Cosas de R

> Por: Mapachana

Operaciones

Símbolo	Operación	Ejemplo
+	Suma	2+2
-	Resta	3-1
*	Multiplicación	2*4
/	División	5/4
^	Potencia	2^3
log(x)	Logaritmo	log(2)
%%	Módulo o resto	5%%4
%/%	División entera	10%/%4
exp(x)	Exponencial	exp(2)
log(x,n)	Logaritmo de x en base n	
sqrt(x)	Raíz de x	
abs(x)	Valor absoluto de x	
cos(x)	Coseno	
sin(x)	Seno	
tan(x)	Tangente	
factorial(x)	Factorial	
choose(n,x)	n!/(x!(n-x)!)	
gamma(x)	Función gamma de x	
floor(x)	Mayor entero más cercano a x	
ceiling(x)	Menor entero más cercano a x	
trunc(x)	floor para positivos y ceiling para negativos	
round(x, digits=n)	Redondeo con n decimales	
Re(x)	Parte real	
Im(x)	Parte imaginaria	
Mod(x)	Módulo	

Números complejos

La letra `i` representa la parte imaginaria. Ejemplos:

```
(1+1i)*(3-2i)
(1+1i)/(3-2i)
```

Importante!! No se pone `*` antes de la `i`, entonces no lo coge bien. Solo se pone 1i y 2i.

Operadores lógicos

Símbolo	Operación
!	Negación
&	Y
|	O
xor(x,y)	xor (O exclusivo)

Valores predefinidos

Símbolo	Valor
pi	Letra pi, 3.14blabla
Inf	Infinito
NaN	Not a Number
NA	Not available (valor perdido)

Cualquier operación con NaN devuelve NaN.

Funciones de ayuda

Si escribimos `help` devuelve el código de la función help (su implementación).

Si escribimos `help()` muestra la ayuda en línea de R.

Si escribimos `help(log)` o `help("log")` (hacen lo mismo con o sin comillas) (log puede cambiarse por cualquier nombre de función) nos muestra la ayuda de esa función.

Si escribimos `help.search("read data")` nos muestra resultados de búsqueda de read data, esto se puede cmabiar por otro término. (No es muy útil).

La función `example()` nos da ejemplos de una función, por ejemplo para obtener ejemplos de la media sería `example(mean)`.

La función `demo()` hace demostraciones de funciones gráfiacs, como `demo(graphics)` o `demo(persp)`.

Asignaciones

Se hacen con `<-`. Por ejemplo `x<-exp(2)`. Así ahora x guarda la exponencial de 2.

Los nombres de variables pueden contener letras, números, . y _, y es importante que nunca empiecen por _ o por un número.

Comparaciones

Al comparar hay que tener en cuenta que `==` comprueba si son exactamente iguales, y como al hacer operaciones hay errores de redondeo hay que usar `all.equal(exp, res)`

Por ejemplo `all.equal(0.3-0.2, 0.1)`.

Gestionar objetos

Con `ls()` se listan los objetos creados, y con `rm()` se borra un objeto.

Por ejemplo para borrar x e y es `rm(x,y)`.

Para borrar todos los objetos se puede ejecutar `rm(list=ls())`.

Vectores

Conjunto de valores del mismo tipo (nmérico, carácter, lógico, etc) de dimensión 1.

c()

La función `c()` concatena los valores que se le pasan Puede recibir elementos (2, 5, num) u otros vectores. Si se le pasan valores de distinto tipo los convierte todos al mismo. Ejemplo `x<-c(1,2,3,4,5); z<-c(1,2,x,6)`.

names

Se le puede dar nombre a los elementos de un vector con names. `names(x)<-c("x1", "x2")`.

seq: Secuencias

Se puede usar `:` para poner rangos, por ejemplo `1:10` son los valores del 1 al 10 (ambos incluidos).

También en orden inverso: 5:1.

También se puede usar `seq` especificando el primer valor (from), el último (to) y la longitud del vector a generar (length.out) y el paso (by). Por ejemplo `seq(-pi, pi, lenght.out=10)`.

```
seq(1,10)
seq(1,10, by=2)
seq(1,10, length.out=15)
```

rep

rep permite crear un vector repitiendo valores:

```
rep(1,3) # 1 1 1
x <- 10:12
rep(x, each=2) # 10 10 11 11 12 12
rep(x, length.out=7) # 10 11 12 10 11 12 10
rep(x, each=2, length.out=9) # 10 10 11 11 12 12 10 10 11
rep(1:4, each=2, times=3) # 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

En el primer ejemplo, recibiendo dos argumetnos simples, `rep(x,y)` se repite x y veces.

Si rep recibe un vector x, el argumento each indica cuántas veces se repite cada elemento de x, length.out indica la longitud dle vector a generar y times es cuántas veces de va a repetir. Los argumentos times y length.out no deberían ir juntos (creo).

vector

Se crean vectores indicando el tipo y la longitud. Los posibles tipos son `numeric` (engloba int y double), `logical`, `complex` y `character`.

`vector("numeric", 2)`

Se inicializan a 0 en numeric, FALSE en logical y "" en character.

Generar vectores lógicos

Se pueden generar a partir de condiciones de vectores de otros tipos. Por ejemplo si x es un vector numérico:

```
temp <- x>0 & x<=3
```

Esto genera un vector de la misma longitud de x que contiene FALSE o TRUE según si cada elemento de esa posición de x cumple o no esa condición.

Paste

Sirve para crear vectores de tipo `character` (cadenas de caracteres). Pega varios vectores. Un argumento relevantes es sep, que indica la separación a usar.

Una variante es `paste0`, que (si no me equivoco) lleva implicito el separador vacío.

```
nombres <- paste(c("X", "Y"), 1:10, sep="")
# X1 Y2 X3 Y4 X5 Y6 X7 Y8 X9 Y10
nth <- paste0(1:10, c("st", "nd", "rd", rep("th",7)))
# 1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
```

months y letters

`letters` y `LETTERS` son vectores que contienen todas las letras del abecesario (en minúscula y mayúscula respectivamente), mientras `month.name` tiene los meses el año.

```
letters[1:5] # "a" "b" "c" "d" "e"
LETTERS[1:5] # "A" "B" "C" "D" "E"
month.name[1:2] # "January" "February"
```

Acceder a los elementos

Se accede con `[]`, y dentro puedes poner un número o un vector con los elemetnos a los que quieres acceder:

```
x <- seq(0,1, length.out=10)
x[2] # accede segundo elemento
x[c(2,4)] # accede elementos 2 y 4
x[1:3] # accede elementos del 1 al 3
x[seq(1,10, by=2)] # accede elementos del 1 al 10 sumando 2
```

Si en los corchetes ponemos elementos negativos, eliminamos esas posiciones (no se accede a ellas).

```
x[-c(2,5)] # Accede a todas menos 2 y 5
```

También se puede acceder a elementos que cumplan una condición lógica:

```
x <- -5:5
x[x>0]
x[x>0 & x<=3]

peso <- c(60,70,56,70)
sexo <- c("F", "M", "F", "M")
peso[sexo=="F"]
# Si peso y sexo no tienen misma longitud aparecen NA si se accede fuera del rango del vector
```

Si se han definido nombres se puede acceder mediante ellos también con:

```
names(peso) <- c("Marta", "Jose", "Paula", "Paco")
peso[c("Marta", "Paula")]
```

Valores perdidos (NA)

Para localizar los valores perdidos usams la función `is.na(x)`. Devuelve un vector lógico indicando TRUE si es NA y FALSE si no.

```
is.na(x) <- c(2,3) # Define como NA las posiciones 2 y 3
x[is.na(x)] <- 0 # Asigna 0 a los NA de x
```

¡Ojo! Hay una función `is.nan(x)` para detectar los NaN.

is.na devuelve true tanto en Na como NaN, mientras is.nan solo devuelve true en NaN.

Operaciones con vectores

Las operaciones se hacen elemento a elemento (sumas el primer elemento de uno con el primero de otro, luego sel segundo etc.).

Si los vectores tienen distinta longitud, R repite los elementos del más corto ed ellos.

```
x <- 1:5
y <- seq(0,1,length.out=5)
x+y # 1 2.25 3.5 4.75 6
x*y # 0 0.5 1.5 3 5
exp(x)+log(1+y) # cosas
x^2 # 1 4 9 16 25
z <- c(-2, -1)
x+z # considera z -2 -1 -2 -1 -2
```

También se pueden considerar los valores TRUE o FALSE como valores aritméticos, esto resulta útil por ejemplo para funciones a trozos:

```
x<-(1:10)/10
(x<0.3)*(2*x)+(0.3<=x & x<0.7)*x^2+(x>=0.7)*2
```

Función	Explicación
sum	Suma de todos los elementos del vector
prod	Producto de todos los elementos del vector
cumsum	Sumas acumuladas de los elementos (1 1+2 1+2+3)
cumprod	Productos acumulados de los elementos (1, 1* 2, 1* 2 *3)
min	Elemento con valor mínimo del vector
max	Elemento con valor máximo del vector
which.min	Posición en la que está el mínimo
which.max	Posición en la que está el máximo
sort	Ordena un vector (de menor a mayor, salvo uso del parámetro decreasing)
order	Posiciones de los elementos en el vector ordenado. Si hacemos rev delante de order hace el orden inverso `rev(order(x))`
which	Devuelve la posición del elemento que cumple una condición

```r
x[which.min(x)] # Equivalente a min(x)
x[order(x)] # Equivalente a sort(x)

set.seed(1)
x <- runif(100)
mx <- mean(x)
which(abs(x-mx)==min(abs(x-mx)))
# Devuelve el elemento del vector más cercano a su media
```

Atributos

`mode` y `typeof` (`mode(x)` y `typeof(x)`) indican el tipo de vector (numeric, character, etc. Ojo que mode dice numeric y typeof integer).

`length` indica la longitud delvector. Se puede obetener con `length(x)` y se puede sobreescribir. SI se asigna una longitud mayor a la que tiene se rellena con NA, si se reduce simplemente se eliminan valores.

Para comprobar el tipo de objeto, además, está la función `class(x)` y para ver los atributos `attributes(x)`.

```r
x <- numeric() # creo vector de numeros x con longitud 0
length(x) # 0
x[3] <- 3 # Na NA 3

x <- 1:10 # 1 2 3 4 5 6 7 8 9 10
x <- x[3*(1:3)] # 3 6 9

x <- numeric()
length(x) <- 3 # x es NA NA NA
x <- 1:5
length(x) <- 10 # x es 1 2 3 4 5 NA NA NA NA NA
length(x) <- 3 # x es 1 2 3
```

Factores (factor)

Son un tipo especial de vectores, son variables nominales con un número fijo de valores (levels).

Crear factores

Se usa `factor(x, levels=, labels=)`. Solo es imprescindible el vector x del que se extraen los factores.

```r
civil<-c('soltero/a','viudo/a', 'casado/a',
'soltero/a','viudo/a', 'divorciado/a', 'soltero/a', 'casado/a',
'soltero/a', 'divorciado/a')
civil.f <- factor(civil)
```

En el argumento levels podemos especificar los niveles que queremos y en qué orden:

```r
factor(civil,levels=c('soltero/a', 'casado/a',
'divorciado/a','viudo/a'))
```

Si el vector es de tipo numérico, podemos asignarle etiquetas para hacerlo más fácil de manejar:

```r
sexo.f<-
factor(c(1,1,2,1,1,2,2,1,2,1),labels=c('hombre','mujer'))
```

Para convertir los niveles en números (ya estuvieran antes como úmeros, como en sexo, o no) usamos `unclass(civil.f)`, que asigna un número a cada nivel y devuelve el vector de estos.

Acceder a los elementos de un factor

Para acceder a los elementos se hace igual que los vectores.

Para acceder y modificar los niveles se usa `levels`:

```
levels(sexo.f) # "hombre" "mujer"
levels(sexo.f)[2] # "mujer" (segundo nivel)
levels(sexo.f) <- c('masculino', 'femenino') # cambia los
niveles
```

Nota: Aunque aquí se usen comillas simples lo he comprobado y no parece afectar si usamos comillas dobles.

Aplicaciones

Se puede usar `tapply` para hacer funciones a datos agrupados por los niveles:

```
edad<-c(23,25,20,19,20,22,24,20,23,19)
tapply(edad,sexo.f,mean)
tapply(edad,sexo.f,summary)
```

También son útiles para boxplot: `boxplot(edad~sexo.f)`.

Matrices

Almacenan datos en dos o más dimensiones (si son más se llaman arrays).

Tienen dimensiones (dim) y nombres para cada dimensión (dimnames).

Creación de matrices

A partir de un vector añadiendo el atributo dimensión:

```
x <- 1:10
dim(x) <- c(2,5)
# 1 3 5 7 9
# 2 4 6 8 10
```

Escriibir una matriz especificando los elementos y el número de filas y columas: `matrix(x, 2, 5)`. ¡ojo! Los elementos se ponen por columnas, es decir, si x tiene los números del 1 al 10, la matriz es:

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

Si en vez de un vector se usa un solo dato, se repite siempre ese dato como valor en la matriz: `matrix(data=pi,nrow=1,ncol=3)`.

También puede ser indicando los valores, y el argumento `byrow=TRUE` hace que se rellene por filas (por defecto lo hace por columnas). Además con nrow y ncol indicamos le número de filas y columnas.

```
matrix(1:10, nrow=2, ncol=5, byrow=TRUE)
# 1 2 3 4 5
# 6 7 8 9 10
```

A partir de varios vectores por columnas, con `cbind`: `cbind(x,y)`. Para que se rellen por filas, se usa `rbind`.

```
cbind(1:3, 4:6, 7:9)
# 1 4 7
# 2 5 8
# 3 6 9
rbind(1:3, 4:6, 7:9)
# 1 2 3
# 4 5 6
# 7 8 9
cbind(matrix(1:9, 3, 3), c(1,0,1))
# 1 4 7 1
# 2 5 8 0
# 3 6 9 1
```

Nombres de filas y columnas

Se asignan usando `rownames(A)` y `colnames(A)`.

Acceder a elementos

Se puede acceder a un elemento concreto especificando fila y columna con corchetes como `A[1,1]`, o a la columna 1 completa `A[,1]` o la fila 1 completa `A[1,]`.

También se pueden obtener submatrices poniendo rangos o no acceder a una columna concreta usando índices negativos: `A[1:2, 1:2]` y `A[,-1]`. También se pueden obtener matrices de expresiones lógicas como `A>3` y trabajar con esto para hacer asignaciones: `B[A>3]<-NA`.

También se pueden obtener sus elementos como si fuera un vector (concatenando sus columnas), y se accede al vector resultado como `A[1:5]`.

Operaciones con matrices

Función	Explicación
*	Multiplicación elemento a elemento
%*%	Multiplicación matricial
t()	Traspuesta
diag()	Diagonal
solve(A)	Inversa de A
solve(A,b)	Solución del sistema Ax=b
qr()	Descomposición de Cholesky
eigen()	Autovalores y autovectores
svd()	Descomposición singular
crossprod(A1, A2)	calcula t(A1)%*%A2
rowSums(A), colSums(A)	Sumas por filas y columnas
rowMeans(A), colMeans(A)	Medias por filas y columnas

Para hacer la desomposición de Cholesky de una matriz A sería:

```
AA <- crossprod(A,A)
solve(AA)
chol2inv(chol(AA))
```

Otra función es `outer`, que proporciona el producto exterior de dos matrices `outer(A,B,*)`.

Arrays

Son matrices de más de dos dimesiones.

Creación

Se pueden crear a partir de un vector con dim `dim(x) <- c(3,4,2)`, o con array `x <- array(1:24, dim=c(3,4,2))`.

Acceso

Sigue las mismas normas de las matrices.

Listas

Las listas permiten almacenar datos de distinto tipo.

Crear una lista

Usando `list` y pasandole las componentes:

```
A<-matrix(1:6,3,2)
m1<-rowMeans(A)
m2<-colMeans(A)
minA<-min(A)
maxA<-max(A)
nomb<-c('col1','col2','col3')
cond<-A > mean(A)+sd(A)
lista<-list(A,m1,m2,minA,maxA,nomb,cond)
```

También se puede crear una lista vacía con `vector(mode='list')`. o `vector(mode=''list'', length=num)` (especificando longitud num).

Nombres

Se puede dar nombre a algunas o todas las componentes de una lista. Se puede hacer al crearla especificando su nombre como `list(nombre1=objeto1, nombre2=objeto2, ...)`.

O bien usando `names` como `names(lista)<-c('nombre1', 'nombre2', ...)`.

La función length nos devuelve la longitud de la lista.

Acceder componentes

Se puede usar `[]`, `[[]]` o `$`:

Con `[]` se pueden seleccionar varios elementos o uno solo, mientras que `[[]]` y `$` solo permiten seleccionar una única componente.

```r
lista<-
list(matriz=A,med_fil=m1,med_col=m2,minimo=minA,maximo=maxA,
nombres=nomb,cond=cond)
lista[[1]] # Accede a la primera componente, la martriz A
lista[['matriz']] # Accede a la componente con nombre matriz,
que es la primera (como antes)
lista$matriz # Equivale a la anterior
lista[1] # Devuelve una lista con unico elemetno el primer
elemento de lista
lista[2:3] # lista con los elementos 2 y 3 de lista

lista[[1]][1,2] # Accede al elemetno 1, 2 de la matriz A
```

Se pueden añadir listas a una lista, no las concatena. (se añaden como los vectores, `lista[pos]<- cosa` aunque pos no este aun en la lista).

Concatenar listas

`c(lista1, lista2)` devuelve una lista que es la concatenación de las dos dadas.

Aplicaciones

Para darle nombres a las dimensiones de las matrices es necesario usar listas:

`matrix(1:4,2,2,dimnames=list(c('fil1','fil2'),c('col1','col2')))`.

Otro ejemplo es que la función `hist(x)` devuelve una lista (para hacer histogramas).

Dataframes

Almacena varios tipos distintos de datos en forma similar a una matriz.

Crear dataframes

Usando la función data.frame pasándole los datos por columnas como argumento como sigue:

```r
dni<-
c('22456715A','22456716B','22456717C','22456718D','22456719E')
edad<-c(45,35,52,60,25)
sexo<-factor(c('Hombre','Mujer','Hombre','Mujer','Hombre'))
estudios<-
factor(c('superior','superior','profesional','medio','profesion
al'))
salario<-c(2500,1500,2000,1200,1800)
datos<-data.frame(dni,edad,sexo,estudios,salario)
```

De esta forma toma por defecto como nombres de las columnas los nombres de los argumentos dados. Se puede cambiar especificando el nombre en la declaración como `data.frame(nombre1=cosa1, nombre2=cosa2, ...)`.

También se puede cambiar usando `names` como `names(datos) <- c('nombre1', nombre2)`. Si se quiere cambiar el de unas columnas concretas `names(datos)[c(1,5)]<-c('nombre1', 'nombre2')`.

Funciones útiles

La estructura del dataframe se ve con `str`: `str(datos)`.

También se puede ver un resumen con `summary(datos)`.

Otra es `searchpaths`, que devuelve el camino (path) a ñlos paquetes cargados.

Acceder a los datos

Es similar a las listas:

- `datos[n]` selecciona las columnas n, por ejemplo `datos[1:3]` selecciona las tres primeras columnas.
- datos[n1, n2], selecciona el elemento n1 de la columna n2, como una matriz. Así `datos[1:3, 1]` selecciona el primer elemento de las tres primeras filas.
- También se pueden seleccionar de forma que cumplan condiciones lógicas, como `datos[datos$sexo=='Hombre']`.

Para preservar la dimensión original de una selección se usa `drop` como :

`datos[,2,drop=FALSE]`. También se puede aplicar a matrices.

Mediante la función `subset` también se pueden seleccionar subconjuntos, como:

```r
subset(mtcars, subset= vs==0 & hp>90)
# Selecciono coches con cambio automatico y potencia mayor a 90
subset(mtcars,subset= vs==0 & hp>90, select=c(-vs))
# Seleccionamos todas las columnas menos vs
```

Gestión y manipulación

Se pueden transformar columnas existentes (usando el mismo nombre) o añadir nuevas (usando otro nombre) con `transform`. Para guardar los cambios debe asignarse al dataframe:

`transform(dataframe, expresion1, expresion2, ...)`

```r
transform(mtcars,wt=wt/2.2046) # modifica wt
transform(mtcars,wt2=wt/2.2046) # crea wt2
```

Otra opción para hacer transformaciones es within: `within(dataframe, {expr1, expr2, ...})`:

```r
mtcars2 <- within(mtcars, {
vs <- factor(vs, labels = c("V", "S"))
am <- factor(am, labels = c("automatic", "manual"))
cyl <- ordered(cyl)
gear <- ordered(gear)
carb <- ordered(carb)
})
```

Esta función devuelve el dataframe entero.

Otra función similar ws `with`, con la que podemos aplicar funciones sobre e dataframe: `with(datafame, funcion(columnas))`:

```r
with(mtcars2, boxplot(mpg~vs))
with(subset(mtcars2,vs=='V'),hist(mpg,main='vs=V'))
with(subset(mtcars2,vs=='S'),hist(mpg,main='vs=S'))
```

Esta función devuelve el valor de la expresión a evaluar.

Otras funciones son `merge` (unir dataframes), na.omit y complete.cases (gestionar valores perdidos), unique (borra columnas duplicadads) y match (compara dos dataframes).

Cambios y comprobación de tipo de objeto

Tipo	Comprobación	Cambio
Array	is.array	as.array
Carácter	is.character	as.character
Complejo	is.complex	as.complex
Dataframe	is.data.frame	as.data.frame
Double	is.double	as.double
Factor	is.factor	as.factor
Lista	is.list	as.list
Lógico	is.logical	as.logical
Matriz	is.matrix	as.matrix
Numérico	is.numeric	as.numeric
Vector	is.vector	as.vector

Para comprobar si un objeto está ausente o es nulo se usa `is.null`.

Leer y escribir datos de ficheros

read.table

Esta diseñada para leer en formato dataframe (las filas son observaciones y las columnas datos de estas).

```
1 datos<-read.table(file="datos.txt",header=TRUE)
```

`file` tiene el nombre del fichero de donde se va a leer y `header` indica si la primera línea tiene los nombre sde las variables. Otro argumento es `sep` que indica qué elemento se usa como separador. Para indicar el separador de números dedimales se usa `dec`, indicar el nombre de filas y columnas es `row.names` y `col.names`, `na.strings` especifica un vector de caracteres que serán interpretados como NA y `as.is` para indicar si las columnas de caracteres se tratarán como factopres.

Un ejemplo de as.is: `datos<-read.table(file="datos.txt",header=TRUE, as.is=c(1))`.

Si se va a usar la coma como delimitador se puede usar directamente `read.csv`.

El fichero debe estar en el directorio de trabajo, este pùede cambiarse con `setwd("/home/mapachana/examen")`. Y puede obtenerse con `getwd()`.

scan

Se asume que leemos números de doble precisión y se debe especificar cualquier otro tipo con `what`. Se asume los datos separados por espacios en blanco o tabuladores, salvo que se especifique con `sep`. No se pueden leer los nombrs de las variables, y si están se salta esa línea usando `skip=1`.

```
datos<-
scan(file="datos.txt",skip=1,what=list(DNI='',edad=0,sexo='',
estudios='',salario=0))
datos <- as.data.frame(datos)
```

Los datos se almacenan en una lista, luego para tener un dataframe se hace la última línea

Esta función también sirve para guardar todos los datos de forma interactiva cual cin usando `scan()`. Se para al leer una línea en blanco.

```
datos2 <- scan()
1
2 3 4
5


```

write.table

Tiene especificaciones similares a `read.table`:

```
write.table(cars[1:10,],file='datos1.txt')
```

Por defecto imprime los nombres de filas y columnas, pero se puede modificar con `row.names` y `col.names`

write

Para escribir se especifica el número de columnas.

```
write(runif(10),file='datos2.txt',ncolumns=1)
```

Si queremos escribir una matriz se especifica `ncol` para escribirla. Además hay que tener en cuenta que R traspone la matriz para escribirla.

Por tanto el análogo a la funcion de write.table es

```
write(t(as.matrix(datos1)),file='datos3.txt',ncolumns=2)
```

Para añadir al final del fichero en lugar de sobreescribir se usa `append=TRUE`.

Para borrar un ficheor desde R se usa `unlink(datos3)`.

Obtener datos de distribuciones

rnorm

Para obtener datos de una distribución normal se usa `rnorm(n)`, siendo `n` el número de datos a obtener. Por ejemplo, si quiero 50 datos `rnorm(50)`. Se puede especificar la media y desviación típica que queremos mediante `mean` y `sd` por ejemplo: `rnorm(50, mean=10, sd=2)`.

runif

Para obtener datos de una uniforme (0,1). `runif(n)`, con n el número de datos a obtener.

Gráficas

hist

Crea histogramas. Usa un argumento, `hist(x)`, en el eje x va x y en el eje y la frecuencia. Son barritas.

plot

Representa la distribución. `plot(x,y)`, en el eje x va x y en el eje y va y, se representa mediante puntos.

abline

Superpone una recta al gráfico anterior. `abline(fit)`.

contour

Representación tridimensional de f como función de x e y.

```
contour(x,y,f) # diagrama de contornos
contour(x,y,f, nlevels=15, add=TRUE) # añadir mas niveles
image(x,y,f) # mapa de colores
```

Funciones

Definición

```
f <- function(argumentos){
    sentencias
    valor devuelto
}
```

¡Ojo! El valor devuelto siempre coincide con la última de las sentencias. Si se quiere devolver otro se usa `return()`.

Se invoca como cualquier función ya hecha `funcion(argumentos)`.

Se indican valores por defecto como:

```
f.suma <- function(x=0, y=0){x+y}
```

Asi si no se da alguno de los valores se toma el por defecto.

No hace falta darles nombre a estas funciones, aunque luego no se puedan llamar de forma independiente. Si no tienen nombre se dicen funcionas anónimas. Se usan por ejemplo como argumento de apply o similares.

Componentes

Se pueden consultar y modificar las componentes de las funciones (argumentos o formals, cuerpo o body y entorno o environment).

```r
formals(f.suma)
body(f.suma)
environment(f.suma)
# Modificamos argumentos
argum.original <- formals(f.suma) # Almacenamos los originales
formals(f.suma) <- alist(x=, y=, z=)
# Modificar el cuerpo
cuerpo.original <- body(f.suma)
body(f.suma) <- expression({
    suma1 <- x+y
    suma2 <- x+z
    return(list(suma1=suma1, suma2=suma2))
})
```

Aunque el entorno dentro de la función es local (variables locales), se puede definir permanentemente una variable con `<<-` o `assign`.

Argumento especial ...

Existe un argumento epecial `...` que sirve para indicar cualquier número de argumentos. Por ejemplo lo tiene paste.

Cuando se usa el argumento `...` en una función, el resto de argumentos que le siguen se deben pasar siempre por nombre y no por posición.

Otro uso común de este argumento especial está relacionado con la posibilidad de
pasar argumentos opcionales a otras funciones utilizadas dentro del cuerpo de la función.

```r
grafico<-function(x,...)
{
z<-(x-mean(x))/sd(x)
plot(x,z,...)
}
```

missing

Una forma de comprobar si un argumento no está es con la función `missing`:

```r
factorial.d<-function(n){
    if (missing(n) || !is.numeric(n)) return(NA)
    n<-as.integer(n)
    fact<-1
    if (n>1) for (i in 1:n) fact<-fact*i
    return(fact)
}
factorial.d(5)
```

Estructuras de control

if

```r
if (condición1) {
    acción-verdad1
} else if (condición2) {
    acción-verdad2
} else if (condición3) {
    acción-verdad3
} else {
    acción-falso
}
```

Si la sentencia que sigue al `if` es de una sola línea no son necesarias las llaves.

Para las compuestas con operadores lógicos, se pueden usar doble s `&&` y `||`, de forma que con las dobles si no es necesario no se comprueba la segunda condición, mientras con las simples siempre se comprueban ambas.

```r
x<-5
if (x>4 | n>1) "se comprueban las dos"
# Error in eval(expr, envir, enclos): objeto 'n' no encontrado
if (x>4 || n>1) "se comprueba sólo la primera"
# Esto no da error porque solo se comprueba la primera
```

ifelse

La estructura ifelse sirve para condicionnes que devuelven vectores en vez de TRUE o FALSE.

```r
ifelse (condicion, a, b)
```

La condición se comprueba para cada elemento, y se realiza a si es verdadero y b si es falso.

```
milog<-function(x) ifelse(x==0,NA,log(x))
milog(c(1,0,2)) #0.0000000 NA 0.6931472
```

switch

Compacta el código de if else:

```
switch(expresión,valor_1=acción_1,valor_2=acción_2,...,valor_n=acción_n)
```

Un ejemplo:

```
opciones <- function(x) {
    if (x == "a") {
        "opción 1"
    } else if (x == "b") {
        "opción 2"
    } else if (x == "c") {
        "opción 3"
    } else {
        stop("Valor de x no válido")
    }
}

# Equivalente a
opciones <- function(x) {
    switch(x,
        a = "opción 1",
        b = "opción 2",
        c = "opción 3",
        stop("Valor de x no válido")
    )
}
```

Estructuras de repetición (bucles)

for

```
for (variable in valores)
{
sentencias
}
```

Ejemplos:

```
for (i in 1:3) print(i^2)
for (i in letters[1:3]) print(i)
for (i in lista) print(i)
```

El bucle se puede interrumpir usando `break`, `next` o `return`.

```
for (i in 1:5) if (i%%2==0) print(i) else next

for (i in 1:100)
{
    suma<-suma+x[i]
    if (suma>0.5) {
        print(paste("Me paro en i=",i,"porque la suma supera 0.5"))
        break
    }
}
```

while

```
while (condición)
{
sentencias
}
```

Se interrumpe igual que el for.

Ejemplos:

```
while (suma<0.5)
{
    i<-i+1
    x[i]<-runif(1,0,0.1) # uniforme en (0,0.1)
    suma<-suma+x[i]
}
```

repeat

Permite repeticiones infnitas, aunque se entiende que se va a salir con break o similares.

Un ejemplo equivalnete al while anterior:

```
repeat
{
    i<-i+1
    x[i]<-runif(1,0,0.1) # uniforme en (0,0.1)
    suma<-suma+x[i]
    if (suma>=0.5) break
}
```

Familia apply

Opern sobre estructuras de datos completas como matrices, dataframes o listas.

lapply y sapply

```
lapply(x, funcion, ...)
```

Devuelve una lista de la misma longitud de x resultado de aplicar funcion a cada componente.

```
sapply(x, funcion, simplify=TRUE, ...)
```

Igual que apply, pero contiene simplify igual a TRUE por defecto, que hace que siempre devuelva la estructura de datos más simple posible (vector o matriz).

apply

Permite realizar operaciones marginales sobre matrices o arrays.

```
apply(x, margin, funcion, ..., simplify=TRUE)
```

x es la matriz o array, funcion es la función que implementa las operaciones a realizar y margin la dimensión donde se opera. Si x es una matriz, margin=1 opera por filas, margin 2 por columnas y margin=c(1,2) por filas y columnas.

split

Clasifica datos consistentes en vectores, matrices o dataframes de acuerdo a un criterio de clasificación.

```
split(x, f, ...)
```

x es el objeto a clasificar y f el criterio.

Por ejemplo:

```
arboles<-split(Orange,Orange$Tree)
# Clasifica según el tipo de árbol
```

Clasificación y agrupación by, aggregate y tapply

`aggregate` permite resumir columnas de un dataframe para cada uno de los niveles de un factor. Devuelve un dataframe (se puede simplificar con simplify).

```
aggregate(Orange$age,by=list(Orange$Tree),summary)
```

`by` es similar, tiene tres argumentos principales: el objeto al que se aplica, factor o vector para clasificar y función que se aplica a cada elemento resultante de clasificar.

```
by(Orange$age,Orange$Tree,summary)
```

`tapply` es similar, pero devuelve una lista.

```
tapply(Orange$age,Orange$Tree,summary)
```

Media y Desviación típica

Media (mean)

Media de un vector con mean. Si el vector tiene valores perdidos se puede decir que se ignoren con:

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

Cuasivarianza (var)

Regresión

Recta de regresión

Para ajustar una recta de regresión se usa `lm(y~x)`.

Batiburrillo

summary

La función summary proporciona información de los datos como la media, mediana, mínimo, máximo y los cuartiles.

Instalar y cargar paquetes

Para instalar se usa `install.packages("maxLik")`.

Para cargar un paquete ya instalado se usa `library(maxLik)`.

Para descargar un paquete se usa `detach(package:maxLik)`.

Paquetes de datos

Podemos ver los disponibles escribiendo `data()`.

Para hacer accesibles las columnas de un dataframe directamenste usamos `attach(cars)` y para eliminarlo usamos `detach(cars)`.

Debugging

stop y warning

Las funciones `stop` y `warning` ayudan con la gstión de errores.

`stop` interrumpe la ejecución del código y devuelve la cadena de caracteres que s le pase como argumento.

`warning` devuelve el mensaje (argumento) pero no interrumpe la ejecución.

```
stop("Debe proporcionar un argumento 'x' numérico")
warning("Se han eliminado los valores negativos o cero")
```

traceback

Permite reconstruir el camino seguido por R hasta que encontró el error.

Se escribe `traceback()` tras encontrase un error.

browser

Se escribe `browser` en mitad de la función y se podrá ir haciendo cosas:

```
f<-function(x)
{
browser()
n<-round(x)
set.seed(n)
u<-runif(n)
return(u)
}
mean(f(-2))
```

debug

`debug` aplicada a una función concreta hace que su evaluación se realice paso a paso.

Dicha función se ejecutará en modo debug hasta que indiquemos que deje de hacerse con `undebug`.

```
f<-function(x)
{
n<-round(x)
set.seed(n)
u<-runif(n)
return(u)
}
debug(f)
mean(f(2))
```

También se puede usar `debugonce`, que solo debuggea la primera vez que se llama a esa función.