones.
- Hablamos de *long datasets* cuando tienen más observaciones que atributos.
- Un *wide dataset* puede indicar que tenemos valores almacenados como columnas. Trataremos de convertirlo en un *long dataset*.

## *tidyr*: quitando valores como columnas

- Para corregir los problemas anteriores usaremos las funciones **pivot\_longer/gather** del paquete *tidyr*. Ambas funciones unen columnas en pares clave/valor.
- Los parámetros de la función **gather** son:
  - **data**: el *data frame* de datos a corregir.
  - **key**: el nombre de la nueva columna que contendrá la clave.
  - **value**: el nombre de la nueva columna que contendrá el valor.
  - **...:** nombres de las columnas a juntar (o no).
- Los parámetros de la función **pivot\_longer** son:
  - **data**: el *data frame* de datos a corregir.
  - **cols**: las columnas a juntar.
  - **names\_to**: el nombre de la nueva columna que contendrá la clave.
  - **values\_to**: el nombre de la nueva columna que contendrá el valor.

# See vignette("pivot") for examples and explanation

## *reshape2: wide dataset to long dataset*

- Dentro del paquete *reshape2* existe una función, análoga a **pivot\_longer/gather** de *tidyr*, para pasar de un *wide dataset* a un *long dataset*: **melt**.
- La función **melt** recibe los siguientes parámetros:
  - **data**: el *data frame* de datos a pivotar.
  - **id.vars**: un vector con el nombre de las variables a mantener.
  - **variable.name**: el nombre de la nueva variable que contendrá las variables fundidas.
  - **value.name**: el nombre de la nueva variable que contendrá los valores.

## *tidyr*: quitando variables en filas

- Para corregir los problemas anteriores usaremos las funciones **pivot\_wider/spread** del paquete *tidyr*. Ambas funciones separan pares clave/valor en columnas.
- Los parámetros de la función **spread** son:
  - **data**: el *data frame* de datos a corregir.
  - **key**: el nombre de la columna que contiene la clave.
  - **value**: el nombre de la columna que contiene el valor.
- Los parámetros de la función **pivot\_wider** son:
  - **data**: el *data frame* de datos a corregir.
  - **names\_from**: el nombre de la columna que contiene la clave.
  - **values\_from**: el nombre de la columna que contiene el valor.

# See vignette("pivot") for examples and explanation

## *reshape2: long dataset to wide dataset*

- Dentro del paquete *reshape2* existe otra función, análoga a **pivot\_wider/spread** de *tidyr*, para pasar de un *wide dataset* a un *long dataset*: **dcast**.
- La función **dcast** recibe los siguientes parámetros:
  - **data**: el *data frame* de datos a pivotar.
  - **variable1 + variable2 + ... ~ variable**: **variable1**, **variable2**... son las columnas que se mantendrán y **variable** la columna que contiene los valores que generarán nuevas columnas.
  - **value.var**: nombre de la columna donde están los valores.
  - **fun.agregate**: en el caso de tener múltiples valores por fila especifica la función de agregación. Por ejemplo: **sum**, **mean**, **max**...

# Preparación y limpieza: dos variables en la misma columna

name	sex_age		eye_color	height
Jake	M.34		Other	6'1"
Alice	F.55		Blue	5'9"
Tim	M.76		Brown	5'7"
Denise	F.19		Other	5'1"

↓

name	sex	age	eye_color	height
Jake	M	34	Other	6'1"
Alice	F	55	Blue	5'9"
Tim	M	76	Brown	5'7"
Denise	F	19	Other	5'1"

## *tidyr*: separando columnas

- Para corregir el anterior problema usaremos la función **separate** del paquete *tidyr*. Esta función separa una columna en múltiples columnas. Los parámetros de la función son:
  - **data**: el *data frame* de datos a corregir.
  - **col**: el nombre de la columna a separar.
  - **into**: vector de caracteres con el nombre de las nuevas columnas.
  - **sep**: el separador a emplear. Por defecto se usa como separador cualquier carácter no alfanumérico.
- Existe otra función en el paquete *tidyr* para realizar el proceso inverso, **unite**. Esta función une varias columnas en una única columna. Los parámetros de la función son:
  - **data**: el *data frame* de datos a corregir.
  - **col**: el nombre de la nueva columna que contendrá la unión.
  - **...:** el nombre de las columnas a unir.
  - **sep**: el separador a emplear. Por defecto: "\_".

# Preparación y limpieza: más de una medida observacional en la misma tabla

name	age	height	pet_name	pet_type	pet_height
Jake	34	6'1"	Larry	Dog	25"
Jake	34	6'1"	Chirp	Bird	3"
Alice	55	5'9"	Wally	Dog	30"
Alice	55	5'9"	Sugar	Cat	10"
Alice	55	5'9"	Spice	Cat	12"



- Existen personas duplicadas (x3). Esto puede indicar que tenemos que separar en dos *data frames* los datos.
- OJO → A la hora de separar en varios *data frames* debemos crear una PK que nos permita cruzarlos. Similar a las *Primary Keys* de una base de datos relacional.



# Preparación y limpieza: *tidyr*

```
library(tidyr)

wide_df <- data.frame(col = c('X', 'Y'), A = c(1, 4), B = c(2, 5), C = c(3, 6))

# Gather
long_df <- gather(wide_df, my_key, my_val, -col)

# Spread
new_wide_df <- spread(long_df, my_key, my_val) #Obtenemos el data.frame original

treatments <- data.frame(patient = c('X', 'Y', 'X', 'Y', 'X', 'Y'),
                          treatment = c('A', 'A', 'B', 'B', 'c', 'C'),
                          year_mo = c('2010-10', '2010-10', '2012-08',
                                       '2012-08', '2014-12', '2014-12'),
                          responde = c(1, 4, 2, 5, 3, 6))

# Separate
treatments_sep <- separate(treatments, year_mo, c("year", "month"))

# Unite
new_treatments <- unite(treatments_sep, year_mo, year, month, sep = "-")
```

# Preparación y limpieza: *reshape2*

```
library(reshape2)

wide_df <- data.frame(col = c('X', 'Y'), A = c(1, 4), B = c(2, 5), C = c(3, 6))

# melt
long_df <- melt(wide_df, id.vars = c("col"), variable.name = "my_key", value.name = "my_val")

# dcast
new_wide_df <- dcast(long_df, col ~ my_key, value.var = "my_val")
```

# Ejercicio 10

- Realiza el ejercicio del fichero *Ejercicio10\_tidyr.R*



# Preparación y limpieza: preparando los datos para el análisis

- El último paso del proceso de limpieza es la preparación de los datos para el análisis.
- Normalmente, nuestro *dataset* contendrá distintos tipos de variables (cadenas de caracteres, numéricas, lógicas, fechas...).
- Es importante asegurar que cada variable está almacenada en el formato adecuado.
- Otro punto especialmente importante es el tratamiento de *missing values* y *outliers* (valores atípicos).

# Preparación y limpieza: conversión de tipos (I)

Función	Descripción
<code>as.factor(x)</code>	Realiza la coerción de x a factor.
<code>as.character(x)</code>	Realiza la coerción de x a cadena de caracteres.
<code>as.numeric(x)</code>	Realiza la coerción de x a número real.
<code>as.integer(x)</code>	Realiza la coerción de x a entero.
<code>as.logical(x)</code>	Realiza la coerción de x a valor lógico.
<code>as.Date(x, f)</code>	Parsea una fecha según el formato establecido por parámetro.
<code>as.POSIXct(x, f)</code>	Parsea una fecha y hora según el formato establecido por parámetro.

# Preparación y limpieza: conversión de tipos (II)

- Para manipulación de fechas es recomendable utilizar el paquete **lubridate** de **Hadley Wickham**.
- Contiene numerosas funciones para *parsear* y operar con fechas.

```
library(lubridate)

ymd("2015-08-25")

ymd("2015 August 25")

mdy("August 25, 2015")

hms("14:17:07")

ymd_hms("2015/08/25 13.33.09")
```

# Preparación y limpieza: conversión de tipos (III)

- Para manipulación de cadenas de caracteres existe el paquete **stringr** de **Hadley Wickham**.

```
library(stringr)
str_trim("  this is a test  ")
str_pad("244493", width = 7, side = "left", pad = "0")
names <- c("Sarah", "Tom", "Alice")
str_detect(names, "Alice")
str_replace(names, "Alice", "David")
```

- También podemos emplear las funciones que hemos visto del paquete base para manipulación de cadenas de caracteres.

# Preparación y limpieza: *missing values*

- ¿Por qué no están los datos? Es importante descubrir y saber el motivo por el cual faltan datos en un *dataset*.
- Dependiendo del motivo tendremos principalmente dos opciones:
  - **Borrarlos**
  - **Imputarlos**
    - Ceros
    - Media/Mediana
    - Empleando un modelo para rellanarlos
- En R los *missing values* se representan con NA.
- Otros valores especiales:
  - Inf: ¿pueden representar *outliers*? Divisiones por cero
  - NaN: ¿pueden representar errores?



## Preparación y limpieza: ¿cómo encontrar *missing values*?

- Para encontrar *missing values* podemos utilizar algunas de las funciones que hemos visto anteriormente. Por ejemplo: `is.na(x)`, `summary(x)`...

```
#NAs
df <- data.frame(A = c(1, NA, 8, NA),
                 B = c(3, NA, 88, 23),
                 C = c(2, 45, 3, 1))

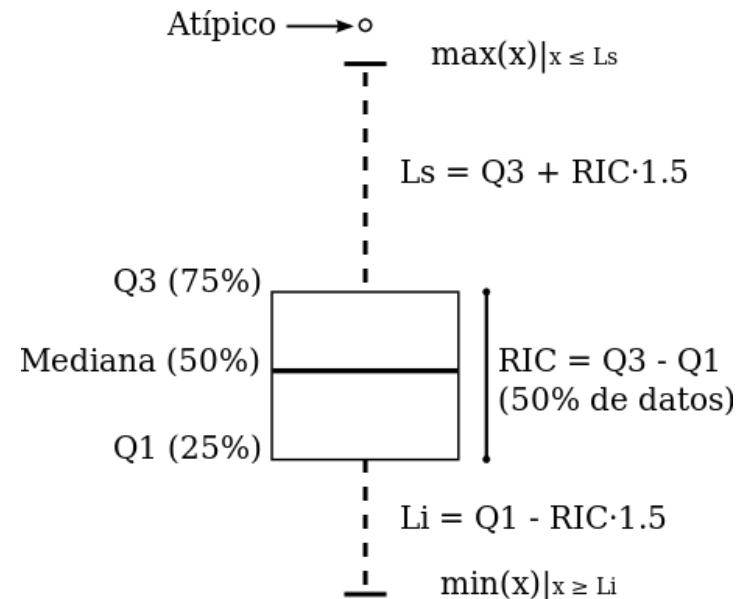
#Detección
is.na(df) #Util en dataset pequeños.
any(is.na(df))
sum(is.na(df))
summary(df)
```

- Si finalmente decidimos eliminarlos podemos utilizar la función `complete.cases(x)` para indexar y eliminar o directamente `na.omit(x)`

```
#Eliminación
df[complete.cases(df), ]
na.omit(df)
```

# Preparación y limpieza: *outliers*

- Los *outliers*, o valores atípicos, son valores extremos, distantes del resto de los valores.
- Algunas causas por las que pueden existir *outliers*:
  - Son errores de medición
  - Son errores producidos al transcribir los datos
  - Son medidas válidas
  - Representar valores por defecto
  - ...
- Una de las herramientas mas útiles será el diagrama de caja o *boxplot*.



# Preparación y limpieza: ¿cómo encontrar *outliers*?

- Existen varias herramientas para encontrar *outliers*:

- `boxplot`
- `hist`
- `summary`

```
#Outliers
set.seed(10)
x <- c(rnorm(30, mean = 15, sd = 5), -5, 28, 35)

boxplot(x, horizontal = T)

df2 <- data.frame(A = rnorm(100, 50, 100),
                  B = c(rnorm(99, 50, 10), 500),
                  C = c(rnorm(99, 50, 10), -1))

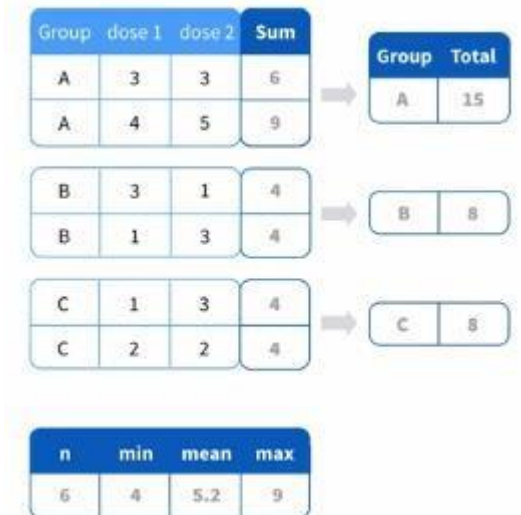
hist(df2$B, 20) #¿Qué le pasa al 500? Es un error de medición, 50
boxplot(df2)
```

- Dependiendo del contexto debemos decidir que hacer:

- Eliminarlos
  - ¿Puede la edad de una persona ser negativa? Valores sin sentido.
  - ¿Y superior a 200? Valores tan extremos que no tienen sentido.
- Dejarlos ¿Puede la edad de una persona ser 110?

# Manipulación

- La mayoría de los *datasets* contienen pequeños *insights* que no están accesibles de manera inmediata.
- No hablamos del tipo de conocimiento que nos proporcionan los algoritmos de *machine learning* o de modelado, sino de cosas más simples. Por ejemplo:
  - Nuevas variables.
  - Estadísticas básicas.
  - Diferencias entre grupos.
  - ...
- Existen paquetes que nos ayudarán a extraer esta información:
  - *dplyr*
  - *data.table*



Group	dose 1	dose 2	Sum
A	3	3	6
A	4	5	9

Group	Total
A	15

Group	dose 1	dose 2	Sum
B	3	1	4
B	1	3	4

Group	Total
B	8

Group	dose 1	dose 2	Sum
C	1	3	4
C	2	2	4

Group	Total
C	8

n	min	mean	max
6	4	5.2	9

## dplyr

- Paquete de R que define una gramática de manipulación de datos. Creado por **Hadley Wickham**. Las operaciones básicas que realizar son:
  - Selección de variables (**select**).
  - Creación de nuevas variables (**mutate**).
  - Filtrado de observaciones (**filter**).
  - Ordenación de observaciones (**arrange**).
  - Agrupación de observaciones y cálculo de estadísticos agregados (**group\_by & summarise**).
- Todas las funciones anteriores devuelven copias del *dataset* original.
- Funciona muy rápido (está implementado en C++).
- Su sintaxis puede recordar a SQL.
- El paquete *dplyr* funciona tanto con objetos *data.frame* como *tbl*.

## *dplyr*: tbl, tipo especial de data.frame

- Dentro del paquete *dplyr* existe un nuevo tipo de objeto, la tabla (**tbl**), similar a un *data frame*.
- Tiene una gran ventaja al trabajar con *datasets* grandes, ya que permite visualizar su contenido de una forma más *amigable*. Al mostrar el contenido de una variable del tipo tabla, el contenido se adapta al tamaño de la pantalla (*responsive*).
- Para ver el contenido de una tabla existe el comando `glimpse(x)`, cuyo resultado es similar al `str(x)` sobre un *data frame*.

```
library(hflights)
library(dplyr)

hflights

hflights <- tbl_df(hflights)

hflights

glimpse(hflights)
```

## *dplyr*: selección de variables

- Para hacer selección de variables utilizaremos la función **select** del paquete *dplyr*. Los parámetros de la función son:
  - *df*: el *data frame* o tabla sobre la que actuar.
  - ...: nombres de las variables a seleccionar. Se puede combinar con el operador `:` para definir rangos de variables o `-` para decidir las que no.
- Existe un conjunto de funciones auxiliares que permiten hacer la selección de las variables de manera más ágil:
  - `starts_with("x")`: selecciona las variables que comienzan con "x".
  - `ends_with("x")`: selecciona las variables que terminan con "x".
  - `contains("x")`: selecciona las variables que contienen "x".
  - `matches("x")`: selecciona las variables que casan con "x". Se puede combinar con expresiones regulares.
  - `num_range("x", 1:5)`: variables de la lista x01, x02, ..., x05.
  - `one_of(x)`: aquellas variables en el vector de caracteres x.

## *dplyr*: creación de variables

- Para añadir nuevas variables utilizaremos la función **mutate** del paquete *dplyr*. Los parámetros de la función son:
  - *df*: el *data frame* o tabla sobre la que actuar.
  - *new\_column = expresion*: el nombre de la nueva variable y la expresión que la calcula.
  - *...*: se pueden añadir más de una variable nueva al mismo tiempo.
- **Tip:** puedes emplear las nuevas variables para crear otras en la misma llamada.



## *dplyr*: filtrar observaciones

- Para filtrar las filas de un *dataset* utilizaremos la función **filter** del paquete *dplyr*. Los parámetros de la función son:
  - *df*: el *data frame* o tabla sobre la que filtrar.
  - *logical test*: condición de filtrado sobre aquellas filas que cumplan el test.
- La función **filter** permite utilizar los operadores relacionales y lógicos de R.

## *dplyr*: ordenando observaciones

- Para ordenar las filas de un *dataset* utilizaremos la función **arrange** del paquete *dplyr*. Los parámetros de la función son:
  - **df**: el *data frame* o tabla sobre el que ordenar.
  - **...**: las variables sobre las que se ordenará.
- Por defecto, la función **arrange** ordena de menor a mayor (ascendentemente).
- Para ordenar de mayor a menor (descendentemente), utilizamos la función **desc(x)**.

## *dplyr*: agregar observaciones

- Para agrupar las filas de un *dataset* utilizaremos la función **group\_by** del paquete *dplyr*. Los parámetros de la función son:
  - *df*: el *data frame* o tabla sobre la que se realizará la agrupación.
  - ...: las variables sobre las que se agrupará.
- La función **group\_by** se suele usar conjuntamente con otra función, **summarise**. Dicha función, permite calcular un resumen de estadísticos básicos que describen el *dataset* o la agrupación. Los parámetros de la función son:
  - *df*: el *data frame* o tabla sobre la que actuar.
  - *new\_column = expresion*: el nombre del nuevo estadístico y la expresión que lo calcula.
  - ...: se pueden añadir más de una variable nueva al mismo tiempo.
- **Tip**: al igual que con **mutate**, al usar **summarise** puedes emplear las nuevas variables para crear otras en la misma llamada.

## *dplyr*: summarise (I)

- La función **summarise** permite utilizar funciones de agregación de R:
  - `min(x)`: valor mínimo del vector `x`.
  - `max(x)`: valor máximo del vector `x`.
  - `mean(x)`: valor medio del vector `x`.
  - `median(x)`: valor mediano del vector `x`.
  - `quantile(x, p)`: el cuantil `p` del vector `x`.
  - `sd(x)`: la desviación estándar del vector `x`.
  - `var(x)`: la varianza del vector `x`.
  - `IQR(x)`: el rango intercuartílico del vector `x`.

## *dplyr*. summarise (II)

- El paquete *dplyr* completa la lista de funciones anteriores con algunas propias:
  - `first(x)`: primer elemento del vector *x*.
  - `last(x)`: último elemento del vector *x*.
  - `nth(x, n)`: el enésimo elemento del vector *x*.
  - `n()`: número de filas del *data frame*, tabla o grupo.
  - `n_distinct(x)`: número de valores únicos del vector *x*.

## *dplyr: pipes operator*

- El paquete *dplyr* permite utilizar el **pipe operator** de R para encadenar llamadas a funciones.
  - Envía la salida de una función a la siguiente.
  - Ahorra espacio (no es necesario declarar variables intermedias).
  - Simplifica la lectura de código.

```
summarise(  
  mutate(  
    filter(  
      select(a, X, Y, Z),  
        X > Y),  
    Q = X + Y + Z),  
  all = sum(Q))  
)
```



```
a %>%  
  select(X, Y, Z) %>%  
  filter(X > Y) %>%  
  mutate(Q = X + Y + Z) %>%  
  summarise(all = sum(Q))
```

## *dplyr*: combinando

- El paquete *dplyr* también contiene algunas funciones para realizar operaciones de unión o combinación de *data frames* o tablas.
- Las funciones son **inner\_join**, **left\_join**, **right\_join**, **full\_join**. Los parámetros son comunes para todas ellas:
  - *x, y*: *data frames* o tablas a unir.
  - *by*: vector de caracteres con las columnas por las que se realizará la unión. Por defecto, buscará aquellas variables con el mismo nombre.

# Manipulación: *dplyr*

```
# Select
hflights.select <- select(hflights.tbl, ActualElapsedTime, AirTime, ArrDelay, DepDelay)
hflights.select

# Mutate
hflights.mutate <- mutate(hflights.select, loss = ArrDelay - DepDelay)
hflights.mutate

# Filter
hflights.select <- select(hflights.tbl, starts_with("Cancel"), DepDelay)
hflights.select
hflights.filter <- filter(hflights.select, Cancelled == 1)
hflights.filter

# Arrange
hflights.select <- select(hflights.tbl, TailNum, contains("Delay"))
hflights.select
hflights.arrange <- arrange(hflights.select, DepDelay)
hflights.arrange
hflights.arrange <- arrange(hflights.select, DepDelay, ArrDelay)
hflights.arrange

# Summarise
hflights.select <- select(hflights.tbl, TailNum, contains("Delay"))
hflights.select <- filter(hflights.select, !is.na(DepDelay))
hflights.summarise <- summarise(hflights.select, min = min(DepDelay), max = max(DepDelay), mean = mean(DepDelay),
                                median = median(DepDelay))
hflights.summarise

# Group by
hflights.group <- group_by(hflights.tbl, UniqueCarrier)
hflights.summarise.group <- summarise(hflights.group,
                                       avgDep = mean(DepDelay, na.rm = T),
                                       avgArr = mean(ArrDelay, na.rm = T))
hflights.summarise.group

# %>%
hflights.tbl %>%
  filter(!is.na(DepDelay)) %>%
  summarise(min = min(DepDelay), max = max(DepDelay), mean = mean(DepDelay), median = median(DepDelay))
```



# Ejercicio 11

- Realiza el ejercicio del fichero *Ejercicio11\_dplyr.R*




## *data.table*

- Paquete de R para manipulación de datos. Podríamos decir que es una versión mejorada de los *data frames*.
- Objetivos del paquete:
  - Reducir el tiempo de programación (sintaxis más compacta que *data.frame*).
  - Reducir el tiempo de procesamiento (agregaciones rápidas).
- Funcionamiento: **DT[i, j, by]**
  - **i: selección de filas (WHERE en SQL).**
  - **j: selección de columnas (SELECT en SQL).**
  - **by: agrupación de filas (GROUP BY en SQL).**
- Un *data.table* es también un *data.frame*. Esto puede resultar de utilidad ya que puede ser empleado con aquellos paquetes que usan *data.frames*.

## *data.table*: selección de filas en i

- Diferencias al indexar filas entre *data.frame* y *data.table*

data.table 	data.frame
<pre>&gt; DT   A B 1: 1 a 2: 2 b 3: 3 c 4: 4 a 5: 5 b 6: 6 c</pre>	<pre>&gt; DF   A B 1: 1 a 2: 2 b 3: 3 c 4: 4 a 5: 5 b 6: 6 c</pre>
<pre>&gt; DT[3:5,]   A B 1: 3 c 2: 4 a 3: 5 b</pre>	<pre>&gt; DF[3:5,]   A B 3: 3 c 4: 4 a 5: 5 b</pre>
<pre>&gt; DT[3:5]   A B 1: 3 c 2: 4 a 3: 5 b</pre>	<pre>&gt; DF[3:5]  Error: undefined columns selected</pre>

# *data.table*: selección y creación de columnas en j

```
> DT
```

	A	B	C
1:	1	a	6
2:	2	b	7
3:	3	c	8
4:	4	d	9
5:	5	e	10

```
> DT[, .(B, C)]
```

	B	C
1:	a	6
2:	b	7
3:	c	8
4:	d	9
5:	e	10

NB: `.( )` is an alias to `list()` in `data.tables` and they mean the same.

- Selección de columnas

- Creación de nuevas columnas

```
> DT
```

	A	B	C
1:	1	a	6
2:	2	b	7
3:	3	c	8
4:	4	d	9
5:	5	e	10

```
> DT[, .(Total = sum(A), Mean = mea
```

	Total	Mean
1:	15	8

## data.table: agregando en by

```
> DT
   A B
1: c 1
2: b 2
3: a 3
4: c 4
5: b 5
6: a 6
```

```
> DT[, .(MySum = sum(B),
        MyMean = mean(B)),
      by = .(A)]
```

	A	MySum	MyMean
1:	c	5	2.5
2:	b	7	3.5
3:	a	9	4.5

- Agrupando por una columna

- Agrupando por el resultado de una función

```
> DT
   A B
1: 1 10
2: 2 11
3: 3 12
4: 4 13
5: 5 14
```

```
> DT[, .(MySum = sum(B)), by = .(Grp = A%%2)]
```

	Grp	MySum
1:	1	36
2:	0	24

## *data.table*: juntándolo todo **DT[i, j, by]**

- Take **DT**, subset rows using **i**, then calculate **j** grouped by **by**.

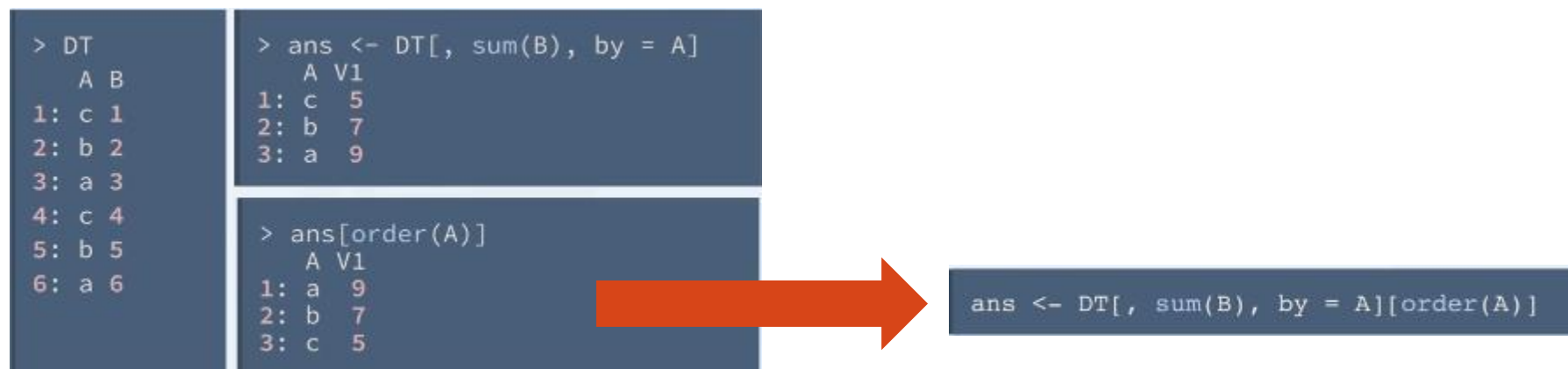
```
> DT
   A  B
1: 1 10
2: 2 11
3: 3 12
4: 4 13
5: 5 14
```

```
> DT[2:4, sum(B), by = A%%2]

   A V1
1: 0 24
2: 1 12
```

## *data.table*: encadenando llamadas

- Del mismo modo que en *dplyr*, con *data.table* podemos encadenar llamadas para ahorrarnos la creación de variables intermedias.



## *data.table: .SD (Subset of Data)*

- **.SD** es una variable especial que contiene un *data.table* con el subconjunto de datos de cada grupo (excluyendo las columnas usadas en el **by**).
- Permite simplificar el código escrito.
- **.SDCols** permite establecer las variables incluidas **.SD**.

```
> DT <- as.data.table(iris)
> DT[, .(Sepal.Length = median(Sepal.Length),
        Sepal.Width = median(Sepal.Width),
        Petal.Length = median(Petal.Length),
        Petal.Width = median(Petal.Width)),
      by = Species]
```

```
> DT[, lapply(.SD, median), by = Species]
```

	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1:	setosa	5.0	3.4	1.50	0.2
2:	versicolor	5.9	2.8	4.35	1.3
3:	virginica	6.5	3.0	5.55	2.0

NB: `.`() is an alias to `list()` and `lapply()` returns a list.



## *data.table*: el operador `:=` (I)

- El operador `:=` de *data.table* permite modificar (crear, actualizar o eliminar) el contenido de una columna. Existe otra versión “funcional” (más simple) del operador, ``:=``.
- Actualizando y creando columnas:

```
> DT
```

	x	y
1:	1	6
2:	1	7
3:	1	8
4:	2	9
5:	2	10

```
> DT[, c("x", "z") := .(rev(x), 10:6)]
```

	x	y	z
1:	2	6	10
2:	2	7	9
3:	1	8	8
4:	1	9	7
5:	1	10	6

Shortcut if just one :

```
DT[, x := rev(x)]
```

```
> DT
```

	x	a
1:	2	10
2:	2	9
3:	1	8
4:	1	7
5:	1	6

```
> DT[, `:=`(y = 6:10, # y (kg)
        z = 1)]
```

## *data.table*: el operador := (II)

- Eliminando columnas:

```
> DT
```

	x	a
1:	2	10
2:	2	9
3:	1	8
4:	1	7
5:	1	6

```
> DT[, c("y", "z") := NULL]
```

Shortcut if just one :

```
DT[, y := NULL]
```

```
> DT
```

	x	a
1:	2	10
2:	2	9
3:	1	8
4:	1	7
5:	1	6

```
MyCols = c("y", "z")
DT[, (MyCols) := NULL]
```

```
DT[, paste0("colNamePrefix", 1:4) := NULL ]
```

## *data.table*: `set()`, `setnames()` y `setcolorder()`

- **`set()`** es una versión rápida del operador de asignación `:=` de *data.table*. Se utilizar dentro de bucles.
- **`setnames()`** permite cambiar o asignar el nombre de las columnas de un *data.table*.
- **`setcolorder()`** permite reordenar las columnas de un *data.table*.

```
> DT
```

	x	y	z
1:	1	1	2
2:	2	8	5
3:	3	1	4
4:	4	1	2
5:	5	1	3

```
> for (i in 1:5) DT[i, z := i+1]
> for (i in 1:5) set(DT, i, 3L, i+1)
```

	x	y	z
1:	1	1	2
2:	2	8	3
3:	3	1	4
4:	4	1	5
5:	5	1	6

```
> DT
```

	x	y
1:	1	a
2:	2	b
3:	3	c
4:	4	d
5:	5	e

```
> setnames(DT, "y", "z")
```

	x	z
1:	1	a
2:	2	b
3:	3	c
4:	4	d
5:	5	e

```
> DT
```

	x	y
1:	1	a
2:	2	b
3:	3	c
4:	4	d
5:	5	e

```
> setcolorder(DT, c("y", "x"))
```

	y	x
1:	a	1
2:	b	2
3:	c	3
4:	d	4
5:	e	5

## *data.table*: creación de índices (*keys*) (I)

- *data.table* permite crear índices (**keys**) para optimizar los filtrados de filas (al estilo de los índices de base de datos). Al crear el índice el *data.table* se reordena según el mismo.
- Cuando filtremos la primera vez por una columna en el *data.table* se creará un índice para dicha columna. Desde ese momento la selección de filas funcionará mucho más rápido.

```
> DT[A == "a"]  
> DT[A %in% c("a", "c")]
```

These don't actually vector scan.  
They create an index automatically (by default) on A, the first time you use column A.

```
> DT[A == "b"]    # second time much faster
```

## *data.table*: creación de índices (*keys*) (II)

- Para crear los índices usamos la función **setkey()**. Esta función recibe como parámetros:
  - DT: el *data.table* sobre el que se creará el índice.
  - ...: las variables a utilizar en la creación del índice.

```
> DT
```

	A	B
1:	c	1
2:	b	2
3:	a	3
4:	c	4
5:	b	5
6:	a	6

```
> setkey(DT, A)
```

	A	B
1:	a	3
2:	a	6
3:	b	2
4:	b	5
5:	c	1
6:	c	4

```
> DT
```

	A	B
1:	a	3
2:	a	6
3:	b	2
4:	b	5
5:	c	1
6:	c	4

```
> DT["b"]
```

	A	B
1:	b	2
2:	b	5

```
> DT
```

	A	B	C
1:	c	4	1
2:	b	1	2
3:	a	6	3
4:	c	3	4
5:	b	5	5
6:	a	2	6

```
> setkey(DT, A, B)
```

	A	B	C
1:	a	2	6
2:	a	6	3
3:	b	1	2
4:	b	5	5
5:	c	3	4
6:	c	4	1

```
> DT[.("b", 5)]
```

	A	B	C
1:	b	5	5

```
> DT[.("b", 6)]
```

	A	B	C
1:	b	6	NA

```
> DT[.("b")]
```

	A	B	C
1:	b	1	2
2:	b	5	5

## *data.table*: mult & nomatch (keys) (III)

- Con el parámetro **mult** seleccionamos el primer o último elemento de la selección filtrada.
- Con **nomatch** controlamos los resultados devueltos cuando no casa el filtrado.

```
> setkey(DT, A)

  A B
1: a 3
2: a 6
3: b 2
4: b 5
5: c 1
6: c 4
```

```
> DT["b", mult = "first"]

  A B
1: b 2
```

```
> DT["b", mult = "last"]

  A B
1: b 5
```

```
> setkey(DT, A)

  A B
1: a 3
2: a 6
3: b 2
4: b 5
5: c 1
6: c 4
```

```
> DT[c("b", "d"), nomatch = NA] # default

  A B
1: b 2
2: b 5
3: d NA
```

```
> DT[c("b", "d"), nomatch = 0]

  A B
1: b 2
2: b 5
```

## Ejercicio 12

- Realiza el ejercicio del fichero *Ejercicio12\_data\_tables.R*



# 8 | Traps & tips en R



## Traps & tips: *coercion*

- **Trap:** objetos R cambian el tipo de manera “silenciosa”.

```
# Coercion
c(1, TRUE)

c(1, TRUE, "cat")

30 < "8"
```

- **Tip:** inspeccionar objetos con `str(x)`, `mode(x)`, `class(x)`, `typeof(x)`, `dput(x)`, `attributes(x)`.

## Traps & tips: *factor*

- **Trap:** generan gran cantidad de *bugs* en R. Especialmente cuando *characters* se convierten en *factors*.
- **Tip:** comprobar con `is.factor(df$col)`.
- **Tip:** usa `stringAsFactors = FALSE` con `read.table(fileName)`.
- **Trap:** las funciones matemáticas no funcionan con *factors*.
- **Tip:** `as.numeric(as.character(factor))`
- **Trap:** añadir filas a un *data frame* con columnas de tipo *factor* puede dar problemas.
- **Trap:** la función `c(x, y, ...)` permite combinar *factors* en un vector de códigos enteros.
- **Tip:** convertir el factor a *character* o *integer* antes de combinar.

## Traps & tips: *vector*

- **Trap:** en R los escalares son vectores de longitud 1. La mayoría de funciones trabajan con vectores.
- **Tip:** no utilizar bucles para recorrer listas o vectores. Es conveniente aprender a utilizar las funciones de la familia *apply*. En R, se dice que el código es más elegante y fácil de entender si usas funciones *apply* en lugar de bucles. El rendimiento de las funciones *apply* es **ligeramente** mejor que el de los bucles (aunque no es determinante). Se debe tratar de utilizar **funciones “vectorizadas”** para obtener un rendimiento óptimo.
- **Trap:** las matemáticas con vectores de diferente longitud hacen que el de menor longitud se “recicle”.

# Traps & tips: operadores lógicos

- **Tip:** `!` y `&` son “*vectorizados*”.
- **Tip:** `!!` y `&&` no son “*vectorizados*”.
- **Trap:** `==` igualdad booleana.
- **Trap:** `=` asignación.
- **Trap:** `==` y `!=` comprueban igualdad/desigualdad cercana. Por ejemplo:  
`as.double(8)==as.integer(8) #TRUE`.
- **Tip:** `identical(x, y)` es más riguroso.

# Traps & tips: NA, NaN y NULL

- **Trap:** NA y NaN son valores válidos. Por ejemplo: `c(1, 2) == c(1, NA) #TRUE, NA`
- **Trap:** muchas funciones fallan al tener valores NA de entrada.
- **Tip:** utilizar el parámetro `na.rm = TRUE`
- **Tip:** para comprobar si hay NAs `any(is.na(y))`
- **Trap:** `x == NA` no es lo mismo que `is.na(x)`
- **Trap:** `x == NULL` no es lo mismo que `is.null(x)`
- **Trap:** `is.numeric(NaN) #TRUE`

# Traps & tips: Indexación

- **Tip:** en R se indexa de 1:N
- **Trap:** existen diferencia sutiles en la indexación de vectores, matrices, *arrays*, *data frames* y listas. Los tipos devueltos dependen del objeto indexado y el método de indexación.
- **Trap:** cuidado con el índices 0. `c(1, 2, 3)[c(0, 1, 2, 0, 2, 3)]` # 1, 2, 2, 3
- **Trap:** índices negativos devuelven todos menos ellos mismos.
- **Trap:** NA es un índice booleano válido. `c(1, 2)[c(TRUE, NA)]` #1, NA
- **Trap:** índices booleanos mal asignados funcionan. `c(1, 2, 3)[c(T, F, T, F, T)]` # 1, 3, NA

## Traps & tips: *workspace*

- **Trap:** R guarda el entorno de trabajo al finalizar la sesión y lo vuelve a cargar al principio. Con variables grandes se puede convertir en una pérdida de tiempo.
- **Tip:** `ls()` para comprobar las variables.
- **Tip:** `rm(list=ls())` limpia el entorno.
- **Tip:** con `library()` comprobamos los paquetes cargados.

# Traps & tips: *coding*

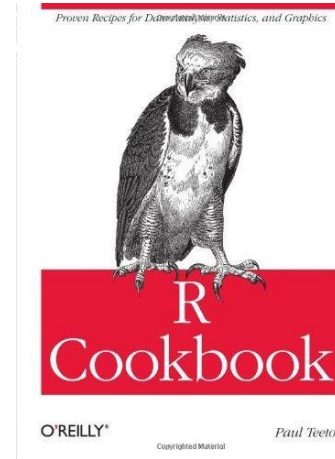
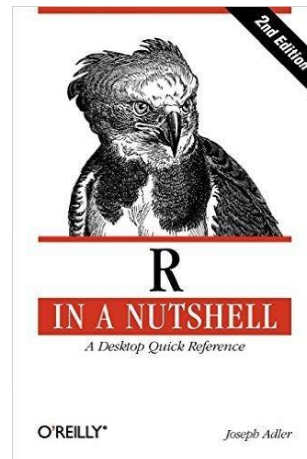
- **Tip:** utilizar <- para asignaciones. [Difference between assignment operators in R](#)
- **Tip:** emplear las llaves ( ) y los corchetes { } en el código.
- **Tip:** [Error Handling in R](#)
- **Tip:** [Learn R, in R \(swirl\)](#)
- **Tip:** [Google's R Style Guide](#) vs [Hadley Wickham Style Guide](#)
- **Tip:** [RStudio Cheat Sheets](#)
- **Tip:** [R Reference Card](#)
- **Tip:** [R vs Python](#)



# 9 | Materiales

# Materiales

- DataCamp: <https://www.datacamp.com/>
  - R Programming: <https://es.coursera.org/learn/r-programming>
  - RStudio Cheat Sheets: <https://rstudio.com/resources/cheatsheets/>
- 
- R in a nutshell
  - R cookbook





**Afi** Escuela  
de Finanzas

---

© 2021 Afi Escuela de Finanzas. Todos los derechos reservados.