Estructuras de datos R

Tipos de datos

Un vector es una secuencia ordenada de datos. R dispone de los siguientes tipos de datos:

- **logical**: (True or False)
- **integer**: $\mathbb{Z}$
- **numeric**: $\mathbb{R}$
- **complex**: $\mathbb{C}$

En los vectores de `R`, todos sus objetos han de ser del mismo tipo: todos números, todos palabras, etc. Cuando queramos usar vectores formados por objetos de diferentes tipos, tendremos que usar **listas generalizadas**, `lists` que veremos al final del tema.

Creacions básicas

- **c()**: crear un vector
- **scan(0)**: definir un vector
- **fix(x)**: modificar visualmente un vector `x`
- **rep(a,n)**: crear un vector que contiene el dato a repetido n veces

Ejemplo

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

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

```
rep("Mates",7)
```

```
[1] "Mates" "Mates" "Mates" "Mates" "Mates" "Mates" "Mates"
```

Pogresiones y secuencias

Una progresión aritmética es una sucesión de números tales que la **diferencia**, d, de cualquier par de términos sucesivos de la secuencia es constante.

$$a_n = a_1 + (n-1) \cdot d$$

- **seq(a,b,by=d)**: para generear una progresión aritmética de diferencia d que empiezaen a y termina en b

```
seq(5,60,by=5)
```

```
 [1]  5 10 15 20 25 30 35 40 45 50 55 60
```

- **seq(a,b,length.out=n)**: define una progresión aritmética de longitud n que va de a a b con difernecia $d = \frac{b-a}{n-1}$. (n es la cantidad de elementos del array)

```
seq(4, 35, length.out=7)
```

```
[1]  4.000000  9.166667 14.333333 19.500000 24.666667 29.833333 35.000000
```

- **seq(a,by=d, length.out=n)**: define la progresión aritmética de longitud n y diferencia d que empieza en a

```
seq(4, by=25, length.out=3)
```

```
[1]  4 29 54
```

- **a:b**: define la secuencia de números **enteros** ($\mathbb{Z}$) consecutivos entre dos números a y b

```
1:9
```

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

Funciones

Cuando queremos aplicar una función a cada uno de los elementos de un vector de datos, la función `sapply` nos ahorra tener que programar con bucles en `R`:

- **sapply(nombre_del_vector, FUN=nombre_de_la_función)**: para aplicar dicha función a dodos los elementos del vector

```
x = 1:10
sapply(x, FUN=function(x){sqrt(x)})
```

```
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

- **sqrt(x)**‘: calcula un nuevo vector con las raíces cuadradas de cada uno de los elementos del vector x

```
x = 1:10
x
```

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

```
x + pi
```

```
 [1]  4.141593  5.141593  6.141593  7.141593  8.141593  9.141593 10.141593
 [8] 11.141593 12.141593 13.141593
```

```
x * pi
```

```
 [1]  3.141593  6.283185  9.424778 12.566371 15.707963 18.849556 21.991149
 [8] 25.132741 28.274334 31.415927
```

```
sqrt(x)
```

```
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

```
2^x
```

```
 [1]    2    4    8   16   32   64  128  256  512 1024
```

```
x^2
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

- **mean(x)**: calcula la media arimtética de las entradas del vector x

```
mean(1:10)
```

```
[1] 5.5
```

- **diff(x)**: cañcula el vector formado por las diferencias sucesivas entre entradas del vector original

```
diff(1:10)
```

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

- **cumsum(x)**: calcula el vector formado por las sumas acumuladas de las entradas del vector original x
  - Permite definir sucesiones descritas mediante sumatorios
  - Cada entrada de *cumsum(x)* es la suma de las entradas de x hasta su posición

```
cumsum(1:10)
```

```
 [1]  1  3  6 10 15 21 28 36 45 55
```

Orden

- **sort(x)**: ordena el vector en orden natural de los objetos que lo forman: el orden numérico creciente, orden alfabético...
- **rev(x)**: invierte el orden de los elementos del vector x

```
v = c(1,7,5,2,4,6,3)
sort(v)
```

```
[1] 1 2 3 4 5 6 7
```

```
rev(v)
```

```
[1] 3 6 4 2 5 7 1
```

Otros

- **length(x)** : longitud del vector
- **max(x)** : elemento máximo del vector
- **min(x)** : mínimo del vector
- **sum(x)** : suma de todos los elementos del vector
- **prod(x)** : producto de todos los elementos del vector

```
x = 1:10
length(x)
```

```
[1] 10
```

```
max(x)
```

```
[1] 10
```

```
min(x)
```

```
[1] 1
```

```
sum(x)
```

```
[1] 55
```

```
prod(x)
```

```
[1] 3628800
```

Subvectores

- **Vector[i]**: da la i-esima entrada del vector
  - Los índices en R empiezan en 1
  - **vector(length(vector))** nos da la última entrada del vector
  - **vecntor[a:b]**: si a y b son naturales, nos da el subvector con las entradas del vector original que van de la posición a-ésima hasta la b-ésima

  - **vector[-i]**: si i es un número, este subvector está formado por todas las entradas del vector original menos la entrada i-ésima. Si i resulta ser un vector, entonces es un vector de índices y crea un nuevo vector con las entradas del vector original,cuyos índices pertenecen a i
  - **vector[-x]**: si x es un vector (de índices), entonces este es el complementario de vector[x]

- También podemos utilizar operadores lógicos:

  - **==**: $=$
  - **!=**: $\neq$
  - **>=**: $\geq$
  - **<=**: $\leq$
  - **<**: $<$
  - **>**: $>$
  - **!**: NO lógico
  - **&**: Y lógico
  - **|**: O lógico

```
v = c(14,5,6,19,32,0,8)
v[2]
```

```
[1] 5
```

```
v[-c(3,5)] # no aparecen los elementos de las posiciones 3 y 5
```

```
[1] 14  5 19  0  8
```

```
v[v != 19 & v > 15] # elementos que no  son 19 y mayores a 15
```

```
[1] 32
```

Condicionales

- **which(x cumple condición)**: para obtener los índices de las entradas del vector x que satisfacen la condición dada
- **which.min(x)**: nos da la primera posición en la que el vector x toma su valor mínimo
- **which(x==min(x))**: da todas las posiciones en las que el vector x toma sus valores mínimos
- **which.max(x)**: nos da la primera posición en la que el vector x toma su valor máximo
- **which(x==max(x))**: da todas las posiciones en las que el vector x toma sus valores máximos

Ejemplos

```
x = seq(3, 50, by=3.5)
x
```

```
 [1]  3.0  6.5 10.0 13.5 17.0 20.5 24.0 27.5 31.0 34.5 38.0 41.5 45.0 48.5
```

```
x[3]
```

```
[1] 10
```

```r
x[length(x)]
```

```
[1] 48.5
```

```r
x[length(x)-2]
```

```
[1] 41.5
```

```r
x[-3] # Se quita la tercer entrada
```

```
 [1]  3.0  6.5 13.5 17.0 20.5 24.0 27.5 31.0 34.5 38.0 41.5 45.0 48.5
```

```r
x[8:4] # da de la posición 8 a la 4
```

```
[1] 27.5 24.0 20.5 17.0 13.5
```

```r
x[seq(2,length(x), by=2)] # los de posición par
```

```
[1]  6.5 13.5 20.5 27.5 34.5 41.5 48.5
```

```r
x[-seq(2,length(x), by=2)] # quitar los de posición par
```

```
[1]  3 10 17 24 31 38 45
```

```r
x[c(1,5,6)] #Da los de posición 1, 5 y 6
```

```
[1]  3.0 17.0 20.5
```

```r
# Da los elementos que son mayores a 5 (x>5 da una lista booleana de los elementos de x que
# son mayores  5)
x[x>5]
```

```
 [1]  6.5 10.0 13.5 17.0 20.5 24.0 27.5 31.0 34.5 38.0 41.5 45.0 48.5
```

```r
x>5
```

```
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE
```

```r
x = c(6,7,4,2,4,8,9,2,9)
y = c(5,2,-3,5,-1,4-2,7,1)
# y%%2==0 da las posiciones de y cuyos valores son pares y tomamos esas posiciones y
x[y%%2==0]
```

```
[1] 7 8
```

```r
which(x>4) # la las posiciones de los elementos de x que son mayores que 4
```

```
[1] 1 2 6 7 9
```

```r
which.max(x) # Da la primera posición del elemento más grande en x
```

```
[1] 7
```

```r
which(x == max(x)) # Da todas las posiciones del elemento máximo de x
```

```
[1] 7 9
```

```r
x[x<0] # es vacío
```

```
numeric(0)
```

```r
which(x<0) #En posiciones
```

```
integer(0)
```

Valores NA

```r
x = seq(1, 10, by=1)
x[11] # no existe elemento en la posición 11
```

```
[1] NA
```

```r
x[11]=11 # se le agrega un elemento en la posición 11
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

```r
x[15]=15 # Se le agrega un elemento en la posición 15
x # se rellena con NA
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 NA NA NA 15
```

```r
mean(x) # no puede operar porque tiene valores NA
```

```
[1] NA
```

```r
mean(x, na.rm = TRUE) # quita loe elementos NA
```

```
[1] 6.75
```

```
which(is.na(x)) # Da las posiciones de los elementos NA
```

```
[1] 12 13 14
```

```
which(x==NA) # Da vacío
```

```
integer(0)
```

```
y=x
y[is.na(y)] = mean(y,na.rm = TRUE) # Cambiamos los NA por la media
y
```

```
 [1]  1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00 10.00 11.00  6.75
[13]  6.75  6.75 15.00
```

```
x[!is.na(x)] # da los elementos que no son NA
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 15
```

```
cumsum(x[!is.na(x)]) #Para funciones que no admiten na.rm
```

```
 [1]  1  3  6 10 15 21 28 36 45 55 66 81
```

```
y=na.omit(x) # Se quitan los NA
cumsum(y)
```

```
 [1]  1  3  6 10 15 21 28 36 45 55 66 81
```

```
y # tiene atributos "na.accion" y "class"
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 15
attr(,"na.action")
[1] 12 13 14
attr(,"class")
[1] "omit"
```

```
attr(y,"na.action")= NULL # se quita el atributo
attr(y,"class")= NULL # se quita el atributo
y
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 15
```

Factor

Factor: es como un vector, pero con una estructura interna más rica que permite usarlo para clasificar observaciones

- **levels**: atributo del factor. Cada elemento del factor es igual a un nivel. Los niveles clasifican las entradas del factor. Se ordenan por orden alfabético
- Para definir un factor, primero hemos de definir un vector y trasformarlo por medio de una de las funciones `factor()` o `as.factor()`.

La función factor()

- `factor(vector,levels=...)`: define un factor a partir del vector y dispone de algunos parámetros que permiten modificar el factor que se crea:
  - `levels`: permite especificar los niveles e incluso añadir niveles que no aparecen en el vector
  - `labels`: permite cambiar los nombres de los niveles
- `levels(factor)`: para obtener los niveles del factor

```
nombres = c("Juan",  "Antonio", "Ricardo", "Juan", "Juan", "María", "María")
nombres
```

```
[1] "Juan"    "Antonio" "Ricardo" "Juan"    "Juan"    "María"   "María"
```

```
nombres.factor = factor(nombres)
nombres.factor
```

```
[1] Juan    Antonio Ricardo Juan    Juan    María   María
Levels: Antonio Juan María Ricardo
```

```
gender = c("M", "H", "H", "M","M","M", "M","H", "H")
gender.factor = factor(gender)
gender.factor
```

```
[1] M H H M M M M H H
Levels: H M
```

```
gender.fact2 = as.factor(gender) # En este caso es lo mismo
gender.fact2
```

```
[1] M H H M M M M H H
Levels: H M
```

```
gender.fact3 = factor(gender, levels = c("M", "H", "B")) # Se pueden asignar más niveles
gender.fact3
```

```
[1] M H H M M M M H H
Levels: M H B
```

```
gender.fact4 = factor(gender, levels =c("M", "H", "B"), labels = c("Mujer", "Hombre", "Hermafrodita"))
gender.fact4
```

```
[1] Mujer  Hombre Hombre Mujer  Mujer  Mujer  Mujer  Hombre Hombre
Levels: Mujer Hombre Hermafrodita
```

```
levels(gender.fact4)
```

```
[1] "Mujer"        "Hombre"       "Hermafrodita"
```

```
levels(gender.fact4) = c("Femenino", "Masculino", "Híbrido")
gender.fact4
```

```
[1] Femenino  Masculino Masculino Femenino  Femenino  Femenino  Femenino
[8] Masculino Masculino
Levels: Femenino Masculino Híbrido
```

```
notas = c(1,2,3,4,1,3,4,1,3,1,3,2,4,1,3,2,4,2,3)
notas.factor = factor(notas)
notas.factor
```

```
 [1] 1 2 3 4 1 3 4 1 3 1 3 2 4 1 3 2 4 2 3
Levels: 1 2 3 4
```

```
levels(notas.factor)
```

```
[1] "1" "2" "3" "4"
```

```
levels(notas.factor) = c("Suspendido", "suficiente", "Notable", "Excelente")
notas.factor
```

```
 [1] Suspendido suficiente Notable    Excelente  Suspendido Notable
 [7] Excelente  Suspendido Notable    Suspendido Notable    suficiente
[13] Excelente  Suspendido Notable    suficiente Excelente  suficiente
[19] Notable
Levels: Suspendido suficiente Notable Excelente
```

Factor ordenado

Factor ordenado. Es un factor donde los niveles siguen un orden

- **ordered(vector,levels=...)***: función que define un factor ordenado y tiene los mismos parámetros que factor

```
notas
```

```
 [1] 1 2 3 4 1 3 4 1 3 1 3 2 4 1 3 2 4 2 3
```

```
ordered(notas, levels = c("Suspenso", "Suficiente", "Notable", "Excelente"))
```

```
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[16] <NA> <NA> <NA> <NA>
Levels: Suspenso < Suficiente < Notable < Excelente
```

```
ordered(notas, labels = c("Suspenso", "Suficiente", "Notable", "Excelente"))
```

```
 [1] Suspenso   Suficiente Notable    Excelente  Suspenso   Notable
 [7] Excelente  Suspenso   Notable    Suspenso   Notable    Suficiente
[13] Excelente  Suspenso   Notable    Suficiente Excelente  Suficiente
[19] Notable
Levels: Suspenso < Suficiente < Notable < Excelente
```

List

List. Lista formada por diferentes objetos, no necesariamente del mismo tipo, cada cual con un nombre interno

- `list(...)`: función que crea una list
  - Para obtener una componente concreta usamos la instrucción `list$componente`
  - También podemos indicar el objeto por su posición usando dobles corchetes: `list[[i]]`. Lo que obtendremos es una list formada por esa única componente, no el objeto que forma la componente

```
x = c(2,5,3,-3,9,7,4,-6,2,-9,2,-56)
x
```

```
 [1]   2   5   3  -3   9   7   4  -6   2  -9   2 -56
```

```
L = list(nombre = "temperaturas", datos = x, media = mean(x), sumas = cumsum(x))
L
```

```
$nombre
[1] "temperaturas"

$datos
 [1]   2   5   3  -3   9   7   4  -6   2  -9   2 -56

$media
[1] -3.333333

$sumas
 [1]   2   7  10   7  16  23  27  21  23  14  16 -40
```

```
L$media #Acceder a los datos
```

```
[1] -3.333333
```

```
L$datos
```

```
 [1]   2   5   3  -3   9   7   4  -6   2  -9   2 -56
```

```
L[[2]] #Acceder a los componentes del vector
```

```
 [1]   2   5   3  -3   9   7   4  -6   2  -9   2 -56
```

```
L[2] #Esto se accede como lista mas no como vector
```

```
$datos
 [1]   2   5   3  -3   9   7   4  -6   2  -9   2 -56
```

Obtener información de una list

- `str(list)`: para conocer la estructura interna de una list
- `names(list)`: para saber los nombres de la list

```
names(L)
```

```
[1] "nombre" "datos"  "media"  "sumas"
```

```
str(L)
```

```
List of 4
 $ nombre: chr "temperaturas"
 $ datos : num [1:12] 2 5 3 -3 9 7 4 -6 2 -9 ...
 $ media : num -3.33
 $ sumas : num [1:12] 2 7 10 7 16 23 27 21 23 14 ...
```

Obtener información de una list

```
x = c(1,-2,3,4,-5,6,7,-8,-9,0)
miLista = list(nombre = "X", vector = x, media = mean(x), sumas = cumsum(x))
miLista
```

```
$nombre
[1] "X"

$vector
 [1]  1 -2  3  4 -5  6  7 -8 -9  0

$media
[1] -0.3

$sumas
 [1]  1 -1  2  6  1  7 14  6 -3 -3
```

Obtener información de una list

```
str(miLista)
```

```
List of 4
 $ nombre: chr "X"
 $ vector: num [1:10] 1 -2 3 4 -5 6 7 -8 -9 0
 $ media : num -0.3
 $ sumas : num [1:10] 1 -1 2 6 1 7 14 6 -3 -3
```

```
names(miLista)
```

```
[1] "nombre" "vector" "media"  "sumas"
```

Matrices

Cómo definirlas

- `matrix(vector, nrow=n, byrow=valor_lógico)`: para definir una matriz de n filas formada por las entradas del vector
  - `nrow`: número de filas
  - `byrow`: si se iguala a TRUE, la matriz se construye por filas; si se iguala a FALSE (valor por defecto), se construye por columnas. `-ncol`: número de columnas (puede usarse en lugar de nrow)
  - R muestra las matrices indicando como $[i,]$ la fila i-ésima y $[,j]$ la columna j-ésima
  - Todas las entradas de una matriz han de ser del mismo tipo de datos

Cómo definirlas

Ejercicio

- ¿Cómo definirías una matriz constante? Es decir, ¿cómo definirías una matriz A tal que $\forall\ i = 1, ..., n; j = 1, ..., m,\ a_{i,j} = k$ siendo $k \in \mathbb{R}$? Como R no admite incógnitas, prueba para el caso específico $n = 3, m = 5, k = 0$
- Con el vector vec = (1,2,3,4,5,6,7,8,9,10,11,12) crea la matriz

$$\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

Cómo construirlas

- `rbind(vector1, vector2, ...)`: construye la matriz de filas vector1, vector2,...
- `cbind(vector1, vector2, ...)`: construye la matriz de columnas vector1, vector2,...
  - Los vectores han de tener la misma longitud
  - También sirve para añadir columnas (filas) a una matriz o concatenar por columnas (filas) matrices con el mismo número de filas (columnas)
- `diag(vector)`: para construir una matriz diagonal con un vector dado
  - Si aplicamos diag a un número n, produce una matriz identidad de orden n

Submatrices

- `matriz[i,j]`: indica la entrada (i, j) de la matriz, siendo $i, j \in \mathbb{N}$. Si i y j son vectores de índices, estaremos definiendo la submatriz con las filas pertenecientes al vector i y columnas pertenecientes al vector j
- `matriz[i,]`: indica la fila i-ésima de la matriz, siendo $i \in \mathbb{N}$
- `matriz[,j]`: indica la columna j-ésima de la siendo $j \in \mathbb{N}$
  - Si i (j) es un vector de índices, estaremos definiendo la submatriz con las filas (columnas) pertenecientes al vector i (j)

Funciones

- `diag(matriz)`: para obtener la diagonal de la matriz
- `nrow(matriz)`: nos devuelve el número de filas de la matriz
- `ncol(matriz)`: nos devuelve el número de columnas de la matriz
- `dim(matriz)`: nos devuelve las dimensiones de la matriz
- `sum(matriz)`: obtenemos la suma de todas las entradas de la matriz
- `prod(matriz)`: obtenemos el producto de todas las entradas de la matriz
- `mean(matriz)`: obtenemos la media aritmética de todas las entradas de la matriz

Funciones

- `colSums(matriz)`: obtenemos las sumas por columnas de la matriz
- `rowSums(matriz)`: obtenemos las sumas por filas de la matriz
- `colMeans(matriz)`: obtenemos las medias aritméticas por columnas de la matriz
- `rowMeans(matriz)`: obtenemos las medias aritméticas por filas de la matriz

Funciones

Ejemplo

Dada la matriz

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

```
A = matrix(c(1,2,3,4,5,6,7,8,9), ncol = 3)
dim(A)
```

```
[1] 3 3
```

```
diag(A)
```

```
[1] 1 5 9
```

Función apply()

- `apply(matriz, MARGIN=..., FUN=función)`: para aplicar otras funciones a las filas o las columnas de una matriz
  - `MARGIN`: ha de ser 1 si queremos aplicar la función por filas; 2 si queremos aplicarla por columnas; o c(1,2) si la queremos aplicar a cada entrada

Función apply()

```
#apply(A, MARGIN = c(1,2), FUN = cuadrado)
apply(A, MARGIN = 1, FUN = sum)
```

```
[1] 12 15 18
```

```r
apply(A, MARGIN = 2, FUN = sum)
```

```
[1]  6 15 24
```

Operaciones

- `t(matriz)`: para obtener la transpuesta de la matriz
- `+`: para sumar matrices
- `*`: para el producto de un escalar por una matriz
- `%*%`: para multiplicar matrices
- `mtx.exp(matriz,n)`: para elevar la matriz a n
  - Del paquete `Biodem`
    - No calcula las potencias exactas, las aproxima
- `%^%`: para elevar matrices
  - Del paquete `expm`
    - No calcula las potencias exactas, las aproxima

Operaciones

Ejercicio

Observad qué ocurre si, siendo $A = \begin{pmatrix} 2 & 0 & 2 \\ 1 & 2 & 3 \\ 0 & 1 & 3 \end{pmatrix}$ y $B = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$, realizamos las operaciones $A * B$, A^2 y B^3