



École d'Ingénierie Digitale  
et d'Intelligence Artificielle

# R Programming

Manipulating **data** using R



الجامعة الأوروبية بـفاس  
EUROMED UNIVERSITY OF FES  
UNIVERSITÉ EUROMED DE FÈS

# Objects

---

R manipulates objects. So, when we import a file into R, we get in R an object named data frame. Variables, data, functions, analysis results are stored in objects. There are several types of objects: vectors, factors, etc. The main objects are presented in this part.

The objects are characterized by their name, their content and attributes that will specify the type of data represented by the object.

The name of an object must start with a letter and can include letters, numbers, periods (.), and underscores (\_). R distinguishes, for the names of the objects, the upper case letters, that is to say that x and X will name distinct objects.

Objects all have at least two attributes: mode and length. The mode is the type of the elements of an object. There are four main ones: numeric, character, complex, and logical (FALSE or TRUE). To find out the mode or the length of an object, we use the mode () and length () functions respectively.

# Objects

## Basic objects

---

The most basic object is a constant, which can be numeric, complex, character, or logical.

We directly assign a value to an object. The object does not need to be declared.

For example, we enter on the console:

```
n = 8
```

We then type `n` to display its value. We obtain the following result:

```
[1] 8
```

*The symbol [1] indicates that the display starts at the first element of `n`.*

# Objects

## Basic objects

---

We could also have assigned a value to object n using the <- sign (minus sign attached to a square bracket) or ->:

```
n <- 8
```

```
n
```

```
[1] 8
```

```
8 -> n
```

```
n
```

```
[1] 8
```

# Objects

## Basic objects



---

*Other examples of basic objects:*

```
x=1
```

```
x  
[1] 1
```

*If you assign a value to an existing object, its previous value is deleted:*

```
x = 10
```

```
x  
[1] 10
```

```
y = 10 + 2
```

```
y  
[1] 12
```

# Objects

## Basic objects

---

We use ";" to separate different commands on the same line:

```
w = 8; name = "Wikistat"; dicton = "Aide-toi, autrui t'aidera";
```

```
w ; name ; dicton
```

```
[1] 8
```

```
[1] "Wikistat"
```

```
[1] "Aide-toi, autrui t'aidera"
```

# Other objects

## Objects and their attributes

*The following table indicates the possible modes for the vector, factor, array, matrix, data.frame, ts and list objects.*

Objet	Modes	Plusieurs modes possibles dans le même objet
vecteur	num, car, comp, log	Non
facteur	num, car	Non
array	num, car, comp, log	Non
matrice	num, car, comp, log	Non
data.frame	num, car, comp, log	Oui
ts	num, car, comp, log	Oui
liste	num, car, comp, log, fonction, expression	Oui

*num = numérique, car = caractère, comp = complexe, log = logique*

# Objects

# Vectors

---

*The vector () function has two arguments: the mode of the elements that make up the vector and the length of the vector.*

- `a = vector("numeric", 5)`
- `b = vector("character", 5)`
- `c = vector("logical", 5)`
- `d = vector("complex", 5)`

`a = numeric(5)`

`b = character(5)`

`c = logical(5)`

`d = complex(5)`



# Objects Vectors

---

*If you type a; b; c; d on the console, the following result is displayed:*

```
[1] 0 0 0 0 0  
[1] "" "" "" "" ""  
[1] FALSE FALSE FALSE FALSE FALSE  
[1] 0+0i 0+0i 0+0i 0+0i
```

We can also construct a vector using the function `c ()`

```
vector = c(5, 7.2, 3.6, 4.9); vector  
[1] 5.0 7.2 3.6 4.9
```

# Objects

## Factors

---

*The factor () function creates nominal categorical variables.*

```
factor( x, levels = sort(unique(x), na.last = TRUE), labels = levels, exclude = NA, ordered =  
is.ordered(x))
```

- **levels**: specifies what are the possible levels of the factor (by default the unique values of the vector x), i.e. the values that the elements of the factor can take.
- **labels**: defines the names of the levels
- **exclude**: the values of x not to be included in the levels
- **ordered**: logical argument specifying whether the levels of the factor are ordered.

# Objects

## Factors

```
factor(1:3)
```

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

---

```
Levels: 1 2 3
```

```
factor(1:3, levels = 1:5)
```

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

```
Levels: 1 2 3 4 5
```

```
factor(1:3, levels = 1:5, labels = c("A", "B", "C", "D", "E"))
```

```
[1] A B C
```

```
Levels: A B C D E
```

```
factor(c(5,7,8,9), levels=5 :8)
```

```
[1] 5 7 8 <NA>
```

```
Levels : 5 6 7 8
```

**9 is not part of the levels (according to the definition set). The number 9 is therefore coded as a missing value.**

# Objects

# Matrices

---

***A matrix is a vector that has an additional argument that defines the dimensions of the matrix. All the elements of a matrix must be the same mode.***

`matrix( data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)`

**byrow**: indicates whether the values given by data must successively fill the columns (FALSE, by default) or the rows (if TRUE).

**dimnames**: allows you to give names to rows and columns. You can also give names to the columns or rows of the matrix using the **rownames ()** and **colnames ()** functions.

# Objects Matrices

```
matrix(0, 5, 7)
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]
[1,]	0	0	0	0	0	0	0
[2,]	0	0	0	0	0	0	0
[3,]	0	0	0	0	0	0	0
[4,]	0	0	0	0	0	0	0
[5,]	0	0	0	0	0	0	0

```
x = 1:20
```

```
> x
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12  
[13] 13 14 15 16 17 18 19 20
```

```
mat1 = matrix(x, 4, 5)
```

```
> mat1
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	5	9	13	17
[2,]	2	6	10	14	18
[3,]	3	7	11	15	19
[4,]	4	8	12	16	20

```
mat2 = matrix(x, 4, 5, byrow = TRUE)
```

```
> mat2
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	5
[2,]	6	7	8	9	10
[3,]	11	12	13	14	15
[4,]	16	17	18	19	20

# Objects

# Matrices

The `paste ()` function is useful in a situation where you want to name the rows and/or columns of a matrix. Indeed, the `paste ()` function allows you to concatenate objects.

```
nom_var = paste("V", 1:5, sep = "")
```

```
> nom_var
```

```
[1] "V1" "V2" "V3" "V4" "V5"
```

```
nom_ind = paste("I", 1:4, sep = "")
```

```
> nom_ind
```

```
[1] "I1" "I2" "I3" "I4"
```


```
colnames(mat2) = nom_var
```

```
rownames(mat2) = nom_ind
```

```
(or dimnames(mat2) = list(nom_ind, nom_var))
```

# Objects

# Matrices



---

```
mat2  
> mat2
```

	V1	V2	V3	V4	V5
I1	1	2	3	4	5
I2	6	7	8	9	10
I3	11	12	13	14	15
I4	16	17	18	19	20

# Objects

## Data frames

---

An array of data is implicitly created by the **read.table** function. You can also create a data table with the **data.frame** function.

The elements of a data.frame do not have to have the same mode. Character mode items are considered factors.

All elements of the data.frame must be the same length. Otherwise, the shorter item is "recycled" a whole number of times.

To create a data.frame from a numeric vector and a character vector, we proceed as follows:



# Objects

## Data frames

To create a data.frame from a numeric vector and a character vector, we proceed as follows:

```
a = c(1, 2, 3) ; a ; mode(a);
```

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

```
[1] "numeric"
```

```
b = c("a", "b", "c") ; b ; mode(b)
```

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

```
[1] "character"
```

```
df = data.frame(a, b)
```

```
> df
```

	a	b
1	1	a
2	2	b
3	3	c

# Objects

## Data frames

---

To create with a single instruction a data.frame with a numeric variable and a character variable:

```
df2 = data.frame(a = 1:6, b = letters[1:6]);
```

```
> df2
```

	a	b
1	1	a
2	2	b
3	3	c
4	4	d
5	5	e
6	6	f

# Objects

## Data frames

If we juxtapose a vector of length 6 and a vector of length 3, the second vector is duplicated:

```
a = c(1, 2, 3, 4, 5, 6)
```

```
b = c("a", "b", "c")
```

```
df = data.frame(a, b)
```

```
>df
```

	a	b
1	1	a
2	2	b
3	3	c
4	4	d
5	5	e
6	6	f

# Objects

## Data frames

---

Therefore, the length of one of the vectors must be a multiple of the length of the other vector:

```
a = c(1, 2, 3, 4, 5)
```

```
b = c("a", "b", "c")
```

```
df = data.frame(a, b)
```

```
Error in data.frame(a, b) : arguments imply differing number of rows: 5, 3
```

# Objects

## Lists

---

A list is created in the same way as a data.frame. Like the data.frame, the elements that compose it are not necessarily of the same mode. They are not necessarily the same length, as the following example illustrates:

```
a = c(1, 2, 3, 4, 5)
```

```
b = c("a", "b", "c")
```

```
liste1 = list(a, b)
```

```
> liste1
```

```
[[1]]
```

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

```
[[2]]
```

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

# Objects

## Lists

We can name the elements of a list as follows:

```
names(liste1) = c("L1", "L2") ;liste1
```

---

```
$L1
```

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

```
$L2
```

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

```
liste2 = list(L1 = a, L2 = b)
```

```
liste2
```

```
$L1
```

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

```
$L2
```

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

# Conversion

## Modes conversion

In many practical situations, it is useful to convert the mode from one object to another. Such a conversion will be possible thanks to a function of the form: `as.mode` (`as.numeric`, `as.logical`, `as.character`, ...).

Convert into	Fonction	Rule
numérique	<code>as.numeric</code>	<code>FALSE</code> → 0 <code>TRUE</code> → 1 "1", "2", ... → 1, 2,... "A", ... → NA
logique	<code>as.logical</code>	0 → <code>FALSE</code> autres nombres → <code>TRUE</code> "FALSE" → <code>FALSE</code> "TRUE" → <code>TRUE</code> autres caractères → NA
caractère	<code>as.character</code>	1, 2, ... → "1", "2", ... <code>FALSE</code> → "FALSE" <code>TRUE</code> → "TRUE"

# Conversion

## Modes conversion

---

Consider a few simple examples:

**Case 1: We want to convert objects from logical mode to numeric mode**

```
logic = c(FALSE, FALSE, TRUE, TRUE, FALSE, TRUE)
```

```
conversion_numeric = as.numeric(logic)
```

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



# Conversion

## Modes conversion

---

**Case 2: We want to convert objects from character mode to numeric mode**

```
character = c("1", "2", "3", "A", "/", "T", "%", "-")
```

```
conversion_numeric = as.numeric(character)
```

Warning message:

NAs introduced by coercion

```
conversion_numeric
```

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

# Conversion

## Modes conversion

---

### **Case 3: We want to convert objects from numeric mode to logical mode**

```
numeric = 0:5  
conversion_logique1 = as.logical(numeric)  
conversion_logique1  
[1] FALSE TRUE TRUE TRUE TRUE TRUE
```

# Conversion

## Modes conversion

---

**Case 4: We want to convert objects from character mode to logical mode**

```
caractere = c("FALSE", "TRUE", "F", "T", "false", "t", "A", "(")
```

```
conversion_logique2 = as.logical(caractere)
```

```
conversion_logique2
```

```
[1] FALSE TRUE FALSE TRUE FALSE NA NA NA
```

# Conversion

## Modes conversion

---

**Case 5: We want to convert objects from numeric mode to character mode**

```
numerique = 1:8
```

```
conversion_caractere1 = as.character(numerique)
```

```
conversion_caractere1
```

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

# Conversion

## Modes conversion

---

**Case 6: We want to convert objects from logical mode to character mode**

```
logique = c(TRUE, FALSE)
```

```
conversion_caractere2 = as.character(logique)
```

```
conversion_caractere2
```

```
[1] "TRUE" "FALSE"
```

# Conversion

## Objects conversion

In many practical situations, it is useful to convert one object to another. Such a conversion will be possible thanks to the function: `as.objet` (`as.matrix`, `as.data.frame`, `as.list`, `as.factor`, ...).

We want to convert the matrix *a* into a data.frame *b*:

```
a = matrix(1:25, nrow = 5, ncol = 5);a;
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	12	17	22
[3,]	3	8	13	18	23
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

# Conversion

## Objects conversion

---

`b = as.data.frame(a);b;`

`b = as.data.frame(a);b;`

	V1	V2	V3	V4	V5
1	1	6	11	16	21
2	2	7	12	17	22
3	3	8	13	18	23
4	4	9	14	19	24
5	5	10	15	20	25

# Conversion

## Objects conversion

---

There are two steps to convert factor to digital. In fact, R converts the levels of the factor with the digital coding.

Examples:

```
facteur = factor(c(1, 5, 10));facteur
```

```
[1] 1 5 10
```

```
Levels: 1 5 10
```

```
facteur_numerique1 = as.numeric(facteur);facteur_numerique1;
```

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



# Conversion

## Objects conversion

---

In the following example, the factor has the Male (level 2) and Female (level 1) modalities. The digital conversion therefore gives the factor 2 1.

```
fac2 = factor(c("Male", "Female"))
```

```
fac2
```

```
[1] Male Female
```

```
Levels: Female Male
```

```
as.numeric(fac2)
```

```
[1] 2 1
```

# Conversion

## Objects conversion

---

To keep a factor (in numeric) while keeping the levels as they are specified, we will convert first to character and then to numeric.

```
facteur_caractere = as.character(facteur);facteur_caractere
```

```
[1] "1" "5" "10"
```

```
facteur_numerique = as.numeric(facteur_caractere);facteur_numerique
```

```
[1] 1 5 10
```

# Having access to Data

## Listing objects in memory

Suppose we have the following objects in memory:

```
x = 10 ; X = 5
```

```
w=8
```

```
name = "Wikistat"
```

```
dicton = "Aide-toi, autrui t'aidera"
```

```
y = 10 + 2
```

The `ls()` function is used to display the list of objects in memory (only the name of the objects is displayed):

```
ls()
```

```
[1] "X" "dicton" "name" "w" "x" "y"
```

# Having access to Data

## Listing objects in memory

---

If we want to list only the objects **containing a given character** in their name, we will use the option `pat` (abbreviation of pattern):

```
ls(pat = "n")  
[1] "dicton" "name"
```

To restrict the list to objects that **start with** a given character, we will use the `"^"` option:

```
ls(pat = "^n")  
[1] "name"
```

# Having access to Data

## Listing objects in memory

---

By default, `ls.str` displays details of all objects in memory:

```
> ls.str()
dicton : chr "Aide-toi, autrui t'aidera"
name : chr "Wikistat"
w: num8
x: num10
X: num5
y: num12
```

# Having access to Data

## Listing objects in memory

---

A useful option of `ls.str` is `max.level` which specifies the level of detail for the display of composite objects. We avoid displaying all the details with the option `max.level = -1`:

```
print(ls.str(), max.level = -1)
```

# Having access to Data

## Access through the indexing system

---

Indexing is an efficient and flexible way to selectively access elements of an object. Indexing is done using square brackets [], parentheses being reserved for function arguments.

- **Vectors**

Let the following vector x be:

```
x = 1:20 ; x  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

To access the fifth element of the vector x, just type the following command:

```
x[5]  
[1] 5
```

# Having access to Data

## Access through the indexing system

---

For vectors, matrices and arrays, it is possible to access the values of these elements using a comparison expression:

```
x = c(1.1, 5.3, 9, 4.2, 3.6, 7.2, 8.4, 1.6, 8.8, 3.5);x
```

```
[1] 1.1 5.3 9.0 4.2 3.6 7.2 8.4 1.6 8.8 3.5
```

If you enter `x < 5` on the console, the following values are displayed:

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

Indeed, `1.1 < 5` (the expression `x < 5` is verified, therefore TRUE), `5.3 > 5` (FALSE) ...



# Having access to Data

## Access through the indexing system

The vector y includes only those elements of x that are less than 5:

```
y = x[x < 5];y  
[1] 1.1 4.2 3.6 1.6 3.5
```

The vector z includes the 2nd and 6th elements of the vector x:

```
z = x[c(2, 6)];z  
[1] 5.3 7.2
```

We access the elements greater than 10 of the vector x by the following command:

```
x = 1 :20  
x[x > 10]  
[1] 11 12 13 14 15 16 17 18 19 20
```

# Having access to Data

## Access through the indexing system

---

We replace the elements of x greater than 10 by the value 20 by the following command:

```
x[x > 10] = 20 ; x
```

```
[1] 1 2 3 4 5 6 7 8 9 10 20 20 20 20 20 20 20 20 20 20
```

We replace the elements equal to 20 by the value 0 by the following command:

```
x[x==20] = 0; x
```

```
[1] 1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0
```

Note the use of "==" which is not an assignment but a test of equality.

# Having access to Data

## Access through the indexing system

Indexing is an efficient and flexible way to selectively access elements of an object. Indexing is done using square brackets [], parentheses being reserved for function arguments.

- ***Matrices***

Let the matrix a

```
a = matrix(1:25, 5, 5);
```

```
a
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	12	17	22
[3,]	3	8	13	18	23
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

# Having access to Data

## Access through the indexing system

The value of the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column is accessed by `x [i, j]`. We can access the values of the  $j^{\text{th}}$  column of the matrix `X` by the command `X [, j]`. In the same way, we access the values of the  $i^{\text{th}}$  row of the matrix `X` by the command `X [i,]`.

We access the element of line 2, column 3 by the command `a [2,3]`

`a[2, 3] = 2; a`

`a`

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	2	17	22
[3,]	3	8	13	18	23
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

# Having access to Data

## Access through the indexing system

We want to access the values of the third column of the matrix a

```
a[, 3]  
[1] 11 2 12 14 15
```

The last result is a vector and not a matrix. By default, R returns an object of the smallest possible dimension. This can be modified with the drop option whose default is TRUE:

```
> a[, 3, drop = FALSE]
```

```
      [,1]  
[1,]  11  
[2,]   2  
[3,]  13  
[4,]  14  
[5,]  15
```

# Having access to Data Access through the indexing system

You can also assign values to rows or columns of a matrix.

`a[ , 3] = 3; a`

`a`

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	3	16	21
[2,]	2	7	3	17	22
[3,]	3	8	3	18	23
[4,]	4	9	3	19	24
[5,]	5	10	3	20	25

`a[3, ] = 3; a`

`a`

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	3	16	21
[2,]	2	7	3	17	22
[3,]	3	3	3	3	3
[4,]	4	9	3	19	24
[5,]	5	10	3	20	25

`a[ , 3] = 11:15 ; a`

`a`

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	12	17	22
[3,]	3	3	13	3	3
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

# Having access to Data

## Access through the indexing system

---

The indexing system is also used to delete rows or columns. We can remove the  $j^{\text{th}}$  column of the matrix  $X$  by the command  $X[:, -j]$ . In the same way, we can delete the  $i^{\text{th}}$  row of the matrix  $X$  by the command  $X[-i, :]$ .

We delete the third row of the matrix  $a$  with the following command

```
a[-3, :];
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	12	17	22
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

# Arithmetic calculations and functions

Simple arithmetic calculations and functions:

`sum(x)`

`prod(x)`

`max(x)`

`which.min(x)`

`Which.max(x)`

`range(x)`

`length(x)`

`mean(x)`

`median(x)`

`var(x) ou cov(x)`



# Operations on matrices and vectors

R offers facilities for calculating and manipulating matrices. The following paragraph illustrates frequently encountered situations.

Let mat1 and mat2 be the matrices:

```
mat1 = matrix(1 : 4, nrow = 2, ncol = 2)
```

```
mat1
```

	[,1]	[,2]
[1,]	1	3
[2,]	2	4

```
mat2 = matrix(5 : 8, nr = 2, nc = 2)
```

```
mat2
```

	[,1]	[,2]
[1,]	5	7
[2,]	6	8

# Operations on matrices and vectors

The `rbind()` function stacks the matrices

```
rbind(mat1, mat2)
```

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

The `cbind()` function juxtaposes matrices while keeping the columns

```
cbind(mat1, mat2)
```

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

# Operations on matrices and vectors

The operator for the term-to-term product of matrices is "\*" while the operator for the product of two matrices is "% \*%". The operators for addition and subtraction of term-to-term matrices are "+" and "-" respectively.

```
rbind(mat1, mat2) %*% cbind(mat1, mat2)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	7	15	23	31
[2,]	10	22	34	46
[3,]	19	43	67	91
[4,]	22	50	78	106

```
cbind(mat1, mat2) %*% rbind(mat1, mat2)
```

	[,1]	[,2]
[1,]	74	106
[2,]	88	128

# Operations on matrices and vectors

---

The transpose of a matrix is done with the function `t()`.

`t(rbind(mat1, mat2) %*% cbind(mat1, mat2))`

	[,1]	[,2]	[,3]	[,4]
[1,]	7	10	19	22
[2,]	15	22	43	50
[3,]	23	34	67	78
[4,]	31	46	91	106

# Operations on matrices and vectors

```
diag(mat1)
```

```
[1] 1 4
```

```
diag(rbind(mat1, mat2) %*% cbind(mat1, mat2))
```

```
[1] 7 22 67 106
```

```
diag(4)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	0	0	0
[2,]	0	1	0	0
[3,]	0	0	1	0
[4,]	0	0	0	1

```
v = c(1, 2, 3, 4)
```

```
diag(v)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	0	0	0
[2,]	0	2	0	0
[3,]	0	0	3	0
[4,]	0	0	0	4

The `diag ()` function is used to extract and/or modify the diagonal of a matrix. It also makes it possible to construct diagonal matrices.

# Operations on matrices and vectors

We use `solve ()` for matrix inversion, `qr ()` for decomposition, `eigen ()` for eigenvalue calculation, and `svd ()` for singular value decomposition.

```
a = solve(diag(v)); a
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	0	0	0
[2,]	0	0.5	0	0
[3,]	0	0	0.33333	0
[4,]	0	0	0	0.25

```
eigen(diag(v))
```

```
$values
```

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

```
$vectors
```

	[,1]	[,2]	[,3]	[,4]
[1,]	0	0	0	1
[2,]	0	0	1	0
[3,]	0	1	0	0
[4,]	1	0	0	0

# Deleting objects

---

The `rm ()` function allows you to delete objects from memory; these options are similar to those of `ls ()`

```
[1] "dicton" "name" "w" "x" "y" "z"
```

To erase the object named `x` from memory, type the following command:

```
rm(x);ls()
```

```
[1] "dicton" "name" "w" "y" "z"
```

To delete all objects whose name begins with `"n"`:

```
> rm(list = ls(pat = "^n")); ls()
```

```
[1] "dicton" "w" "z" "y"
```

To erase all objects from memory:

```
> rm(list = ls()); ls()
```

```
character(0)
```

# Saving data

---

To save objects, this time of any type, we will use the command

```
save(x, y, z, file="xyz.RData")
```

To facilitate the exchange of files between machines and operating systems, you can use the option `ascii=TRUE`. The data can later be loaded into memory with `load("xyz.RData")`.

The `save.image` function is a shortcut for `save(list=ls (all=TRUE),file=".RData")`.