

# Cómo sobrevivir a una pelea con

## Curso de introducción a

Pablo Cabrera-Álvarez

 |  @pablocalv

Julio 2019

# El curso

1. Cómo sobrevivir a una pelea con 
2. Manipulación de datos, lo básico pero en 
3. Manipulación de datos, más allá de lo básico
4. *Outputs*: ¿Y todo esto para hacer una tabla?

- 
- Proyecto: Crear una BD a partir de fuentes web

# Dinámica de trabajo

- Reproducir → Aplicar → Insertar
- Jugar con vuestros propios datos
- Dudas a pablocal@usal.es | @pablocalv
- En  hay que perderse
- ¿Para qué es  útil?
- Materiales y todo el código disponible en 

# Cómo sobrevivir a una pelea con



# Cómo sobrevivir a una pelea con

- Tipos de **objetos** en : vectors, matrices, data frames, lists, functions
- **Indexación** en 
  - Vectors
  - Matrices
  - Data frames
  - Lists
- **Modificar** vectores:
  - Operadores
  - Factor
  - Character
  - Date
  - Valores **perdidos** NA
- **Transformar** objetos
- **Proyecto** y workflow en 

# Antes de empezar, consejos generales



- Empezar un script ([.R](#))
  - Título y metadatos
  - Limpiar el espacio
  - Cargar paquetes que van a ser usados

```
##### CURSO DE R - Sesión 1: Objetos #####
# by PCA on 15th jul. 19

rm(list = ls()) # limpiar el espacio

library(tidyverse) # carga paquete tidyverse
library(lubridate) # carga paquete lubridate
```

# Antes de empezar, consejos generales 😎

- Nombrar objetos
  - 2Cs: Conciso y consistente
  - Minúscula
  - Estilos: `sexoEdad`, `sexo_edad` o `sexo.edad`
- Comentar `#` [`Ctrl + Mayús + C`]
  - Explicar el por qué
  - Usar separadores para organizar el scrip [`Ctrl + Mayús + R`]
- Espacios
  - Usar espacios en torno a los operadores
  - Evitar espacios en el principio/final de los paréntesis

```
a<-22+23 # mal  
a <- 22 + 23 # bien
```

# Help!

help()

- Ayuda con las funciones:

```
help(mean)  
?mean
```

- Vignettes y ejemplos
- Stackoverflow
- Cheatsheets
- Quick 

# Recursos: aprender

- Libros: R for Data Science, Advanced R
- Cursos web: DataCamp
- Twitter: @RLadiesGlobal, @hadleywickham, @dataandme...
- Rpckages: swirl
- Rbloggers: <https://www.r-bloggers.com>



File Edit Code View Plots Session Build Debug Tools Help

cismay Sessions R 3.3.1



first\_rmarkdown.Rmd x

```
1 ---  
2 title: "First RMarkdown"  
3 output: html_document  
4 ---  
5  
6 ```{r setup, include=FALSE}  
7 knitr::opts_chunk$set(echo = TRUE)  
8 ---  
9  
10 ## R Markdown  
11  
12 This is an R Markdown document. Markdown is a simple formatting syntax for  
authoring HTML, PDF, and MS Word documents. For more details on using R Markdown  
see <http://rmarkdown.rstudio.com>.
```

13

2:1 # First RMarkdown

Console ~/initial/ ↵

Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.

Type 'q()' to quit R.

Attaching package: 'dplyr'

The following objects are masked from 'package:base':

intersect, setdiff, setequal, union

> |

Environment History

Import Dataset

Global Environment

Environment is empty

Files Plots Packages Help Viewer

	Name	Size	Modified
	..		
	Initial.Rproj	205 B	Aug 14, 2016, 2:00 PM
	first_rmarkdown.Rmd	803 B	Aug 14, 2016, 1:38 PM
	.Ruserdata		

# Directorio de trabajo

```
setwd(dir) | getwd()
```

Antes de comenzar a trabajar en  es preciso **designar una carpeta de trabajo**, para lo que existen dos opciones.

La primera opción es utilizar un **proyecto de RStudio**, que fijará el directorio de trabajo en la carpeta escogida:

File → New project... → New directory

La segunda opción consiste en **fijar manualmente el directorio**<sup>1</sup> de trabajo con la función `setwd(dir)`. Para conocer el directorio de trabajo actual se puede utilizar `getwd()`.

```
setwd("c:/user/pablo/documentos/mi proyecto de R")
```

[1] Nótese el sentido de las barras (/).

# Objetos en

# R como lenguaje orientado a objetos

R es un **lenguaje orientado a objetos**: todos los elementos son almacenables como un objeto...

```
number_a <- 5  
number_b <- 2
```

...y los objetos pueden ser utilizados posteriormente en otras operaciones:

```
number_c <- number_a*number_b
```

```
number_c
```

```
## [1] 10
```

# Objetos en

# Vector

# Vector

`c(...)` | `names(x)`

Un **vector** es el objeto básico de R, un conjunto de números o caracteres. Todos los elementos de un vector deben de ser del mismo tipo.

```
my_vector <- c(10, 20, 10) # crear objeto mi_vector
```

Para **imprimir el vector**:

```
my_vector # imprimir mi_vector
```

```
## [1] 10 20 10
```

Los elementos de los vectores pueden tener **atributos**, como **nombres**:

```
names(my_vector) <- c("Juan", "Marta", "Pablo") # asignar nombres  
my_vector # imprimir vector con nombres
```

```
## Juan Marta Pablo
```

# Numeric

Los vectores **integer** están compuestos por números enteros:

```
vector_integer <- c(1, 2, 5)
vector_integer
```

```
## [1] 1 2 5
```

Los vectores de tipo **double** están compuestos por números decimales:

```
vector_double <- c(1.87, 2, 5.8754)
vector_double
```

```
## [1] 1.8700 2.0000 5.8754
```

# Character

Los vectores **character** contienen cadenas de texto:

```
vector_character <- c("Hola", "Casa", "Terreno en Madrid")
vector_character

## [1] "Hola"           "Casa"          "Terreno en Madrid"
```

# Logical

Los vectores de tipo **logical** contienen los elemento *booleanos* **TRUE** y **FALSE**:

```
vector_logical <- c(TRUE, FALSE, TRUE)  
vector_logical
```

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

# Factor

```
factor(x, levels, labels, ordered) | str(object)
```

Los **factores** son un tipo concreto de vector en el que unos **valores numéricos son asociados a unas etiquetas**.

```
## 1 = Trabaja, 2 = Desemp., 3 = Retirado
vector_values <- c(1, 2, 2, 3, 1, 2)

value_factor <- factor(x = vector_values, levels = c(1, 2, 3),
                       labels = c("Trabaja", "Desemp.", "Retirado"))
str(object = value_factor)
```

```
## Factor w/ 3 levels "Trabaja","Desemp.",...: 1 2 2 3 1 2
```

# Factor

Los factores también pueden ser generados a partir de vectores de tipo *character*. En ese caso los **levels** son obtenidos de las categorías de texto del vector.

```
vector_labels <- c("Trabaja", "Desemp.", "Trabaja", "Trabaja", "Retirado")
labels_factor <- factor(x = vector_labels)
str(object = labels_factor)

## Factor w/ 3 levels "Desemp." , "Retirado" , ... : 3 1 3 3 2
```



# Quiz

En el siguiente vector lógico: `c(TRUE, TRUE, FALSE)`, ¿cuál será el resultado de la suma de sus elementos?

```
sum(c(TRUE, TRUE, FALSE))
```

```
## [1] 2
```

# Objetos en

# Matrix

```
matrix(x, nrow, ncol)
```

Los **matrices** son un conjunto de datos del mismo tipo organizados en más de una dimensión:

```
my_matrix <- matrix(1:20, nrow = 4) # genera una matriz (4 x 5)  
my_matrix
```

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

# Objetos en

# Data frame

```
data.frame(..., stringsAsFactors = TRUE) | rnorm(n, mean = 1, sd = 0) | runif(n, min = 0, max = 1)
```

Los **data frames** son conjuntos de datos en los que las columnas pueden ser vectores de diferente tipo. Equiparable al conjunto de datos estándar utilizado en las Ciencias Sociales, **casos** en las **filas** y **variables** en las **columnas**.

```
my_df <- data.frame(var_char= letters[1:5],  
                      var_num = rnorm(5),  
                      var_logic = runif(5) > .5)  
my_df
```

```
##   var_char     var_num var_logic  
## 1      a  0.77608922    FALSE  
## 2      b -0.14854701     TRUE  
## 3      c -0.38244872    FALSE  
## 4      d -0.11902709    FALSE  
## 5      e  0.06831824     TRUE
```

# Objetos en

# List

`list(...)`

Las **listas** son conjuntos de diferentes tipos de elementos (e.g. vectores, matrices, df, funciones, listas...):

```
my_list <- list(vector = my_vector, df = my_df) # generar lista  
my_list
```

```
## $vector  
##   Juan Marta Pablo  
##   10      20      10  
  
##  
## $df  
##   var_char      var_num var_logic  
## 1       a  0.77608922    FALSE  
## 2       b -0.14854701     TRUE  
## 3       c -0.38244872    FALSE  
## 4       d -0.11902709    FALSE  
## 5       e  0.06831824     TRUE
```



# Quiz

¿Es posible que una lista contenga otra lista dentro como en el siguiente código?

```
my_list2 <- list(list_old = my_list,  
                 df = my_df)
```

# Function

```
mean(x, na.rm = FALSE)
```

Las **funciones** son las **herramientas** que sirven para trabajar con los datos, también son objetos almacenables, modificables y pueden ser creadas por el usuario:

```
mean(my_vector)
```

```
#> [1] 13.33333
```

```
mean_my_vector <- mean(my_vector)  
mean_my_vector
```

```
#> [1] 13.33333
```

# Function

Las **funciones** siempre tienen la misma estructura:

```
function(arg1 = x1, arg2 = x2, ...)
```

Algunos argumentos son **obligatorios**, otros opcionales; utilizar el nombre del argumento es útil para hacer más legible el código, especialmente en el caso de funciones poco utilizadas.

```
runif(n = 5, min = 0, max = 1)
```

```
## [1] 0.1324611 0.8815074 0.4423002 0.7949167 0.4191350
```

Los argumentos también pueden ser interpretados según su **posición** en la función:

```
runif(5, 0, 1)
```

```
## [1] 0.42745405 0.61604122 0.05450059 0.89956295 0.82168076
```

# Function

Algunos argumentos vienen definidos **por defecto**:

```
runif(5) # min = 0; max = 1  
## [1] 0.3907304 0.1041690 0.2038754 0.9706789 0.9953343
```

# Packages

```
install.packages(pkgs) | library(pkgs)
```

Algunas funciones vienen por defecto en **R** (8 paquetes) y otras están en paquetes creados por terceros. Los paquetes se encuentran disponibles en un repositorio llamado **CRAN**.

Para **instalar un paquete**:

```
install.packages("package")
```

Una vez instalado, para **cargar un paquete**:

```
library(package)
```

Alternativamente se puede **extraer una función** de un paquete instalado **sin necesidad de cargarlo**:

```
package::function()
```

Indexar en 

# Indexar vectores

! Es recurrente la selección de **elementos dentro de los vectores**. En este caso se quiere seleccionar los valores correspondientes a Marta y Pablo.

```
my_vector
```

```
## Juan Marta Pablo  
##     10      20      10
```

✓ Para indexar un vector se utiliza el corchete `[]`. Hay dos formas de hacerlo, si el vector no tiene nombres se utiliza la **posición del elemento**. En el caso de Marta es posición `2` mientras que Pablo está en la posición `3`.

```
my_vector[2:3]
```

```
## Marta Pablo  
##     20      10
```

# Indexar vectores

- ✓ La otra opción consiste en utilizar **los nombres de los elementos** del vector:

```
my_vector[c("Marta", "Pablo")]
```

```
## Marta Pablo  
##     20    10
```

# Indexar matrices

! A veces es necesario **seleccionar parte de las matrices**. En este caso se seleccionará las filas 1 y 3, y las columnas 3-5.

```
my_matrix
```

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

✓ La indexación de matrices se hace a partir del `[]`, separando con una coma `(,)` la selección de las **filas**, primero, de la selección de las **columnas**, segundo. En ambos casos según su **posición**:

```
my_matrix[c(1, 3), 3:5]
```

```
##      [,1] [,2] [,3]
## [1,]    9   13   17
## [2,]   11   15   19
```

# Indexar data frames

! En los data frames se suelen **seleccionar variables** o **filtrar casos**. Como ejemplo vamos a utilizar `my_df`. Para empezar vamos a subseleccionar la variable `var_num`.

```
my_df
```

```
##   var_char      var_num var_logic
## 1      a  0.77608922    FALSE
## 2      b -0.14854701     TRUE
## 3      c -0.38244872    FALSE
## 4      d -0.11902709    FALSE
## 5      e  0.06831824     TRUE
```

# Filtrar columnas

- ✓ Las columnas se pueden indexar de hasta tres maneras diferentes, según se utilice el **nombre** (`var_num`) o la **posición** (2) de la variable en el data frame y el **tipo de indexación** [ , ] o \$.

```
my_df$var_num # produce un vector  
my_df[, "var_num"] # produce un df  
my_df[, 2] # produce un df
```

# Filtrar filas por posición

! Ahora, además de escoger la variables `var_num` vamos a **seleccionar las filas** 1 y 3

- Para ello en el caso de haber utilizado `$` se emplean los **corchetes []**; en el caso de haber utilizado corchetes `[ , ]`, se completa la **parte izquierda**:

```
my_df$var_num[c(1, 3)] # produce un vector  
my_df[c(1, 3), ]$var_num # produce un vector  
  
my_df[c(1, 3), "var_num"] # produce un vector  
my_df[c(1, 3), 2] # produce un vector
```

# Filtrar filas por característica

!Lo más común es que las filas sean seleccionadas a partir de una **característica**, por ejemplo, si en `var_logic` es `FALSE`

```
my_df
```

```
##   var_char      var_num var_logic
## 1      a  0.77608922     FALSE
## 2      b -0.14854701      TRUE
## 3      c -0.38244872     FALSE
## 4      d -0.11902709     FALSE
## 5      e  0.06831824      TRUE
```

# Filtrar filas por característica

- Para ello en el caso de haber utilizado \$ se emplean los **corchetes [ ]**, mientras que en el caso de haber utilizado corchetes [ , ], se completa la **parte izquierda**:

```
my_df$var_num[my_df$var_logic == FALSE] # produce un vector  
my_df[my_df$var_logic == FALSE, ]$var_num # produce un vector  
  
my_df[my_df$var_logic == FALSE, "var_num"] # produce un vector  
my_df[my_df$var_logic == FALSE, 2] # produce un vector
```



# Quiz

¿Cómo se seleccionarían los casos para los que `var_logic == TRUE` y que `var_num < 0`?

```
my_df[my_df$var_logic == TRUE & my_df$var_num < 0,]
```

```
##   var_char   var_num var_logic
## 2          b -0.148547    TRUE
```

# Indexar lists

! También puede ser necesario acceder a **elementos concretos de las listas**. Por ejemplo, queremos **extraer el data frame** de `my_list`:

```
my_list  
## $vector  
##   Juan Marta Pablo  
##   10     20     10  
##  
## $df  
##   var_char      var_num var_logic  
## 1       a  0.77608922    FALSE  
## 2       b -0.14854701     TRUE  
## 3       c -0.38244872    FALSE  
## 4       d -0.11902709    FALSE  
## 5       e  0.06831824     TRUE
```

# Indexar lists

- Para acceder a los elementos de una lista existen dos sistemas: `$` y `[[[]]]`. El uso de `$` solo es posible **si los elementos de la lista tienen nombres**, de lo contrario habrá que seleccionarlos por su **posición** utilizando `[[[]]]`. Para comprobar si una lista tiene nombres se utiliza la función `names()`:

```
names(my_list)
```

```
## [1] "vector" "df"
```

```
my_list$df
```

```
my_list[[2]]
```

```
##   var_char      var_num var_logic
## 1      a  0.77608922    FALSE
## 2      b -0.14854701     TRUE
## 3      c -0.38244872    FALSE
## 4      d -0.11902709    FALSE
## 5      e  0.06831824     TRUE
```

# Modificar vectores

# Operadores

# Operaciones aritméticas

Las **operaciones aritméticas** son las más comunes, `+`, `-`, `/`, `*`, `^`, `%%`, `%/%`. Estas operaciones tienen sentidos con vectores numéricos o factores ordinales. Para ejemplificar las operaciones con vectores se crean dos vectores, el `v1` y `v2`:

```
v1 <- c(10, 5, 3)
v2 <- c(4, 5, 20)
```

Y ahora operamos con ellos:

```
v1+v2
```

```
## [1] 14 10 23
```

```
v1/v2
```

```
## [1] 2.50 1.00 0.15
```

# Operaciones relacionales

Los **operadores relacionales** (`< >`, `<= >=`, `== !=`) funcionan con cualquier tipo de vector comparan dos elementos y devuelven un vector de tipo lógico como resultado:

```
v1 == v2
```

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

```
v1 >= v2
```

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

# Operaciones lógicas

Los operadores lógicos (`!`, `&`, `|`, `%in%`) sirven para establecer relaciones entre elementos:

```
v1 | v2 == c(10, 5, 20)
```

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

```
10 %in% v1
```

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



# Quiz

¿Por qué el resultado de `v1 | v2 == c(10, 5, 20)` es igual a `TRUE TRUE TRUE` dados `V1` y `V2`?

```
v1
```

```
## [1] 10  5  3
```

```
v2
```

```
## [1] 4  5 20
```

```
v1 | v2 == c(10, 5, 20)
```

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

# Factor

# ⚠️ ⚠️ ⚠️ Atención ⚠️ ⚠️ ⚠️



- Los factores son recodificados a partir de los niveles
- La intervención de los factores en fórmulas lógicas se hace a través de los niveles
- Ejemplo:

```
region <- recode(ciudad,  
                  "Valladolid" = "CyL",  
                  "Salamanca" = "CyL")
```

## SPSS o Stata

- Las variables son recodificadas en base a los valores
- La intervención de las variables en fórmulas lógicas se hace a través de los valores
- Ejemplo:

```
# SPSS  
recode ciudad (3 = 2) (else = copy) into region.  
val labels region 1"Madrid" 2"CyL".
```

# Cambiar las etiquetas de un factor

```
recode(x, ..., .default, .missing) | table(x, y)
```

! Cambiar las etiquetas de labels\_factor\_recode de forma que "Desemp." → "Desempleado" y "Retirado" → "Jubilado".

- Una de las operaciones básicas con los factores es cambiar las etiquetas, para ello se utiliza la función `recode()`:

```
labels_factor_recode <- recode(labels_factor,
                                "Desemp." = "Desempleado",
                                "Retirado" = "Jubilado")

table(labels_factor, labels_factor_recode)
```

```
##          labels_factor_recode
## labels_factor Desempleado Jubilado Trabaja
##   Desemp.        1        0        0
##   Retirado       0        1        0
##   Trabaja        0        0        3
```

# Reordenar los niveles de un factor

```
fct_relevel(x, ...)
```

! Hay que **reordenar los niveles** de un factor. Asociar el nivel "**Trabaja**" con el valor **1**.

- ✓ Los niveles de un factor se pueden reordenar con la función `fct_relevel()`. Los niveles no explicitados en la función son colocados al final en el mismo orden que estaban:

```
levels(labels_factor_recode)
```

```
## [1] "Desempleado" "Jubilado"     "Trabaja"
```

```
labels_factor_recode <- fct_relevel(labels_factor_recode, "Trabaja")  
levels(labels_factor_recode)
```

```
## [1] "Trabaja"      "Desempleado" "Jubilado"
```

# Combinar varios niveles de un factor

! Hay que recodificar el vector `labels_factor_recode` de forma que "Desempleado" = "No trabaja" y "Jubilado" = "No trabaja".

- ✓ La función `recode()` sirve para unir categorías de los factores, recodificar:

```
labels_factor_recode <- recode(labels_factor_recode,  
                                "Desempleado" = "No trabaja",  
                                "Jubilado" = "No trabaja")  
  
levels(labels_factor_recode)
```

```
## [1] "Trabaja"     "No trabaja"
```



# Acceder a los elementos de un data frame

Ahora sabemos modificar factores y hacer operaciones numéricas, pero los vectores que utilizamos se **encuentran generalmente dentro de un data frame**. ¿Cómo se modifica o crea un vector nuevo dentro de un data frame?

```
my_df$new_var <- my_df$var_num + 5
```

Utilizaremos el **operador \$ para acceder a los vectores** y modificarlos o emplearlos en otras transformaciones.

# Character

# Cadenas de texto en R

Uno de los problemas usuales que se plantean al manipular datos es el manejo de las cadenas de texto del tipo:

```
centros <- c("CEIP María de Molina: Madrid", "IES Juan de la Cierva: Barcelona", "IES Juan Robles: ")
```

```
...>
```

# Localizar

`str_locate(string, pattern) | str_locate_all(string, pattern)`

! Para separar los los nombres de los centros educativos de los municipios es necesario localizar `:`.

- ✓ En su forma más sencilla `str_locate()` permite localizar expresiones de texto (*p. ej.* "Madrid") o símbolos antecedidos por una doble barra (*p. ej.* "\\\":"). En este caso de lo que se trata es de localizar la posición de los dos puntos en cada cadena:

```
str_locate(string = centros, pattern = "\\\\":")
```

```
##      start end
## [1,]    21  21
## [2,]    22  22
## [3,]    16  16
```

# Remplazar

`str_replace(string, pattern, replacement) | str_replace_all(string, pattern, replacement)`

! Otra forma de limpiar texto es **sustituyendo** partes del mismo. Por ejemplo, en el caso de `centros` se podría sustituir `:` por la expresión `" de"`.

- Para ello, la función `str_replace()` puede sustituir una expresión o símbolo por otros:

```
str_replace(string = centros, pattern = "\\\:", replacement = " de") # nótese el espacio en blanco
```

```
## [1] "CEIP María de Molina de Madrid"
## [2] "IES Juan de la Cierva de Barcelona"
## [3] "IES Juan Robles de Madrid"
```

# Eliminar

`str_remove(string, pattern) | str_remove_all(string, pattern)`

Otra forma operación habitual es la **eliminación** de determinados elementos con el fin de limpiar el texto. En el caso de `centros` se trata de eliminar los dos puntos (:).

Para ello se utilizan las funciones `str_remove()` para eliminar la primera ocurrencia o `str_remove_all()` para eliminar todas las ocurrencias de la expresión:

```
str_remove(string = centros, pattern = "\\:")
```

```
## [1] "CEIP María de Molina Madrid"      "IES Juan de la Cierva Barcelona"  
## [3] "IES Juan Robles Madrid"
```

# Seleccionar

`str_sub(string, start, end)`

! La otra operación básica consiste en **seleccionar fragmentos** de las cadenas de texto. En este caso se trata, por ejemplo, de seleccionar el nombre de los centros omitiendo las ciudades.

✓ En este caso habría que emplear la función `str_sub()`, estableciendo el comienzo en el carácter `1` y el final en el carácter en el que se localiza : menos `1`.

```
ends <- str_locate(string = centros, pattern = "\\:")[,1]  
str_sub(string = centros, start = 1, end = ends-1)
```

```
## [1] "CEIP María de Molina" "IES Juan de la Cierva" "IES Juan Robles"
```

 Quiz

¿Por qué es necesario utilizar [,1] al final de la expresión `ends <- str_locate(centros, "\\:")[,1]` para que `str_sub(string = centros, start = 1, end = ends-1)` funcione?

```
str_locate(string = centros, pattern = "\\:")
```

```
##      start end
## [1,]    21  21
## [2,]    22  22
## [3,]    16  16
```

**Date**

# Formatear fechas

`parse_date_time()`

! En ocasiones las **fechas son cargadas como texto** y deben ser convertidas a formato fechas.

```
dates <- c("10/07/2019", "11/08/2019")
class(dates)
```

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

✓ Para convertir a fecha el texto se puede utilizar la función `parse_date_time()`, especificando el formato de la fecha del tipo "`dmy`" (día-mes-año):

```
lubridate::parse_date_time(x = dates, orders = "dmy")
```

```
## [1] "2019-07-10 UTC" "2019-08-11 UTC"
```

# Valores perdidos

# Los valores perdidos en

En  los **valores perdidos** toman la forma de `NA`, perdidos por el sistema. Para que códigos de encuesta como el 98 o 99 fueran tratados como perdidos habría que convertirlos en `NA`. Para exemplificar el tratamiento vamos a partir del siguiente vector que representa la variable `age`:

```
# variable age en la que 999 es NC  
age <- c(36, 44, NA, 999, 67, NA)
```

# Localizar los valores perdidos

`is.na(x) | sum(x)`

! Un problema es saber cuantos valores perdidos hay en cada objeto.

✓ Utilizar la función `is.na()` en combinación con `sum()`. Por un lado, `is.na()` produce un vector lógico en el que `TRUE` se refiere a los valores perdidos. Un vector lógico se puede transformar en numérico, en el que `TRUE` es 1 y `FALSE` es 0.

```
is.na(age)
```

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

```
sum(is.na(age))
```

```
## [1] 2
```

# Convertir valores en perdidos

! En ocasiones hay valores que deben ser transformados en perdidos, por ejemplo, en el caso de `age` el código `999`.

- ✓ Seleccionar los casos que tengan código `999` y asignarles `NA`.

```
age[age == 999] <- NA  
age
```

```
## [1] 36 44 NA NA 67 NA
```

# Tratar los valores perdidos

```
mean(x, na.rm)
```

! También hay que tratar los valores perdidos para el análisis. Al usarlo en algunas funciones, el resultado es **NA**:

```
mean(age)
```

```
## [1] NA
```

✓ En las funciones como `mean()` existe una opción `na.rm` que cuando se le asigna el valor `TRUE` excluye los valores `NA` del análisis:

```
mean(age, na.rm = TRUE)
```

```
## [1] 49
```

# Transformar objetos

# Paso 1: ¿qué tipo de objeto es?

```
class(x)
```

La función `class()` devuelve el tipo de objeto:

```
class(my_vector)
```

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

```
class(my_list)
```

```
## [1] "list"
```

# Paso 1: ¿qué tipo de objeto es?

Por su parte, `str()` nos informa de la estructura del objeto:

```
str(my_vector)
```

```
## Named num [1:3] 10 20 10
## - attr(*, "names")= chr [1:3] "Juan" "Marta" "Pablo"
```

```
str(my_list)
```

```
## List of 2
## $ vector: Named num [1:3] 10 20 10
##   ..- attr(*, "names")= chr [1:3] "Juan" "Marta" "Pablo"
## $ df    :'data.frame': 5 obs. of 3 variables:
##   ..$ var_char : Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
##   ..$ var_num  : num [1:5] 0.7761 -0.1485 -0.3824 -0.119 0.0683
##   ..$ var_logic: logi [1:5] FALSE TRUE FALSE FALSE TRUE
```

# Paso 2: transformar objetos

Las funciones de tipo `as.data.frame()` sirven para transformar los objetos de  sin embargo todas las transformaciones no son posibles. Principales transformaciones:

```
as.character()  
as.factor()  
as.numeric()  
as.data.frame()  
as.list()
```

# character → numeric

! Tras cargar un conjunto de datos una variable de tipo numérico, como puede ser edad, aparece **almacenada como texto**. El objetivo es calcular la media.

```
numbers <- c("19.0", "22.0", "33.0")
```

Partimos de un vector de tipo character que contiene números:

```
mean(numbers)
```

```
## Warning in mean.default(numbers): argument is not numeric or logical:  
## returning NA  
  
## [1] NA
```

# character → numeric

- ✓ Transformar el vector en numeric:

```
numbers_new <- as.numeric(numbers)  
mean(numbers_new)
```

```
## [1] 24.66667
```

# character → factor

`as.factor(x)`

!  R trata los **factores de forma específica** en algunos procedimientos, como por ejemplo en las regresiones `lm()`. Las variables de tipo factor son dicotomizadas, sin embargo, puede ocurrir que al cargar el conjunto de datos la variable categórica aparezca almacenada como character.

Partimos de un vector de tipo *character*, en este caso ciudades.

```
char <- c("Madrid", "Barcelona", "Madrid")
str(char)
```

```
## chr [1:3] "Madrid" "Barcelona" "Madrid"
```

 Transformar la variable en factor usando `as.factor()`:

```
factor_new <- as.factor(char)
str(factor_new)
```

```
## Factor w/ 2 levels "Barcelona","Madrid": 2 1 2
```

# Proyecto

# Proyecto y workflow

**Objetivo:** Crear una base de datos con las **notas de la EVaU** de los centros educativos de Castilla-La Mancha y las características de los centros.

1. Notas por centro educativo de la EVaU disponible en la UCLM en PDF.
2. Notas de los centros educativos disponibles en una base de datos de consulta individual de la Consejería de Educación.
3. Combinar las extracciones de las dos fuentes de información.

# Hoy..

- Conocer la información y planificar las tareas