Módulo de adaptación

Master en Business Intelligence y Big Data

PROFESOR/A
Antonio Sarasa Cabezuelo

Escuela de organización industrial

www.eoi.es

El lenguaje R

Objetivos del tema

- Conocer las características básicas del lenguaje R
- Realizar cálculos estadísticos básicos con R

Índice de contenidos

- Tipos de datos
- Vectores
- Matrices
- Listas
- Factores.
- Valores desconocidos
- Data frames.
- Nombres
- Subconjuntos
- Entrada y salida de datos.
- Funciones.
- Estructuras de control
- Gráficos.
- Datos temporales.
- Generación de datos aleatorios.
- Otros operadores.

Tipos de datos

- R tiene 5 tipos básicos:
  - character
  - numeric(real numbers)
  - Integer
  - Complex
  - Logical(True/False)

Números

- Los números en R son tratados como objetos numéricos(números reales de doble precisión).
- Si se quiere usar un número entero, es necesario especificar el sufijo L.
- Por ejemplo 1 es un objeto numérico, sin embargo 1L es un entero.
- Existe un número especial Inf que representa el infinito. Por ejemplo 1/0 sería Inf.
- Inf puede ser usado en cualquier cálculo como por ejemplo 1 / Inf que equivale a 0
- Existe un valor NaN que representa un valor indefinido o desconocido.

Atributos

- Los objetos en R tienen un conjunto de atributos:
  - Nombre(names)
  - Dimensiones(dimensions)
  - Clase(class)
  - Longitud(length)
  - Otros
- Se puede acceder a los atributos de un objeto usando la función attributes()

Entrada de datos

- El símbolo usado para introducir datos en R es <-
- Por ejemplo:

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
```

Evaluación

- Cuando se introduce una expresión, entonces el sistema la evalúa y devuelve el resultado de la misma. El resultado puede que se imprima automáticamente.

```
> x <- 5
> x
[1] 5
> print(x)
[1] 5
```

- En el ejemplo, [1] indica que x es un vector y que 5 es su primer elemento.

Vectores

Vectores

- Un vector es una colección de elementos del mismo tipo.
- Se puede crear de diferentes formas:
  - Usando la función c()

```
> x <- c(0.5, 0.6)
> x <- c(TRUE, FALSE)
> x <- c(T, F)
> x <- c("a", "b", "c")
> x <- 9:29
> x <- c(1+0i, 2+4i)
```

  - Usando la función vector()

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

  - Si se usa la función vector() sin argumentos, se crea un vector vacío.

Vectores

- Cuando se mezclan diferentes objetos en un vector, se produce una transformación de los elementos de forma que todos sean de la misma clase.

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

Vectores

- Los objetos pueden ser explícitamente cambiados su tipo mediante las funciones as.*

```
> 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"
```

Vectores

- Si la transformación que se quiere hacer no tiene sentido, entonces el sistema introduce el valor NA

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Matrices

- Las matrices son vectores con un atributo dimensión.
- La dimensión es un vector de enteros de longitud 2 que representa el número de filas y el número de columnas.

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices

Matrices

- Las matrices se rellenar comenzando por la parte superior izquierda, por columnas y hacia abajo.
- Por ejemplo:

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Observar que este ejemplo y algunos anteriores se ha usado el operador : , que permite crear secuencias determinas por el conjunto de números que hay entre el número de la izquierda y el número de la derecha.

Matrices

- Las matrices también pueden ser creadas desde un vector al que se le añade un atributo dimensión.
- Por ejemplo:

```
> 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
```

Matrices

- Otra forma de crear matrices es combinando vectores como si fueran columnas o como si fueran filas.
- El operador cbind() combina un conjunto de vectores como si fueran columnas y el operador rbind() combina un conjunto de vectores como si fueran filas.

```
> 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
```

Listas

Listas

- Las listas es un tipo de vector que contiene elementos de diferentes tipos.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

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

Factores

Factores

- Los factores es un tipo de datos que permite representar datos que representan categorías. Pueden entenderse como si fuera un vector de enteros donde cada entero tiene una etiqueta.
- Pueden ser ordenados o no.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
 2   3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Factores

- El orden se puede configurar mediante el argumento levels en el constructor del factor.
- Por ejemplo:

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Valores desconocidos

Valores desconocidos

- Los valores desconocidos son denotados por NA o NaN para operaciones matemáticas:
  - is.na() es usado para testear si un objeto es NA
  - is.nan() es usado para testear si un objeto es NaN
  - Los valores NA tienen una clase como integer NA, character NA,etc
  - Un valor NaN es también NA pero el contrario no es cierto.

Valores desconocidos

```
> 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
```

Data frames

Data Frames

- Los Data Frames son usados para almacenar valores tabulares:
  - Representan un tipo especial de lista donde cada elemento de la lista tiene que tener la misma longitud.
  - Cada elemento de la lista se puede entender como una columna y la longitud de cada elemento es el número de filas.
  - A diferencia de las matrices, los data frames pueden almacenar objetos de diferentes clases en cada columna(como las listas). Sin embargo las matrices deben tener todos los elementos de la misma clase.
  - Los Data Frames tienen un atributo especial denominado row.names que indica el nombre de las filas.

Data Frames

- Los Data Frames son creados usualmente mediante una llamada a read.table() o read.cvs()
- Un Data Frame puede ser convertido a una matriz mediante la función 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
```

Nombres

Nombres

- En R los objetos tienen nombres.

```
> 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
```

Nombres

- Y de la misma forma les pasa a las matrices:

```
> 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
```

Subconjuntos

Subconjuntos

- Existen operadores que pueden ser usados para extraer subconjuntos de los objetos de R:
  - [retorna un objeto de la misma clase que el original.Puede ser usado para seleccionar más de un elemento.
  - [[es usado para extraer elementos de una lista o data frame. Solo puede ser usado para extraer un único elemento, y la clase del elemento retornado no será necesariamente una lista o data frame.
  - $ es usado para extraer elementos de una lista o data frame por nombre. Es similar al [[

Subconjuntos

```
> 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"
```

Subconjuntos

- En las matrices se pueden obtener subconjuntos utilizando los índices. Como por ejemplo.

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

- También se pueden utilizad índices incompletos que indican todos los elementos.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Subconjuntos

- Por defecto cuando se retorna un único elemento de una matriz, se retorna como un vector de longitud 1 en vez de una matriz 1x1. Este efecto se puede configurar mediante el atributo drop=FALSE.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,] 3
```

Subconjuntos

- Por defecto cuando se retorna un único elemento de una matriz, se retorna como un vector de longitud 1 en vez de una matriz 1x1. Este efecto se puede configurar mediante el atributo drop=FALSE.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,] 3
```

- De la misma forma ocurre cuando se obtiene un subconjunto de una fila o de una columna(devolverá un vector y no una matriz).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Subconjuntos

- Algunos ejemplos de subconjuntos de listas

```
> 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"
```

Subconjuntos

- El operador [[puede ser usado con índices, y $ sólo puede ser usado con nombres literales.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]]
[1] 1 2 3 4
> x$name
NULL
> x$foo
[1] 1 2 3 4
```

- El operador [[puede recibir 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
```

Subconjuntos

- También el operador [[permite realizar un emparejamiento parcial de nombres. Como por ejemplo en:

```
> 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
```

Eliminación de valores Na

- Una tarea bastante común en R es eliminar valores desconocidos. Se pueden tener varios casos:
  - En un único objeto.

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

  - En varios objetos a la vez.

```
> 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"
```

Entrada y salida de datos

Lectura de datos

- En R existen varias funciones para la lectura de datos:
  - read.table, read.cvs para la lectura de datos tabulares(data frames).
  - readLines para la líneas de un archivo de texto.
  - source, dget para la lectura de archivos que contienen código en R.
  - load para la lectura de un espacio de trabajo.
  - unserializa para la lectura de objetos R en formato binario.

Escritura de datos

- En R existen funciones análogas para la escritura:
  - write.table.
  - writeLines
  - dump
  - dput
  - save
  - serialize

Lectura de datos con read.table

- La función read.table es una de las funciones más comunes para la lectura de datos, y tiene los siguientes argumentos:
  - file es el nombre del archivo o conexión.
  - header es un indicador lógico que indica si el archivo tiene una cabecera.
  - sep es una cadena indicando como están separadas las columnas.
  - colClasses es un vector de caracteres indicando la clase de cada columna en el conjunto de datos.
  - nrows indica el número de filas en el conjunto de datos.

Lectura de datos con read.table

- comment.char es una cadena de caracteres indicando el carácter comentario.
- skip, indica el número de líneas que debe no leer desde el comienzo del archivo.
- stringsAsFactors indica que variables deberían ser codificadas como factores.

Lectura de datos con read.table

- comment.char es una cadena de caracteres indicando el carácter comentario.
- skip, indica el número de líneas que debe no leer desde el comienzo del archivo.
- stringsAsFactors indica que variables deberían ser codificadas como factores.

- Para conjuntos de datos con tamaños pequeños se puede usar read.table sin especificar argumentos, y en este caso:
  - Se saltan las líneas del comienzo con un #
  - Descifra cuantas filas tiene
  - Descifra que tipo de variable hay en cada columna
- La función read.cvs es igual que read.table pero usa como separador la coma.

Lectura de datos con read.table

- Cuando se van a leer conjuntos de datos muy grandes, entonces hay que tener en cuenta algunos detalles:
  - Configurar el argumento comment.char="" en caso de que no existan líneas con comentarios.
  - Configurar el argumento nrows.
  - Usar el argumento colClasses. En este sentido, a veces cuando no se conoce, se puede usar la siguiente estrategia:

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

Lectura de objetos de R

- Una forma de gestionar objetos de R es mediante las instrucciones dput y dget.

```
> 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
```

Lectura de objetos de R

- También puede usarse dump y source.

```
> 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"
```

Interfaces de conexión

- Para poder realizar la lectura de un archivo previamente es necesario realizar una conexión, para lo cual existen varias formas:
  - file abre una conexión a un archivo.
  - gzfile abre una conexión a un archivo comprimido con gzip.
  - bzfile abre una conexión a un archivo comprimido con bzip2.
  - url abre una conexión a una página web.

Conexión file

- La conexión file tiene la siguiente estructura:

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

donde:

- description es el nombre del archivo.
- open es un código indicando:
  - “r” indica solo lectura.
  - “w” indica escritura e inicialización de un nuevo archivo.
  - “a” indica añadir.
  - “rb”,”wb”,”ab” indica escribir, leer o añadir en modo binario.

Conexión file

- En general, en las conexiones no es necesario tratar con las interfaces directamente. Por ejemplo la conexión siguiente:

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

- Es equivalente a:

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

Lectura de líneas de un archivo de texto

- Para leer líneas de un archivo de texto se puede usar la función readLines.
- En el siguiente ejemplo se lee información de un archivo comprimido.

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"      "10-point" "10th"      "11-point"
 [5] "12-point" "16-point" "18-point" "1st"
 [9] "2"         "20-point"
```

- En el siguiente ejemplo se lee información de una página web.

```
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
> head(x)
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8
```

Funciones

Funciones

- Las funciones en R se crean usando la directiva function(), y se almacenan como objetos R.

```
f <- function(<arguments>) {

}
```

- Pueden ser pasadas como argumentos a otras funciones.
- Pueden estar anidados, por lo que se puede definir una función en el interior de otra función.
- El valor retornado por una función es la última expresión evaluada en el cuerpo de la función.

Funciones

- Las funciones pueden tener argumentos con nombres y pueden tomar valores por defecto:
  - Los argumentos formales están incluidos en la definición de la función.
  - La función formals retorna una lista con todos los argumentos formales.
  - Los argumentos de una función podrían ser desconocidos o tener valores por defecto.

Funciones

- Los argumentos de una función pueden ser emparejados posicionalmente o por nombre.
- En el siguiente ejemplo, todas las llamadas a sd son 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)
```

- Se puede mezclar emparejamiento por nombre con posicional. Para ello si un argumento encaja por nombre se considera así, y el resto de argumentos se tratan de emparejar posicionalmente.

Funciones

- Vamos a considerar el siguiente ejemplo:

```
> 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, ...)
```

- Las siguiente dos llamadas son equivalentes:

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

Funciones

- Vamos a considerar el siguiente ejemplo:

```
> 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, ...)
```

- Las siguiente dos llamadas son equivalentes:

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

- También es posible un emparejamiento parcial. Así cuando se tiene que hacer un emparejamiento, el orden que sigue el lenguaje es:
  - Emparejamiento por nombre.
  - Emparejamiento parcial.
  - Emparejamiento posicional.

Funciones

- Cuando se define una función si no se define un valor por defecto, es posible asignarle el valor NULL.

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

}
```

- Los argumentos en las funciones se evalúan de forma perezosa, es decir que sólo se evalúa lo que es necesario.

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

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

- En esta función no se usa el argumento b, por lo que en la llamada 2 se empareja posicionalmente con a.

Funciones

- Sin embargo, en el siguiente ejemplo se produce un error dado que es necesario el argumento b.

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

```
## [1] 45
```

```
## Error: argument "b" is missing, with no default
```

El argumento …

- El argumento … indica un número variable de argumentos que son pasados a otras funciones.
- Por ejemplo un uso típico de este argumento, es cuando se extiende una función y no se quieren copiar la lista de los argumentos de la función original.

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

- También se usa cuando el número de argumentos no se conoce.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

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

El argumento …

- Una excepción con el argumento …, es que cualquier argumento que aparezca después de … , debe aparecer nombrado explícitamente, y no puede haber un emparejamiento parcial.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

Operaciones vectorizadas

- R admite operaciones sobre vectores.

```
> 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
```

Operaciones vectorizadas

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Estructuras de control

Estructuras de control

- R tiene un conjunto de estructuras de control que organizan el flujo de la ejecución de un programa dependiendo de las condiciones del entorno:
  - if,else: Se testea una condición.
  - for: ejecuta un bucle un número de veces fijo.
  - while: ejecuta un bucle mientras que una condición sea verdad.
  - repeat: ejecuta un bucle infinitamente.
  - break: interrumpe la ejecución de un bucle.
  - next: salta una iteración de un bucle.
  - return: abandona una función.

Estructuras de control: if

- La estructura de un if es la siguiente:

```
if (<condición>) {
        Hacer algo
} else {
        Hacer algo
}
if (<condición>) {
        Hacer algo
} else if (<condición>) {
        Hacer algo
} else {
        Hacer algo
 }
```

Estructuras de control: if

- El siguiente ejemplo representa una construcción válida:

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

- Sin embargo el siguiente ejemplo no es válido:

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

- Observar que la clausula else no es necesaria.

Estructuras de control: for

- El for toma una variable iterador y le asigna sucesivos valores de una secuencia o vector.
- Suele utilizarse para iterar sobre los elementos de un objeto tales como listas vectores.

```
for(i in 1:10) {
        print(i)
}
```

- En este ejemplo la variable i toma los valores 1,2,..10, los va imprimiendo, y a continuación sale del bucle

Estructuras de control: for

- En el siguiente ejemplo, los tres bucles hacen lo mismo.

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

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

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

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

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

Estructuras de control: for

- Los bucles for pueden anidarse entre si como en el siguiente ejemplo.

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

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Estructuras de control: while

- Los bucles while comienzan testeando el valor de una condición, de forma que si es cierta, entonces se ejecuta el cuerpo del bucle. Una vez que se ha ejecutado el cuerpo,la condición es testeada nuevamente, y así hasta que se hace falsa.

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

Estructuras de control: while

- Puede existir más de una condición a evaluar, y en este caso se evalúan de izquierda a derecha.

```
z <- 5

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

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

Estructuras de control: repeat

- Inicia un bucle infinito, de manera que la única forma de salir del mismo es mediante la instrucción break.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()

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

Estructuras de control: next

- En el siguiente ejemplo se ilustra el uso de next, en el cual se salta 20 iteraciones.

```
for(i in 1:100) {
        if(i <= 20) {

                next
        }

}
```

Gráficos

Gráficos

- R incluye muchas funciones para hacer gráficos.
- El comando plot es uno de los más utilizados. Si escribimos plot(**x**, **y**) donde **x** e **y** son vectores con *n* coordenadas, entonces R representa el gráfico de dispersión con los puntos de coordenadas (*xi*, *yi*).
- Se puede configurar el gráfico para establecer las leyendas y las etiquetas. Los argumentos pueden consultarse en help(plot)

```
x <- runif(50, 0, 4); y <- runif(50, 0, 4)
plot(x, y, main = "Título principal",sub = "subtítulo", xlab =
"eje x", ylab = "eje y",xlim = c(-5,5),ylim = c (-5,5))
```

Gráficos

- Algunos ejemplos de uso de plot:
  - z <- cbind(x,y)
  - plot(z)
  - plot(y ~ x)
  - plot(log(y + 1) ~ x) # transformaci´on de y
  - plot(x, y, type = "p")
  - plot(x, y, type = "l")
  - plot(x, y, type = "b")
  - plot(c(1,5), c(1,5))
  - legend(1, 4, c("uno", "dos", "tres"), lty = 1:3, col = c("red", "blue", "green"), pch = 15:17, cex = 2)

Gráficos

- Con text se pueden representar caracteres de texto directamente:
  - altura <- c(rep("v", 20), rep("m", 30))
  - plot(x, y, type = "n")
  - text(x, y, labels = altura)
- Puntos:
  - points(x, y, pch = 3, col = "red")
- Tipos de puntos.
  - plot(c(1, 10), c(1, 3), type = "n", axes = FALSE,xlab = "", ylab="")
  - points(1:10, rep(1, 10), pch = 1:10, cex = 2, col = "blue")
  - points(1:10, rep(2, 10), pch = 11:20, cex = 2, col = "red")
  - points(1:10, rep(3, 10), pch = 21:30, cex = 2,col = "blue", bg = "yellow")

Gráficos

- Tipos de líneas:
  - plot(c(0, 10), c(0, 10), type = "n", xlab ="",ylab ="")
  - for(i in 1:10) abline(0, i/5, lty = i, lwd = 2)
- Observar que lty permite especificaciones más complejas (longitud de los segmentos que son alternativamente dibujados y no dibujados).

Gráficos

- par controla muchos parámetros gráficos. Por ejemplo, cex puede referirse a los “labels” (cex.lab), otro, cex.axis, a la anotación de los ejes, etc.
- Por ejemplo mediante par(mfrow=c(*filas*,*columnas*)) se pueden mostrar muchos gráficos en el mismo dispositivo.
  - par(mfrow = c(2, 2))
  - plot(rnorm(10))
  - plot(runif(5), rnorm(5))
  - plot(runif(10))
  - plot(rnorm(10), rnorm(10))

Gráficos

- Otros gráficos que pueden realizarse son:
  - Diagrama de dispersión múltiple.
    - X <- matrix(rnorm(1000), ncol = 5)
    - colnames(X) <- c("a", "id", "edad", "loc","weight")
    - pairs(X)
  - Gráficos condicionados.
    - Y <- as.data.frame(X)
    - Y$altura <- as.factor(c(rep("Bajo", 80),rep("Alto", 120)))
    - coplot(weight ~ edad | altura, data = Y)
    - coplot(weight ~ edad | loc, data = Y)
    - coplot(weight ~ edad | loc * altura, data = Y)

Gráficos

- Boxplot

  Los diagramas de caja son muy útiles para ver rápidamente las principales características de una variable cuantitativa, o comparar entre variables.
  - attach(Y)
  - boxplot(weight)
  - plot(altura, weight)
  - detach()
  - boxplot(weight ~ altura, data = Y, col = c("red", "blue"))

Gráficos

- Se pueden añadir muchos elementos a un gráfico, tales como leyendas o líneas rectas:
  - x <- rnorm(50)
  - y <- rnorm(50)
  - plot(x, y)
  - lines(lowess(x, y), lty = 2)
  - plot(x, y)
  - abline(lm(y ~ x), lty = 3)

Gráficos

- A los datos cuantitativos se les puede añadir un poco de ruido con el comando jitter.
  - dc1 <- sample(1:5, 500, replace = TRUE)
  - dc2 <- dc1 + sample(-2:2, 500, replace = TRUE,prob = c(1, 2, 3, 2, 1)/9)
  - plot(dc1, dc2)
  - plot(jitter(dc1), jitter(dc2))

Gráficos

- Se pueden modificar márgenes exteriores de figuras y entre figuras. Ver ?par y buscar oma, omi, mar y mai.
- Algunos gráficos:
  - Gráficos 3D: persp, image, contour.
  - Histogramas: hist
  - Gráficos de barras:barplot
  - Gráficos de comparación de cuantiles,usados para comparar la distribución de dos variables, o la distribución de unos datos frente a un estándar: qqplot, qqnorm.
  - Notación matemática (plotmath).
  - Gráficos tridimensionales dinámicos con XGobi y GGobi.

Gráficos

- Se puede especificar donde se quiere guardar el gráfico.
  - pdf(file = "f1.pdf", width = 8, height = 10)
  - plot(rnorm(10))
  - dev.off()
- Se puede copiar una figura a un fichero.
  - plot(runif(50))
  - dev.copy2eps()

Datos temporales en R

Datos de tipo temporal en R

- La representación en R de las fechas y el tiempo es especial:
  - Las fechas son representadas usando la clase Date.
  - El tiempo es representado mediante las clases POSIXct o POSIXlt
  - Las fechas son almacenadas internamente como el número de días desde 01-01-1970.
  - El tiempo es almacenado internamente como el número de segundos desde el 01-01-1970

Datos de tipo temporal en R

- Las fechas se pueden crear a partir de una cadena, como en el siguiente ejemplo:

```
x <- as.Date("1970-01-01")
x
## [1] "1970-01-01"
unclass(x)
## [1] 0
unclass(as.Date("1970-01-02"))
## [1] 1
```

- El tiempo es representado mediante las clases:
  - POSIXct es un entero muy largo, que es útil cuando se almacena tiempo en un data frame.
  - POSIXlt es una lista y almacena información tal como semana, día del año, mes y día del mes.
- Existe un conjunto de funciones genéricas que usan fechas y tiempo:
  - weekdays: da el día de la semana.
  - months: da el nombre del mes.
  - quarters: da el trimestre

Datos de tipo temporal en R

- El tiempo puede ser transformada una cadena de caracteres usando la función as.POSIXlt o as.POSIXct.

```
x <- Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
p <- as.POSIXlt(x)
names(unclass(p))
## [1] "sec"   "min"   "hour"  "mday"  "mon"
## [6] "year"  "wday"  "yday"  "isdst"
p$sec
## [1] 14.34
```

Datos de tipo temporal en R

- Se puede usar el formato as.POSIXct.

```
x <- Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
unclass(x)
## [1] 1359083054
x$sec
## Error: $ operator is invalid for atomic vectors
p <- as.POSIXlt(x)
p$sec
## [1] 14.37
```

- Existe la función strptime en el caso de que las fechas estén escritas en diferente formato.

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
## [1] "2012-01-10 10:40:00" "2011-12-09 09:10:00"
class(x)
## [1] "POSIXlt" "POSIXt"
```

Datos de tipo temporal en R

- Algunas operaciones que pueden realizarse sobre las fechas y el tiempo son: +,-,==,<=,…

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x-y
## Warning: Incompatible methods ("-.Date",
## "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator
x <- as.POSIXlt(x)
x-y
## Time difference of 356.3 days
```

Generación de datos aleatorios

Generación de datos aleatorios

- Las distribuciones de probabilidad usualmente asociadas 4 funciones:
  - d: densidad
  - r: generación aleatoria de números.
  - p: distribución acumulada
  - q: función cuantil.

Generación de datos aleatorios

- En el caso de la distribución Normal se requiere usar las siguientes funciones:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

- Observar que si Φ es la función de distribución acumulada de un distribución Normal, entonces pnorm(q)= Φ (q) y qnorm(p)= Φ^{-1} (p)

```
> x <- rnorm(10)
> x
 [1] 1.38380206 0.48772671 0.53403109 0.66721944
 [5] 0.01585029 0.37945986 1.31096736 0.55330472
 [9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
 [1] 23.38812 20.16846 21.87999 20.73813 19.59020
 [6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.32   19.73   20.55   20.67   21.67   23.39
```

Generación de datos aleatorios

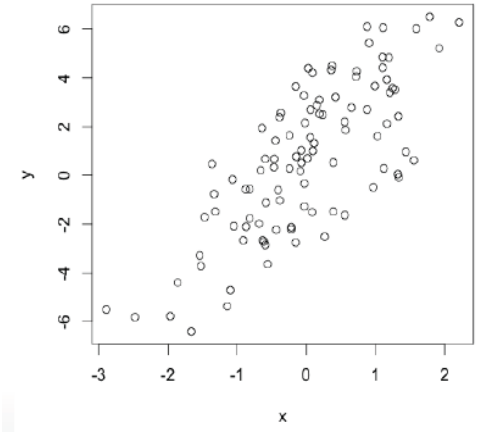
- Hay que configurar la semilla usando set.seed, antes de usar estas funciones:

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Generación de datos aleatorios

- Supóngase que se quiere simular un modelo lineal de la forma $y = \beta_0 + \beta_1 x + \varepsilon$

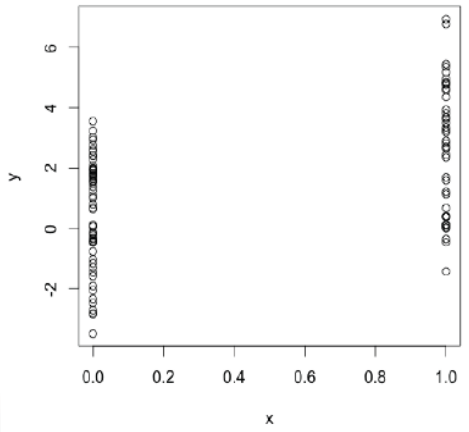
```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```



Generación de datos aleatorios

- Y si en el ejemplo anterior la x fuera binaria, entonces se tendría:

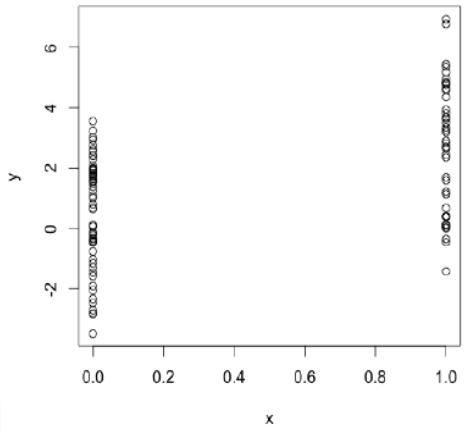
```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```



Generación de datos aleatorios

- Y si en el ejemplo anterior la x fuera binaria, entonces se tendría:

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```



Generación de datos aleatorios

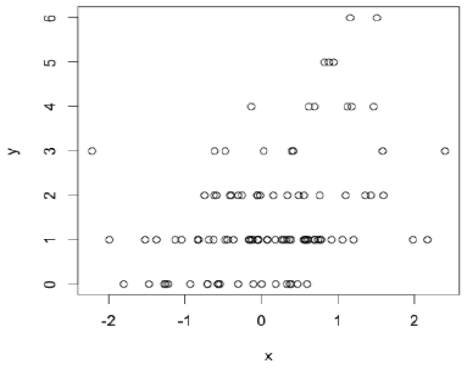
- Ahora supongamos que se quiere simular una poisson:

$$Y \sim \text{Poisson}(\mu)$$

$$\log \mu = \beta_0 + \beta_1 x$$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$.

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    1.00    1.55    2.00    6.00
> plot(x, y)
```



Otros operadores

Otros operadores:lapply

- En R hay un conjunto de operadores que implementan bucles:
  - lapply: Bucle sobre una lista que evalúa una función sobre cada elemento
  - sapply: Igual que lapply pero intenta simplificar el resultado.
  - apply: aplica una función sobre los márgenes de un array.
  - tapply: aplicar una función sobre subconjuntos de un vector.
  - mapply: Versión multivariante de lapply
  - split: es una función auxiliar que se usa con lapply

Otros operadores:lapply

- lapply toma tres argumentos:
  - Una lista X
  - Una función FUN
  - Y otros argumentos representados como …
- Si X no es una lista, entonces se le transforma usando as.list

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x7ff7a1951c00>
## <environment: namespace:base>
```

Otros operadores:lapply

- lapply devuelve una lista como resultado:

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.4671
```

- Otro ejemplo sería:

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.5261
##
## $c
## [1] 1.421
##
## $d
## [1] 4.927
```

Otros operadores:lapply

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

Otros operadores:lapply

- lapply hace uso de funciones anónimas:

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

- **En el siguiente ejemplo se hace uso de una función anónima para extraer la primera columna de cada matriz**

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

Otros operadores:sapply

- sapply intenta simplificar el resultado de lapply si es posible:
  - Si el resultado es una lista donde cada elemento es de longitud 1, entonces se devuelve un vector.
  - Si el resultado es una lista donde cada elemento es un vector de la misma longitud(> 1), entonces se devuelve una matriz.
  - En caso contrario se devuelve una lista.

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749


> sapply(x, mean)
        a         b         c         d
2.5000000 0.0608267 1.4670828 5.0747495

> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

Otros operadores:apply

- apply es usado para evaluar una función sobre los márgenes de un array:
  - A veces se usa para evaluar una función sobre las filas y columnas de una matriz.
  - Puede ser usado con array generales, como tomar la media de un array de matrics.
- Tiene la estructura:

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- X es un array.
- MARGIN es un vector de enteros que indica los márgenes.
- FUN es la función que debe ser aplicada.
- … es para otros argumentos que deben ser pasados a la función.

Otros operadores:apply

- Por ejemplo:

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```

- Por ejemplo para sumas o medias de dimensiones de una matriz:

```
rowSums = apply(x, 1, sum)
rowMeans = apply(x, 1, mean)
colSums = apply(x, 2, sum)
colMeans = apply(x, 2, mean)
```

Otros operadores:apply

- Para los cuantiles de las filas de una matriz.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]        [,2]       [,3]        [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75%  0.9258157  0.07065724  0.3050407 -0.06585436
          [,5]       [,6]      [,7]       [,8]
25% -0.05999553 -0.6588380 -0.653250 0.01749997
75%  0.52928743  0.3727449  1.255089 0.72318419
          [,9]      [,10]      [,11]      [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75%  0.3352377  0.7297176  0.3113434  0.4581150
         [,13]      [,14]      [,15]      [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75%  0.5326299  0.5064267  0.4933852  0.8868922
         [,17]      [,18]      [,19]     [,20]
```

- La matriz de las medias de un array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908

> rowMeans(a, dims = 2)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```

Otros operadores:mapply

- Es una aplicación multivariada de tipos los cuales son aplicados a una función en paralelo sobre un conjunto de argumentos.
- Tiene la estructura:

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
        USE.NAMES = TRUE)
```

- Donde:
  - FUN es la función a aplicar.
  - … contiene los argumentos sobre los que se aplica.
  - MoreArgs es una lista de otros argumentos de FUN
  - SIMPLIFY indica si el resultado debe ser simplificado.

Otros operadores:mapply

- Por ejemplo la siguiente operación:

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

- Se puede hacer de la siguiente manera usando mapply:

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

- Otra aplicación es vectorizar una función:

```
> noise <- function(n, mean, sd) {
+ rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1]  2.4831198  2.4790100  0.4855190 -1.2117759
[5] -0.2743532

> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266  4.2507057  1.1572738
[5]  3.7413584
```

Otros operadores:mapply

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.658653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

Otros operadores:mapply

- Esta operación es la misma que:

```
list(noise(1, 1, 2), noise(2, 2, 2),
     noise(3, 3, 2), noise(4, 4, 2),
     noise(5, 5, 2))
```

Otros operadores:tapply

- tapply es usado para aplicar una función sobre subconjuntos de un vectos.
- Tiene la estructura:

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- Donde:
  - X es un vector.
  - INDEX es un factor o una lista de factores.
  - FUN es la función a ser aplicada
  - … contiene los argumentos pasados a la función
  - simplify indica si el resultado debería ser simplifcado

Otros operadores:tapply

- Por ejemplo para tomar grupos de medias:

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1144464 0.5163468 1.2463678
```

- Y si queremos que se realice sin simplificación:

```
> tapply(x, f, mean, simplify = FALSE)
$'1'
[1] 0.1144464

$'2'
[1] 0.5163468

$'3'
[1] 1.246368
```

Otros operadores:tapply

- Otro ejemplo puede ser el encontrar grupos de rangos

```
> tapply(x, f, range)
$`1`
[1] -1.097309  2.694970

$`2`
[1] 0.09479023 0.79107293

$`3`
[1] 0.4717443 2.5887025
```

Otros operadores:split

- split toma un vector u otros objetos y los divide en grupos determinados por un factor o lista de factores.
- Tiene la estructura:

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- Donde:
  - X es un vector(o lista) o un data frame
  - f es un factor o lista de factores
  - drop indica que si los niveles de factores vacíos deben ser eliminados

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
 [5]  0.2849881  0.9383361 -1.0973089  2.6949703
 [9]  1.5976789 -0.1321970

$`2`
 [1] 0.09479023 0.79107293 0.45857419 0.74849293
 [5] 0.34936491 0.35842084 0.78541705 0.57732081
 [9] 0.46817559 0.53183823

$`3`
 [1] 0.6795651 0.9293171 1.0318103 0.4717443
 [5] 2.5887025 1.5975774 1.3246333 1.4372701
```

Otros operadores:split

- split se suele usar en combinación con lapply

```
> lapply(split(x, f), mean)
$'1'
[1] 0.1144464

$'2'
[1] 0.5163468

$'3'
[1] 1.246368
```

- En el siguiente ejemplo se usa sobre un data frame

```
> library(datasets)
> head(airquality)
  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
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$'5'
   Ozone  Solar.R     Wind
      NA       NA 11.62258
$'6'
    Ozone   Solar.R     Wind
       NA 190.16667 10.26667
$'7'
    Ozone    Solar.R      Wind
       NA 216.483871  8.941935
```

Otros operadores:split

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
              5         6         7        8        9
Ozone        NA        NA        NA       NA       NA
Solar.R      NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667  8.941935 8.793548  10.1800

> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
                                 na.rm = TRUE))
                5          6           7           8          9
Ozone    23.61538   29.44444   59.115385   59.961538   31.44828
Solar.R 181.29630  190.16667  216.483871  171.857143  167.43333
Wind     11.62258   10.26667    8.941935    8.793548   10.18000
```

- Se puede dividir más de un nivel

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Otros operadores:split

- Se pueden crear niveles vacíos

```
> str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] -0.378  0.445
 $ 2.1: num(0)
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 2.2: num(0)
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 1.4: num(0)
 $ 2.4: num [1:2] -0.907  0.723
 $ 1.5: num(0)
 $ 2.5: num [1:2] 0.732 0.360
```

- Y también se pueden eliminar los niveles vacíos:

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
 $ 1.1: num [1:2] -0.378  0.445
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 2.4: num [1:2] -0.907  0.723
 $ 2.5: num [1:2] 0.732 0.360
```

Más información

- Para profundizar en R:
  http://cran.r-project.org/doc/manuals/r-release/R-intro.html