Introducción a R (I)

FSC

Contents

Antes de empezar...

Los materiales de esta clase han sido preparados utilizando dos libros que están disponibles bajo licencia OUP y que os invito a explorar:

Hands-on programming with R https://rstudio-education.github.io/hopr/

Data Science with R https://rafalab.github.io/dsbook/

Qué es R

- R es un lenguaje de programacion interpretado (no precisa compilacion) que permite manipular data, hacer cálculos (de muy sencillos a muy complejos) y con una gran capacidad para realizar gráficos de alta calidad y complejidad.
- Está escrito en R, Fortran y C para funciones que requieren una mayor rapidez.
- Es la versión Open source de un software S desarrollado en los Bell Laboratories (formerly AT&T, now Lucent Technologies) por John Chambers y co en 1976. Ross Ihaka y Robert Gentleman (de ahi su nombre) de la universidad de Auckland, NZ, fueron sus creadores en 1992, aunque la primera version beta es de 2000.
- Fue desarrollado por estadisticos y analistas de datos, no por ingenieros, lo cual explica mucha de su esencia.
- Puntos fuertes:
  - Gráficos de alta calidad con no demasiado esfuerzo
  - Una gran comunidad desarrollando nuevos paquetes
  - Casi cualquier problema estadistico puede ser resuelto con las utilidades generadas por algun grupo en el mundo
  - Es multiplatform: puede correr en todos los SO
  - Es interactivo: la idea es ser capaz de ir implementando y viendo el resultado en tiempo real, lo cual permite un buen análisis de los datos.
- Características computacionales
  - Cada funcion que se genera es un nuevo objeto que ademas puede ser compartido con el resto de la comunidad.
  - La mayor parte del codigo fuente está escrito en R, lo cual hace muy sencillo entenderlo y tomar decisiones acerca de cada algoritmo utilizado.
  - Se pueden escribir funciones con C para mejorar la velocidad. A traves de suites como RStudio se puede integrar R con Python y con otros lenguajes mas especificos como STAN.
  - R y Latex se pueden utilizar de manera combinada para producir reports de gran calidad y fomentar reproducibilidad.

Estructura de R

- Con codigo de R se escriben funciones que tambien son objetos de R y que a su vez constituyen los paquetes.
- La distribucion base de R contiene 8 paquetes: stats, graphics, grDevices, utils, datasets, methods, base. Puedes verlos abriendo la consola y escribiendo `sessionInfo()`
- CRAN (The Comprehensive R Archive Network) es una red de repositorios conectados a traves de ftp por todo el mundo y que contienen versiones idénticas, documentadas y "up to date" de los paquetes oficiales de R, mantenidos por el equipo de desarrolladores de R
- Otros paquetes pueden descargarse de las páginas web de sus creadores, de github, etc.

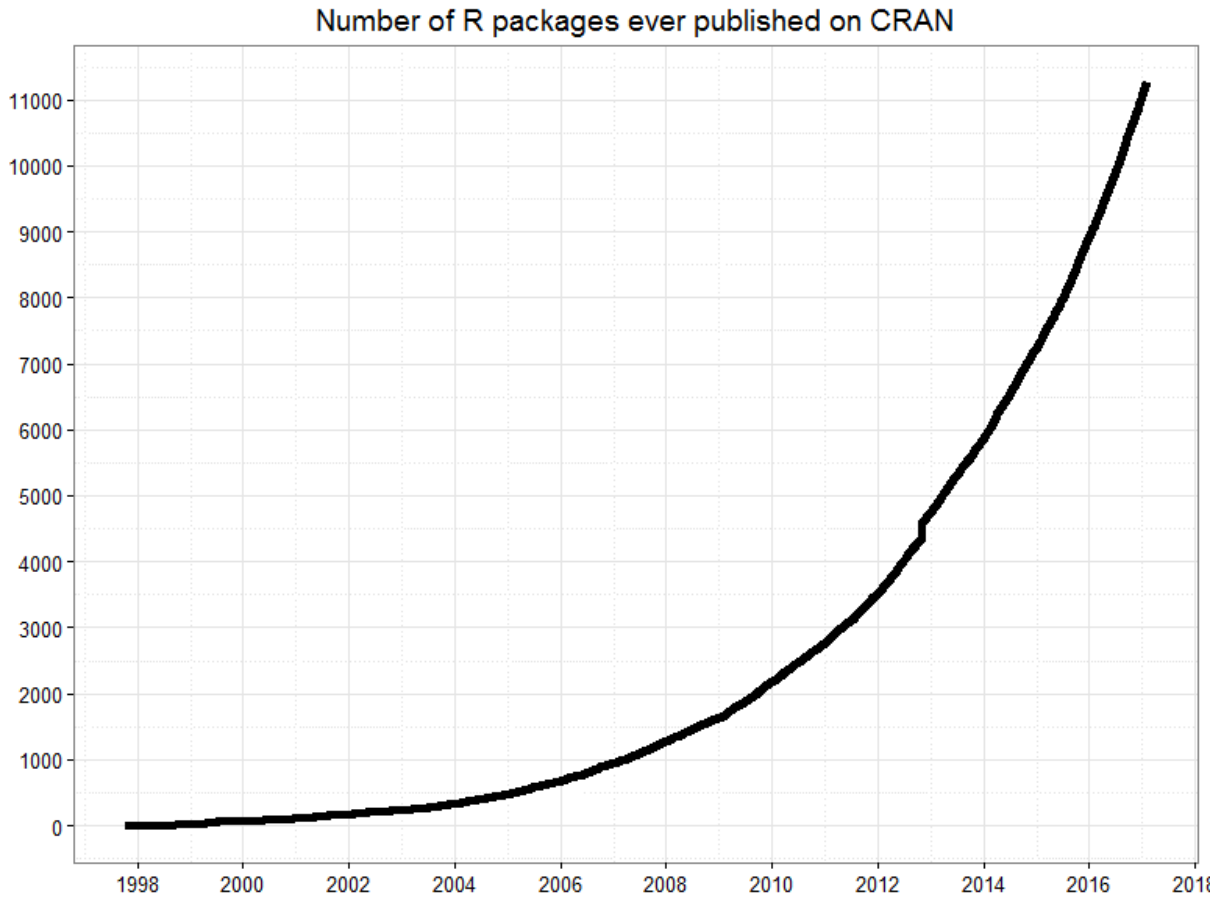


Figure 1: Number of packages

- Bioconductor es un conjunto de paquetes especificamente desarrollados para el análisis de datos biomédicos.

Getting started

Abre la consola de R, vamos a realizar un cálculo simple en la consola.

Calcula el precio final de la factura del albañil si la obra ha costado 871€

```
871* 1.21
```

```
## [1] 1053.91
```

A no ser que todo nuestro trabajo como data scientists vaya a ser tan sencillo como esto, tendremos que utilizar un IDE (Interactive Development Environment). El más popular es RStudio. Para instalarlo, sigue las instrucciones en: https://rstudio-education.github.io/hopr/starting.html#starting

Un IDE tiene integrada una consola y un editor para producir scripts que pueden ser grabados para ser usados posteriormente.

Operaciones aritméticas

```
1+2
```

```
## [1] 3
```

```
1-2
```

```
## [1] -1
```

```
1/(1-2)
```

```
## [1] -1
```

Troubleshooting

1. Qué sucede en la consola si tecleamos un comando incompleto?

```
871*
```

```
## Error: <text>:2:0: unexpected end of input
## 1: 871*
##    ^
```

2. Si tecleamos un comando que R no reconoce entonces nos devuelve un error

```
871 % 2
```

```
## Error: <text>:1:5: unexpected input
## 1: 871 % 2
##         ^
```

Ejercicio 1: Magia con números

1. Elige un número del 1 al 10 y súmale 2
2. Multiplica el resultado por 3
3. Réstale 6
4. Divide lo que obtienes por 3

Objetos en R

Vectores atomicos

Son vectores de con varios componentes, todos del mismo tipo de manera que cada uno de ellos ocupa una unica posición. Por ejemplo, si queremos construir un vector con los 6 posibles resultados que surgen de tirar un dado:

```
1:6
```

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

Otra forma de escribirlo es:

```
c(1,2,3,4,5,6)
```

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

Si miramos a la izquierda, en el environment, no vemos que se haya almacenado esta información. Para usar otra vez este vector que hemos creado se lo asignamos a un *objeto* de nombre *dado*.

```
dado=1:6
```

Ahora ya aparece en el environment. Podemos ver su tamaño usando la funcion *length()*

```
length(dado)
```

```
## [1] 6
```

Un valor único es un vector atómico de longitud 1:

```
a<-1
length(a)
```

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

```
is.vector(a)
```

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

Y podemos realizar operaciones aritmeticas con un vector, al fin y al cabo es sólo una repetición de números:

```
dado+2
```

```
## [1] 3 4 5 6 7 8
```

```
dado*2
```

```
## [1]  2  4  6  8 10 12
```

```
dado+dado
```

```
## [1]  2  4  6  8 10 12
```

```
dado*dado
```

```
## [1]  1  4  9 16 25 36
```

```
dado/dado
```

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

Estas operaciones se hacen elemento a elemento. ¿Qué pasa si le damos a R un vector más corto que otro y le pedimos que haga alguna operacion aritmética? R repetirá el vector corto tantas veces necesite para ser del mismo tamaño que el largo. Esto se conoce como *vector recycling*, and it helps R do element-wise operations:

```
moneda=c(1,2)
moneda+dado
```

```
## [1] 2 4 4 6 6 8
```

¿Pero, no puedo usar operaciones matriciales en R? Multiplicación matricial y outer multiplication:

```
dado%*%dado
```

```
##      [,1]
## [1,]   91
```

```
dado%o%dado
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    2    4    6    8   10   12
## [3,]    3    6    9   12   15   18
## [4,]    4    8   12   16   20   24
## [5,]    5   10   15   20   25   30
## [6,]    6   12   18   24   30   36
```

Nota: R es case-sensitive. Asi que un vector llamado "dado" no es el mismo que uno llamado "Dado"

```
Dado
```

```
## Error in eval(expr, envir, enclos): object 'Dado' not found
```

Nota 2: En cualquier momento puedes ver los objetos que hay en tu entorno usando ls()

```
ls()
```

```
## [1] "a"      "dado"   "moneda"
```

Tipos de vectores:

R tiene seis tipos distintos de vectores atómicos: Hay seis tipos de vectores atómicos: doubles, integers, characters, logicals, complex, and raw.

Double & integer

Un vector de dobles almacena números de cualquier tipo. Los enteros son un subconjunto de los dobles, de ahi que a veces nos refiramos a un vector de dobles como *numérico*. El término *doble* se refiere al número de bits que usa el ordenador para guardar esos números

Si un número no tiene parte decimal entonces es entero, sea positivo o negativo. Para que al generar un número lo trate como entero y no como doble podemos especificarlo con una L:

```
int <- c(-1L, 2L, 4L)
int
```

```
## [1] -1  2  4
```

```
## -1  2  4

typeof(int)
```

```
## [1] "integer"
```

```
typeof(4)
```

```
## [1] "double"
```

```
typeof(4L)
```

```
## [1] "integer"
```

A no ser que lo especifiquemos, R guarda los números como doble por defecto. ¿Por qué necesitamos a veces entonces guardar algunos números como enteros? Cada doble se guarda con una precisión de 16 decimales que no necesitamos si son todos ceros. Será más eficiente computacionalmente guardar el número sin esa precisión. Por otro lado, números irracionales como *pi* con un número infinito de decimales no tienen suficiente precisión almacenándolos solo con 16 dígitos. Ese redondeo a veces tiene resultados inesperados, por ejemplo:

```
sqrt(2)^2 - 2
```

```
## [1] 4.440892e-16
```

```
## 4.440892e-16
```

No es 0, que es lo que esperaríamos. Esto son *floating point errors* y *floating point arithmetic* que no es parte del objetivo de R.

Vectores de caracteres

Si los elementos de un vector son pequeños textos hablamos de un vector de tipo *character*

```r
text<-c("Hello","World")
text
```

```
## [1] "Hello" "World"
```

```r
typeof(text)
```

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

¿Cuáles son caracteres? ¿Cuál doble? 1, "1", "uno". Compruébalo.

Vectores lógicos

Las entradas de un vector lógico pueden tomar dos valores: TRUE o FALSE.

```r
1>2
```

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

```r
1<2
```

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

```r
typeof(1<2)
```

```
## [1] "logical"
```

Vectores complejos y crudos

```r
comp <- c(1 + 1i, 1 + 2i, 1 + 3i)
comp
```

```
## [1] 1+1i 1+2i 1+3i
```

```r
## 1+1i 1+2i 1+3i

typeof(comp)
```

```
## [1] "complex"
```

```r
## "complex"
```

Los vectores de datos crudos *raw* almacenan bytes. No vamos a profundizar mucho mas en ello porque no los usaremos, pero puedes quedar un vector vacio de tamaño 3 con el comando:

```r
raw(3)
```

```
## [1] 00 00 00
```

```r
## 00 00 00

typeof(raw(3))
```

```
## [1] "raw"
```

```r
## "raw"
```

Ejercicio 2: Una mano de cartas

Genera un vector que reproduzca las cartas repartidas a un jugador de mus en una mano jugando a la baraja española. Recuerda que hay cuatro palos: bastos, oros, copas y espadas. ¿Qué tipo de vector es? ¿Con que funcion lo confirmamos?

```r
mano1<-c("as_bastos","as_copas","as_oros","as_espadas")
is.character(mano1)
```

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

Matrix y array

Un matriz es un conjunto de datos de dos dimensiones, como una tabla. Al igual que los vectores todas las entradas de una matriz tienen que ser del mismo tipo. El array es como una matriz pero de mas dimensiones. Una vez mas todos los datos tienen que ser del mismo tipo.

Las funciones *matrix()* y *array()* sirven para transformar un vector numérico en una matriz o en un array:

```r
matrix(dado,nrow=2,byrow = T)
```

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

```r
ar <- array(c(11:14, 21:24, 31:34), dim = c(2, 2, 3))
```

Ejercicio 3: Jugando al mus

Construye una matriz que contenga una mano de una pareja jugando al mus:

(Jugador 1): los cuatro reyes de los cuatro palos

(Jugador 2): escalera de oros

```r
jugador1<-c("rey_bastos","rey_copas","rey_oros","rey_espadas")
jugador2<-c("sota_oros","caballo_oros","rey_oros","as_oros")
mano<-cbind(jugador1,jugador2)
```

Atributos

Los atributos son piezas de información de las entradas de los elementos de un objeto en R. Por ejemplo, podemos generar un vector con dos entradas: 0 y 1 para cara y cruz de una moneda y añadir el nombre para tenerlo claro:

```r
moneda<-c(0,1)
moneda
```

```
## [1] 0 1
```

```r
attributes(moneda)=list("cruz"=0,"cara"=1)
moneda
```

```
## [1] 0 1
## attr(,"cruz")
## [1] 0
## attr(,"cara")
## [1] 1
```

Cuando no tiene atributos el objeto devuelve un valor *NULL*.

```r
attributes(dado)
```

```
## NULL
```

NULL es un vector vacío. Lo encontraras a menudo por ejemplo al intentar llamar a un objeto que no se ha definido previamente.

names

Un tipo de atributo de un vector son los nombres de cada una de sus entradas. En el caso de la moneda podríamos haberlos usado para definir ese tipo de atributo:

```
moneda<-c(0,1)
names(moneda)=c("cruz","cara")
moneda
```

```
## cruz cara
##    0    1
```

```
names(moneda)
```

```
## [1] "cruz" "cara"
```

```
names(dado)
```

```
## NULL
```

dim

La dimension es otro atributo que se puede asignar a un vector y asi transformarlo en una matriz de la dimensionalidad deseada:

```
dim(dado)=c(2,3)
dado
```

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

Nota como se van colocando los datos del vector, por columnas de arriba abajo. Una matriz con mas de dos dimensiones es un array:

```
dim(dado)=c(1,2,3)
dado
```

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

class

Otro tipo de atributo de los objetos de R es la clase. Mientras que el tipo no cambia entre

```
typeof(dado)
```

```
## [1] "integer"
```

```r
typeof(matrix(dado,nrow = 2))
```

```
## [1] "integer"
```

Si que cambia su clase:

```r
class(dado)
```

```
## [1] "array"
```

```r
class(matrix(dado,nrow = 2))
```

```
## [1] "matrix"
```

En realidad un entero es un tipo especial de matriz, pero tienen dos clases distintas.

Factores

Los factores son un tipo de objeto muy controvertido pero muy utilizado en R. Almacenan información categórica como el color de ojos, la raza, el género... Este tipo de objeto es difícil de manejar, sobre todo si no nos damos cuenta de que lo tenemos entre nuestros datos. Sin embargo, es esencial en modelado estadístico:

```r
gender<-factor(c(rep("M",6),rep("F",6)))
weight<-c(rnorm(6,75,5),rnorm(6,65,3))
lm(weight~gender)
```

```
##
## Call:
## lm(formula = weight ~ gender)
##
## Coefficients:
## (Intercept)      genderM
##      66.069        7.559
```

Independientemente del tipo de vector que estamos transformando en factor, R lo transformará en un entero y los almacenará en un vector de enteros. Además añadirá un atributo especifico de los factores: los niveles de cada categoría del factor *levels* :

```r
typeof(gender)
```

```
## [1] "integer"
```

```r
## "integer"

attributes(gender)
```

```
## $levels
## [1] "F" "M"
##
## $class
## [1] "factor"
```

¿Cómo almacena R exactamente el factor?

```r
unclass(gender)
```

```
##  [1] 2 2 2 2 2 2 1 1 1 1 1 1
## attr(,"levels")
## [1] "F" "M"
```

Manipular factores cambiándolos de tipo es muy peligroso y es una de las mayores fuentes de errores en R:

```r
as.numeric(gender)[1]+as.numeric(gender)[2]
```

```
## [1] 4
```

Se pueden transformar en vectores de caracteres usando:

```r
as.character(gender)
```

```
##  [1] "M" "M" "M" "M" "M" "M" "F" "F" "F" "F" "F" "F"
```

Importante: Ten siempre en mente que los factores parecen caracteres pero en realidad son enteros. Intenta evitarlos en programación con R, reservándolos para el modelado estadístico

data.frames

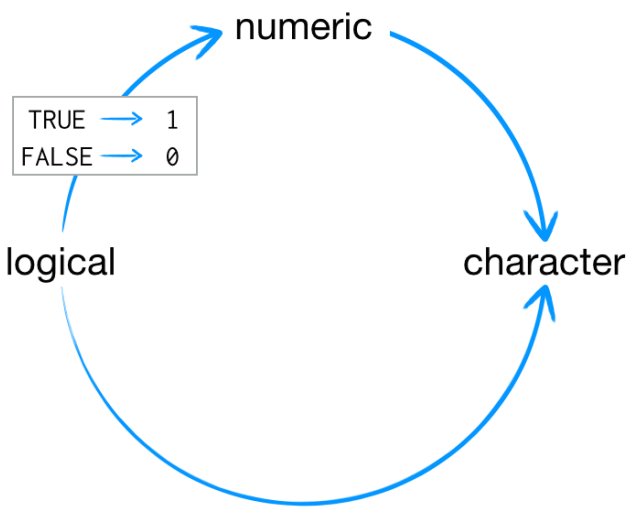
Coerción

Hasta ahora hemos estado trabajando con vectores en los que todos los elementos eran del mismo tipo. QUe pasa si intentamos mezclar por ejemplo caracteres con enteros?

```r
c("F","M",1)
```

```
## [1] "F" "M" "1"
```

R transforma el numérico en un caracter, la opción más sencilla. A esto se le llama *coertiony* en R sigue las siguientes reglas:



Lo mismo sucede si intentamos operar con vectores lógicos:

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

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

En R los datos se pueden convertir de un tipo a otro utilizando *as.character()*, *as.numeric()*, *as.logical()*

Si queremos almacenar distintos tipos de datos en un mismo objeto, necesitamos un objeto de R llamado data.frame. Imaginemos que queremos definir un objeto de R en el que tengamos la puntuación que se le otorga en un juego de cartas a la sota, el caballo y el rey. NO podemos hacerlo con una matriz, porque transforma las puntuaciones en caracteres, impidiendonos operar con ellas:

```r
p<-cbind(c("sota","caballo","rey"),c(10,11,12))
p[1,2]+p[2,2]
```

```
## Error in p[1, 2] + p[2, 2]: non-numeric argument to binary operator
```

Sin embargo, si usamos un data.frame:

```
p<-data.frame(c("sota","caballo","rey"),c(10,11,12))
p[1,2]+p[2,2]
```

```
## [1] 21
```

Podemos incluso asociar distintos nombres a las columnas, que en realidad serán tratadas como vectores atomicos independientes:

```
p<-data.frame(figura=c("sota","caballo","rey"),puntos=c(10,11,12))
sum(p$puntos[1:2])
```

```
## [1] 21
```

Un nuevo tipo de atributo para los data.frames es *str()* que nos define la estructura del objeto:

```
attributes(p)
```

```
## $names
## [1] "figura" "puntos"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

```
str(p)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ figura: Factor w/ 3 levels "caballo","rey",..: 3 1 2
##  $ puntos: num  10 11 12
```

Fijaos que el data.frame ha transformado la variable caracter en un factor y esto es peligroso! Podemos reconducirlo haciendo:

```
p<-data.frame(figura=c("sota","caballo","rey"),puntos=c(10,11,12),stringsAsFactors = FALSE)
str(p)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ figura: chr  "sota" "caballo" "rey"
##  $ puntos: num  10 11 12
```

Listas

Pero, ¿qué sucede si necesitamos un objeto en el que tengamos que aglutinar vectores atómicos de distintos tipos y tamaños? Para eso tenemos las listas. Realmente los data.frames son un tipo de lista:

```
attributes(p)
```

```
## $names
## [1] "figura" "puntos"
##
## $class
## [1] "data.frame"
##
## $row.names
```

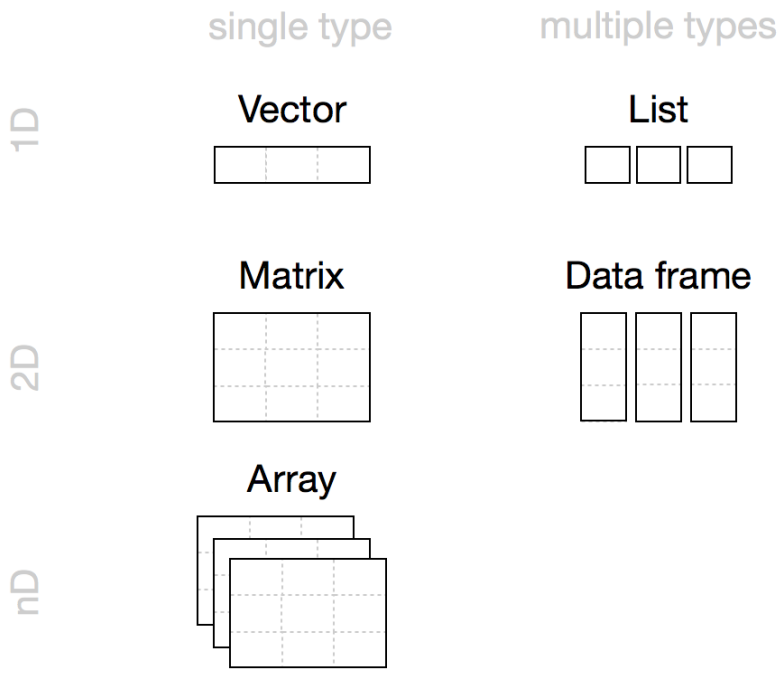


Figure 2: Resumen de los tipos de objetos en R

```
## [1] 1 2 3
```

```
typeof(p)
```

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

```
str(p)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ figura: chr  "sota" "caballo" "rey"
##  $ puntos: num  10 11 12
```

Por ejemplo, algo muy complicado sería:

```
list1 <- list(100:130, "R", list(TRUE, FALSE))
```

Ejercicio 4: Construye una baraja española de cartas

La baraja española tiene cuatro palos y 10 cartas por palo. Cada una de ellas tiene una puntuación: As (11 puntos), Tres (10 puntos), Rey (4 puntos), Caballo (3 puntos) y Sota (2 puntos). El resto de cartas no tienen valor puntuable.

4.1. Construye una lista que contenga todas las posibles cartas que hay en la baraja y sus puntuaciones

4.2. Construye un data frame donde cada fila represente una combinacion de carta/palo/puntuacion

4.3. Cuantos puntos suma en total la baraja española?

```
## [1] 120
```

Funciones

Esta es quizas una de las mayores peculiaridades de R frente a otros lenguajes de programación: podemos construir nuestras propias funciones en R que son, a su vez, también objetos de R. Podemos hacer el mismo tipo de cosas con las funciones que con el resto de objetos.

Por ejemplo, throw puede ser una funcion que nos de el primer valor de nuestro objeto dado:

```
throw<-function(x){return(x[1])}
throw(dado)
```

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

o

```
throw<-function(x){
  a<-sample(x,1)
  b<-2*a
  return(b)
  }
throw(dado)
```

```
## [1] 12
```

El resultado de "throw" es un objeto de R que puede ser manipulado como el resto de objetos.

R tiene muchas funciones implementadas. Por ejemplo, min nos da el menor número de una serie, max el maximo. . . tienen la misma sintaxis que una función implementada por nosotros.

```
min(dado)
```

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

```
max(dado)
```

```
## [1] 6
```

```
which.min(dado)
```

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

```
which.max(dado)
```

```
## [1] 6
```

Entornos en R (environment)

Para seguir avanzando es importante que entendamos cómo R guarda los objetos. En realidad funciona de una forma parecida al sistema de ficheros de un ordenador. Cada objeto se guarda dentro de un environment (o entorno) que a su vez puede estar dentro de otro, con una estructura jerárquica. Vamos a ver la estructura de nuestro entorno en esta sesion:

```
#install.packages("pryr")
library(pryr)
```

```
## Warning: package 'pryr' was built under R version 3.6.2
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
parenvs(all = TRUE)
```

```
##    label                               name
## 1  <environment: R_GlobalEnv>          ""
## 2  <environment: package:pryr>         "package:pryr"
## 3  <environment: package:stats>        "package:stats"
## 4  <environment: package:graphics>     "package:graphics"
## 5  <environment: package:grDevices>    "package:grDevices"
## 6  <environment: package:utils>        "package:utils"
## 7  <environment: package:datasets>     "package:datasets"
## 8  <environment: package:methods>      "package:methods"
## 9  <environment: 0x00000000130695f0>   "Autoloads"
## 10 <environment: base>                 ""
## 11 <environment: R_EmptyEnv>           ""
```

```
x <- 5
where("x")
```

```
## <environment: R_GlobalEnv>
```

```
#> <environment: R_GlobalEnv>
where("mean")
```

```
## <environment: base>
```

R_EmptyEnv es el único entorno que no tiene ningún padre. Los entornos de R no se almacenan físicamente en tu ordenador, están en memoria RAM.

Para explorar la jerarquía de nuestro entorno tenemos algunas funciones:

```
as.environment("package:stats")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "C:/Users/fscabo/Documents/R/R-3.6.1/library/stats"
```

```
globalenv()
```

```
## <environment: R_GlobalEnv>
```

```
baseenv()
```

```
## <environment: base>
```

```
emptyenv()
```

```
## <environment: R_EmptyEnv>
```

```
parent.env(globalenv())
```

```
## <environment: package:pryr>
## attr(,"name")
## [1] "package:pryr"
## attr(,"path")
## [1] "C:/Users/fscabo/Documents/R/R-3.6.1/library/pryr"
```

```
search()
```

```
##  [1] ".GlobalEnv"        "package:pryr"      "package:stats"
```

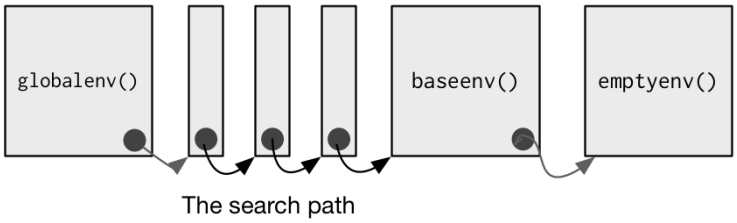


Figure 3: Jerarquía de entornos

```
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

globalenv(), baseenv(), los environments en el search path, y emptyenv() se conectan como se muestra en la figura. Cada vez que se abre un nuevo paquete con library() se inserta entre el global environment y el ultimo entorno de paquete generado.

Los objetos guardados en un entorno pueden ser vistos con:

```
ls(emptyenv())
```

```
## character(0)
```

```
ls(globalenv())
```

```
##  [1] "a"           "ar"          "baraja"      "baraja.list" "carta"
##  [6] "carta1"      "comp"        "dado"        "gender"      "int"
## [11] "jugador1"    "jugador2"    "list1"       "mano"        "mano1"
## [16] "moneda"      "p"           "palo"        "palo1"       "puntos"
## [21] "text"        "throw"       "weight"      "x"
```

En cualquier momento podemos saber cual es el entorno con el que está trabajando R:

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
new<-"Hello Global"
```

Si guardamos algo en ese objeto new R lo sobreescribira:

```
new <- "Hello Active"

new
```

```
## [1] "Hello Active"
```

```
## "Hello Active"
```

Algunas funciones guardan objetos temporales.

```
roll <- function() {
  die <- 1:6
  dice <- sample(die, size = 2, replace = TRUE)
  sum(dice)
  return(environment())
}
```

Para no sobreescribir un posible objeto que se llame “dice”, R crea un nuevo entorno cada vez que se evalua

una funcion. De ahi las dependencias que hemos visto al principio.Al finalizar la evaluación, R vuelve al entorno global:

```
show_env <- function(){
  list(ran.in = environment(),
    parent = parent.env(environment()),
    objects = ls.str(environment()))
}
show_env()
```

```
## $ran.in
## <environment: 0x00000000154366f0>
##
## $parent
## <environment: R_GlobalEnv>
##
## $objects
```

R ha creado un nuevo *runtime environment* durante el tiempo que ha estado ejecutando la función.

Notación en R

Seleccionando datos

Utilizaremos brackets *[ ]* para acceder a los elementos de un objeto en R (excepto listas) con tantas comas como dimensiones tenga el objeto. Por ejemplo,

```
baraja[1,]
```

```
##   carta1  palo1 puntos
## 1     As bastos     11
```

```
baraja<-data.frame(
carta=rep(c("as",as.character(seq(2,7)),"sota","caballo","rey"),4),
palo=rep(c("bastos","copas","espadas","oros"),each=10),
puntos=rep(c(11,0,10,0,0,0,0,2,3,4),4),stringsAsFactors = F)

baraja[1,]
```

```
##   carta   palo puntos
## 1    as bastos     11
```

```
baraja[,1]
```

```
##  [1] "as"      "2"       "3"       "4"       "5"       "6"       "7"
##  [8] "sota"    "caballo" "rey"     "as"      "2"       "3"       "4"
## [15] "5"       "6"       "7"       "sota"    "caballo" "rey"     "as"
## [22] "2"       "3"       "4"       "5"       "6"       "7"       "sota"
## [29] "caballo" "rey"     "as"      "2"       "3"       "4"       "5"
## [36] "6"       "7"       "sota"    "caballo" "rey"
```

```
baraja[1,1]
```

```
## [1] "as"
```

Nos dan la primera fila, la primera columna o el valor que ocupa la posicion (1,1) en el objeto baraja

Hay seis tipos de índices que se pueden usar en R:

- Enteros positivos

```
baraja[1,1]
```

```
## [1] "as"
```

Podemos tambien usar vectores de enteros positivos como indices:

```
baraja[1,c(1,2,3)]
```

```
##   carta   palo puntos
## 1    as bastos     11
```

Importante: si seleccionamos toda una columna R nos devolverá un vector a menos que le digamos que nos mantenga el objeto resultante como un data.frame para lo que usamos el comando *drop=FALSE*

```
baraja[1:10,1]
```

```
##  [1] "as"      "2"       "3"       "4"       "5"       "6"       "7"
##  [8] "sota"    "caballo" "rey"
```

```
baraja[1:10,1,drop=F]
```

```
##      carta
## 1       as
## 2        2
## 3        3
## 4        4
## 5        5
## 6        6
## 7        7
## 8     sota
## 9  caballo
## 10     rey
```

- Enteros negativos

```
baraja[1,-1]
```

```
##     palo puntos
## 1 bastos     11
```

```
baraja[1,]
```

```
##   carta   palo puntos
## 1    as bastos     11
```

- Zero

En R el indexado comienza en 1.

```
baraja[0,0]
```

```
## data frame with 0 columns and 0 rows
```

- Espacio Selecciona toda la fila o toda la columna
- Logicos Nos ayuda a seleccionar filas o columnas, es como si preguntaramos para por ejemplo cada columna si queremos seleccionarla o no.

```
baraja[1,c(TRUE,FALSE,TRUE)]
```

```
##   carta puntos
```

```
## 1    as     11
```

- Nombres Las columnas o filas pueden accederse directamente usando el nombre.

```
baraja[1,c("carta","puntos")]
```

```
##   carta puntos
## 1    as     11
```

Si hablamos de data.frames, no necesitamos los corchetes nos basta con:

```
baraja$carta[1]
```

```
## [1] "as"
```

Para listas utilizamos doble corchete:

```
baraja.list[[1]]
```

```
##  [1] "As"      "2"       "3"       "4"       "5"       "6"       "7"
##  [8] "sota"    "caballo" "rey"
```

Ejercicio 5: Baraja y reparte

5.1. Construye una función *deal<-function(){return()}* que seleccione la primera carta de tu baraja

5.2. Baraja de manera que queden ordenadas de otra forma. Genera una función llamada *shuffle* que pueda ser utilizada para esto más veces.

5.3. Selecciona la primera carta de esta baraja revuelta

```
deal<-function(x){return(x[1,])}
deal(baraja)
```

```
##   carta   palo puntos
## 1    as bastos     11
```

```
shuffle<-function(x){return(x[c(10:20,20:30,1:10),])}
baraja2<-shuffle(baraja)
deal(baraja2)
```

```
##    carta   palo puntos
## 10   rey bastos      4
```

```
baraja3<-shuffle(baraja2)
deal(baraja3)
```

```
##      carta  palo puntos
## 19 caballo copas      3
```

Modificando los objetos

Imagina que quieres cambiar la puntuación de tu baraja porque vas a jugar a otro juego donde por ejemplo cada carta tiene una puntuación equivalente a su numero real, de 1 a 9. Es conveniente, si vamos a modificar un objeto que ya existe y que es complejo, crear una copia de el con un nombre distinto:

```
baraja2 <- baraja
```

Cambiar los valores en su posición

Vamos a definir un vector atómico de 0s.

```r
vec <- c(0, 0, 0, 0, 0, 0)
vec
```

```
## [1] 0 0 0 0 0 0
```

```
##  0 0 0 0 0 0
```

Accedemos al primer valor de `vec`:

```r
vec[1]
```

```
## [1] 0
```

```
##  0
```

¿Cómo lo modificamos, asignándole un nuevo valor?:

```r
vec[1] <- 1000
vec
```

```
## [1] 1000    0    0    0    0    0
```

```
## 1000    0    0    0    0    0
```

Podemos reemplazar varios valores a la vez usando vectores de índices:

```r
vec[c(1, 3, 5)] <- c(1, 1, 1)
vec
```

```
## [1] 1 0 1 0 1 0
```

```
##  1 0 1 0 1 0
```

```r
vec[4:6] <- vec[4:6] + 1
vec
```

```
## [1] 1 0 1 1 2 1
```

```
## 1 0 1 1 2 1
```

Podemos también expandir nuestro vector, añadirle entradas:

```r
vec[7] <- 0
vec
```

```
## [1] 1 0 1 1 2 1 0
```

```
## 1 0 1 1 2 1 0
```

De la misma forma, podemos añadirle nuevas variables a un data.frame:

```r
baraja2<-baraja
baraja2$puntos2 <- rep(1:10,4)
head(baraja2)
```

```
##   carta   palo puntos puntos2
## 1    as bastos     11       1
## 2     2 bastos      0       2
## 3     3 bastos     10       3
## 4     4 bastos      0       4
## 5     5 bastos      0       5
## 6     6 bastos      0       6
```

Se pueden también quitar columnas asignándoles `NULL`, que ya habíamos visto que era el vector vacío:

```
baraja2$puntos2 <- NULL
head(baraja2)
```

```
##   carta   palo puntos
## 1    as bastos     11
## 2     2 bastos      0
## 3     3 bastos     10
## 4     4 bastos      0
## 5     5 bastos      0
## 6     6 bastos      0
```

Imagina que quieres asignarle a los ases y a los treses una puntuación de 0 porque en otro juego no valen nada:

```
baraja2$puntos[c(1,11,21,31)] <- c(0,0,0,0)
head(baraja2)
```

```
##   carta   palo puntos
## 1    as bastos      0
## 2     2 bastos      0
## 3     3 bastos     10
## 4     4 bastos      0
## 5     5 bastos      0
## 6     6 bastos      0
```

```
baraja2$puntos[c(3,13,23,33)] <- 0
head(baraja2)
```

```
##   carta   palo puntos
## 1    as bastos      0
## 2     2 bastos      0
## 3     3 bastos      0
## 4     4 bastos      0
## 5     5 bastos      0
## 6     6 bastos      0
```

Importante: esto funciona para todo tipo de objetos en R: selecciona los valores que quieres cambiar usando la notacion de R y asígnale el valor que quieras usando el operador <-

Recuerdas que hicimos una función para barajar las cartas? Vamos a usar ahora esa baraja revuelta:

```
baraja3 <- shuffle(baraja)
```

Cómo encontramos ahora los ases para cambiar su valor?

```
head(baraja3)
```

```
##    carta   palo puntos
## 10   rey bastos      4
## 11    as  copas     11
## 12     2  copas      0
## 13     3  copas     10
## 14     4  copas      0
## 15     5  copas      0
```

Vamos a preguntarle a R dónde están los ases y pedirle que modifique sus valores. Para eso necesitamos logical subsetting.

Logical Subsetting

En R teníamos un tipo de vectores que eran los lógicos y que podían usarse como indices. R devuelve los valores en las posiciones indexadas con *TRUE*. Ese vector de índices, a diferencia de los índices enteros, tiene que ser de la misma longitud que el vector sobre el que hacemos la selección:

```
vec
```

```
## [1] 1 0 1 1 2 1 0
```

```
## 1 0 1 1 2 1 0
vec[c(FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE)]
```

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

```
## 2
```

Los *TRUES* y *FALSE* pueden ser a su vez obtenidos de tests lógicos del tipo: es 1 mayor que 2? O más en general, es este valor del vector vec mayor que 2?

Logical Tests

R proporciona 7 operadores lógicos para hacer comparaciones, Table @ref(tab:logop).

Table 1: (#tab:logop) R's Logical Operators

Operator	Syntax	Tests
`>`	`a > b`	Is a greater than b?
`>=`	`a >= b`	Is a greater than or equal to b?
`<`	`a < b`	Is a less than b?
`<=`	`a <= b`	Is a less than or equal to b?
`==`	`a == b`	Is a equal to b?
`!=`	`a != b`	Is a not equal to b?
`%in%`	`a %in% c(a, b, c)`	Is a in the group c(a, b, c)?

Cada operador devuelve `TRUE` o `FALSE`. Si los usamos para comparar vectores la comparación será elemento a elemento, como en las operaciones aritméticas:

```
1 > 2
```

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

```
## FALSE
1 > c(0, 1, 2)
```

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

```
## TRUE FALSE FALSE
c(1, 2, 3) == c(3, 2, 1)
```

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

```
## FALSE  TRUE FALSE
```

`%in%` no compara elemento a elemento. Es similar a *lookup* en excel: busca si el valor de la izquierda está en el vector de la derecha. Si tenemos un vector también en la izquierda `%in%`buscará si cada valor en el vector de la izquierda está en el vector de la derecha independientemente unos de otros:

```
1 %in% c(3, 4, 5)
```

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

```
## FALSE
c(1, 2) %in% c(3, 4, 5)
```

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

```
## FALSE FALSE
c(1, 2, 3) %in% c(3, 4, 5)
```

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

```
## FALSE FALSE  TRUE
c(1, 2, 3, 4) %in% c(3, 4, 5)
```

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

```
## FALSE FALSE  TRUE  TRUE
```

Importante: Para testar si dos objetos son iguales usamos ==y no =. El último es otra forma de definir el operador de asignación -> y el valor de la derecha reemplazará al de la izquierda.

Si usamos == para comparar objetos de distinta clase R tirará de coerciónpara que ambos tengan el mismo tipo.

Ejercicio 6: ¿Cuántos ases hay en mi baraja?

6.1. Selecciona la columna carta del objeto *baraja* que contiene los nombres de las cartas

6.2. Usa el operador lógico correspondiente para identificar todos los que tengan como nombre "as". (Hint: recuerda que los caracteres se escriben entre comillas)

6.3. Crea un nuevo objeto baraja2 en el que pongas a 0 la puntuacion de los ases

```
##  [1] "as"      "2"       "3"       "4"       "5"       "6"       "7"
##  [8] "sota"    "caballo" "rey"     "as"      "2"       "3"       "4"
## [15] "5"       "6"       "7"       "sota"    "caballo" "rey"     "as"
## [22] "2"       "3"       "4"       "5"       "6"       "7"       "sota"
## [29] "caballo" "rey"     "as"      "2"       "3"       "4"       "5"
## [36] "6"       "7"       "sota"    "caballo" "rey"
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
## [1] 40
```

```
## [1] 4
```

```
##   carta   palo puntos
## 1    as bastos      0
## 2     2 bastos      0
## 3     3 bastos     10
## 4     4 bastos      0
## 5     5 bastos      0
## 6     6 bastos      0
```

Logical subsetting es una de las mejores características de R, aunque como veremos en la clase 3 hay ahora nuevas formas de hacerlo mucho mas eficientes y seguras.

Si no queremos arrastrar el vector de TRUE y FALSE podemos usar la función *which()* que nos devuelve el índice de aquellos elementos que cumplen la condición lógica:

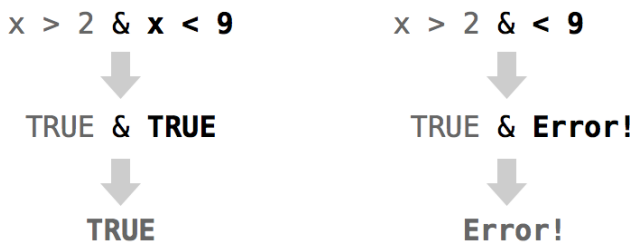


Figure 4: R will evaluate the expressions on each side of a Boolean operator separately, and then combine the results into a single TRUE or FALSE. If you do not supply a complete test to each side of the operator, R will return an error.

```
## [1]  1 11 21 31
```

```
##   carta   palo puntos
## 1    as bastos      0
## 2     2 bastos      0
## 3     3 bastos     10
## 4     4 bastos      0
## 5     5 bastos      0
## 6     6 bastos      0
```

Operadores booleanos

Hasta ahora hemos realizado operaciones aritméticas con R cuando teníamos números. Podemos también hacer operaciones booleanas como las descritas en Table @ref(tab:boole).

Table 2: (#tab:boole) Boolean operators

Operator	Syntax	Tests
`&`	`cond1 & cond2`	Are both `cond1` and `cond2` true?
`|`	`cond1 | cond2`	Is one or more of `cond1` and `cond2` true?
`xor`	`xor(cond1, cond2)`	Is exactly one of `cond1` and `cond2` true?
`!`	`!cond1`	Is `cond1` false? (e.g., `!` flips the results of a logical test)
`any`	`any(cond1, cond2, cond3, ...)`	Are any of the conditions true?
`all`	`all(cond1, cond2, cond3, ...)`	Are all of the conditions true?

Los operadores booleanos se situan entre dos tests lógicos como se indica en la Figura @ref(fig:boolean).

Por ejemplo:

```
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 4)
a == b
```

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

```
##  TRUE TRUE TRUE
b == c
```

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

```
##  TRUE  TRUE FALSE
a == b & b == c
```

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

```
##  TRUE  TRUE FALSE
```

Utilicemos operadores booleanos para encontrar el as de oros en nuestra baraja:

```
baraja$carta=="as" & baraja$palo=="oros"
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Vamos a guardar este vector de lógicos en un vector:

```
AsOro <- baraja$carta=="as" & baraja$palo=="oros"
```

Y podemos usar este vector de lógicos para ver si estamos en lo cierto:

```
baraja[AsOro,]
```

```
##    carta palo puntos
## 31    as oros     11
```

En un juego el As de Oros vale 21 en lugar de 11, hacemos una nueva baraja con esa puntuación:

```
baraja4<-baraja
baraja4$puntos[AsOro]<-21
baraja4[AsOro,]
```

```
##    carta palo puntos
## 31    as oros     21
```

- Ejercicio 6: Convierte en operaciones lógicas las siguientes frases para los objetos:

```
w <- c(-1, 0, 1)
x <- c(5, 15)
y <- "February"
z <- c("Monday", "Tuesday", "Friday")
```

- Que valores de w son positivos?
- Es x mayor que 10 y menor que 200?
- El objeto y es la palabra February?
- Son todos los valores de z dias de la semana?

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

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

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

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

Missing Information en R (Valores perdidos)

En data science es común encontrarnos con valores perdidos: porque no han podido ser recogidos, porque la medicion no era correcta o no era fiable. . . en estos casos es preferible no usar nada a usar un valor que

no es correcto. Y por supuesto no queremos descartar un individuo por completo solo porque una de sus mediciones no esta disponible.

En R el símbolo `NA` se utiliza para denotar un "missing value". La mayor parte de las funciones y operadores de R saben como lidiar con este problema. Por ejemplo:

```
1 + NA
```

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

```
## NA
NA == 1
```

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

```
## NA
```

La respuesta en ambos casos es "No se". La información de que no disponemos de ese valor se propaga.

na.rm

Este es el comando por el que le indicamos a R que quite (rm=remove) los missing values de un vector:

```
c(NA, 1:50)
```

```
##  [1] NA  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
## [24] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
## [47] 46 47 48 49 50
```

```
mean(c(NA, 1:50))
```

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

```
## NA
mean(c(NA,1:50),na.rm=T)
```

```
## [1] 25.5
```

is.na

Nos devuelve un lógico indicando si alguna de las componentes del vector son valores perdidos:

```
NA == NA
```

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

```
c(1, 2, 3, NA) == NA
```

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

```
is.na(NA)
```

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

```
vec <- c(1, 2, 3, NA)
is.na(vec)
```

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

```
!is.na(vec)
```

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

```
vec[!is.na(vec)]
```

```
## [1] 1 2 3
```