R for Data Analysis in Agrobiodiversity 2

Data management

Recap

- We discussed about the structure of computers
- We saw some features of computer languages

- We discussed object-oriented programming in R
- We learnt about operators
- We learnt about data types
- We learnt about data classes
- We learnt about data subsetting

R objects may belong to a number of different classes

«integer»
«logical»
«character»
«numeric»
«factor»
[…]

Classes dictate what you can do on a specific object; you cannot multiply characters, you cannot search for strings in logicals, etc.

Data structures

Data loaded in R may assume different strucures, dipending on their dimensionality and features

- Vectors
- Matrices
- Dataframes
- Lists
- Arrays

Data indexing

- To access individual elements of data structures, we may use the square brackets operators: [ ]
- Depending on dimensionality, brackets may be used to slice different components of data using the comma separator, the colon operator, the dollar symbol, the dash symbol

1. Numeric indexing
2. Name indexing
3. Logical indexing

Functions in R

- In the past lecture, we briefly discussed functions

- Functions in R are used to perform complex tasks
- They are identified by:
  - A function name
  - A couple of brackets ()
  - One ore more arguments, separated by commas and included by brackets. Arguments are named, and their value may be assigned by an equal sign

- Functions work by taking an **input**, elaborating it through a certain set of **specific operations**, and producing an **output**
- Arguments are used to modulate the operations run by the function, as well as to indicate input and output
- The input, output, and arguments of a function, of course, may be objects.

- It all boils down to a complex interrelation of simple arithmetic / logical operations that R is able to run.
- Functions are buildings, operations are bricks

Argument *data*

matrix(data= paste(“entry”, 1:15, sep=“”), nrow=5, ncol=3)

Argument *nrow*

Argument *ncol*

Another function (its result may have been put into an object)

Function name

```
> matrix(data = paste("entry", 1:15, sep=""), nrow=5, ncol=3)
     [,1]     [,2]      [,3]
[1,] "entry1" "entry6"  "entry11"
[2,] "entry2" "entry7"  "entry12"
[3,] "entry3" "entry8"  "entry13"
[4,] "entry4" "entry9"  "entry14"
[5,] "entry5" "entry10" "entry15"
```

Two different functions here:

- *matrix()*
  - Takes a number of rows, number of columns, a content, and produces a matrix with those
- *paste()*
  - Concatenates strings and values of choice using a specific separator to create a new set of strings

Help on functions

- If you forget how a function works, or if you want a few more informations about its compoments, you may type

  ?function.name

- This will open your internet browser with an explanatory page

R: Matrices

127.0.0.1:11101/library/base/html/matrix.html

matrix {base} R Documentation

Function name and the {package} where it belongs

Matrices

Description

`matrix` creates a matrix from the given set of values.

`as.matrix` attempts to turn its argument into a matrix.

`is.matrix` tests if its argument is a (strict) matrix.

Brief description of the function and those related to it

Usage

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

as.matrix(x, ...)
## S3 method for class 'data.frame'
as.matrix(x, rownames.force = NA, ...)

is.matrix(x)
```

How to use it, with a list of arguments

The three points stand for «additional arguments» not listed here and borrowed from other related functions

Arguments

Detailed explanation of the arguments expected by the function (see «Usage» above)

`data`
: an optional data vector (including a list or `expression` vector). Non-atomic classed R objects are coerced by `as.vector` and all attributes discarded.

Hyperlinks will redirect you to other help pages

`nrow`
: the desired number of rows.

`ncol`
: the desired number of columns.

`byrow`
: logical. If `FALSE` (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.

`dimnames`
: A `dimnames` attribute for the matrix: `NULL` or a `list` of length 2 giving the row and column names respectively. An empty list is treated as `NULL`, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.

`x`
: an R object.

`...`
: additional arguments to be passed to or from methods.

`rownames.force`
: logical indicating if the resulting matrix should have character (rather than `NULL`) `rownames`. The default, `NA`, uses `NULL` rownames if the data frame has 'automatic' row.names or for a zero-row data frame.

Details

Verbose description of the function

If one of `nrow` or `ncol` is not given, an attempt is made to infer it from the length of `data` and the other parameter. If neither is given, a one-column matrix is returned.

If there are too few elements in `data` to fill the matrix, then the elements in `data` are recycled. If `data` has length zero, `NA` of an appropriate type is used for atomic vectors (`0` for raw vectors) and `NULL` for lists.

`is.matrix` returns `TRUE` if `x` is a vector and has a "`dim`" attribute of length 2 and `FALSE` otherwise. Note that a `data.frame` is **not** a matrix by this test. The function is generic: you can write methods to handle specific classes of objects, see InternalMethods.

`as.matrix` is a generic function. The method for data frames will return a character matrix if there is only atomic columns and any non-(numeric/logical/complex) column, applying `as.vector` to factors and `format` to other non-character columns. Otherwise, the usual coercion hierarchy (logical < integer < double < complex) will be used, e.g., all-logical data frames will be coerced to a logical matrix, mixed logical-integer will give a integer matrix, etc.

The default method for `as.matrix` calls `as.vector(x)`, and hence e.g. coerces factors to character vectors.

When coercing a vector, it produces a one-column matrix, and promotes the names (if any) of the vector to the rownames of the matrix.

`is.matrix` is a primitive function.

The `print` method for a matrix gives a rectangular layout with dimnames or indices. For a list matrix, the entries of length not one are printed in the form `integer,7` indicating the type and length.

Note

If you just want to convert a vector to a matrix, something like

```
dim(x) <- c(nx, ny)
dimnames(x) <- list(row_names, col_names)
```

will avoid duplicating `x`.

Note

Any additional info that may be needed

If you just want to convert a vector to a matrix, something like

```
dim(x) <- c(nx, ny)
dimnames(x) <- list(row_names, col_names)
```

will avoid duplicating x.

References

Who did the job to make it work

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

A list of related functions

data.matrix, which attempts to convert to a numeric matrix.

A matrix is the special case of a two-dimensional array.

Examples

A few easy usage examples

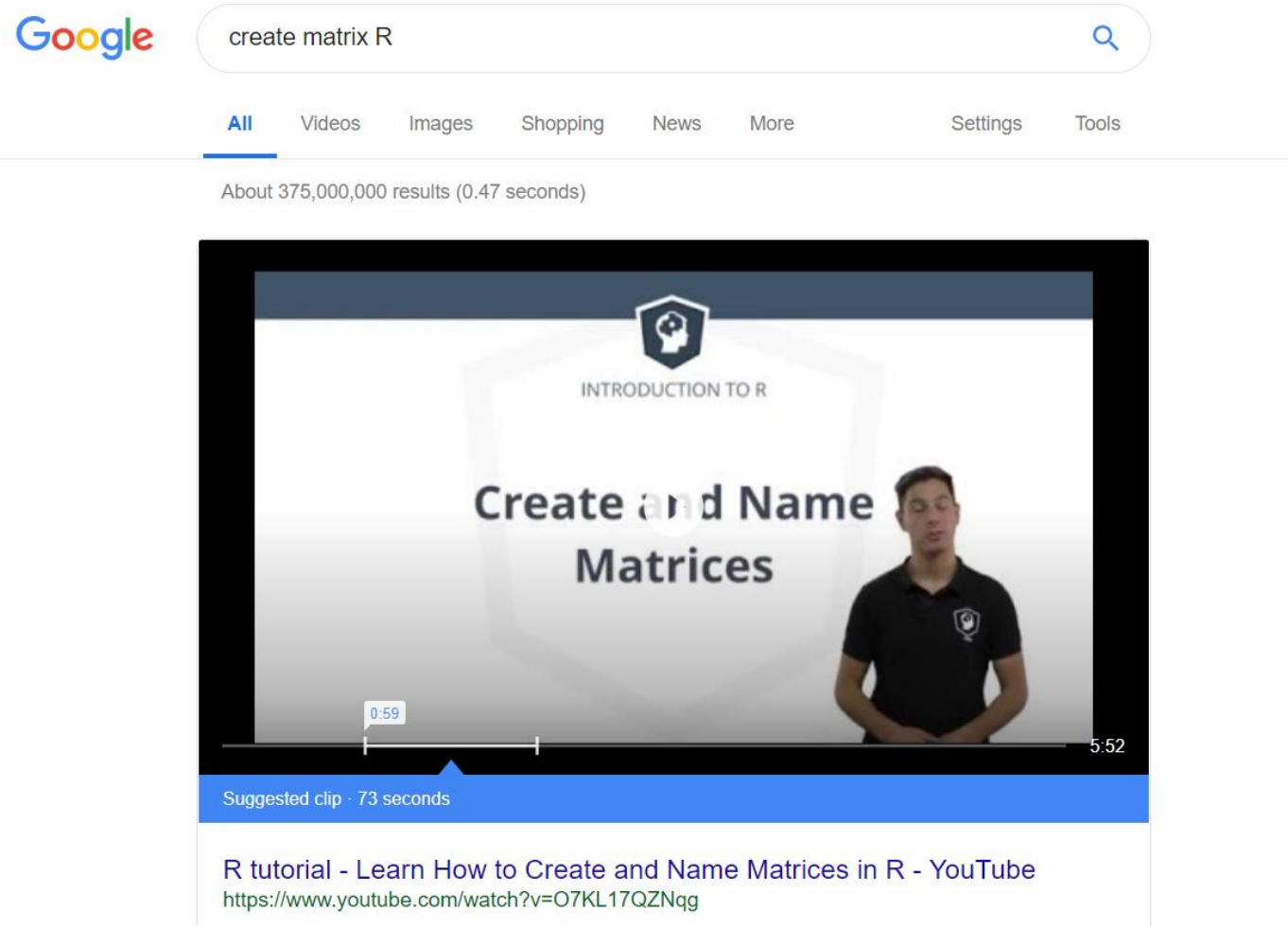
```
is.matrix(as.matrix(1:10))
!is.matrix(warpbreaks)  # data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,])  # using as.matrix.data.frame(.) method

## Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
               dimnames = list(c("row1", "row2"),
                               c("C.1", "C.2", "C.3")))
mdat
```

Google is your friend

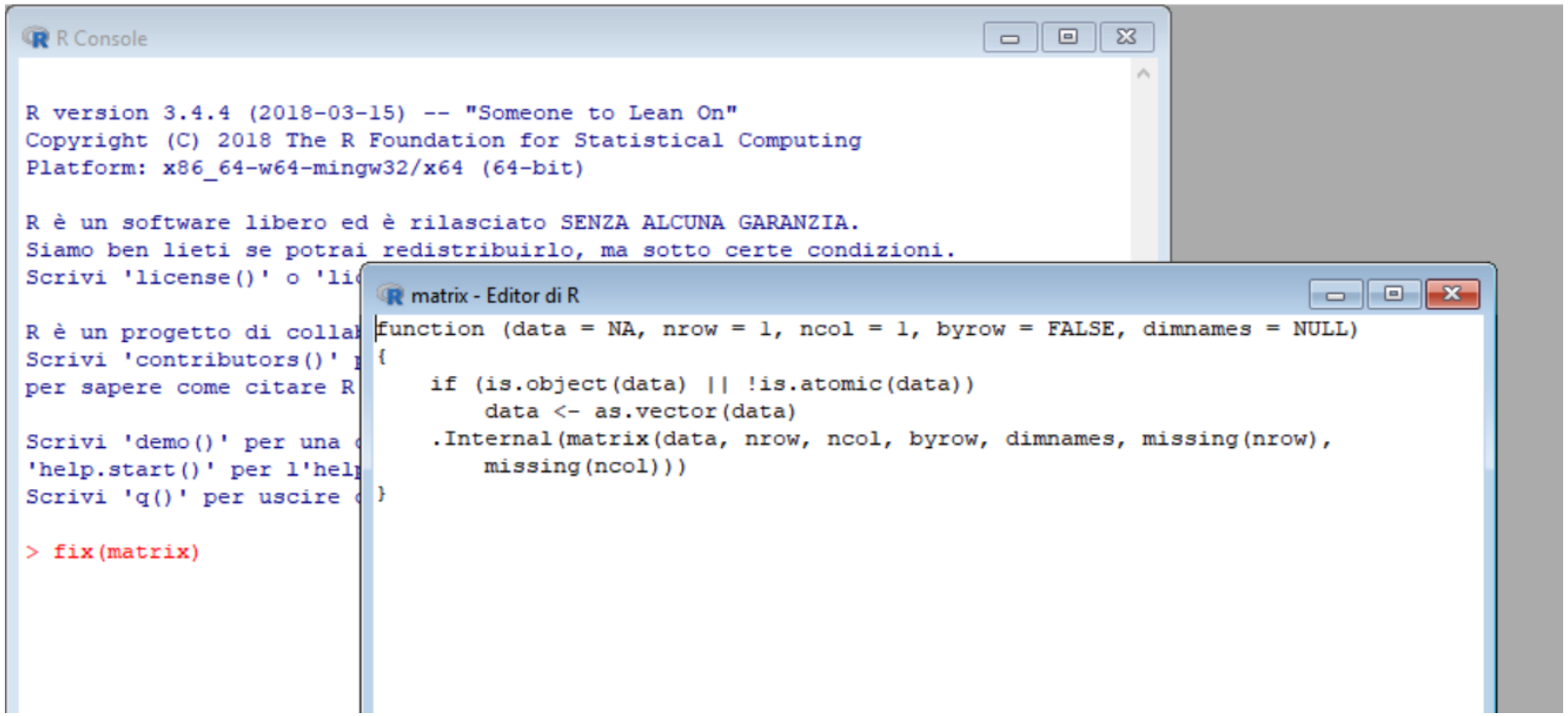
- There is no shame in googling things

Knowing how to make questions may be more valuable that knowing answers



What’s inside functions?

To know how a function is made, you may use the *fix()* function. It will reveal the internal structure of any function and even **allow you to modify it** in text edit



- Functions may be stated by the following synthax

A name is given

We specify that this object is to be intended as a function. Expected arguments are given

```
example.function <- function(arg1, arg2){
    res<-arg1 * arg2
    return(res)
}
```

Operations are listed

This specifies the output value

Indentation is not mandatory, but eases your reading

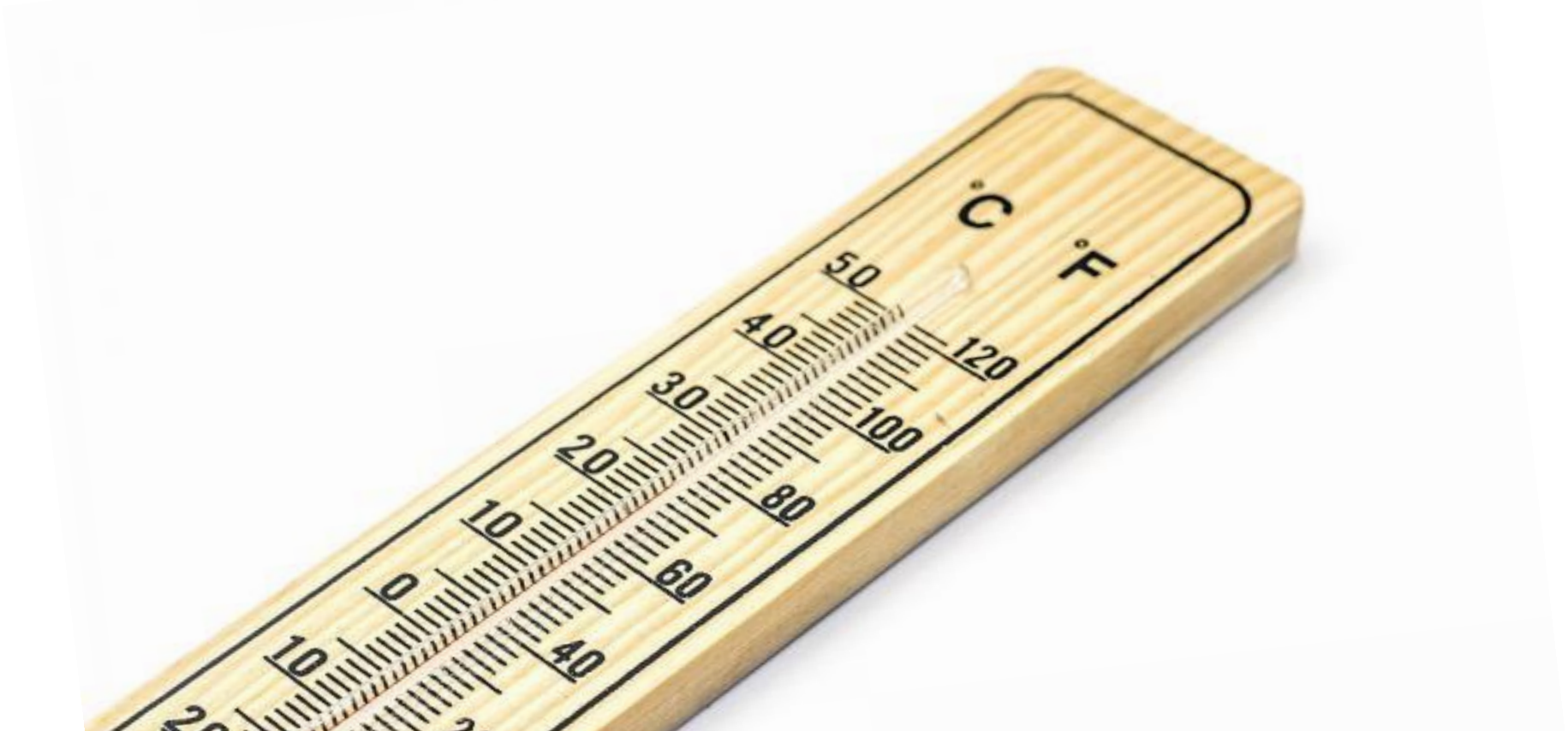
Curly braces precede and follow the operations performed by the function

So that example.function(2, 3) will result in the value 6

Let’s say that I want to convert between °C and F

In Farenheit, water freezes at 32 F and boils at 212 F

In Celsius, water freezes at 0 °C and boils at 100 °C



- We may use the difference b/w boiling and freezing in the two scales;
  - 100-0=100 for °C
  - 212-32=180 for F

- We then get a ratio among scales;
  - 100/180= 0.5555556 °C for each F

- We then may convert F to °C considering the ratio and offset among scales;
  - C° = (F-32) * 0.5555556

How many °C are 73 F?

```
>(73-32)*0.5555556
[1] 22.77778
```

We can make things clearer using objects

```
>F <- 73
>ratio <- 100/180
>C <- (F-32)*ratio
>C
[1] 22.77778
```

We may then think of creating a function doing the conversion in a consistent way

```
f.to.c<- function(x) {
    outc<-(x-32)*(100/180)
    return(outc)
}
```

In this case, the object may be a number as well as an atomic vector

°C can be easily converted in K by adding 273.15

```
c.to.k<- function(x) {
    outk <- x+273.15
    return(outk)
}
```

Functions may be wrapped one in another

```
f.to.k<- function(x) {
      outc<-f.to.c(x)
      outk<-c.to.k(outc)
      return(data.frame(C=outc, K=outk))
}
```

We can now easily convert to C and K the temperature in Anchorage, AK, during Christmas day 2018

Min	6.8 F
Average	23 F
Max	44.6 F

```
>akf<-c(6.8,23, 44.6)
>ack<-f.to.k(akf)
> ack
    C      K
1 -14 259.15
2  -5 268.15
3   7 280.15
```

Some good advice

Tim "Agile Otter" Ottinger
@tottinge

The reason for writing a function is not to reuse its code, but to name the operation it performs.

29 6:44 PM - Jan 22, 2013

Gustavo Rod. Baldera
@grodbaldera

"The name of a variable, function, or class, should answer all the big questions." - Uncle Bob Martin, Clean Code

3 3:15 PM - Apr 24, 2013

M Butcher
@technosophos

If you've written a function whose body is 2,996 lines of code, you're doing it wrong.

9 5:55 PM - Apr 11, 2013

R packages

- R packages are collections of functions assembled by R users and shared with the community
- They are not loaded by default
- They are not natively installed in the R interpreter
- R packages are free and available at CRAN - Comprehensive R Archive Network
- While CRAN is the official repository, other repository exist: bioconductor, GitHub, etc

Get R packages

- To get an R package from CRAN, you will need to type in

  install.packages(“package name“, dependencies==T)

  - Do not forget quotes
  - Packages may be depending one on another, so setting the argument dependencies as TRUE make sure you will install all you need
  - You need to run the installation only once (for each R installation)
  - When installing, you will be asked to select a mirror (=download point); choose the closest one and you are all set

Once an R package is installed, you may load it using:

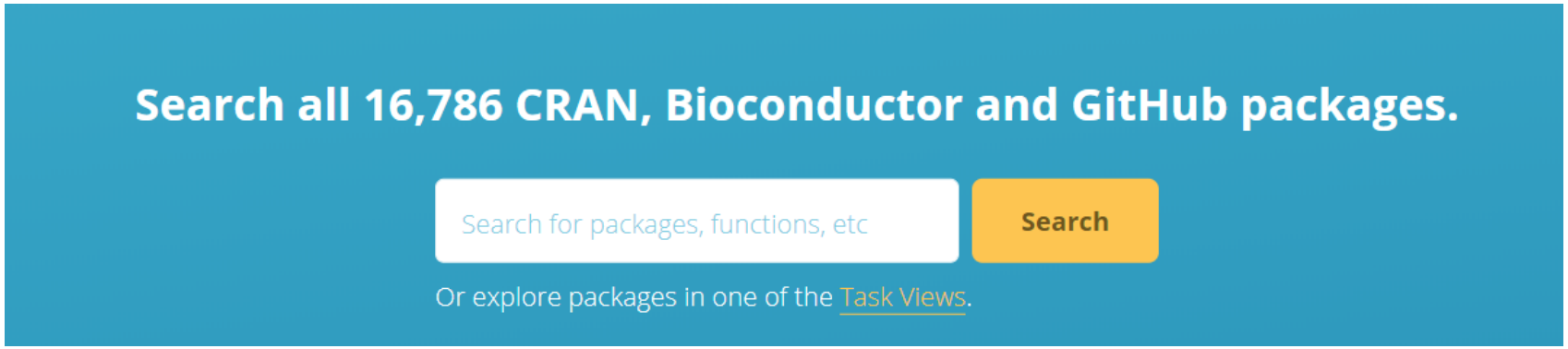
library(package name)

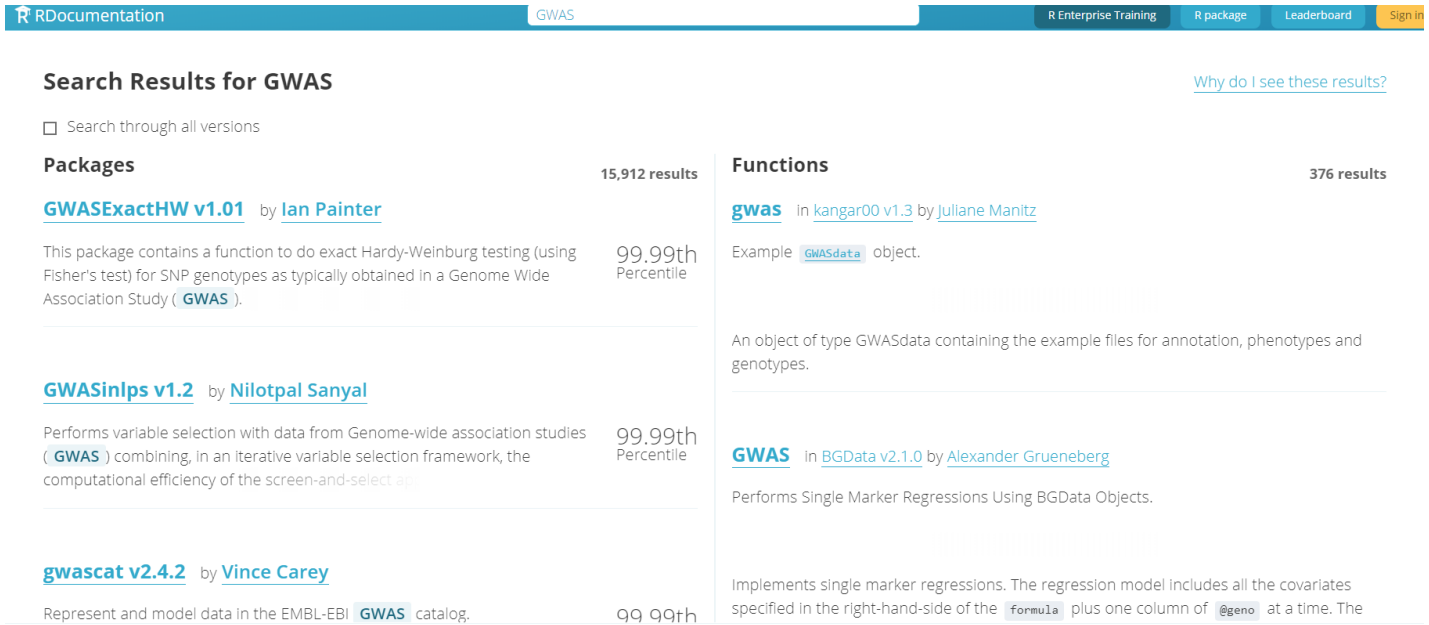
- No quotes needed this time
- Needs to be run in every new R session in which you need that specific package
- You may also use the notation *require(package name)* even if this will provide a logical value (T/F) reporting the presence of that package on your installation. If a package is not found, *library()* will trow an error whilte *require()* will attach a FALSE value and continue

In case you don’t need the entire package, you may load only selected functions/datasets using the notation

packagename::functionname()

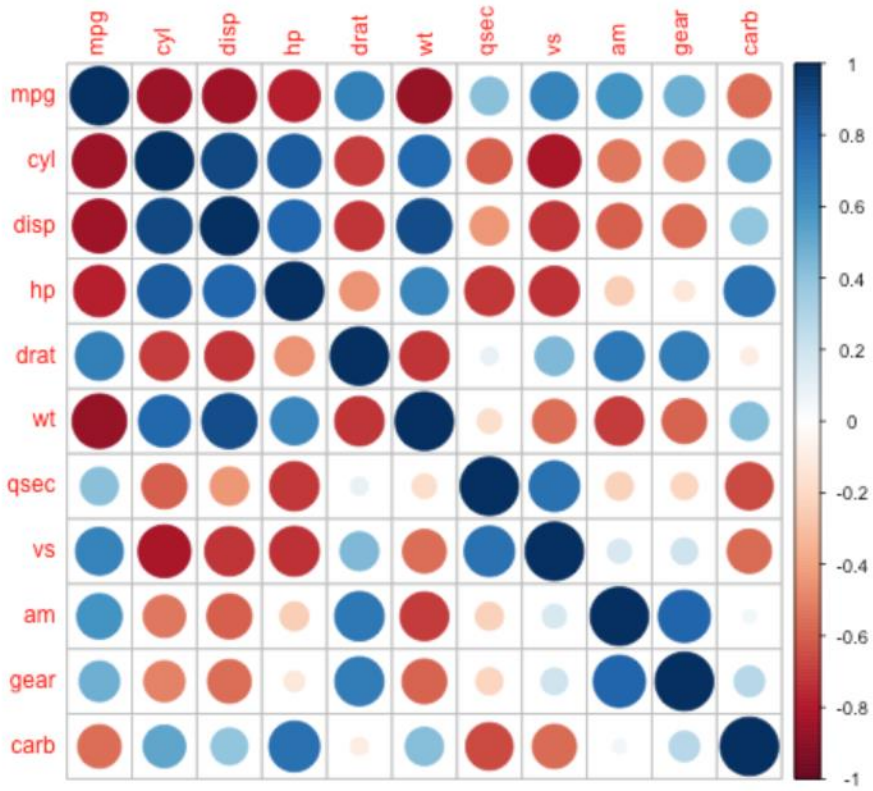
Nice places to look for R packages are literature and Google. If you are at loss, you may also look into rdocumentation.org



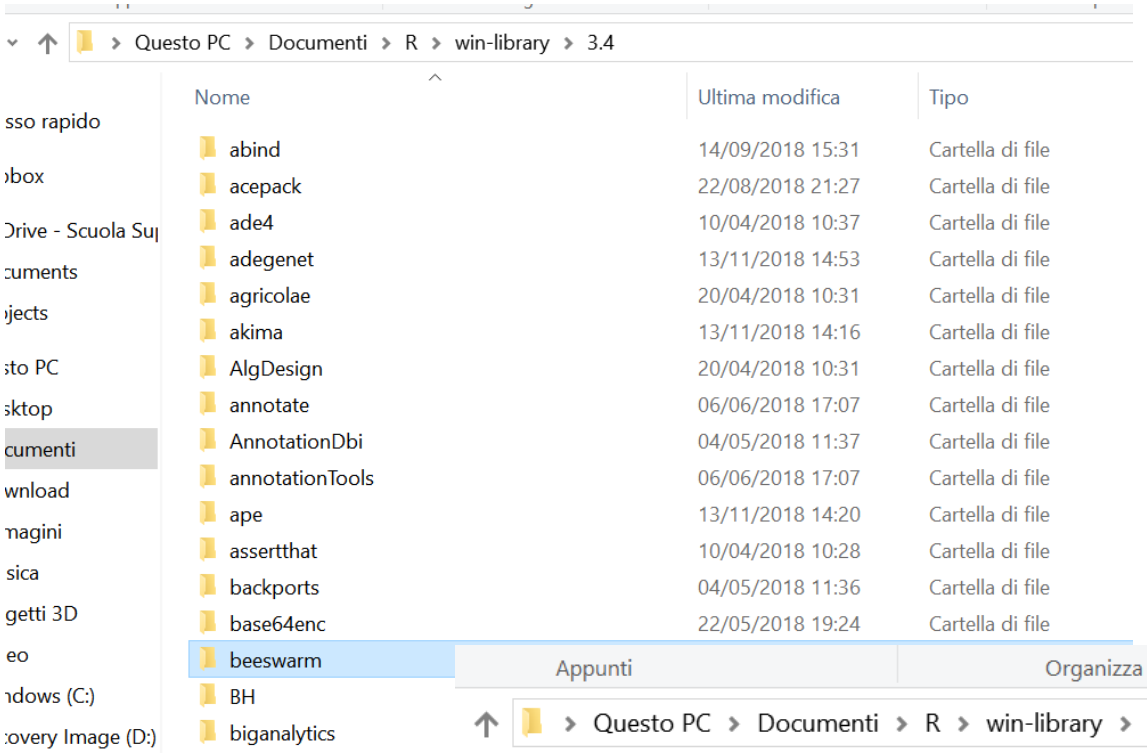


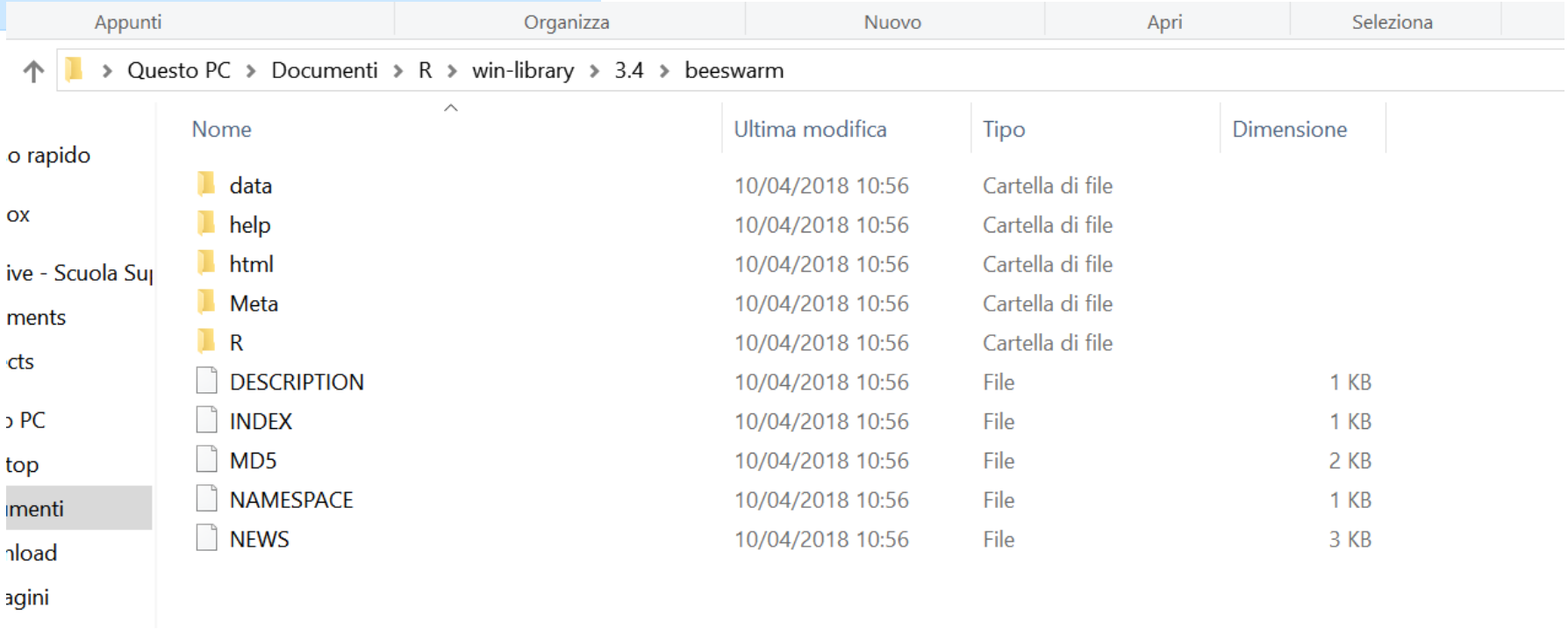
Try it

Install and load the package "corrplot"



R library structure on your PC





The R workspace

- Every object(variable) that you create will be stored in the **workspace** (also named environment)
- Objects remain untouched in the workspace until you
  - Overwrite them
  - Remove them using *rm()*
- At any given moment, you can list the object available in the workspace using the *ls()* function
- Functions inclueded in loaded packages are also temporarily stored in the workspace
- When you terminate your session, all your workspace will be permanently lost, unless you save it

Every once in a while, it is good to save your workspaces in order to facilitate a restart. You may do so by:

save.image(file=“filename”)

This will save the entire workspace. Not the packages that are loaded!

save(object, file=“filename”)

This will save object(s) of your choice.

In both cases, the extension tipically used is *.Rdata*

These files will be saved in the working directory

Of course, you may also save R scripts and functions.

For both, the typical file extension is «.R»

The working directory

- R operations on the disk drive are bound the the working directory
- Any file saved or loaded by R resides in a specific location on the hard disk
- R must be pointed to the correct directory to perform the desired disk operations

getwd()

Will provide you the current working directory*;*

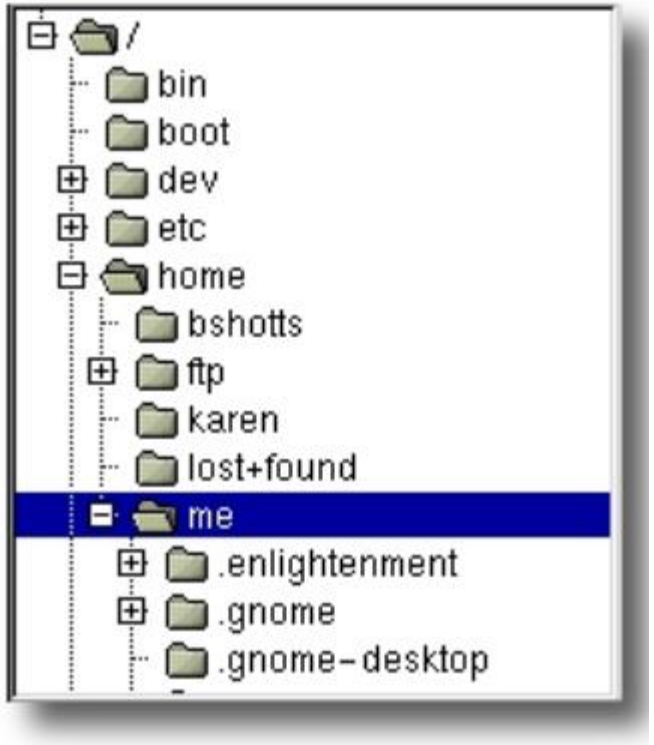
setwd(“path to directory”)

Will allow you to choose a working directory

dir(“parth to directory”)

Will list the contents of a given directory

Navigation through folders(directories)



- The file system follows a **hierarchical folder structure**
- Folders contain data and other folders, and are in turn contained by other folders
- Command line interfaces (including R) allow you to navigate verticallt across hierarichy but not to have an overall vision

- From the current directory, you can only navigate forward or backwards

By default, files are written and loaded in the working directory; it is however possible to specify other directories

```
> getwd()
 [1] "Documents"
```

One dot means current directory

```
> dir(".")
 [1] "30657_fastqc.html"          "Adobe"
 [3] "AdobePhotoshopPortable"     "Altezza2.txt"
 [5] "Custom Office Templates"    "desktop.ini"
```

Two dots mean one step backward (the previous directory in the hierarchical structure)

```
> dir("..")
 [1] "3D Objects"
 [2] "AppData"
 [3] "Contacts"
 [4] "Documents"
```

In order to move forward, you need to specify where to go

```
> dir("./GIS DataBase/")
[1] "spearfish60"
```

R can write and read files, as well as create directories

dir.create(«path to folder»)

File and Directory Manipulation

Description

These functions provide a low-level interface to the computer's file system.

Usage

```
file.create(...)
file.exists(...)
file.remove(...)
file.rename(from, to)
file.append(file1, file2)
file.copy(from, to, overwrite = FALSE)
file.symlink(from, to)
dir.create(path, showWarnings = TRUE, recursive = FALSE)
```

Arguments

`..., file1, file2, from, to`	character vectors, containing file names.
`path`	a character vector containing a single path name.
`overwrite`	logical; should the destination files be overwritten?
`showWarnings`	logical; should the warnings on failure be shown?
`recursive`	logical: should elements of the path other than the last be created? If true, like Unix's `mkdir -p`.

Something that will happen very often:

If you get errors that sound like "cannot change the working directory” or “probable reason 'No such file or directory” it is very likely that you **misspelled file names or paths**

- You may use tab for autocompletion
- Watch out as windows uses “\” instead of “/” in file paths
- Be consistent with lower and upper letters
- Do NOT use spaces in filenames

  “my Favourite File.txt” should really be “my_favourite_file.txt”

Scripting

- Commands may be given to the R console on the fly or by scripting
- In most cases you will deal with **scripts**, that is collection of commands in a sequential order
- Scripts are the recipes, or algorithms, that our analyses follow
- Scripts are good!
  - Consistency
  - Clarity
  - Reproducibility
  - Sharing

Tipically, scripts follow a few conventions

```
maindir<-"C:/Users/admin/OneDrive - Scuola Superiore Sant'Anna/projects/MAIZE_ghent/analyses/7.RNA.pheno.cor"
setwd(maindir)

#check correlation among traits
library(corrplot)
pheno<-read.delim("allpheno.txt", header=T)

#subset to interesting traits
pheno<-pheno[,c(6,10,5,18)]
cr<-cor(pheno, use="complete.obs")

pdf("correlation.among.phenotypes.pdf")
    #corrplot(cr, method="color", diag=F, tl.col="black", tl.pos="d")
    corrplot.mixed(cr, lower.col = "black", tl.cex=1.2, tl.col="black", number.cex=1.5,  upper="pie")
dev.off()
```

- We start setting the working directory
- We load all needed packages
- Each line is an independent instruction
- «#» precede comments; anything writtien after them is NOT run as a command
  - Comments a re VERY important; spend some time to make them as informative as you can
  - You may also use it to temporarly disable lines of code
- Indentation is not mandatory; it is however very useful to highlight multi-line instructions or even nested processes
- You may use as many blank lines to separate blocks of code

Scripts are stored on the computer an are tipically saved with the .R extension

Scripts can be run line by line (best for scripting and debugging) or, when consolidated, as an undivided collection of code

source(«script name»)

This automation is especially useful when launching R routines externally

File manipulation

Remember: file format is not the same as file content. The same text file may be saved with a number of different formats: txt, R, doc, docx, html, …

- R is able to read a lot of different file formats; it is very important to tell R which
- You will mostly deal with tabular data (raster images too can be considered tables filled with numerical values)
- First of all, we need to see where the file that we are looking for is located (by default, R will look in the working directory)
- R will throw and error if the file cannot be found, this is very common. If you are sure the file is there, look for typos and special characters in the path/file name

Loading R files

- The easiest way to go is to use R data files
- These are binary (=compressed) files that can be natively loaded in R using

load(«file»)

- The content of the file will be loaded in the workspace and objects will mantain their original names until reassigned

Loading non-R files

- Files that are not natively R may be loaded with a bunch of link functions
- The most used one is read.table()
  - Presumes space-separated fields, one line per row
  - Main argument is the file name or URL
  - Returns a dataframe
  - Lots of options to specify field separator, column names, forcing or guessing column types, skipping lines at the start of the file….(see help)

Special cases are

read.delim()

read.csv()

That are wrapper functions for read.table defaulting tab separation ("\t") or comma separation (",")

When called, these functions need to state an object in which to transfer the table content, otherwise the result of the function will be simply printed on screen

Format conversion (R/non R) is often tricky, and comes with a number of known pitfalls;

- The region of your operating system will make a difference between «.» and «,» as decimal separators; make sure R is interpreting them correctly (it may be set in options)
- In European computers, CSV are really semicolon separated values («;»)
- Look for trailing white spaces in your tables, as they may be loaded as (blank) values or cause the read functions to fail
- End of line characters are different in Windows and OS/Linux
- Data frames and matrices in R have separated column names and row names; this is not the case of tabular formats. When reading a file, make sure to correctly set the *header* and *row.names* attributes

Load files from URLs

Tables may be loaded from the web using file URLs

```
> url<-"http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv"
> adv<-read.table(url, sep=",", header=T)
> head(adv)
  X    TV radio newspaper sales
1 1 230.1  37.8      69.2  22.1
2 2  44.5  39.3      45.1  10.4
3 3  17.2  45.9      69.3   9.3
4 4 151.5  41.3      58.5  18.5
5 5 180.8  10.8      58.4  12.9
6 6   8.7  48.9      75.0   7.2
```

Same rules as local files apply (separators, headers, etc)

Excel/spreadsheets

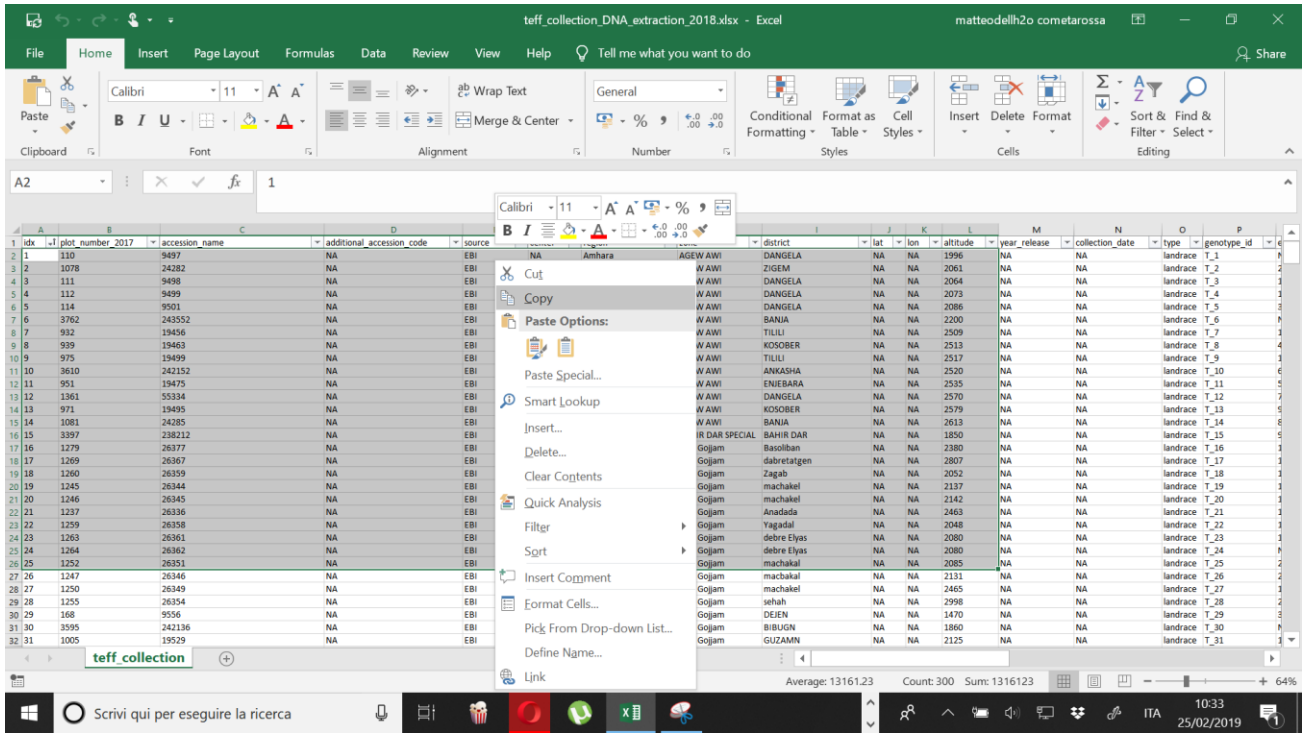
- Please, stop using Excel for your data analysis. Just stop.

- Spreadsheets look like dataframes (different type of data)
- In reality, spreadsheets are full of ugly irregularities
  - Values or formulas?
  - Headers, footers, side-comments, notes
  - Multiple sheets
  - Misassigned data type
  - Columns that change meaning half-way down
- It is always preferred to manage data in simpler formats such as txt or CSV

If you really can’t avoid it

- From Excel, save your files are either txt or csv, then read them with *read.table()* or *read.csv()*
- Use the *read.xls()* function from R/gdata (requires perl interpreter)
  - Be very aware of the multiple sheets to choose from; use the «sheet» argument to specify which
- Check out other packages such as R/readxl

- read.table() is also able to read what you saved in your clipboard (by either copy or ctrl+C)



read.table(«clipboard»)

Data analysis is perfomed on table values, not formats; when using Excel, think of what you will need in R downstream

Avoid using spaces in strings

Use clear naming

Colors and formatting may be converted in dummy variables

Be careful with uppercase & lowercase

Don't leave missing cells; use NA for missing values

Use consistent notation

Plot Number	Accession name	XY	Center	Center_2	region
110	9497	Ciccio	EBI		Amhara
1078	24282	ciccio	EBI		Amhara
111	9498	ciccio	EBI		Amhara
112	9499	NA	EBI		Amhara
114	9501		EBI		Amhara
3762	243552		EBI		Amhara
932	19456	**NA**	EBI		Amhara
939	19463	**NA**	EBI		Tigray
[illegible]	[illegible]	**Ciccio**	EBI		Tigray
3610	242152	NA	EBI		Tigray
951	19475	NA	EBI		Tigray
1361	55334	NA	EBI	X	Tigray
971	19495		EBI	Y	Tigray
1081	24285		EBI		Tigray
3397	238212	Ciccio	EBI		Amhara
[illegible]	[illegible]	?	EBI		[illegible]
1269	26367	?	EBI		Amhara
1260	26359		EBI		Amhara
1245	26344		EBI	X	Amhara
1246	26345		EBI	X	Amhara

AVOID: `?`, `$`, `%`, `^`, `&`, `*`, `(`, `)`, `-`, `#`, `?`, `,`, `<`, `>`, `/`, `|`, `\`, `[`, `]`, `{`, and `}`

File writing

- Once you want to output the result of your job, you may use the native R file saving that we already seen
  - save(object, file=«filename.Rdata»)
  - save.image(file=«filename.Rdata»)

- Pros:
  - Better space usage
  - Faster load/save
  - Consistent object naming
- Cons:
  - Compatibility with other programs

To write a table compatible with other software, you may use *write.table()*

Description

`write.table` prints its required argument `x` (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")

write.csv(...)
write.csv2(...)
```

Arguments

`x`
: the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce `x` to a data frame.

`file`
: either a character string naming a file or a connection open for writing. `""` indicates output to the console.

`append`
: logical. Only relevant if `file` is a character string. If `TRUE`, the output is appended to the file. If `FALSE`, any existing file of the name is destroyed.

`quote`
: a logical value (`TRUE` or `FALSE`) or a numeric vector. If `TRUE`, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If `FALSE`, nothing is quoted.

`sep`
: the field separator string. Values within each row of `x` are separated by this string.

`eol`
: the character(s) to print at the end of each line (row). For example, `eol = "\r\n"` will produce Windows' line endings on a Unix-alike OS, and `eol = "\r"` will produce files as expected by Excel:mac 2004.

`na`
: the string to use for missing values in the data.

`dec`
: the string to use for decimal points in numeric or complex columns: must be a single character.

`row.names`
: either a logical value indicating whether the row names of `x` are to be written along with `x`, or a character vector of row names to be written.

`col.names`
: either a logical value indicating whether the column names of `x` are to be written along with `x`, or a character vector of column names to be written. See the section on 'CSV files' for the meaning of `col.names = NA`.

`qmethod`
: a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of `"escape"` (default for `write.table`), in which case the quote character is escaped in C style by a backslash, or `"double"` (default for `write.csv` and `write.csv2`), in which case it is doubled. You can specify just the initial letter.

`fileEncoding`
: character string: if non-empty declares the encoding to be used on a file (not a connection) so the character data can be re-encoded as they are written. See `file`.

`...`
: arguments to `write.table`: `append`, `col.names`, `sep`, `dec` and `qmethod` cannot be altered.

In most cases, to make it work with Excel

write.table(object, file=«filename.txt», quote=F, row.names=F, sep=«\t»)

Non-tabular formats

In some cases you may still need to read/write files in odd formats, including plain text

```
> filename <- “filename.txt"
> paste(readLines(filename),collapse=" ")
```

Open the connection to a file, write some lines and then close the connection

```
>fileConn<-file("output.txt")
>writeLines(c("Hello","World"), fileConn)
>close(fileConn)
```

You may also use prepacked functions, as *read_file()* in R/readr

Error messages in R

R gives you two types of feedback when something is wrong:

Errors - R is saying "no way I am going to do this". The chunk of code that you provided cannot be executed

Warnings – R is saying "yeah, sure, you are the boss but maybe you won't like what you are going to get". The chunk of code is executable, but is behaving unexpectedly

There are three parts in any error message

- The declaration that this is an error
- The location of the error
- The problem in the code

Errors messages are very sinthetic and general; dont' expect R to tell you how to solve your problem

```
Error in model.frame.default(formula = y ~ female + DNC + SE_region +  :
  could not find function "function (object, ...) \nobject"
```

```
"Error in if (d &lt; delta) { : missing value where TRUE/FALSE needed"
```

```
Error in eval(expr, envir, enclos) : object '1005_at' not found
```

```
Error in file(file, "rt") : cannot open the connection
```

These are a few common errors:

1. "cannot open". You are trying to read a file that doesn't exist or can't be accessed
2. "could not find function". Most likely a typo, or forgetting to load the needed package
3. "subscript out of bounds". You are trying to access an element or dimension that doesn't exist
4. "unexpected character". Most likely a typo
5. "error in eval". You are referring to objects that don't exist
6. "no applicable method". There is a conflict b/w a function and a type of data not supported

Warnings are perhaps more tricky, as may depend from many different causes

```
> cor( c( 1 , 1 ), c( 2 , 3 ) )
[1] NA
Warning message:
In cor(c(1, 1), c(2, 3)) : the standard deviation is zero
```

Very often you may get warnings from packages which are very specific to the functions implemented there; their clarity depend on the completeness of the enclosed documentation

A sensible approach is to consider all warings as errors: to do so, set *options(warn = 2)* at the beginning of your script

Recap exercise

Rookies:

- Create a data frame with two columns and three rows, containing numbers from 6 to 1
- Save it as a text file using your separator of choice (watch out for quotes, row names, etc)
- Load it back in your workspace, then save it in R format, then again load it

Pro users:

- Create a function converting meters to feet (1mt = 3.28 ft). Use it to get the height of Mount Serra in feet.

Recap exercise 2

Rookies:

- Do the following making a stand-alone R script, annotate it
- Set working directory
- Load advertising data from

https://vincentarelbundock.github.io/Rdatasets/csv/Ecdat/Forward.csv

- Save it as an R file
- Run the R script from the terminal

Pro users:

- Use the file above. Load it and create a function converting Euro to BP using the approriate conversion rates; plug in a selector that will choose the line of the file with the conversion rate you want to use.