R programming

José R. Valverde
jrvalverde@cnb.csic.es
CNB/CSIC

R as a calculator

Statistical Facts:

Most people have more legs than the average.

About

- José R. Valverde (*j*)
  - PhD, MD, Msci, Higher Body of Systems and Information Technologies of the Administration
  - Head of Scientific Computing. CNB-CSIC.
- R
  - FLOSS
  - General purpose: not limited to statistics
  - Powerful abstractions

Getting Started

- **R**
- Type "R" and press the "enter" key.
- **>**
- The “**>**” that you see is the computer's way of telling you that it is ready and waiting for your commands

Note: from now on, text in **blue** is intended for you to type

Your first command

- This is the most important command, and one you will be using constantly (believe it or not)
- `1↵`
- What we have typed
  - a one '1', followed by the enter/return key
- What we get
  - [1] 1
- What it means
  - We told the computer "what is the result of *one*?"
  - The computer answers "the result of what you gave me is *one*"

Your second command

- **2 + 2 ↵**
- What we have typed
  - The key “2”, an space, a “+”, an space, key “2” and the ENTER ↵ key.
- What we are telling the computer
  - Here, take “**2**” and add (“**+**”) to it “**2**”
  - Do it! (↵)
  - We can add with “**+**” and substract with “**—**”
- **From now on, we'll ellide the ↵ at the end of a command**

Your second result

- `2 + 2 ↵`
- [1] 4
- What you get:
  - [1] 4
- What it means
  - **[1]** this is R telling you that this is result number 1 (useful when the result constains many values spread over many lines, ignore it for now)
  - **4** this is the only result of this command

Longer commands

- Let's solve $2\ x^2 + 4\ x - 6 = 0$
- We will need to calculate two values:
- $-b \pm \sqrt{\frac{b^2 - 4 \cdot a \cdot c}{2 \cdot a}}$
- That's -b plus/minus the square root of $b^2 - 4 \cdot a \cdot c$ and all of that, divided by $2 \cdot a$
  - We can make it more clear using **parentheses**
- $-b \pm \sqrt{\frac{((b^2) - (4 \cdot a \cdot c))}{(2 \cdot a)}}$
- Now, all we need to learn is how to express multiplication, division and square root.

$2 x^2 + 4 x - 6 = 0$

- We express multiplication using an asterisk (*****)
- We express division with a slash (***/***)
- We'll express the square root as **sqrt()**
- That gives us (after substitution of a, b and c)

Try it now

-2 ; -6

$$-b \pm \sqrt{\frac{b^2 - 4 \cdot a \cdot c}{2 \cdot a}}$$

Hint

- **Use parentheses** to encapsulate parts of your calculations.
- This will make clear the order in which you want them to take place.
  - to you
  - to others
  - to the computer

Hint

- When you enter long or complex commands (such as the preceding formula) you may recover and modify a previous line to save work
  - Press the "**up-arrow**" key:
    - this will recover the last command entered.
  - Press it again:
    - it allows you to go over various previous commands
  - Press the "**down-arrow**" key:
    - this allows you to navigate your command history
  - Use the "left-" and "right-arrow" and deletion keys to move and edit the command.

$2x^2 + 4x - 6 = 0$

- We express multiplication using an asterisk (*****)
- We express division with a slash (***/***)
- We'll express the square root as **sqrt()**
- That gives us (after substitution of a, b and c)
- **(-4 + sqrt((4 * 4) - (4 * 2 * -6)) / (2 * 2))**
- And
- **(-4 - sqrt((4 * 4) - (4 * 2 * -6)) / (2 * 2))**

$$-b \pm \sqrt{\frac{b^2 - 4 \cdot a \cdot c}{2 \cdot a}}$$

f(x)

- We are used to functions in mathematics.
  - They are supposed to carry out some operations on data
- Typically abstracted and expressed as f(x) or g(x)
- In a computer we do real work: each function needs a name.
  - We soon run out of single-letter names
  - And of imagination!
- Most functions will use longer names.

sqrt()

- sqrt(...) is called a "function" in R
- A function takes zero or more values (called arguments) enclosed in parentheses
  - f(x): function(args) ; args $\rightarrow$ { ø , a_1, … , a_n }
- A function returns zero or more values (the result)
- The function and its arguments will be substituted by the result in the command
  - A command may contain many functions
- And then the command will be evaluated

Thinking “computerish”

- We are going to see many new objects
- Whenever we type in one of them, the computer will substitute it for its meaning
- When we press the key labeled <ENTER> the computer will evaluate the line and give us the result:
  - 2 + sqrt(4)
  - sqrt(4) will be substituted by its result, “2”
  - 2 + 2
  - And produce “4”

Hint: splitting commands

- Notice the difficulty with counting all parentheses in the formulas
- Notice the lengthy command line
- You can split commands across several lines if needed...

```
 - 4 +
 ( sqrt(
     (4 * 4) - (4 * 2 * -6) )
           /
       (2 * 2)
 )
```

...as long as R knows that the line is not complete

  - e.g. without the '+' it would think we only want '-4'

- **be careful** (this is a *very bad* example, by the way).

More commands

```
2 + 2 * 2
2 - 2 * 3
(2 + 2) * 2
(2 - 2) * 3
2 - (2 * 3)
3 / 2
2 ** 2
2 ** 3
3 ^ 2
```

- **Observe and note that**
  - Multiplication is computed before addition and substraction
  - Parentheses clarify our intention
  - Power/exponentiation is indicated by "**" (two asterisks) or "^" (caret)
  - Calculations use real numbers
  - Negative numbers use '-'

An exercise

- Calculate the volume of a cylinder 4m high with a base 2m in diameter.

- Calculate the volume of a cell 1um in diameter (assuming it is perfectly spherical)

Hint

- Judicious use of space and parentheses will help you express better what you intend to do, and make it clear and easier to see and interpret.
- This will be quintessential to catch errors!

An exercise

- Calculate the volume of a cylinder 4m high with a base 3.5m in diameter.

```
4 * (3.1416 * ( (3.5 / 2) ** 2) )
```

- Estimate the surface and volume of a cell 1.5µm in diameter (assuming it is perfectly spherical)

```
4 * 3.1416 * ( ( 1.5 / 2) ** 2)
(4 / 3) * 3.1416 * ( (1.5 / 2) ^ 3 )
```

Variables

Compute the volume of a cylinder

- For a cylinder of radius 10 and height 20
- It is the area of the base multiplied by the height
- Where the area of the base is Πr^2
- We can use the full formula

```
20 * ( 3.14159 * 10 * 10 )
```

- As formulas become longer, it soon becomes unreadable

Variables

- Are convenience names that we use to refer to values
- We **store** a value in a variable and write the name of the variable instead

```
pi <- 3.14159  # this is the preferred form
pi = 3.14159
```

- Either form will work. The recommended approach is to use <— to avoid confusion with comparisons (more on them later).
- When the variable is used it will be substituted by its value

```
pi
```

Volume of a cylinder using variables

- **Using variables**, our calculations will be easier to understand
  - We'll make less mistakes
- Variables are remembered by the computer
  - We can use them many times

```
radius <- 10
height <- 20
pi <- 3.14159
base_area <- pi * radius ** 2
volume <- base_area * height
base_area
volume
```

Exercise

- Repeat the cell calculations using variables
- Remember: surface area and volume of a cylindrical cell 1.5 μm in diameter.

Exercise

- Repeat the cell calculations using variables
- Remember: surface area and volume of a cylindrical cell 1.5 μm in diameter.

```
radius <- 1.5 / 2
pi <- 3.14159
circle <- pi * (radius ^ 2)
surface <- 4 * circle
volume <- (4/3) * pi * (radius ^ 3)
surface
volume
```

Hint

- Judiciously choosing variable names will help you make your intent clear
  - And facilitate correcting errors

- Variable names start with a letter and can contain upper and lower case letters, numbers, dots (.) or underscores (_), e.g.

  my_name_is_Jose_R._Valverde

More variables

- **`x <- 1.2345`**
  - That's a less-than “<” and a dash “-” together
- **`y <- 1.e-5`**
  - That's $1.0 \cdot 10^{-5}$
- **`species_name <- 'Homo sapiens'`**
  - A **string** of characters (letters, numbers. symbols...) is enclosed in **matching** quotes, single (') or double (")
- **`is.mammal <- FALSE`**
- **`is.bacteria <- T`**
  - Logical values are actually represented by variables TRUE and FALSE and may be abbreviated to T and F

Logical expressions

(boolean algebra)

(comparisons)

Logical expressions

<	less-than
>	greater-than
<=	less-than or equal to
>=	greater-than or equal to
==	equal to
!=	not equal to
&	and
|	or (vertical bar)
!	not (exclamation mark)

Using logical expressions

- Logical operations will return a boolean value: TRUE or FALSE

```
x <- 3 > 2
x
x = 3 == 2
x
x <- 3 == 4 & 3 < 4
x
x <- 3 == 4 | 3 < 4
x
x <- (3 == 4) & (3 < 4)
x
```

Combining commands in one line

- You can put more than one command in one line if you separate them with semiconlons (;)

```
x <- 3 > 2 ; x
x = 3 == 2 ; x
x <- 3 == 4 & 3 < 4 ; x
x <- 3 == 4 | 3 < 4 ; x
x <- (3 == 4) & (3 < 4) ; x
```

Model formulae

Expressing a model

- We express a relationship among variables using a **model formula**, e.g.:
  - o ~ e1 + e2 +e3
- This means that the outcome variable (“o') depends (“**~**') on the explanatory variables “e1', “e2' and “e3'.
- There may be several kinds of relationship. Here, “+' does **not** mean “add', it is only a way to express the type of dependence relation.

Basic syntax for model formulae

- $O \sim F$ — the outcome is modeled using formula F
- $O \sim F_1 + F_2$ — both F_1 and F_2 should participate in explaining O
- $O \sim F_1 - F_2$ — O is explained by all of F_1 except for F_2
- $O \sim F_1 : F_2$ — Consider the interaction between F_1 and F_2
- $O \sim F_1 * F_2$ — Consider all, F_1, F_2, and the interaction between F_1 and F_2, i.e.: $F_1 + F_2 + (F_1 : F_2)$
- $O \sim F \wedge n$ — Use all terms in F crossed to order n

Functions

Functions

- Just like we talk of f(x) in mathematics as a *function* of x, which produces a new value derived from x, we also have “functions” in R.
  - c( ) — combine
  - sqrt( ) – compute square root
  - etc...
- A function has a name and *may* act on zero, one or more values. These values are enclosed in parentheses and are separated by commas

```
vec <- c(1, 2, 3) ; vec
```

Functions in R

- R comes with many useful functions, and you can add more to it
  - Either written by yourself
  - Or written by others
- All functions *already defined* in R, and many functions provided by others, have been documented.
- You can read this documentation with `help( )`
  - What did c() in the former example do?

```
help(c)
```

Help

- You can get help on any R “object” that has associated documentation.
- Many functions, but also packages (collections of functions) and data sets (collections of data) have associated documentation.
- We *should* make extensive use of the help function in normal practice

```
help(c)
help(help)
help(seq)
```

Arguments and argument lists

- Much like you can have f(x), f(x,y) or f(x,y,z), you can have functions acting on one or more values.
- The values that a function acts upon, that a function uses to generate its result, are called “**arguments**” (here "*1*", "*2*" and "*3*"):

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

- All the arguments taken together (all the values within the parentheses) are called an “**argument list**” (here "*1, 2, 3*")

Variable-length argument lists

- R has provision for **variable-length** argument lists: functions which may take any number of arguments
  - *c(1,2,3, ...etc...)*
- R also allows providing "**default**" values for some arguments:
  - this allows us to define a function for the general case, e.g. rotation in 3D, and use it for more specific instances (like 2D or 1D) without having to specify all the arguments.

Positional arguments

- Assume we have a function point(x, y, z)
  - x is the first argument
  - y is the second
  - z is the third
- If we use point(10, 20)
  - The computer assigns 10 to x, 20 to y, and has no value to assign to z
  - This would be an error unless there is a default value defined for z (a default value is a value that will be used *as default* if no value is provided).
  - Try this: **`sqrt(); help(sqrt)`**

Default arguments explained

- **Imagine** we *had* a function point(x, y, z) to draw a point in 3D
  - point(10, 20, 30)
- We **could** use it to draw points in 2D by setting one of the coordinates to a constant value, e.g. 0
  - point(10, 20, **0**)
- If we will do this often, it makes sense to tell the computer “whenever there is no z coordinate, assume its value is 0”, then we might use
  - point(10, 20)
  - as a shorthand to mean the same as point(10, 20, 0)

Default arguments

- Let us calculate a t-test:

```
a <- c(1, 2, 3, 3, 2, 1)
b <- c(3, 4, 7, 7, 6, 4)
t.test(a, b, "two.sided", 0, FALSE, FALSE, 0.95)
t.test(a, b)
help(t.test)
```

- In the second case, if we do not specify some arguments, the predefined value is used by default. When a predefined value is used, we call it a default argument.

Named arguments

- But, what if we do not remember the order of the arguments or we want to use the default value of one argument in the middle?
- Arguments can be “named” as well. If you know their name, you can specify them **by name**.
  - Then you do not need to remember the order
  - If one is missing, R will know which is it and use the default (if any has been predefined)

Example

```
t.test(a, b, "less", TRUE)
```

- TRUE can be applied to both paired and var.equal. To which one do we want to apply it?
- In principle it will be assigned to the first argument that needs it.
- But we can specify to which it should be assigned using its name (see the difference).

```
t.test(a, b, "less", paired=TRUE)
```

```
t.test(a, b, "less", var.equal=TRUE)
```

help( ) revisited

- That was quite a feat!
- But it will allow us to understand help( )
- Try now to read the help for function “seq( )”

```
help(seq)
```

Generating values with seq( )

- seq( ) is defined as seq(...), which means that it can take a variable number of arguments.
- Read on: you will see that it has various named arguments
  - seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)), length.out = NULL, along.with = NULL, ...)
- As you can see, all the argument names are followed by an “=” and a default value; i.e. we can omit any and all the arguments and then they will take the values defined here.

seq

```
seq( )
seq(10)
seq(10, 20)
seq(from=10, to=20)
seq(10, 20, 2)
seq(from=10, to=20, by=2)
seq(by=2, to=20, from=10)
```

Why use named arguments?

- Note the previous examples: if we do not specify names, we must give the arguments in the correct order (or, if you prefer, arguments will be considered in the order originally specified).
- Named arguments, clearly state which value we want to assign to which argument.
- *We do not need to give named arguments in order*.
- This will be handy when we do not (or do not want to) remember (or use) the correct order of the arguments or all the possible arguments.

Vectors

- Did you hear the joke about statisticians?
- Probably...

Vectors

- Vectors are one-dimensional sets, composed of values of the same data type (e.g. a collection of numbers, or a collection of strings, or of logical values)
- We only call them vectors by analogy to maths.

```
( 1, 2, 3, 4, 5, 6, 7, 8, 9,
  9, 8, 7, 6, 5, 4, 3, 2, 1 )
```

Making vectors

- Ensure that all elements are of the same type
- We may give them a name. I.e. we can have a variable whose contents are the list of values.
- The values can be read from a file (more on this later) or assigned
  - From other variables or vectors
  - With the aid of the **c( )** function (combine):

```
my_vector <- c(1, 2, 3, 4, 5,
               1, 2, 3, 4, 5)
my_vector
```

Using vectors

- You can refer to a vector by its name

```
my_vector + my_vector
my_vector
```

- You can refer to individual items with square brackets [ ].
  - Items in a vector are numbered 1 … n
  - In school you would use my_vector_i
  - In R we will use *my_vector[i]*

```
my_vector[ 5 ]
my_vector[ 5 ] + my_vector[ 6 ]
```

longer vectors

- You can create a vector combining several vectors and values!

```
Long.vec <- c(
              c(1,2,3),
              c(4,5,6),
              c(7,8,9)
             )
Long.vec
```

- Note that we still get one single vector, with all the values concatenated one after the other; not a vector of vectors.
- Also note that we split the command in several lines, this has the potential of making our intention easier to read and more clear.

More on vectors

- We can obtain useful information about vectors

```
v <- c("one", "two", "three")
length(v)
```

- We can try to force the combination of values of different types with c( )
  - `m <- c("v", 1, 2, 3)`
  - But in this case -because of the v (a letter)- all values will be converted to the same type: strings of letters (note the quotes!)

```
m
```

Simple sequences

- You can use seq( ) to generate the values:

```
vec <- c( seq(10, 20, 2) )
```

- For simple consecutive sequences, there is a shorter notation:
  - min:max will generate all values between min and max

```
help(":");   10:20
```

- For repetitive sequences you can use rep( )

```
help(rep) ; rep(10, 5))
```

Combining values

- Function c( ) will combine the values you list as arguments and produce all of them as a collection
- Function seq( ) will generate the values and produce them as a collection
- Function rep( ) will repeat values...
- Each of these collections can act as a single argument as well:

```
c( 1, 2, 3:10, seq(1, 10), rep(1, 10) )
c( rep( seq(1, 10, by=2), 2), 10:1)
```

Repeating values

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

- Here, we generate a vector with three values using seq(3)
- We pass this vector as first argument to rep( )
- Then we pass a second vector as second argument to rep( ), this one generated with c( ).
- `help(rep)`
- Both vectors must be the same length.
- The first vector specifies the values to repeat
- The second vector, the number of times each corresponding value has to be repeated.

Exercise

- Try to create a vector with the contents
  - 1 2 3 4 5 1 2 3 4 5

  - using c()
  - using c() and :
  - using c(), rep() and :

Off you go!

Exercise

- Try to create a vector with the contents
  - 1 2 3 4 5 1 2 3 4 5

  - using c() **c(1,2,3,4,5,1,2,3,4,5)**
  - using c() and : **c(1:5, 1:5)**
  - using c(), rep() and : **c(rep(1:5, 2))**

Using vector data

- We already saw that we can use a variable using its name

```
x <- seq(3)
rep(x, c(1,2,3))
```

- And we can use a single value in a vector by giving its index in square brackets:

```
x[2]
```

- If no value is given, the full vector is returned

```
x[ ]
```

- But wait, there is more...

Using vector data

- We can retrieve more than one value from a vector. All we need to do is list the values we want to retrieve:

```
x <- seq(10,100, 10)
x
3:5
x[3:5]
x [ c(1, 3, 5, 7, 9) ]
x [ seq(1, 9, 2) ]
x[ ]
```

Intersecting vectors

```
v1 <- 1:10
v2 <- 5:15
v1 %in% v2
v2 %in% v2
v1[ v1 %in% v2 ]   # note the order
v2[ v2 %in% v1 ]

v1[ v2 %in% v1 ]   # study this
```

Boring?

- I know.
- What is the use of all this?
  - In many occasions we will work with collections of values
  - And in also many situations we will want to “label” them.
    - indicate source, experiment, characteristics, color, properties, etc...
    - Being able to generate “labels” automatically instead of one by one will save us a lot of time working with large data.
  - Or to use only specific subsets.

Omitting vector data

- We can also retrieve all values except for some using a – sign

```
x [ -3 ]
```

- Returns all values except the third

```
x [ 0:-3 ] ; x [ -1:-3 ]
```

- Omit all values between the first and the third

```
x [ -3:-5 ]
```

- Omit values between the third and fifth

Hint

- Whenever you have trouble understanding what will happen in a complex command, you can always resort to the first kind of command you learnt (viz. “1”, “*give me the result of... 1*”):
  - divide the command in its constituent parts
  - give each part separately and see the result
  - E.g: in the previous slide you could have
    - first tried 0:-3, -1:-3 to see their result
    - then think on the effect when used as subindexes with x[ ]

Naming vector values

- You can give names to vector values and use them to access elements:

```
v <- 1:3
names(v) <-  c("Tom", "Dick", "Harry")
v
v["Dick"]
```

Removing data with logicals

```
v <- c(1:10, NA, 11:20, NA)
is.na(v)
v[ is.na(v) ]
v[ ! is.na(v) ]
v <- v[ ! is.na(v) ]
v < 11
v[ v < 11 ]
v[ v > 10 ]
```

Plotting

- seed() sets a starting value for random number generation (useful for reproducibility)
- rnorm() generates pseudo-random numbers drawn from a normal distribution.

```
vals <- rnorm(n=100, mean=10, sd=3)
vals
print(vals)
plot(vals)
hist(vals)
```

- Compare with the one on your side, then use the following and repeat all the previous commands:

```
seed(202405)
```

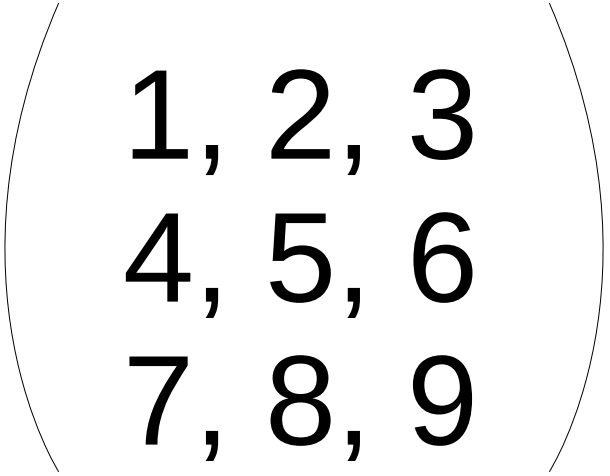
Summary to date

- > value
  - *give me the the result of “value”*
- > var <- value
  - *store “value” in a box named “var”*
- > vector <- c(*a 1D collection of values*)
  - *store a **linear** collection of values, **all of the same type** in a box named “vector”*

Matrices

Matrices

- Are two-dimensional collections of values, all of the same type.
- They are the analogue of a matrix in linear algebra
- But we may have matrices of numbers, matrices of strings, matrices of logical values...



Making matrices

- We use the function matrix( ) to create matrices
  - **`help(matrix)`**
  - data=NA, means that the default value is *NA* (note it has no quotes). This is a special statistical variable meaning "Not Available" (could not be measured).
  - nrow is the number of rows. It comes first because *R matrices are organized first by row*.
  - ncol is the number of columns. Matrices in R are *two-dimensional*!

Understanding matrices

- Try to think of matrices in the same way you usually think of data:
- Normally you organize the measures for each sample in one line, and place each measure in a different column so you can see how the measure changes

Sample	Sex	Height	Weight
1	Male	1.8	70
2	Female	1.7	55

- Individual samples go in rows, hence rows go first

Creating matrices

- We will normally give the values column-by-column as a vector:

```
vals <- c(1, 1, 1, 2, 2, 2, 3, 3, 3)
matrix(vals, 3, 3)
```

- If we want to give the values row-by-row, we must specify **byrow=TRUE**

```
matrix(vals, 3, 3, byrow=T)
matrix(data=vals, nrow=3, ncol=3, byrow=TRUE)
```

Hint

- Whenever possible, use variables and names to make your intention clear
  - You will appreciate it when you have forgotten what it means some time later
  - You will facilitate correcting your mistakes
  - Others will appreciate it when they see your commands
  - Others will be able to help you
- Did we mention the wonders of collaboration?

Better

- **When in doubt, use variables with meaningful names** (e.g.):

```
col.1 <- c(1,1,1)
col.2 <- c(2,2,2)
col.3 <- c(3,3,3)
matrix(c(col.1, col.2, col.3), ncol=3,
       nrow=length(col.1))
row.1 <- c(1,1,1)
row.2 <- c(2,2,2)
row.3 <- c(3,3,3)
matrix(c(row.1, row.2, row.3), nrow=3,
       ncol=length(row.1), byrow=T)
```

Using matrix data

- Is much like using vector data, but this time we have two dimensions: we'll separate each dimension by a comma:
  - Let us create a matrix and access one element:

```
m <- matrix(1:9, 3, 3, byrow=T)
m
m[2, 3]
```

- The first value **always** specifies the row, and the second, the column.

Your first matrix

- Create a matrix with the following values

  1 2 3

  2 2 2

  3 2 1

Off you go!

Your first matrix

- Create a matrix with the following values

  1 2 3

  2 2 2

  3 2 1

```
col.1 <- 1:3
col.2 <- rep(2, times=3)
col.3 <- 3:1
vals <- c(col.1, col.2, col.3)
m <- matrix(vals, ncol=3)
```

Retrieving matrix data

- As with vectors, if we leave the index empty, all the values are returned

```
m[2, ]          # returns row number 2
m[ , 2]         # returns column number 2
```

- And we can use a vector of values to specify several coordinates

```
m[ , c(2,3)]
m[ , 2:3]
m[2:3, ]
m[2:3, 2:3]
```

Identifying data

- Remember logical operations? Those that resulted in a value of TRUE or FALSE?
- There are also *logical functions* such as

```
is.numeric(3)
is.character(3)
is.character("3")
is.infinite(1 / 0)
is.na(NA)        # NA: not available
is.null(NULL)    # NULL: “no value here”
```

Selecting data

- We can use logical values to select elements of collections:

```
x <- seq(20)
x
x < 10    # what is the result of this?
selected <-  x < 10
selected # what does “selected” contain?
x[ selected ]
x [ x < 10 ]
```

Hint

- Observe the value in square brakets at the left of the results.
- Now you can see how it can be helpful when dealing with large collections of data.
- It is an index or place holder that lets you know at which result data point you are.

Matrix operations

- Addition (elementwise)

```
m.1 <-  matrix(1:9, nrow=3)
m.2 <- matrix(9:1, nrow=3)
m.1 + m.2
m.2 + m.1
```

- Substraction (element by element)

```
m.1 <-  matrix(1:9, nrow=3)
m.2 <- matrix(9:1, nrow=3)
m.1 - m.2
m.2 - m.1
```

Matrix multiplication

- * gives the element-wise multiplication

```
m.1 <-  matrix(1:9, nrow=3)
m.2 <- matrix(9:1, nrow=3)
m.1 * m.2
m.2 * m.1
```

- **%*%** gives actual matrix multiplication

```
m1 <- matrix(1:24, ncol=4)    # 6x4
m2 <- matrix(1:32, nrow=4)    # 4x8
m1 %*% m.2                    # 6x4 · 4x8 OK
m2 %*% m.1                    # 4x8 · 6x4 KO
```

Matrix operations

- Outer tensorial product (also for vectors): **%o%**

```
m.1 %o% m.2
1:4 %o% 5:8
```

- Transposition: t( )

```
t(m.1)
```

- crossprod(A, B): cross product of A'B:

```
crossprod(m.1, m.2)
```

- There are many more (for diagonalization, solution of systems of equations, etc...)

Naming matrix values

- It is extremely useful to attach names to rows and columns in a matrix.
- This allows us to know what the values mean.
- And even better yet, it allows us to specify values by name.

One way to add names

- colnames(matrix) gives the column names
- rownames(matrix) gives the row names
- We can assign values to them: try this

```
m <- matrix(1:6, nrow=2, byrow=T)
m
rownames(m) ; colnames(m)
rownames(m) <- c("sane", "sick")
colnames(m) <- c("control", "placebo",
                 "treated")
m
rownames(m) ; colnames(m)
```

Tensors

- A vector is a special, 1D type of matrix
- A matrix is a special, 2D type of tensor
- A tensor can have any number of dimensions
- **TENSORS ARE NOT SUPPORTED IN R BY DEFAULT.**
  - **BUT CAN BE USED WITH THE APPROPRIATE “MAGIC”.**

Summary to date

- 1
  - gives the value
- 1+1
  - gives the value
- var <- 2 * 2
  - saves the value in var
- vec <- 1:10
  - saves a 1-D series of same-type values in vec
- mat <- matrix(1:9, nrow=3, byrow=T)
  - saves a 2-D series of same-type values in mat

Data frames

Data frames

- The next step in complexity is provided by data frames.
- Data frames can be composed of different data types but
  - all data in **one column** must be of the same type.
- Each column and row can have a name
- They are similar to databases, tables and spread sheets.

Why use a data frame

- Data frames bind vectors of different types together
- This allows us to have a dataset with e.g. a column of values, a column of factors, a column of named factors, etc...
- Think of each row as an experiment and each column as a measure/property

ID	SEX	VALUE	LOCATION
1	M	10	SPAIN
2	F	9	FRANCE
3	M	11	USA

Iris

- R comes with many datasets already installed, we can load them with data( ) and get information with help( )

```
data(iris)
help(iris)
print(iris)
head(iris)  # show only first rows
tail(iris)  # show only last rows
summary(iris)
```

Creation

- Data frames are created with data.frame( )

```
height <- seq(150, 190, length.out=6)
weight <-  rnorm(n=6, mean=60, sd=10)
age <- runif(n=6, min=20, max=30)
sex <- rep(c("F","M"), c(3, 3))
data <- data.frame(height, weight, age, sex)
data
```

- Note that, if you use variabes, R automatically takes the variable names as column names.

Hint

- You can also put all the pieces together assigning column names with something like

```
df = data.frame(
     name=c("Joe", "John", "James"),
     age=c(21, 22, 23),
     sex=rep("M", 3)
)
df
```

Accessing data frames

- Data frames are accessed exactly in the same way as matrices are:
  - By using subindices and/or row/column names

```
data[1, 3]
data[ ,"sex"]
```

- Columns within a bidimensional data set can also be accessed by name using **“$” notation**. E.g. columns in a data frame:

```
data$sex
data$height
```

Hint

- If we have several data frames, we may chose one as the default dataset to work with using `attach( )`
- After we attach( ) a data frame, we can access its components directly without the $ notation

```
attach(data)
sex
data$sex
```

- Another way within some function argument lists is using "with=*data.frame.name*"
- When you are done, “detach( )” it:

```
detach(data)
```

Basic descriptives

- **`iris`**
  - This will show the dataset; if it is too large we may not be able to see it entirely
- **`summary(iris)`**
  - This is more useful: it gives us summary information on all the variables of the dataset
- **`iris$Sepal.Width`**
- **`iris$Petal.Length`**
  - This will show the values in the correspondingly named columns of the dataset
- **`table( iris$Species )`**

Hypothesis testing

```
attach(iris)
shapiro.test(Sepal.Length)
bartlett.test(Sepal.Length ~ Species)
# ~ means'by'
t.test(Sepal.Length[1:50],
       Sepal.Length[51:100])
wilcox.test(Sepal.Length[ Species == "setosa"],
            Sepal.Length[ Species=="versicolor"])
a <- aov(Sepal.Length ~ Species)
TukeyHSD(a)
detach(iris)
```

Basic graphics

```
hist( iris$Sepal.Length )
hist( iris$Sepal.Width )
hist( iris$Petal.Length )
hist( iris$Petal.Width )
boxplot( iris$Sepal.Length )
```

- Exercise: repeat all graphics with boxplot(...)
- Plot one column against other

```
plot( iris$Sepal.Length,
      iris$Petal.Length )
```

- plot all combinations

```
plot( iris )
```

Summary so far

- We can have variables with
  - isolated values:
    - var <- 3.14159
  - linear series of similar-type values (vectors, 1D matrices)
    - vec <- 1:9
  - 2D series of similar-type values (matrices)
    - mat <- matrix(1:9, nrow=3, ncol=3, byrow=T)
  - 2D data.frames with columns of different types (all values in a column are of the same type, all columns are of the same length)
    - dat <- data.frame(c1=1:3, c2=c(T,F,T), c3=c(“A”, “B”, C”))

Lists

Lists

- A list is a compound object, i.e. it can contain objects or values of different types and lengths.
- A list can contain any kind of objects, named **components**, all of them mixed together in a bag.
  - matrices are rectangular, all elements of the same type
  - data frames are rectangular, all elements in a column of the same type, all columns same length
  - lists can have components of different type and lengths

Making and using lists

```
x <- seq(10)
y <- matrix(seq(10), nrow=5)
xylist <- list(x, y)
xylist
```

- **Note** that list components are referenced using **double** square brackets

```
xylist[[1]]
xylist[[2]]
```

Naming list components

- We can name list components and reference them using variable names, double square blackets or $ notation

```
xylist = list(X = x, Y = y)
xylist[[1]]
xylist$X
xylist[[2]]
xylist$Y
```

- Note that this departs from matrices and data frames (where we named rows and columns), as we now label components

Kolmogorov-Smirnov test

```
# let us create two simple hypothetical data frames
ref <- data.frame(weight=c(10, 20, 30, 40),
                  height=c(1.1, 2.2, 3.3, 4.4))
exp <- data.frame(weight=c(11, 22, 33, 44),
                  height=c(5.5, 6.6, 7.7, 8.8))
# and now apply the Kolmogorov-Smirnov test
weight.ks <- ks.test(ref$weight, exp$weight)
height.ks <- ks.test(ref$height, exp$height)
# str shows the structure of a variable
str(weight.ks)
# the result of ks.test is a list, we can access its
elements with, e.g.
weight.ks$p.value
```

Adding complexity

- We can also create lists of lists:

```
str(weight.ks)
weight.ks$p.value
# let us now create a list with the results
res <- list(w.ks=weight.ks, h.ks=height.ks)
# res is a list of lists:
res[[1]]
# gives us w.k, and therefore we can use
res[[1]]$p.value
# or to access h.k and its p.value (using $ notation)
res$h.ks
res$h.ks$p.value
```

Generating normal data

- We can generate a set of random data following a specific distribution:

```
help(rnorm)
set.seed(1234)
x <- rnorm(100, 0, 1)
hist(x)
boxplot(x)
y <- 5 + 2 * x + rnorm(100, 0, 1)
plot(x, y)
abline(5, 2)
```

y <- 5 + 2 * x

- Let us make a linear model of y vs. x

```
m <- lm( y ~ x )
plot(m)
```

- And now create a nice plot:

```
plot(x, y)
str(m)
m$coefficients   # m is a list
intercept <- m[[1]][ "(Intercept)" ]
slope <- m$coefficients[ "x" ]
abline(intercept, slope, col="red")
points(x, m$fitted.values, col="green")
title(paste("fit: y =", intercept,
            "+", slope, "* x"))
```

Copying graphics to a file

- We can save any graphics that we have generated.
- R provides various graphics “devices”. By default it will draw to our screen.
- Some "devices" can be associated with files
- We can copy a graphic to a file by sending it to the correct "device". Here we copy the current graphic to a “PNG” device, associated with a file named 'plot.png':

```
dev.copy(png, file='plot.png')
dev.off()
```

- If you can't find it, `getwd()` will show you in which directory/folder you are working now.

Copying graphics explained

- **dev.copy(png, file='plot.png')**
- **dev.off()**
- That means: make a copy of the current graphic device (the one we see on the screen) into a new device (dev.copy)
- The new device makes a copy as a PNG graphic, and will save it on file "*plot.png*"
- We cannot see the file contents until the PNG device is turned off and the file closed
- When we close it, only the **last** graphic that was in the screen will be saved

Saving graphics

```
# This will not show the graphic on
# the screen
png("plot.png")
plot(x, y)
```

We can specify a different (from the screen) output device: the png() command tells R to use a PNG file instead of the screen.

It will not copy what was already visible, but will use the new device (the PNG file) from now on.

Since we are plotting to a file (and not the screen) we will not see it.

```
dev.off()
```

- This will close the **last** open device (png) and now graphics will go back to the screen again.

Summary: operators

<- =	assignment
+	addition
-	subtraction
*	multiplication
/	division
** ^	exponentiation
( )	grouping
%*%	matrix multiplication
%in%	pertenence

Summary: Logical expressions

<	less-than
>	greater-than
<=	less-than or equal to
>=	greater-than or equal to
==	equal to
!=	not equal to
&	and
|	or (vertical bar)
!	not (exclamation mark)

Summary: data types

- **Variables** allow us to give a name to values
- **Values** can be
  - **single values**: numbers, text, logical (T/F)
  - **vectors**: 1D series of values of the same type
  - **matrices**: 2D rectangular series of values of the same type
  - **data.frames**: 2D rectangular series of columns, all columns the same length, all elements in a column are a vector (all the same type) but columns can be of different types
  - **lists**: series of components of any type and length

Summary: accessing values

- by index : **[ ]**, or **[[ ]]** in the case of lists
  - separated by comma; rows first in 2D data
  - indicating items to select or remove (negative)
    - by number
    - by logical values (TRUE / FALSE)
    - by name of row, column or component
    - empty means all
- using **$** notation and a name
  - returns a column in a data frame or a component in a list

Thank you

Questions?