An intRoduction to R

Univerisity of Padova (IT)

Who aRe you?

- R is an open source software for statistical computing, graphics, and so much more
- RStudio is the perfect IDE for R $\rightarrow$ allows for a better, easier use of R
- R runs on Windows, MacOs, Unix

CalculatoR

```
3 + 2  # plus
3 - 2  # minus
3 * 2  # times
3/2  # divide
sqrt(4)  # square root
log(3)  # natural logarithm
exp(3)  # exponential
```

Use brackets as you would do in a normal equation:

```
(3 * 2)/sqrt(25 + 4)
```

R ignores everything after # (it's a comment)

Assign

The results of the operations can be “stored” into objects with specific names defined by the users.

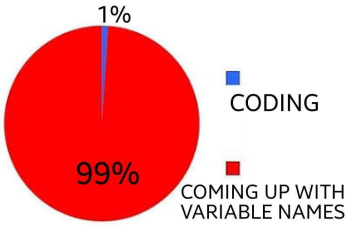
To assign a value to an object, there are two operators:

1. `=` `x = exp(2^2)`
2. `<-` `X <- log(2^2)`

The elements on the right are assigned to the object on the left

Careful! `R` is case sensitive: `x` and `X` are two different objects!!!

Variable names



Valid variable names are letters, numbers, dots, underscores (e. g., `variable_name`)

Variables names cannot start with numbers

Again, `R` is case sensitive

Get help

R is open source and it used world wide $\rightarrow$ there's a huge community ready to help you

Just copy & paste any error message or wanring in google or ask google "how to something in r"

Ask R to help you! Type `?` in your console followed by the name of the function:

```
?mean()
```

Will show you the help page of the `mean()` function

Be tidy

Organize your files

R projects are the best way to organize your files (and your workflow)

It allows you to have all your files in a folder organized in sub folders

You don't have to worry about the wording directories because it's all there!

By creating a nw project, you can also initialize a shiny app

Create a new R project

File → New project and choose what is best for you (unless you have already initialized a directory for your project, select a new directory):

- R poject “basic”
- R package
- `Shiny`

and so much more

Take out the trash

The R environment should be always tidy

If it feels like you're losing it, just clean it up:

```
ls()  # list objects in the envrinoment
rm(A)  # remove object A from the environment
rm(list = ls())  # remove everything from the environment
```

Save the environment

It might be useful to save all the computations you have done:

```
save.image("my-computations.RData")
```

Then you can upload the environment back:

```
load("my-Computations.RData")
```

When to save the environment

The computations are slow and you need them to be always and easily accessible

The best practice wis to save the script and document it in an `RMarkdown` file

Working directories

If you choose not to use the R projects ({what a bad, bad, bad idea}), you need to know your directories:

```
getwd()  # the working directory in which you are right now

dir()  # list of what's inside the current working directory
```

Change your working directory:

```
setwd("C:/Users/huawei/OneDrive/Documenti/GitHub/RcouRse")
```

Structures in R

Functions and arguments (pt. I)

Almost everything in R is done with functions, consisting of:

- a name: `mean`
- a pair of brackets: `()`
- some arguments: `na.rm = TRUE`

```
mean(1:5, trim = 0, na.rm = TRUE)
```

```
[1] 3
```

Arguments may be set to default values; what they are is documented in `?mean`

Functions and arguments (pt. II)

Arguments can be passed

- without name (in the defined order)
- with name (in arbitrary order) → keyword matching

  mean(x, trim = 0.3, na.rm = TRUE)

No arguments? No problems, just brackets:

`ls(), dir(), getwd()`

Want to see the code of a function? Just type its name in the console without brackets:

```
mean
```

Vectors

Vectors are created by **c**ombining together different objects

Vectors are created by using the `c()` function.

All elements inside the `c()` function **must** be separated by a comma

Different types of objects → types of vectors:

- `int`: numeric integers
- `num`: numbers
- `logi`: logical
- `chr`: characters
- `factor`: factor with different levels

int and num

`int`: refers to integer -3, -2, -1, 0, 1, 2, 3

```
months = c(5, 6, 8, 10, 12, 16)
```

```
[1]  5  6  8 10 12 16
```

`num`: refers to all numbers from $-\infty$ to ∞ 1.0840991, 0.8431089, 0.494389, -0.7730161, 2.9038161, 0.9088839

```
weight = seq(3, 11, by = 1.5)
```

```
[1]  3.0  4.5  6.0  7.5  9.0 10.5
```

logi

Logical values can be TRUE (T) or FALSE (F)

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

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

logical vectors are often obtained from a comparison:

```
months > 12
```

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

chr and factor

chr: characters a, b, c, D, E, F

```
v_chr = c(letters[1:3], LETTERS[4:6])
```

```
[1] "a" "b" "c" "D" "E" "F"
```

factor: use numbers or characters to identify the variable levels

```
ses = factor(c(rep(c("low", "medium", "high"), each = 2)))
```

```
[1] low    low    medium medium high   high
Levels: high low medium
```

Change order of the levels:

```
ses1 = factor(ses, levels = c("medium", "high", "low"))
```

```
[1] low    low    medium medium high   high
Levels: medium high low
```

Create vectors

Concatenate elements with `c()`: `vec = c(1, 2, 3, 4, 5)`

Sequences:

```
-5:5  # vector of 11 numbers from -5 to 5
```

```
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
seq(-3, 3, by = 0.5)  # sequence in steps of 0.5 from -3 to 3
```

```
 [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3
```

Repeating elements:

```
rep(1:3, 4)
```

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

Create vectors II

```
rep(c("condA", "condB"), each = 3)

[1] "condA" "condA" "condA" "condB" "condB" "condB"

rep(c("on", "off"), c(3, 2))

[1] "on"  "on"  "on"  "off" "off"

paste0("item", 1:4)

[1] "item1" "item2" "item3" "item4"
```

Don't mix them up unless you truly want to

int + num → num

int/num + logi → int/num

int/num + factor → int/num

int/num + chr → chr

chr + logi → chr

Vectors and operations

Vectors can be summed/subtracted/divided and multiplied with one another

```
a = c(1:8)
a
```

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

```
b = c(4:1)
b
```

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

```
a - b
```

```
[1] -3 -1  1  3  1  3  5  7
```

If the vectors do not have the same length, you get a warning

Vectors and operations PT. II

The function is applied to each value of the vector:

```
sqrt(a)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.6
```

The same operation can be applied to each element of the vector:

```
(a - mean(a))^2  # squared deviation
```

```
[1] 12.25  6.25  2.25  0.25  0.25  2.25  6.25 12.25
```

Matrices and arrays

Create a 3 × 4 matrix:

```
A = matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
```

Label and transpose:

```
rownames(A) = c(paste("a", 1:3))  # colnames()
t(A)  # transpose matrix
```

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

Matrices and arrays

Matrix can be created by concatenating columns or rows:

```
cbind(a1 = 1:4, a2 = 5:8, a3 = 9:12)  # column bind
rbind(a1 = 1:4, a2 = 5.8, a3 = 9:12)  # row bind
```

Matrices and arrays

```
array(data, c(nrow, ncol, ntab))

my_array = array(1:30, c(2, 5, 3))  # 2 x 5 x 3 array

, , 1

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

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   23   25   27   29
```

Work with vectors, atrices, arrays

Index elements in vectors `vector_name[position]`

```
weight[2]  # second element in vector weight
weight[6] = 15.2  # replace sixth element of weight
weight[seq(1, 6, by = 2)]  # elements 1, 3, 5
weight[2:6]  # elements 2 to 6
weight[-2]  # without elemt 2
```

Logic applies as well:

```
weight[weight > 7]  # values greater than 7
weight[weight >= 4.5 & weight < 8]  # values between 4.5 and
```

Work with vectors, atrices, arrays II

Access elements in matrices: `matrix_name[row, column]`

```
A[2, 3]  # cell in row 2 column 3

A[2, ]  # second row

A[, 3]  # third column
```

Work with vectors, atrices, arrays III

Access elements in arrays `array_name[row, col, tab]`

```
my_array[2, 1, 3]  # cell in 2nd row 1st col of 3rd tab

my_array[, , 3]  # 3rd tab

my_array[1, , 2]  # 1st row in tab 2
```

Lists

Can store different objects (e.g., vectors, data frames, other lists):

```
my_list = list(w = weight, m = months, s = ses1, a = A)
```

The components of the list can be extracted with $ or [[]] and the name (or position) of the component:

Extract `months`:

```
my_list[["m"]]  # my_list$a
```

```
[1]  5  6  8 10 12 16
```

Extract `weight`:

```
my_list[[1]]  # my_list$months or my_list[['a']]
```

```
[1]  3.0  4.5  6.0  7.5  9.0 10.5
```

The king of data structure: data frames

Data frames are lists that consist of vectors and factors of equal length. The rows in a data frame refer to one unit:

```
id = paste0("sbj", 1:6)
babies = data.frame(id, months, weight)

babies

    id months weight
1 sbj1      5    3.0
2 sbj2      6    4.5
3 sbj3      8    6.0
4 sbj4     10    7.5
5 sbj5     12    9.0
6 sbj6     16   10.5
```

Working with data frames

Index elements in a data frame:

```
babies$months  # column months of babies
```

```
babies$months[2]  # second element of column months
```

```
babies[, "id"]  # column id
```

```
babies[2, ]  # second row of babies (obs on baby 2)
```

Logic applies:

```
babies[babies$weigth > 7, ]  # all obs above 7 kg
babies[babies$id %in% c("sbj1", "sbj6"), ]  # obs of sbj1 and
```

Working with data frames II

```
dim(babies)  # show the dimensions of the data frame
[1] 6 3
names(babies)  # variable names (= colnames(babies))
[1] "id"     "months" "weight"
View(babies)  # open data viewer

plot(babies)  # pariwise plot
```

You can use these commands also on other R objects

Working with data frames III

```
str(babies)  # show details on babies

'data.frame':   6 obs. of  3 variables:
 $ id    : chr  "sbj1" "sbj2" "sbj3" "sbj4" ...
 $ months: num  5 6 8 10 12 16
 $ weight: num  3 4.5 6 7.5 9 10.5

summary(babies)  # descriptive statistics

      id                 months          weight      
 Length:6           Min.   : 5.0   Min.   : 3.000  
 Class :character   1st Qu.: 6.5   1st Qu.: 4.875  
 Mode  :character   Median : 9.0   Median : 6.750  
                    Mean   : 9.5   Mean   : 6.750  
                    3rd Qu.:11.5   3rd Qu.: 8.625  
                    Max.   :16.0   Max.   :10.500  
```

You can use these commands also on other R objects

Sorting

`order()`:

```
babies[order(babies$weight), ]  # sort by increasing weight
```

```
    id months weight
1 sbj1      5    3.0
2 sbj2      6    4.5
3 sbj3      8    6.0
4 sbj4     10    7.5
5 sbj5     12    9.0
6 sbj6     16   10.5
```

```
babies[order(babies$weight,    # sort by decreasing weight
             decreasing = T), ]
```

Multiple arguments in `order`:

```
babies[order(babies$weight, babies$months, decreasing = TRUE), ]
```

Aggregating

Aggregate a response variable according to grouping variable(s) (e.g., averaging per experimental conditions):

```
# Single response variable, single grouping variable
aggregate(y ~ x, data = data, FUN, ...)

# Multiple response variables, multiple grouping variables
aggregate(cbind(y1, y2) ~ x1 + x2, data = data, FUN, ...)
```

Aggregating: Example

```
head(ToothGrowth)  # Vitamin C and tooth growth (Guinea Pigs)
```

```
   len supp dose
1  4.2   VC  0.5
2 11.5   VC  0.5
3  7.3   VC  0.5
4  5.8   VC  0.5
5  6.4   VC  0.5
6 10.0   VC  0.5
```

Aggregate across supplement and dose of Vitamin C:

```
aggregate(len ~ supp + dose, data = ToothGrowth, mean)
```

```
  supp dose   len
1   OJ  0.5 13.23
2   VC  0.5  7.98
3   OJ  1.0 22.70
4   VC  1.0 16.77
5   OJ  2.0 26.06
```

Reshaping: Long to wide

Data can be organized in wide format (i.e., one line for each statistical unit) or in long format (i.e., one line for each observation).

```
head(Indometh)  # Long format
```

```
  Subject time conc
1       1 0.25 1.50
2       1 0.50 0.94
3       1 0.75 0.78
4       1 1.00 0.48
....
```

Long to wide

```
# From long to wide
df.w <- reshape(Indometh, v.names = "conc", timevar = "time",
    idvar = "Subject", direction = "wide")

   Subject conc.0.25 conc.0.5 conc.0.75 conc.1 conc.1.25 conc
1        1      1.50     0.94      0.78   0.48      0.37   0.
12       2      2.03     1.63      0.71   0.70      0.64   0.
23       3      2.72     1.49      1.16   0.80      0.80   0.
34       4      1.85     1.39      1.02   0.89      0.59   0.
45       5      2.05     1.04      0.81   0.39      0.30   0.
56       6      2.31     1.44      1.03   0.84      0.64   0.
   conc.5 conc.6 conc.8
....
```

Reshaping: Wide to long

```
# From wide to long
df.l <- reshape(df.w, varying = list(2:12), v.names = "conc",
    idvar = "Subject", direction = "long", times = c(0.25, 0.5,
        0.75, 1, 1.25, 2, 3, 4, 5, 6, 8))

       Subject time conc
1.0.25       1 0.25 1.50
2.0.25       2 0.25 2.03
3.0.25       3 0.25 2.72
....

# reorder by subject
df.l[order(df.l$Subject), ]

       Subject time conc
1.0.25       1 0.25 1.50
1.0.5        1 0.50 0.94
1.0.75       1 0.75 0.78
....
```

Execersizes

- Open a new R script
- Create one vector for each type (`int`, `num`, `chr`, `logi`, `factor`) and assign each of them to an object
- Compute the mean of the `int` and `num` vectors
- Standardize the values of the `int` and `num` vectors and store them in two new objects:

$$z = \frac{x_i - \bar{X}}{sd}$$

- Create a new vector by combining together the `logi` and `int` vectors
- Add the `logi` vector to the `num` vector