

# Librerías, funciones propias y estructuras de control

intro-R

Febrero 2018

1. Librerías
2. Funciones propias
3. Estructuras de control
4. Vectorización

## 1. Librerías

## 2. Funciones propias

## 3. Estructuras de control

## 4. Vectorización

A veces las funciones que vienen instaladas con R ("base-R") no son suficientes para cumplir nuestros propósitos. Para instalar nuevas librerías:

- ▶ Instalar el paquete en nuestro ordenador (una vez):  
`install.packages("nombre_del_paquete")`
- ▶ Cargar la librería en nuestra sesión (cada vez que abramos RStudio):  
`library(nombre_del_paquete)`

A veces las funciones que vienen instaladas con R ("base-R") no son suficientes para cumplir nuestros propósitos. Para instalar nuevas librerías:

- ▶ Instalar el paquete en nuestro ordenador (una vez):  
`install.packages("nombre_del_paquete")`
- ▶ Cargar la librería en nuestra sesión (cada vez que abramos RStudio):  
`library(nombre_del_paquete)`

## Ejemplo

```
install.packages("Rcpp")  
library(Rcpp)
```

A veces las funciones que vienen instaladas con R ("base-R") no son suficientes para cumplir nuestros propósitos. Para instalar nuevas librerías:

- ▶ Instalar el paquete en nuestro ordenador (una vez):  
`install.packages("nombre_del_paquete")`
- ▶ Cargar la librería en nuestra sesión (cada vez que abramos RStudio):  
`library(nombre_del_paquete)`

## Ejemplo

```
install.packages("Rcpp")  
library(Rcpp)
```

## No sólo funciones

A través de librerías podemos cargar también objetos (data frames, matrices, vectores...

1. Librerías

2. Funciones propias

3. Estructuras de control

4. Vectorización

# Funciones propias (I)

**No reinventes la rueda.** Cada vez que copies las mismas líneas de código más de dos veces... deberías haber escrito una función.



# Funciones propias (I)

**No reinventes la rueda.** Cada vez que copies las mismas líneas de código más de dos veces... deberías haber escrito una función.

```
presentacion <- function(nombre, comida_preferida){  
  paste("Hola, me llamo", nombre, "y me encanta comer",  
        comida_preferida)  
}  
presentacion("Álvaro", "croquetas")
```

# Funciones propias (II)

Podemos definir un argumento como *opcional*.

```
presentacion <- function(nombre,  
                           comida_preferida = "murciélagos"){  
  paste("Hola, me llamo", nombre, "y me encanta comer",  
        comida_preferida)  
}  
presentacion("Ozzy")  
presentacion("Ozzy", "tofu")
```

## Funciones propias (III): Alcances

Se pueden crear objetos dentro de una función. Sin embargo, solo "existirán" dentro de esa función; si los llamar fuera de ella, dará un error.

```
funcion_ejemplo <- function(x){  
  y <- x^3 + 3  
  abs(y)  
}  
funcion_ejemplo(-2) # Devuelve 11  
y                  # Devuelve un error.
```

### Entorno global

Los objetos creados fuera de una función pertenecen al *entorno global*.

1. Librerías
2. Funciones propias
3. Estructuras de control
4. Vectorización

A la hora de dar órdenes es importante poder especificar condiciones o repeticiones. Estas directrices se llaman **estructuras de control**.

A la hora de dar órdenes es importante poder especificar condiciones o repeticiones. Estas directrices se llaman **estructuras de control**.

## Ejemplo

Si llueve, tiende la ropa. Si no, saca al perro y da tres vueltas a la manzana.

A la hora de dar órdenes es importante poder especificar condiciones o repeticiones. Estas directrices se llaman **estructuras de control**.

## Ejemplo

Si llueve, tiende la ropa. Si no, saca al perro y da tres vueltas a la manzana.

Existen dos tipos principales:

1. Bucles
2. Condiciones

# Bucles *for*

Un bucle define cuántas veces se debe repetir una cierta acción. El bucle más típico es el **for**:

```
for (i in 10:1) {  
  print(paste("Quedan", i, "segundos para el despegue"))  
}
```

En este ejemplo

El contador, *i*, va tomando todos los valores posibles del vector 10:1.



# Inicializando objetos

A veces queremos ir rellenando un objeto a medida que progresa un bucle. Para ello, primero tendremos que **inicializar** el objeto.

```
vector_vacio <- c()
colores_primarios <- c("Magenta", "Cian", "Amarillo")
for (i in 1:100) {
  vector_vacio[i] <- sample(colores_primarios,
                           size = 1, replace = T)
}
```

# Inicializando objetos

A veces queremos ir rellenando un objeto a medida que progresa un bucle. Para ello, primero tendremos que **inicializar** el objeto.

```
vector_vacio <- c()
colores_primarios <- c("Magenta", "Cian", "Amarillo")
for (i in 1:100) {
  vector_vacio[i] <- sample(colores_primarios,
                           size = 1, replace = T)
}
```

## Inicializando otros objetos

Podemos inicializar un vector o una lista sin determinar su tamaño final. Sin embargo, esto no es así con una matriz o data frame.

# Bucles anidados

Podemos usar dos bucles anidados para recorrer los elementos de una matriz.

```
mi_matriz <- matrix(1:10, ncol = 2, byrow = T)
matriz_cuadrados <- matrix(rep(0, times = 10), ncol = 2)
for (i in 1:nrow(mi_matriz)) {
  for (j in 1:ncol(mi_matriz)) {
    matriz_cuadrados[i,j] <- mi_matriz[i,j]^2
  }
}
```

# Bucles anidados

Podemos usar dos bucles anidados para recorrer los elementos de una matriz.

```
mi_matriz <- matrix(1:10, ncol = 2, byrow = T)
matriz_cuadrados <- matrix(rep(0, times = 10), ncol = 2)
for (i in 1:nrow(mi_matriz)) {
  for (j in 1:ncol(mi_matriz)) {
    matriz_cuadrados[i,j] <- mi_matriz[i,j]^2
  }
}
```

Funciona, pero...

...era la mejor manera de hacerlo?

# Bucles en funciones

Es muy frecuente meter un bucle dentro de una función para que el número de repeticiones, o la condición, sean variables.

```
aleatorios_hasta_n <- function(n = 3) {  
  x <- 0  
  while (x != n) {  
    x <- sample(1:10, 1)  
    print(x)  
  }  
}  
  
aleatorios_hasta_n(5)  
aleatorios_hasta_n()  
aleatorios_hasta_n(11)    # Ojo-cuidao!
```

# Bucles en funciones

Es muy frecuente meter un bucle dentro de una función para que el número de repeticiones, o la condición, sean variables.

```
aleatorios_hasta_n <- function(n = 3) {  
  x <- 0  
  while (x != n) {  
    x <- sample(1:10, 1)  
    print(x)  
  }  
}  
  
aleatorios_hasta_n(5)  
aleatorios_hasta_n()  
aleatorios_hasta_n(11)    # Ojo-cuidao!
```

En los bucles for sabemos de antemano cuántas veces queremos repetir las acciones. En cambio, los bucles while paran cuando se alcanza una cierta condición:

```
x <- 0
while (x != 3) {
  print(x)
  x <- sample(1:10, 1)
}
```

# Condiciones (I)

La estructura `if/else` sirve para ordenarle al programa que haga algo solo si se cumple una cierta condición (*if*). Si no, puede hacer otra cosa (*else*).:

```
poblaciones <- c(200, 120000, 1300000, 2300, 9500)
for (municipio in poblaciones) {
  if (municipio > 10000) {
    print("Es ciudad")
  } else {
    print("No es ciudad")
  }
}
```



## Condiciones (II)

Al igual que con los bucles, podemos concatenar o anidar condiciones:

```
poblaciones <- c(200, 120000, 1300000, 23000, 9500)
for (municipio in poblaciones) {
  if (municipio > 10000) {
    if (municipio > 1e6) {
      print("Es una ciudad muy grande!")
    } else {
      print("Es una ciudad normal")
    }
  } else if (municipio > 8000) {
    print("Es un pueblo grandecito")
  } else {
    print("Es un pueblo")
  }
}
```

1. Librerías
2. Funciones propias
3. Estructuras de control
4. Vectorización

R está hecho para trabajar con **vectores**. Cuando una función puede procesar todos los elementos de un vector a la vez se dice que está **vectorizada**.

## ¿Por qué vectorizar?

A R se le da muy bien el trabajo simultáneo, pero no tanto el trabajo secuencial.

Al proceso de adaptar una función para que trabaje con vectores de cualquier longitud se le llama *vectorización*.

# ifelse()

La función ifelse es una versión vectorizada de una condición if/else. Toma tres argumentos: un test lógico, el resultado si el test es verdadero y el resultado si el test es falso.

```
poblaciones <- c(200, 120000, 1300000, 23000)
ifelse(poblaciones > 100000, "Es ciudad", "No es ciudad")
```

# ifelse()

La función ifelse es una versión vectorizada de una condición if/else. Toma tres argumentos: un test lógico, el resultado si el test es verdadero y el resultado si el test es falso.

```
poblaciones <- c(200, 120000, 1300000, 23000)
ifelse(poblaciones > 100000, "Es ciudad", "No es ciudad")
```

También podemos anidar condiciones:

```
poblaciones <- c(200, 120000, 1300000, 23000)
ifelse(poblaciones > 100000, "Es ciudad",
      ifelse(poblaciones > 8000, "Es un pueblo grande",
              "Es un pueblo pequeño"))
```

Las funciones de la familia *apply* sirven para aplicar una función a los elementos de un vector, lista, matriz...

```
alturas <- list("rodal_1" = c(32, 24, 56),  
               "rodal_2" = c(13, 16, 21))  
lapply(alturas, mean) # Devuelve una lista  
sapply(alturas, sd)   # Devuelve un vector
```