

# R Programming

Martín Macías

Diciembre de 2015

## 1 Estableciendo el Directorio de trabajo

La idea es hacer el seguimiento de los comandos en R para establecer el directorio de trabajo.

- Con la opción `getwd()` se obtiene el directorio de trabajo actual:

```
getwd()

[1] "/Users/Martin/Desktop/GitHub/datasciencecoursera"
```

- Para ver los archivos carpetas que hay en el directorio de trabajo actual:

```
dir()

[1] "data.R"
[2] "datasciencecoursera.Rproj"
[3] "functions.R"
[4] "HelloWorld.md"
[5] "hw1_data.csv"
[6] "Quices"
[7] "R Programming.Rnw"
[8] "R_Programming.bbl"
[9] "R_Programming.pdf"
[10] "R_Programming.Rnw"
[11] "R_Programming.tex"
[12] "README.md"
[13] "rprog-032_Basic_Building_Blocks.txt"
[14] "rprog-032_Missing_Values.txt"
[15] "rprog-032_Sequences_of_Numbers.txt"
[16] "rprog-032_Vectors.txt"
[17] "rprog-032_Workspace_and_Files.txt"
[18] "y.R"
```

- El comando `ls()` muestra lo que exista en mi espacio de trabajo:

```
ls()

character(0)
```

## 2 Data types

### 2.1 Vectores y listas

#### 2.1.1 Creando vectores

La función `c()` se usa para crear vectores de objetos:

```
x <- c(0.5, 0.6)      # Numérico
x <- c(TRUE, FALSE)   # Lógico
x <- c(T, F)          # Lógico
x <- c("a", "b", "c") # Caracter
x <- 9:29             # Entero
x <- c(1+0i, 2+4i)    # Complejo
```

Usando la función `vector()`

```
x <- vector("numeric", length = 10)
x

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

#### 2.1.2 Mezclando objetos

Miremos cómo se mezclan objetos en un vector

```
y <- c(1.7, "a")      # Caracter
y <- c(TRUE, 2)       # Numérico
y <- c("a", TRUE)     # Caracter
```

#### 2.1.3 Concatenación explícita

Los objetos pueden concatenarse explícitamente de una clase a otra usando las funciones `as.*`, si están disponibles:

```
x <- 0:6
class(x)
```

```

[1] "integer"

  as.numeric(x)

[1] 0 1 2 3 4 5 6

  as.logical(x)

[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

  as.character(x)

[1] "0" "1" "2" "3" "4" "5" "6"

```

La concatenación sinsentido resulta en NAs

```

x <- c("a", "b", "c")
class(x)

[1] "character"

  as.numeric(x)

Warning:  NAs introduced by coercion

[1] NA NA NA

  as.logical(x)

[1] NA NA NA

  as.complex(x)

Warning:  NAs introduced by coercion

[1] NA NA NA

```

## 2.2 Listas

Las listas son tipos de vectores especiales que pueen contener elementos de diferentes clases.

```

x <- list(1, "a", TRUE, 1 + 4i)
x

[[1]]
[1] 1

```

```
[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

## 2.3 Matrices

Las matrices son vectores con un atributo de dimensión. El atributo de dimensión es, en sí mismo, un vector entero de longitud 2 (`nrow`, `ncol`)

```
m <- matrix(nrow = 2, ncol = 3)
m

      [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA

dim(m)

[1] 2 3

attributes(
  "dim"
)
[1] 2 3
```

Las matrices se contruyen en el sentido de las columnas, es decir, en forma de "zig zag" invertido

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m

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

Las matrices también pueden crearse directamente de vectores añadiendo el atributo de dimensión

```
m <- 1:10
m
```

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

dim(m) <- c(2, 5)
m

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

Las matrices también pueden crearse mediante la unión de filas `rbind()` o la unión de columnas `cbind`

```
x <- 1:3
y <- 10:12
cbind(x, y)

      x y
[1,] 1 10
[2,] 2 11
[3,] 3 12

rbind(x, y)

      [,1] [,2] [,3]
x        1    2    3
y       10   11   12
```

## 2.4 Factores

Los factores se usan para representar datos categóricos. Los factores pueden ser ordenados o desordenados.

- Los factores son utilizados especialmente para modelar funciones como `lm()` y `glm()`
- Usar factores con labels es mejor que usar enteros puesto que los factores son autodescriptivos; tener una variable que tenga como valores `¡¡Masculino!!` y `¡¡Femenino!!` es mejor que tener una variable con 1 y 2.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x

[1] yes yes no  yes no
Levels: no yes

table(x)      # Hace conteo de niveles que hay
```

```
x
no yes
2 3

unclass(x) # Despoja de la clase que tenga el vector

[1] 2 2 1 2 1
attr("levels")
[1] "no" "yes"
```

El orden de los niveles se puede determinar usando el argumento `levels` en `factor()`. Esto puede ser importante en modelación lineal porque el primer nivel se usa como nivel de base.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))

x

[1] yes yes no yes no
Levels: yes no
```

## 2.5 Datos faltantes

Los datos faltantes se denotan por `NA` o `NaN` para operaciones matemáticas indefinidas.

- `is.na()` se usa para probar si los objetos son `NA`
- `is.nan()` se usa para probar si los objetos son `NaN`
- Los valores `NA` también tienen clase. Pueden ser enteros `NA`, carácter `NA`, etc.
- Un valor `NaN` es también `NA` pero el recíproco no es cierto

```
x <- c(1, 2, NA, 10, 3)
is.na(x)

[1] FALSE FALSE TRUE FALSE FALSE

is.nan(x)

[1] FALSE FALSE FALSE FALSE FALSE

x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

```
[1] FALSE FALSE  TRUE  TRUE FALSE

is.nan(x)

[1] FALSE FALSE  TRUE FALSE FALSE
```

## 2.6 Data frames

Los data frames se usan para almacenar datos tabulados.

- Se representan como un tipo especial de lista donde cada elemento de la lista debe tener la misma longitud.
- Cada elemento de la lista se puede pensar como una columna y la longitud de cada elemento de la lista es el número de filas.
- A diferencia de las matrices, los data frames pueden almacenar diferentes clases de objetos en cada columna (así como las listas): los elementos de las matrices deben ser de la misma clase.
- Los data frames también tienen un atributo especial llamado `row.names`
- Los data frames se crean usualmente mediante `read.table()` o `read.csv()`
- Los data frames pueden convertirse en matrices mediante `data.matrix()`

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x

  foo  bar
1   1 TRUE
2   2 TRUE
3   3 FALSE
4   4 FALSE

nrow(x)

[1] 4

ncol(x)

[1] 2
```

## 2.7 Atributos de nombre

Los objetos en R también pueden tener nombres, los cuales son muy útiles para escribir código legible y objetos autodescriptivos.

```
x <- 1:3
names(x)

NULL

names(x) <- c("foo", "bar", "norf")
x

foo bar norf
  1   2   3

names(x)

[1] "foo" "bar" "norf"
```

Las listas también pueden tener nombres.

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

$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Para las matrices funciona de igual forma.

```
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m

  c d
a 1 3
b 2 4
```

## 3 Leer datos tabulados

Existen unas pocas funciones principales para leer datos en R.



- `read.table` y `read.csv` para leer datos tabulados.
- `readLines` para leer líneas de texto.
- `source` para leer archivos de R (*inverse* de `dump`)
- `dget` para leer archivos de R (*inverse* de `dput`)
- `load` para leer de espacios de trabajo guardados
- `unserialize` para leer objetos de R en forma binaria.

### 3.1 `read.table`

La función `read.table` es una de las funciones más comúnmente usadas para leer datos. Tienen unos pocos argumentos:

- `file`, el nombre de un archivo o conexión.
- `header`, tipo lógico e indica si el archivo tiene línea de cabecera
- `sep`, tipo cadena e indica cómo están separadas las columnas.
- `colClasses`, un vector de caracteres que indica la clase de cada columna en el conjunto de datos.
- `nrows`, número de filas en el conjunto de datos.
- `comment.char`, una cadena de caracteres que indica el caracter del comentario.
- `skip`, número de líneas que deben omitirse desde el principio.
- `stringsAsFactors`, ¿deberían las variables carácter ser codificadas como factores?

**Nota:** `read.table` trabaja con archivos separados por espacios mientras que `read.csv` trabaja con archivos separados por comas.

### 3.2 Usando `read.table`

El argumento `colClasses` acelera el uso de la función `read.table` hasta casi el doble. Para usar esta opción, debe conocerse la clase de cada columna de su data frame. Si todas las columnas son numéricas, por ejemplo, entonces debe ajustar el argumento así: `colClasses = "numeric"`.

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt", colClasses = classes)
```

Si configura `nrows` podrá ahorrar memoria. Puede usar la herramienta de Unix `wc` para calcular las líneas en un archivo.

### 3.3 Formatos de datos textuales

- `dumping` y `dputing` son muy útiles porque el formato textual resultante es editable y en caso de que se corrompan, potencialmente recuperable.
- A diferencia de trabajar con una tabla o con un archivo csv, las funciones `dumping` y `dputing` conservan los metadatos (sacrificando algo de legibilidad) para que otro usuario no tenga que especificar todo de nuevo.
- Los formatos textuales pueden trabajar mucho mejor con programas de control de versiones, los cuales pueden rastrear cambios significativos en archivos de texto.
- Los archivos textuales pueden ser más duraderos; en caso de que se corrompan, es más fácil hallar el problema dentro del archivo.
- **Desventaja:** El formato no es eficiente en cuanto a ahorro de espacio.

### 3.4 dput-ing objetos en R

```
options(width = 55)
y <- data.frame(a = 1, b = "a")
dput(y)

structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",
"b"), row.names = c(NA, -1L), class = "data.frame")

dput(y, file="y.R")
new.y <- dget("y.R")
new.y

  a b
1 1 a
```

### 3.5 dump-ing objetos en R

```
x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)
source("data.R")
y

  a b
1 1 a
```

```
x  
[1] "foo"
```

### 3.6 Conexiones a archivos

```
str(file)  
  
function (description = "", open = "", blocking = TRUE,  
         encoding = getOption("encoding"), raw = FALSE)
```

- **description** es el nombre del archivo.
- **open** es un indicador de código.
  - "r" para únicamente lectura
  - "w" escritura e iniciando un nuevo archivo
  - "a" anexar
  - "rb", "wb", "ab" leer, escribir y anexar en modo Binario (Windows)

### 3.7 Conexiones

En general, las conexiones son herramientas poderosas para navegar entre archivos u otros objetos. En la práctica, no se necesita enfrentarse directamente con la interfaz de conexión.

```
con <- file("foo.txt", "r")  
data <- read.csv(con)  
close(con)
```

es lo mismo que

```
data <- read.csv("foo.txt")
```

### 3.8 Leer líneas de un archivo de texto

```
con <- gzfile("words.gz")  
x <- readLines(con, 10)
```

**writeLines** toma un vector de caracteres y escribe cada elemento de una línea al tiempo en un archivo de texto.

**readLines** puede ser útil para leer líneas de páginas web.

```
## Esto puede tomar algunos minutos
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
head(x)

[1] "<!DOCTYPE html>"
[2] "<html lang=\"en\">"
[3] ""
[4] "<head>"
[5] "<meta charset=\"utf-8\" />"
[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

### 3.9 Subconjunto

Existen un gran número de operadores que pueden usarse para extraer subconjuntos de objetos en R.

- `[ ]` siempre arroja un objeto de la misma clase del original; puede usarse para seleccionar más de un elemento (hay una excepción).
- `[[ ]]` se usa para extraer elementos de una lista o de un data frame; únicamente puede usarse un solo elemento y la clase del objeto obtenido no necesariamente es una lista o un data frame.
- `$` se usa para extraer elementos de una lista o un data frame por nombre; la semántica es igual a la de `[[ ]]`.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]

[1] "a"

x[2]

[1] "b"

x[1:4]

[1] "a" "b" "c" "c"

x[x > "a"]

[1] "b" "c" "c" "d"

u <- x > "a"
u

[1] FALSE TRUE TRUE TRUE TRUE FALSE

x[u]

[1] "b" "c" "c" "d"
```

### 3.9.1 Listas de Subconjuntos

```
x <- list(foo = 1:4, bar = 0.6)
x[1]

$foo
[1] 1 2 3 4

x[[1]]

[1] 1 2 3 4

x$bar

[1] 0.6

x[["bar"]]

[1] 0.6

x["bar"]

$bar
[1] 0.6
```

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]

$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

El operador `[[ ]]` puede usarse con índices computados; `$` puede usarse exclusivamente con nombres literales.

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]]    ## índice computado para 'foo'

[1] 1 2 3 4

x$name      ## ¡el elemento 'name' no existe!

NULL

x$foo

[1] 1 2 3 4
```

[[]] puede tomar una secuencia de enteros.

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1, 3)]]

[1] 14

x[[1]][[3]]

[1] 14

x[[c(2, 1)]]

[1] 3.14
```

### 3.9.2 Matrices

De las matrices se puede crear también subconjuntos de la forma usual, con índices de la forma  $(i, j)$

```
x <- matrix(1:6, 2, 3)
x[1, 2]

[1] 3

x[2,1]

[1] 2
```

U omitir los índices si se quiere:

```
x[1, ]

[1] 1 3 5

x[, 2]

[1] 3 4
```

Por defecto, cuando un elemento solo de una matriz es recuperado, éste se retorna como un vector de longitud 1 en vez de ser una matriz de dimensión 1 x 1. Este proceder puede ser modificado mediante el argumento `drop = FALSE`

```
x <- matrix(1:6, 2, 3)
x[1, 2]

[1] 3
```

```
x[1, 2, drop = FALSE]

      [,1]
[1,]     3
```

De la misma manera, al hacer subconjuntos de una sola columna o una sola fila, el programa arrojará un vector, no una matriz (por defecto)

```
x[1, ]
[1] 1 3 5

x[1, , drop = FALSE]

      [,1] [,2] [,3]
[1,]     1     3     5

x[2,1]
[1] 2
```

### 3.9.3 Subconjuntos con nombres

La coincidencia parcial de nombres se permite con `[[ ]]` y `$`

```
x <- list(aardvark = 1:5)
x$a

[1] 1 2 3 4 5

x[["a"]]

NULL

x[["a", exact = FALSE]]

[1] 1 2 3 4 5
```

### 3.9.4 Eliminando valores faltantes

Una tarea común es eliminar valores faltantes (NAS).

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

[1] 1 2 4 5
```

¿Y qué pasaría si existen multiples cosas y usted quiere tomar el subconjunto con los valores no faltantes?

```
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
good

[1] TRUE TRUE FALSE TRUE FALSE TRUE

x[good]

[1] 1 2 4 5

y[good]

[1] "a" "b" "d" "f"
```

```
airquality[1:6, ]

  Ozone Solar.R Wind Temp Month Day
1   41     190  7.4   67     5   1
2   36     118  8.0   72     5   2
3   12     149 12.6   74     5   3
4   18     313 11.5   62     5   4
5   NA       NA 14.3   56     5   5
6   28       NA 14.9   66     5   6

good <- complete.cases(airquality)
airquality[good, ][1:6, ]

  Ozone Solar.R Wind Temp Month Day
1   41     190  7.4   67     5   1
2   36     118  8.0   72     5   2
3   12     149 12.6   74     5   3
4   18     313 11.5   62     5   4
7   23     299  8.6   65     5   7
8   19       99 13.8   59     5   8
```

## 4 Operaciones vectorizadas

Muchas operaciones en [R] son *vectorizadas* volviendo el código más eficiente, conciso y fácil de leer.



```

x <- 1:4; y <- 6:9
x + y

[1]  7  9 11 13

x > 2

[1] FALSE FALSE  TRUE  TRUE

x >= 2

[1] FALSE  TRUE  TRUE  TRUE

y == 8

[1] FALSE FALSE  TRUE FALSE

x * y

[1]  6 14 24 36

x / y

[1] 0.1666667 0.2857143 0.3750000 0.4444444

```

```

x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x * y  ## Multiplicación término a término

      [,1] [,2]
[1,]    10    30
[2,]    20    40

x / y

      [,1] [,2]
[1,]   0.1   0.3
[2,]   0.2   0.4

x %*% y  ## Multiplicación real de matrices

      [,1] [,2]
[1,]    40    40
[2,]    60    60

```

## 5 Estructuras de control

### 5.1 Introducción

Las estructuras de control en R permiten controlar el flujo de ejecución del programa, dependiendo de las condiciones del tiempo de ejecución. Las estructuras más comunes son:

- **if, else:** para probar una condición
- **for:** ejecuta un bucle un número fijo de veces
- **while:** ejecuta un bucle 'mientras' una condición sea verdadera
- **repeat:** ejecuta un bucle indefinidamente
- **break:** rompe la ejecución de un bucle
- **next:** omite la interacción de un bucle
- **return:** sale de una función

### 5.2 if-else

```
if(<condición>) {  
  ## haga algo  
} else {  
  # haga algo más  
}  
if(<condición1>) {  
  ## haga algo  
} else if (<condición2>) {  
  ## haga algo diferente  
} else {  
  ## haga algo diferente  
}
```

Esta es una estructura **if/else** válida:

```
if(x > 3) {  
  y <- 10  
} else {  
  y <- 0  
}
```

Warning in if (x > 3) {: the condition has length > 1 and only the first element will be used

o esta:

```
y <- if(x > 3) {  
  10  
} else {  
  0  
}
```

Warning in if (x > 3) {: the condition has length > 1 and only the first element will be used

Obviamente, `else` no es estrictamente necesario:

```
if(<condición1>) {  
  
}  
if(<condición2>) {  
  
}
```

### 5.3 Bucles for

Los bucles `for` toma una variable iteradora a quien le es asignada valores sucesivos de un vector o una secuencia. Los bucles `for` son comúnmente usados para iterar sobre los elementos de un objeto (lista, vector, etc.)

```
for(i in 1:10){  
  print(i)  
}  
  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10
```

Este bucle toma la variable `i` y en cada iteración del bucle le da el valor de 1, 2, 3, ..., 10 y luego sale del bucle.

Los siguientes tres bucles hacen lo mismo:

```
x <- c("a", "b", "c", "d")
```

```
for(i in 1:4){  
  print(x[i])  
}
```

```
[1] "a"
```

```
[1] "b"
```

```
[1] "c"
```

```
[1] "d"
```

```
for(i in seq_along(x)){  
  print(x[i])  
}
```

```
[1] "a"
```

```
[1] "b"
```

```
[1] "c"
```

```
[1] "d"
```

```
for(letter in x){  
  print(letter)  
}
```

```
[1] "a"
```

```
[1] "b"
```

```
[1] "c"
```

```
[1] "d"
```

```
for(i in 1:4) print(x[i])
```

```
[1] "a"
```

```
[1] "b"
```

```
[1] "c"
```

```
[1] "d"
```

Los bucles **for** pueden estar anidados

```
x <- matrix (1:6, 2, 3)  
for(i in seq_len(nrow(x))){  
  for(j in seq_len(ncol(x))){  
    print(x[i, j])  
  }  
}
```

```
[1] 1
```

```
[1] 3
```

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

Hay que tener cuidado con los bucles anidados. Anidar más allá de dos o tres pasos muchas veces difícil de leer o entender.

## 5.4 Bucles while

Los bucles **while** comienzan con una condición a evaluar. Si la condición es cierta, ejecutan un cuerpo del bucle. Una vez ese cuerpo es ejecutado, la condición es evaluada nuevamente, etc.

```
count <- 0
while(count < 10){
  print(count)
  count <- count + 1
}

[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

Los bucles **while** pueden resultar potencialmente en bucles infinitos si no se escriben apropiadamente. Hay que usarlo con cuidado.

Puede haber más de una condición para evaluar.

```
z <- 5
while(z >= 3 && z <= 10){
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin == 1) { ## caminata aleatoria
    z <- z + 1
  } else {
    z <- z - 1
  }
}
```

Las condiciones siempre se evalúan de izquierda a derecha.

## 5.5 repeat, next y break

### 5.5.1 repeat

`repeat` inicia un bucle infinito; esto no se utiliza usualmente en aplicaciones estadísticas pero sí tienen sus usos. La única forma de salir de un `repeat` es llamar un `break`.

```
x0 <- 1
tol <- 1e-8
repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1
  }
}
```

### 5.5.2 next, return

`next` se usa para salir de la iteración de un bucle

```
for(i in 1:100) {
  if(i <= 20){
    ## Salta las primeras 20 iteraciones
    next
  }
  ## Haga algo aquí
}
```

`return` señala que una función debe salir y regresar a un valor dado.

## 5.6 Resumen de estructuras de control

- Las estructuras de control como `if`, `while` y `for` permiten controlar el flujo de un programa en R
- Los bucles infinitos deben evitarse aún si son teóricamente correctos.
- Las estructuras de control mencionadas son principalmente utilizadas para escribir programas; para trabajar interactivamente líneas de comando, las funciones `*apply` son más utilizadas.

## 6 Primeras funciones en R

iiiiiii HEAD

A continuación algunas funciones que se escribieron inicialmente en R:

Una simple función que recibe dos objetos y los suma

```
add2 <- function(x, y) {  
  x + y  
}
```

Una función que recibe un objeto, evalúa si lo que compone este objeto tiene componentes mayores a 10 y luego arroja aquellos componentes mayores a 10

```
above10 <- function(x){  
  use <- x > 10  
  x[use]  
}
```

La misma función anterior pero ahora el usuario deberá suministrar el objeto y el número al cual debe evaluarse

```
above <- function(x, n){  
  use <- x > n  
  x[use]  
}
```

Esta función evaluará el promedio de cada columna de un objeto dado.

```
columnmean <- function(y) {  
  nc <- ncol(y)  
  means <- numeric(nc)  
  for(i in 1:nc) {  
    means[i] <- mean(y[, i])  
  }  
  means  
}
```

=====

A continuación algunas funciones que se escribieron inicialmente en R:

Una simple función que recibe dos objetos y los suma

```
add2 <- function(x, y) {  
  x + y  
}
```

Una función que recibe un objeto, evalúa si lo que compone este objeto tiene componentes mayores a 10 y luego arroja aquellos componentes mayores a 10

```
above10 <- function(x){  
  use <- x > 10  
  x[use]  
}
```

La misma función anterior pero ahora el usuario deberá suministrar el objeto y el número al cual debe evaluarse

```
above <- function(x, n){  
  use <- x > n  
  x[use]  
}
```

Esta función evaluará el promedio de cada columna de un objeto dado.

```
columnmean <- function(y) {  
  nc <- ncol(y)  
  means <- numeric(nc)  
  for(i in 1:nc) {  
    means[i] <- mean(y[, i])  
  }  
  means  
}
```

iiiiii 1ed43ad379b5fc490bc47c16b50adc7c3917a80f La misma función anterior sólo que omitiremos los NA para que pueda evaluarse.

```
columnmean2 <- function(y, removeNA = TRUE) {  
  nc <- ncol(y)  
  means <- numeric(nc)  
  for(i in 1:nc) {  
    means[i] <- mean(y[, i], na.rm = removeNA)  
  }  
  means  
}
```

## 7 Funciones en R

Las funciones se crean con la directiva `function()` y se guardan como objetos de R. En particular, son objetos de R de clase "function".



```
f <- function(<argumentos>) {
  ## Que haga algo
}
```

Las funciones en R son "objetos de primera clase", que significa que pueden ser tratados como cualquier otro objeto R. Principalmente,

- Las funciones pueden pasar como argumentos de otras funciones
- Las funciones pueden estar anidadas, es decir que es posible definir una función dentro de otra función. El valor que entrega una función es la última expresión en el cuerpo de la función a ser evaluada.

iiiiiii HEAD =====

## 7.1 Argumentos de las funciones

Las funciones tienen *argumentos de nombre* los cuales potencialmente tienen *valores por defecto*.

- Los *argumentos formales* son los argumentos incluidos en la definición de la función.
- la función `formals` devuelve una lista de todos los argumentos formales de una función.
- No todas las funciones llamadas en R hacen uso de todos los argumentos formales
- Los argumentos de las funciones pueden estar *perdidos* o quizás tengan valores por defecto.

## 7.2 Argumentos coincidentes

Los argumentos de las funciones en R pueden coincidir en posición o en nombre. Por ende, los siguientes llamados a `sd` son todas equivalentes

```
mydata <- rnorm(100)
sd(mydata)
sd(x = mydata)
sd(x = mydata, na.rm = FALSE)
sd(na.rm = FALSE, x = mydata)
sd(na.rm = FALSE, mydata)
```

A pesar de que esto sea posible, no es recomendable manejar las funciones desordenadamente por lo que, naturalmente, causará confusión.

Es posible combinar coincidencias posicionales con coincidencias por nombre. Cuando un argumento es coincido por nombre, éste es "sacado" de la lista de

argumentos y los argumentos sin nombrar restantes se coinciden en el orden en que ellos están listados en la definición de la función.

```
args(lm)

function (formula, data, subset, weights, na.action, method = "qr",
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
  contrasts = NULL, offset, ...)
NULL
```

~~~~~ 1ed43ad379b5fc490bc47c16b50adc7c3917a80f

### 7.3 Argumentos de las funciones

Las funciones tienen *argumentos de nombre* los cuales potencialmente tienen *valores por defecto*.

- Los *argumentos formales* son los argumentos incluidos en la definición de la función.
- la función `formals` devuelve una lista de todos los argumentos formales de una función.
- No todas las funciones llamadas en R hacen uso de todos los argumentos formales
- Los argumentos de las funciones pueden estar *perdidas* o quizás tengan valores por defecto.

### 7.4 Argumentos coincidentes

Los argumentos de las funciones en R pueden coincidir en posición o en nombre. Por ende, los siguientes llamados a `sd` son todas equivalentes

```
mydata <- rnorm(100)
sd(mydata)
sd(x = mydata)
sd(x = mydata, na.rm = FALSE)
sd(na.rm = FALSE, x = mydata)
sd(na.rm = FALSE, mydata)
```

A pesar de que esto sea posible, no es recomendable manejar las funciones desordenadamente por lo que, naturalmente, causará confusión.

Es posible combinar coincidencias posicionales con coincidencias por nombre. Cuando un argumento es coincidido por nombre, éste es "sacado" de la lista de argumentos y los argumentos sin nombrar restantes se coinciden en el orden en que ellos están listados en la definición de la función.

```
args(lm)

function (formula, data, subset, weights, na.action, method = "qr",
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
  contrasts = NULL, offset, ...)
NULL
```

Los siguientes dos llamados son similares:

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

- Muchas veces los argumentos de nombre son útiles en la línea de comandos cuando se tiene una lista de argumentos larga y se pretende usar los valores por defecto de todo excepto por un argumento cercano al final de la lista.
- Los argumentos de nombre también son de ayuda si puede recordarse el nombre del argumento y no su posición en la lista de argumentos (plotting es un excelente ejemplo).

Los argumentos de las funciones también pueden ser parcialmente coincidentes, lo que es muy útil en trabajos interactivos. El orden de operaciones cuando un argumento es dado:

1. Revisa la coincidencia exacta para un argumento de nombre
2. Revisa una coincidencia parcial
3. Revisa una coincidencia de posición

## 7.5 Definiendo una función

Además de no especificar un valor por defecto, también es posible asignarle a un argumento el valor 'NULL'.

```
f <- function(a, b = 1, c = 2, d = NULL){
  }
}
```

## 7.6 Evaluación perezosa

Los argumentos para funciones son evaluadas *perezosamente*, es decir sólo cuando son necesarias.

```
f <- function(a, b){
  a^2
}
f(2)

[1] 4
```

Esta función no usa el argumento `b`, de tal suerte que llamar `f(2)` no producirá un error por cuanto `2` quedará ubicado posicionalmente en `a`.

```
f <- function(a, b){
  print(a)
  print(b)
}
f(45)

[1] 45
```

## 7.7 El argumento "..."

El argumento `"..."` indica un número variable de argumentos que usualmente se pasan a otras funciones.

- ... se usa comúnmente cuando se extienden otras funciones y no se quiere copiar la lista entera de argumentos de la función original.

```
myplot <- function(x, y, type = "l", ...){
  plot(x, y, type = type, ...)
}
```

- Las funciones genéricas usan ..., así esos argumentos extra se pasan a métodos.

```
mean

function (x, ...)
UseMethod("mean")
<bytecode: 0x7fa01d04b390>
<environment: namespace:base>
```

El argumento `...` también es necesario cuando el número de argumentos pasados a la función no se conoce previamente.

```
args(paste)

function (... , sep = " ", collapse = NULL)
NULL

args(cat)

function (... , file = "", sep = " ", fill = FALSE, labels = NULL,
          append = FALSE)
NULL
```

## 7.8 Argumentos que vienen después del argumento “...”

Una captura con ... significa que cualquier argumento que aparezca después de ... sobre la lista de argumentos debe ser llamados explícitamente y no puede ser parcialmente coincido.

```
args(paste)

function (... , sep = " ", collapse = NULL)
NULL

paste("a", "b", sep = ":")

[1] "a:b"

paste("a", "b", se = ":")

[1] "a b :"
```

## 8 Reglas de ámbito

### 8.1 Símbolo de unión

#### 8.1.1 Una desviación en valores de unión a símbolos

¿Cómo sabe R qué valor asignar a qué símbolo?

```
lm <- function(x) {x * x}
lm

function(x) {x * x}
```

¿Cómo sabe R que valor asignar al símbolo `lm`? ¿Por qué no le da el valor de `lm` que está en la librería de R?