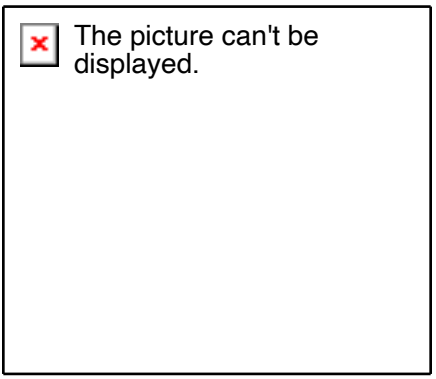



# DATA TYPES EN R

Dept. Ciencias de la Computación e Inteligencia Artificial,
Universidad de Granada

# Indice

- Data types and Structures
  - Vectors
  - Missing and special values
  - Matrices and Arrays
  - Factors
  - Lists
  - Data frames
  - Indexing
  - Conditional indexing

# Espacio de trabajo en R (workspace)

- Los objetos que creas durante una sesión de R se guardan en el workspace
- Este espacio de trabajo no se guarda en el disco a menos que se indique a R que lo haga. *En las diapositivas anteriores te indicabamos como cambiar esto usando las preferencias de RStudio*
- Los objetos se pierden cuando se cierra R
- Los archivos del espacio de trabajo se guardan con la extensión '.Rdata'

# R interactivo

- R evaluates expressions.
- Allowing its use as a calculator.

Tip:

Predefined symbols:

`pi`, `letters`, `month.name`

Special symbols:

`NA`, `NaN`, `Inf`, `NULL`, `TRUE`, `FALSE`

```
> -27*12/21
[1] -15.42857

> sqrt(10)
[1] 3.162278

> log(10)
[1] 2.302585

> log10(2+3*pi)
[1] 1.057848

> exp(2.7689)
[1] 15.94109

> (25 - 5)^3
[1] 8000

> cos(pi)
[1] -1
```

# “atomic” classes of objects in R

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

# Asignación de valores en R

En R los valores se asignan utilizando el símbolo **<-**

```
msg <- "hello”
y <- sin(pi/6)

The # character indicates a comment.
Anything to the right of the # (including the # itself) is
ignored.
```

# Eliminación de variables

- rm()

```
> x <- 2*pi
> x
[1] 6.283185
> rm(x)
> x
Error: object "x" not found
```

**Si quieres borrar todo lo que hay en el espacio de trabajo** con un solo comando ... La función rm() tiene un argumento “list”. A este argumento se le puede dar un vector conteniendo los nombres de todas las variables prsentes en el directorio de trabajo. Mira como funciona:

```
> ls()
[1] "f" "x" "y" "z”

> rm(list=ls())
> ls()
character(0)
```

# Strings/Cadenas

```
> "Hello"
[1] "Hello"
> x <- paste("Hello", "World")
> x
[1] "Hello World”
```

Es posible hacer casi cualquier procesamiento de cadenas con **R**…
PERO existen otros lenguajes de programación mucho mas mejores para ello.

*Por defecto paste() pone un espacio entre los diferentes elementos que hay que unir*

Task:
Crea dos variables, a una asignale tu nombre y a la otra tu apellido.
Crea una tercera variable con ambos valores.

# Números

- Los números en R a generalmente se tratan como objetos numéricos (es decir, números reales de doble precisión)

- También hay un número especial `Inf` que representa el infinito;

```
1 / 0; Inf can be used in ordinary calculations;
1 / Inf is 0
```

- El valor `NaN` representa un valor indefinido (\not a number");

```
0 / 0; NaN
can also be thought of as a missing value
```

# NA

- Que los sets de datos con los que trabajamos esten incompletos y les falten valores ”Missing values” es algo común
- En R estos valores se marcan como NA o NaN en el caso de operaciones matemáticas no definidas.
- `>x<-c(4,2,NA,10,8)`

```
> 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
```

# ¿Como afecta NA a las funciones?

- Las funciones estadísticas de R pueden indicar a la función que omita cualquier valor faltante, or NAs:

```
> x <- c(88,NA,12,168,13)
> x
[1] 88 NA 12 168 13
> mean(x)
[1] NA
> mean(x, na.rm=T)
[1] 70.25
> x <- c(88,NULL,12,168,13)
> mean(x)
[1] 70.25
```

- La primera llamada mean()  no funciona ya que un valor de x es NA.
- **PERO usando el argumento na.rm** (NA remove ) con valor true (T ), es posible hacer funcionar la funcion calculando la media para el resto de los elementos.

# Funciones numéricas

| Function | Description |
|---|---|
| **abs(*x*)** | absolute value |
| **sqrt(*x*)** | square root |
| **ceiling(*x*)** | ceiling(3.475) is 4 |
| **floor(*x*)** | floor(3.475) is 3 |
| **trunc(*x*)** | trunc(5.99) is 5 |
| **round(*x*, digits=*n*)** | round(3.475, digits=2) is 3.48 |
| **signif(*x*, digits=*n*)** | signif(3.475, digits=2) is 3.5 |
| **cos(*x*), sin(*x*), tan(*x*)** | also acos(*x*), cosh(*x*), acosh(*x*), etc. |
| **log(*x*)** | natural logarithm |
| **log10(*x*)** | common logarithm |
| **exp(*x*)** | e^*x* |

# Operadores aritméticos

| Operator | Description |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ^ or ** | exponentiation |
| x %% y | modulus (x mod y) 5%%2 is 1 |
| x %/% y | integer division 5%/%2 is 2 |

# Ejemplos

```
# An addition
>3 + 4

# A multiplication
>3*5

# A division
>(5+5)/2

# Exponentiation
>2^5

# Modulo
>28%%6
```

# Operadores Lógicos

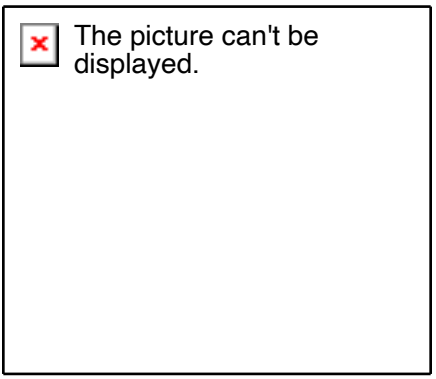
| Operator | Description |
|---|---|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |
| !x | Not x |
| x \| y | x OR y |
| x & y | x AND y |
| isTRUE(x) | test if x is TRUE |

# Ejemplos

```
>x <-1:3;
>y<-1:3
>x == y
[1] TRUE   TRUE    TRUE
```

Para comparar en su totalidad dos objetos tenemos las funcion identical() y all.equal().

```
>x <-1:3
>y<-1:3
>identical(x,y)
[1] TRUE
>all.equal(x,y)
[1] TRUE
```




# DATA TYPES:

- Vectors
- Matrices
- Factors
- Lists
- Dataframes
- Names

# Vectors

- La función  c() se usa para crear vectores de objetos.

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

- También se pueden crear vectores de forma automática con la función vector()

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

# Indexing

Mediante el indexado podemos extraer elementos de los vectores

```
# Indexing a vector
> Num <- c(0,3,2,2,1)
> Num[1]
[1] 0
> Num[2:3]
[1] 3 2
> Num[c(1,3)]
[1] 0 2
```

# Manipulación de vectores

- Our vector: `x=c(100,101,102,103)`
- `[]` are used to access elements in `x`
- Extract 2nd element in `x`

```
> x[2]
[1] 101
```

- Extract 3rd and 4th elements in `x`

```
> x[3:4] # or x[c(3,4)]
[1] 102 103
```

# Manipulación de vectores

```
> x
[1] 100 101 102 103
```

- Add 1 to all elements in **x**:

```
> x+1
[1] 101 102 103 104
```

- Multiply all elements in **x** by 2:

```
> x*2
[1] 200 202 204 206
```

# Comparación de vectores

| Operator | Result |
|---|---|
| x == y | Returns TRUE if x exactly equals y |
| x != y | Returns TRUE if x differs from y |
| x > y | Returns TRUE if x is larger than y |
| x >= y | Returns TRUE if x is larger than or exactly equal to y |
| x < y | Returns TRUE if x is smaller than y |
| x <= y | Returns TRUE if x is smaller than or exactly equal to y |
| x & y | Returns the result of x and y |
| x \| y | Returns the result of x or y |
| ! x | Returns not x |
| xor( x,y) | Returns the result of x xor y (x or y but not x and y) |

# “Recycling Rule” en R

- Las operaciones vectoriales aritmética R las realiza elemento por elemento. Esto funciona bien cuando ambos vectores son de la misma longitud.
- Pero cuando los vectores no tienen la misma longitud invoca la *“Recycling Rule”.*
- Cuando el vector más corto se agota, mientras que el vector más largo todavía tiene elementos sin procesar, R vuelve al principio del vector más corto, "reciclando" sus elementos.

```
> x=c(2:5)
> y=3
> x
[1] 2 3 4 5
> y
[1] 3 8
> x>y
[1] FALSE FALSE  TRUE FALSE
```

# ¿se puede mezclar?

- ¿Que pasa cuando?

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

- Cuando se mezclan diferentes objetos en un vector, la *coercion* se produce de modo que cada elemento en el vector es de la misma clase

# Mexclando objetos de forma dirigida

- Los objetos pueden convertirse en objetos de otro tipo utilizando las funciones de tipo: as.* functions

```
> 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"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
10
```

# Coercing types

| Tipo | Comprobación | Coerción |
|---|---|---|
| array | is.array() | as.array() |
| character | is.character() | as.character() |
| complex | is.complex() | as.complex() |
| double | is.double() | as.double() |
| factor | is.factor() | as.factor() |
| integer | is.integer() | as.integer() |
| list | is.list() | as.list() |
| logical | is.logical() | as.logical() |
| matrix | is.matrix() | as.matrix() |
| NA | is.na() | - |
| NaN | is.nan() | - |
| NULL | is.null() | as.null() |
| numeric | is.numeric() | as.numeric() |
| ts | is.ts() | as.ts() |
| vector | is.vector() | as.vector() |

*Tabla de David Conesa, GE**ei** **tE**ma (UV)*

# Genera vectores con seq()

- La funcion seq()  (or sequence ) genera una secuencia que sigue una progresión aritmética.

```
> seq(from=12,to=30,by=3)
[1] 12 15 18 21 24 27 30
```

The spacing can be a non-integer value, too, say 0.1.

```
> seq(from=1.1,to=2,length=10)
[1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

# Crea un vector con valores repetidos rep()

- La función rep() (or repeat ) function nos permite poner la misma constante en vectores largos. La forma de llamada es `rep (x, times)`. Aquí hay un ejemplo

```
> x <- rep(8,4)
> x
[1] 8 8 8 8
> rep(c(5,12,13),3)
[1] 5 12 13 5 12 13 5 12 13
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

- La función rep() tiene un argumento ”each”, que modifica el comportamiento de la función, por ejemplo `rep(c(5,12,13),each=2)`repite x cada 2 números

```
> rep(c(5,12,13),each=2)
[1] 5 5 12 12 13 13
```

# all() y any()

- Las funciones any () y all () son atajos útiles. Informan si alguno o todos sus argumentos son TRUE.

```
> x <- 1:10
> any(x > 8)
[1] TRUE
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
```

# Manipulación de vectores

Our vector: `x=100:150`

Elements of **x** higher than 145

```
> x[x>145]
[1] 146 147 148 149 150
```

Elements of **x** higher than 135 and lower than 140

```
> x[ x>135 & x<140 ]
[1] 136 137 138 139
```

# Vector In, Vector Out

- You saw examples of vectorized functions earlier with the + and * operators. Another example is > .

```
> u <- c(5,2,8)
> v <- c(1,3,9)
> u > v
[1] TRUE FALSE FALSE
```

# Funciones para operaciones aritméticas de vectores

| Function | What it does |
|---|---|
| **sum(x)** | Calculates the sum of all values in x |
| **prod(x)** | Calculates the product of all values in x |
| **min(x)** | Gives the minimum of all values in x |
| **max(x)** | Gives the maximum of all values in x |
| **cumsum(x)** | Gives the cumulative sum of all values in x |
| **cumprod(x)** | Gives the cumulative product of all values in x |
| **cummin(x)** | Gives the minimum for all values in x from the start of the vector until the position of that value |
| **cummax(x)** | Gives the maximum for all values in x from the start of the vector until the position of that value |
| **diff(x)** | Gives for every value the difference between that value and the next value in the vector |

# Ejercicio con vectores

- Write a R program to create a vector and find the length and the dimension of the vector.

- `print("Original vectors:")`

# Ejercicio con vectores

- Write a R program to create a vector and find the length and the dimension of the vector.

```
v = c(1,3,5,7,9)
print("Original vectors:")
print(v)
print("Dimension of the vector:")
print(dim(v))
print("length of the vector:")
print(length(v))
```

```
v = c(1,3,5,7,9)
dim(v)
length(v)
```

# Ejercicio con vectores

- Write a R program to add 3 to each element in a given vector. Print the original and new vector.

```
v = c(1, 2, NULL, 3, 4, NULL)
print("Original vector:")
print(v)
new_v = (v+3)
print("New vector:")
print(new_v)
```

# Ejercicios vectores

Considera el vector x.

```
x <- 1:10
```

¿que hace cada uno de los siguientes commandos?

- `x[3]`
- `x[c(2, 4)]`
- `x[-1]`
- `x[c(2, -4)]`
- `x[c(2.4, 3.54)]`

# Matrices

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

- Todas las columnas de una matriz deben tener el mismo modo (numérico, carácter, etc.) y la misma longitud.

- El formato general es:

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

# Matrices

- Matrices are constructed column-wise, so entries can be thought of starting in the “upper left" corner and running down the columns.
- **byrow=TRUE** indicates that the matrix should be filled by rows.
- **byrow=FALSE** indicates that the matrix should be filled by columns (the default).
- **dimnames** provides optional labels for the columns and rows.

```
> a<-matrix(1:12,nrow=3,byrow=TRUE)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> a<-matrix(1:12,nrow=3,byrow=FALSE)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> rownames(a)<-c("A","B","C")
> a
  [,1] [,2] [,3] [,4]
A    1    4    7   10
B    2    5    8   11
C    3    6    9   12

> colnames(a)<-c("1","2","x","y")
> a
  1 2 x  y
A 1 4 7 10
B 2 5 8 11
C 3 6 9 12
```

# Matrices: propiedades

- To look at the structure of an object using the **`str()`** function. It looks similar to the output for a vector, with the difference that R gives the indices for the rows and for the columns.
- To look at the number of rows and columns without looking at the structure, use the **`dim()`**or the **`attributes()`** functions.
- To find the total number of values in a matrix use the **`length()`** function.

```
➢ m<-matrix(1:12, ncol=4, byrow=TRUE)

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> str(m)
 int [1:3, 1:4] 1 5 9 2 6 10 3 7 11 4
...

 > dim(m)
 [1] 3 4

 > attributes(m)
 $dim
 [1] 3 4
 > length(m)
 [1] 12
```

# Matrices: Combinando vectores en una matriz

- Matrices can be created by column-binding or row-binding with **`cbind()`** and **`rbind()`**.The rows take the names of the original vectors.

```
> objects <- 1:3
> counts <- 10:12

> cbind(objects, counts)
 objects counts
[1,]       1     10
[2,]       2     11
[3,]       3     12

>rbind(objects, counts)
         [,1] [,2] [,3]
objects    1    2    3
counts    10   11   12
```

# Indexado de Matrices

Indexing allows to directly extract elements

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> # Indexing a matrix
> a[1,1]
[1] 1

# First row
>a[1,]
 1  2  3  4
 # First column
>a[,1]
 1  5 9
```

# Indexado de Matrices

Extract the values of the last two columns for the first two rows

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> a[1:2,3:4]
     [,1] [,2]
[1,]    3    4
[2,]    7    8
```

R returns you a matrix again. **Pay attention:** the indices of this new matrix are not the indices of the original matrix anymore.

# Eliminando valores

Get all the values except the second row and the third column

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> a[-2,-3]
     [,1] [,2] [,3]
[1,]    1    2    4
[2,]    9   10   12
```

R returns you a matrix again. **Pay attention:** the indices of this new matrix are not the indices of the original matrix anymore.

# R and dimensions

Get the the second row

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> a[-c(1, 3), ]
```

```
> a[-c(1, 3), ]

[1] 5 6 7 8
```

By default, R always tries to simplify the objects to the smallest number of dimensions possible!!!. So, if you extract from a matrix only one column or row, R will make a vector

# R and dimensions

To force R to keep all dimensions use the extra argument `drop` from the indexing function.

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> a[2, , drop=FALSE]

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

First position: row index
Second position: column index
Third position: argument drop

# Sustituir valores en una Matriz

- Cambia un valor
- Cambia una fila o columna de valores completa sin especificar la dimensión. Presta atención al reciclado de valores
- Substituye un set de valores

```
#Change a subset
> a[1:2, 3:4] <- c(8,4,2,1)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    8    2
[2,]    1    3    4    1
[3,]    9    4   11   12
```

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

# Change a value
> a[3, 2] <- 4
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9    4   11   12

# Change a row
> a[2, ] <- c(1,3)
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    3    1    3
[3,]    9    4   11   12
```

# La traspuesta de una Matriz

```
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> t(a)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

# Funciones para trabajar con matrices

| Función | Utilidad |
|---|---|
| **ncol(x)** | Número de columnas de x. |
| **nrow(x)** | Número de filas de x. |
| **t(x)** | Transpuesta de x |
| **cbind(…)** | Combina secuencias de vectores/matrices por col's. |
| **rbind(…)** | Combina secuencias de vectores/matrices por filas. |
| **diag(x)** | Extrae diagonal de matriz o crea matriz diagonal. |
| **col(x)** | Crea una matriz con elemento ij igual al valor j |
| **row(x)** | Crea una matriz con elemento ij igual al valor i |
| **apply(x,margin,FUN,)** | Aplica la función FUN a la dimensión especificada en margin 1 indica filas, 2 indica columnas. NB. |
| **outer(x,y,fun="*") otra forma x %o %y** | Para dos vectores x e y , crea una matriz A[i,j]=FUN(x[i],y[j] Por defecto crea el producto externo. |

*Tabla de David Conesa, GE**ei** **tE**ma (UV)*

# Ejercicios con Matrices

- R Notebook

# Arrays en R

- Las matrices son los objetos de datos R que pueden almacenar datos en más de dos dimensiones.
- Un array se crea utilizando la función array (). Toma vectores como entrada y usa los valores en el parámetro dim para crear una matriz

```
> my.array <- array(1:24, dim=c(3,4,2))
> my.array
, , 1
  [,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12
, , 2
  [,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24
```

# Pon nombres a filas y columnas en un Array

Usa el parámetro **dimnames**.

```
# Create two vectors of different lengths.

vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(
            c(vector1,vector2),
            dim = c(3,3,2),
            dimnames = list(row.names, column.names, matrix.names)
          )
```

# Assesing Array elements

```
# Print the third row of the second matrix of the array.
 print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st
matrix.

 print(result[1,3,1])

 # Print the 2nd Matrix.

print(result[,,2])
```

# Gracias…