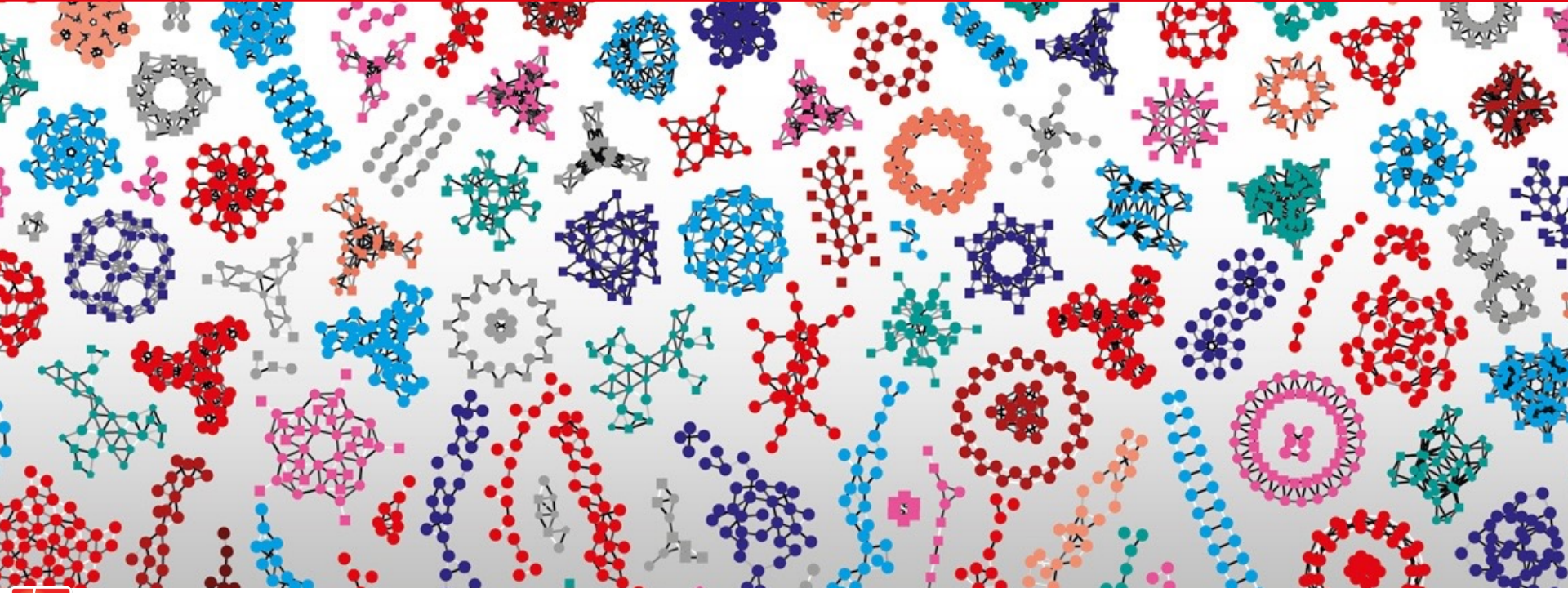


SIB
Swiss Institute of Bioinformatics

First steps with R in Life Sciences: Introduction

Diana Marek & Thomas Junier

-- with slides from Wandrille Duchemin, Leonore Wigger, Diana Marek

General Information

Course page: https://github.com/sib-swiss/first-steps-with-R-training

- Slides
- Data Sets
- Exercises
- solutions

Optional exam, 0.5 ECTS value

Asking questions - communicating

- Raise your hand anytime

Red sticky note

- Done with an exercise ?

green sticky note

Introducing Ourselves

Your experience:
3 quick polls

What type of computer OS are you using for this course?

A) Windows PC

B) Mac

C) Unix/Linux (Redhat, Ubuntu, etc)

Do you know other programming languages?

A) None

B) Yes, but I have only notions

C) Yes, I work regularly with at least one other language

What is your experience with R?

A) I've never tried to use R (apart from installing it).

B) I've tried to run a few commands or do a few exercises.

C) I've used R a little for work or personal projects.

D) I've used R extensively (wait, why am I even here?).

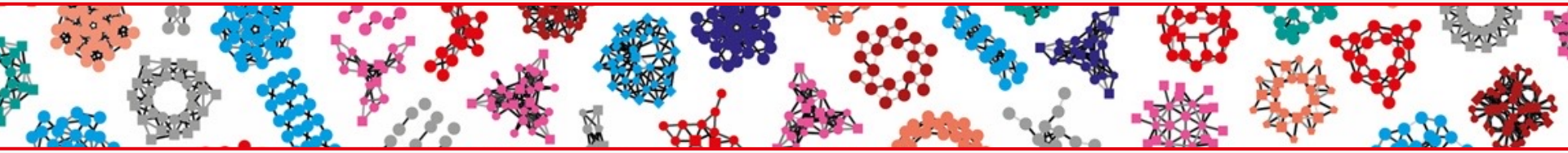
Course Content

R is vast and can't be learned overnight. The scope of this course:

- basic understanding and concepts behind R
- implement and interpret a data analysis workflow

This course is only the first step in your  journey

Outline



Day 1

01 What is R? Introduction

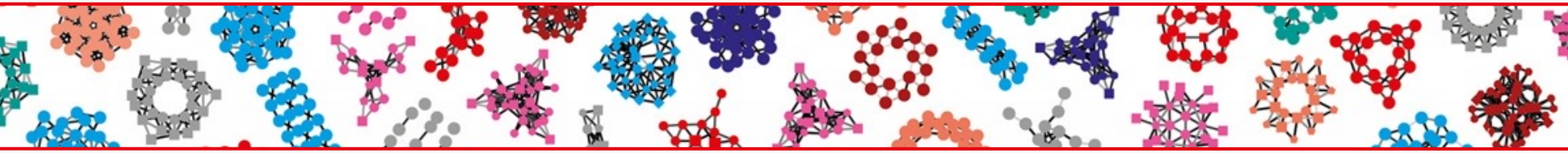
02 Getting familiar with R and the RStudio environment

03 Getting started with R syntax and objects

04 Formatting your data

05 Importing/exporting data with R

Outline

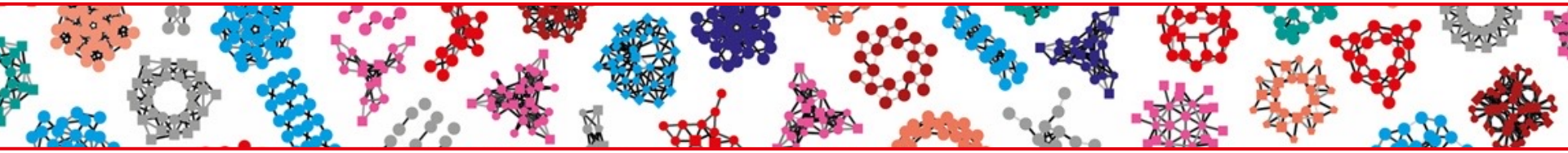


Day 2

06 **Building graphics in R (basic plotting)**

07 **Starting with statistics in R (hypothesis testing, simple linear regression)**

Examples and exercises are integrated in the chapters

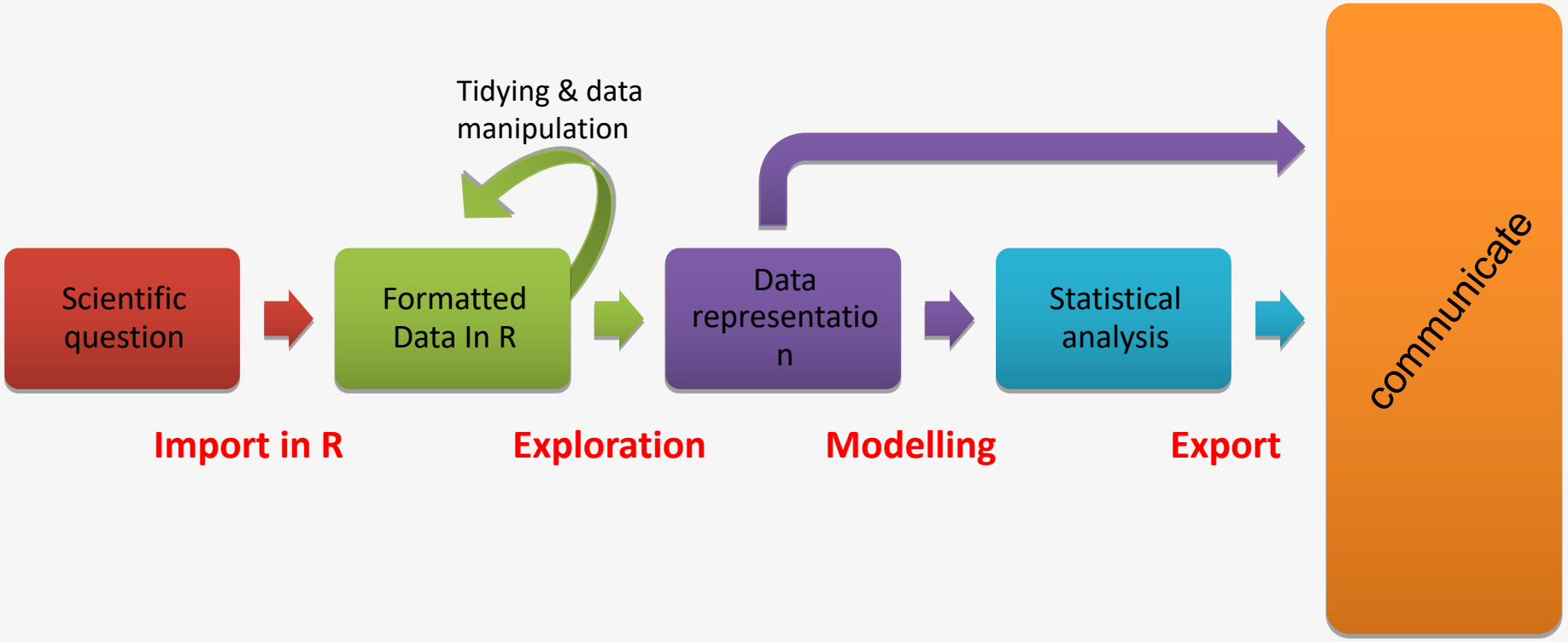


01 What is R? Introduction

What is R?

- R is a programming language and an environment for statistical computation and graphics.
  - A simple development environment with a console and a text editor
  - Facilities for data import, manipulation and storage
  - Functions for calculations on vectors and matrices
  - Large collections of data analysis tools
  - Graphical tools

Taking Advantage of R For Your Work



R's user community

- Group of core developers who maintain and upgrade the basic R installation. New version every 6 months.

- Anyone can contribute with add-on packages which provide additional functionality (thousands of such packages available).

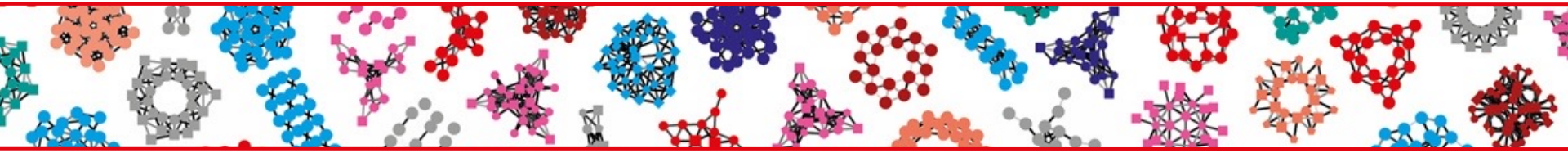
- Online help

  - in user group forums, *eg*:

  https://stat.ethz.ch/mailman/listinfo/r-help

  http://stackoverflow.com/questions/tagged/r

  - in countless online tutorials, books, blogs

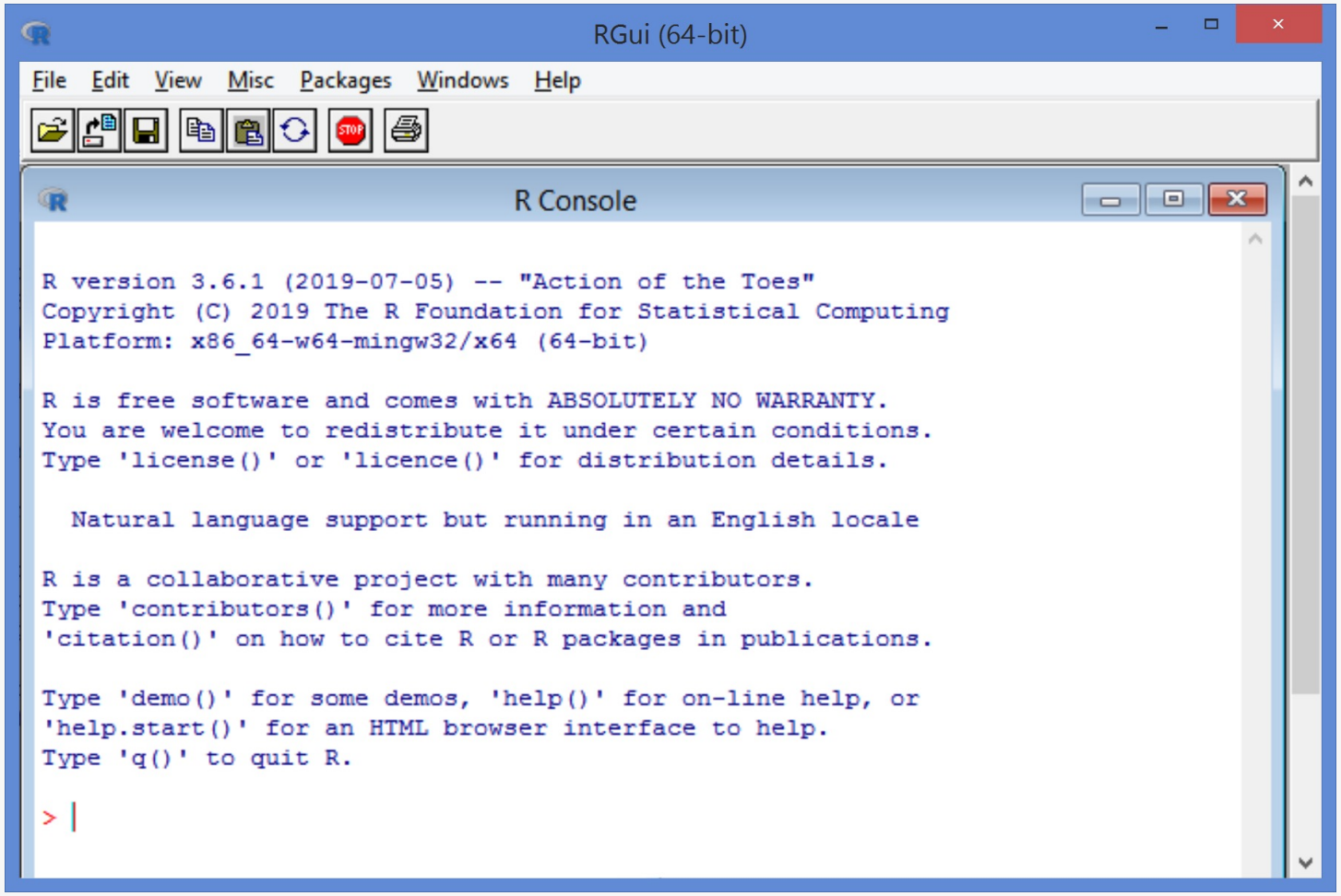


02

Getting familiar with R and RStudio environments

RGui (R Graphical user interface)

- Together with the programming language, a graphical user interface is installed.



R Combined with RStudio

http://www.rstudio.com/

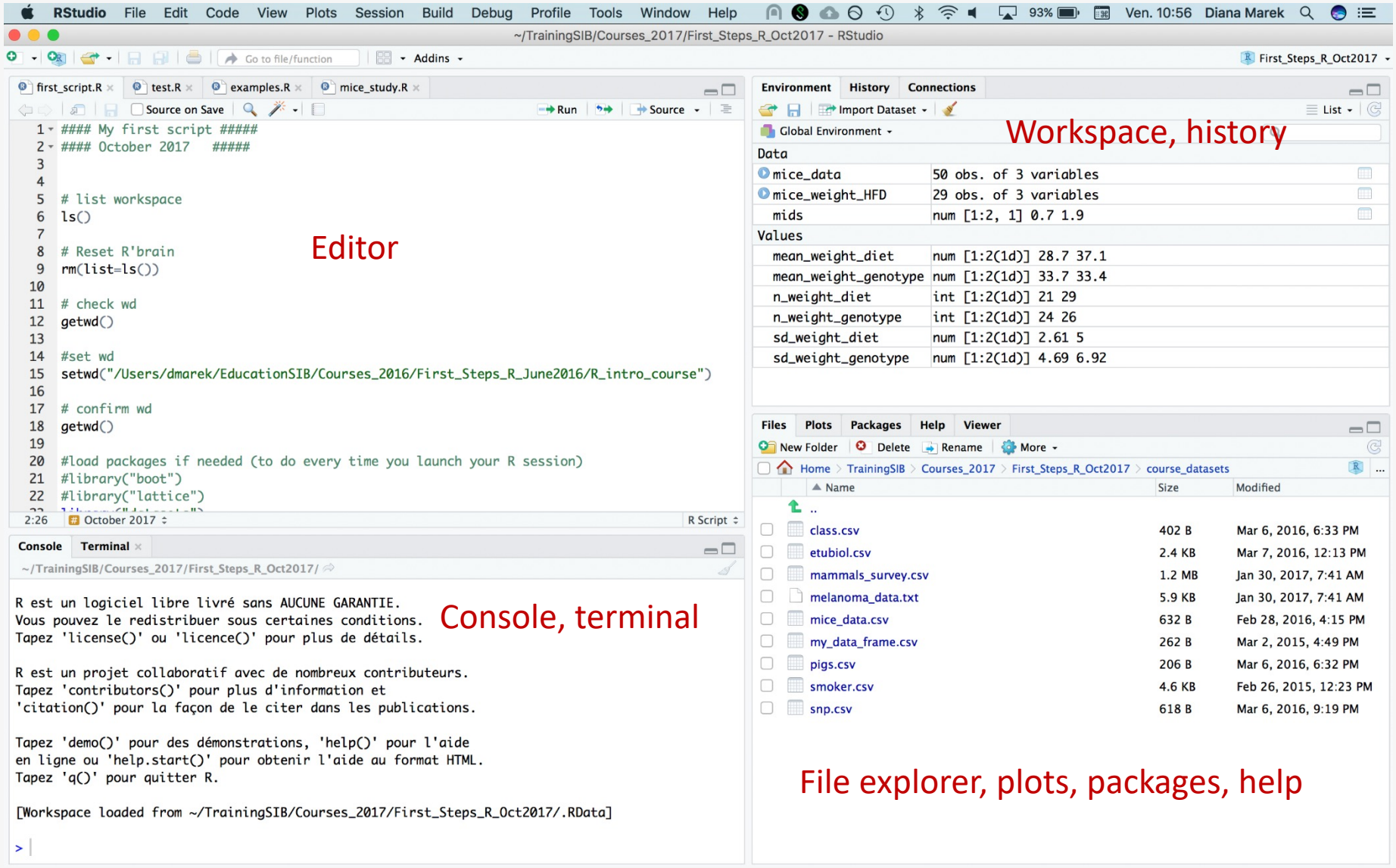
RStudio is an integrated development environment (IDE), designed to help you be more productive with R

It includes:

- A console
- A syntax-highlighting editor that supports direct code execution
- Tools for viewing the workspace and the history
- A file explorer, a package explorer, plot and help display areas

We suggest Rstudio as a more powerful, more comfortable alternative to the RGUI

RStudio interface



Creating an R Project

- RStudio allows organizing your work into projects.

  - Go to File > New project or click on “Project” in the upper right corner of RStudio.
  - Choose New Directory, then New Project, give a name to the directory and set its location.
  - This creates a new directory which contains a .Rproj file (same name as the directory).

  - OR choose Existing Directory, click Browse, navigate to a folder, then click Create Project
  - This creates an .Rproj file inside the directory (same name as the directory).

The project directory becomes automatically the working directory.

This is one of the ways RStudio adds convenience

Let’s practice – 1

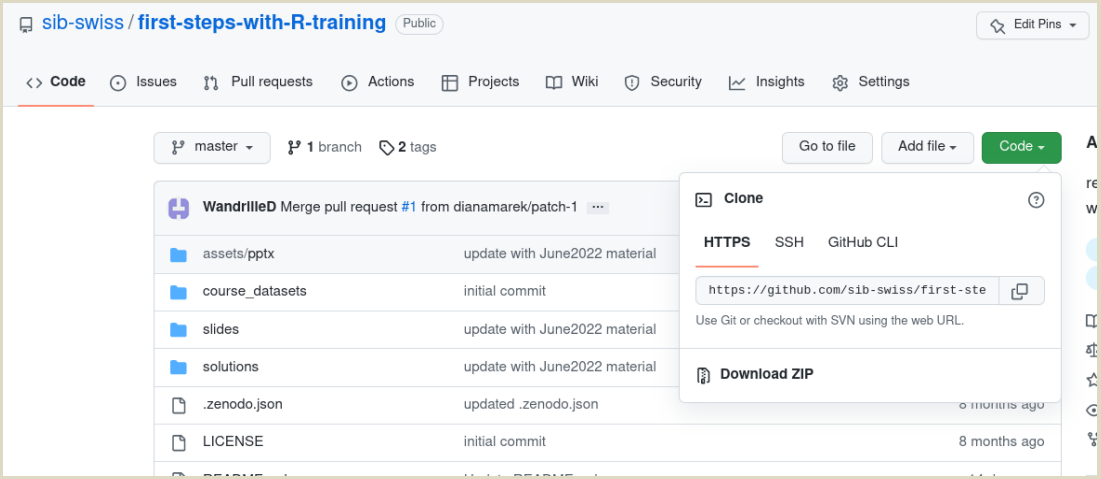
1) Outside Rstudio: Prepare course data for exercises

- Download the course material from :

  https://github.com/sib-swiss/first-steps-with-R-training

  Either use *git clone* if you know how

  OR click **Download ZIP**

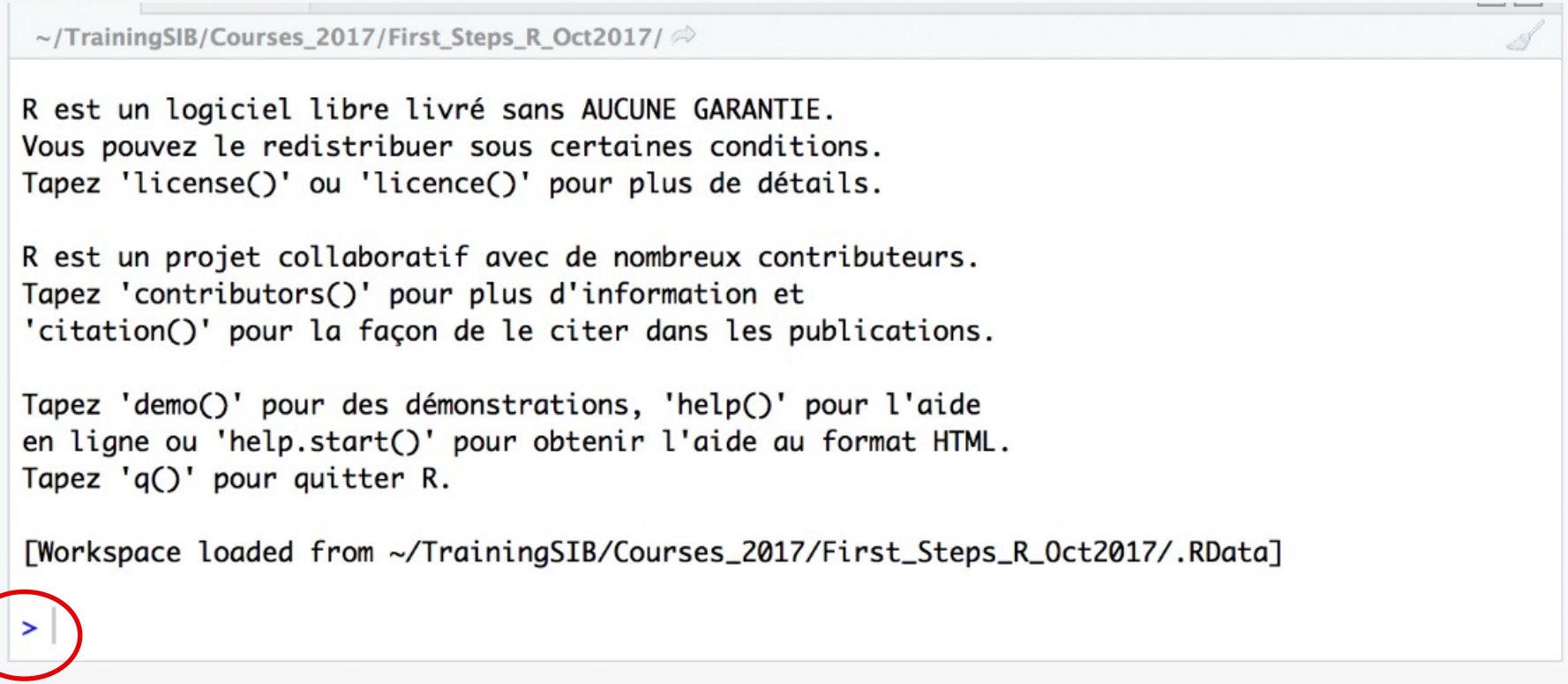


  and **unzip** then **move** folder where you want it

2) Inside Rstudio: Project set-up

In RStudio, create a new project in an existing directory, and set it as the folder of the course material you just recovered.

Console: The Command Line



The prompt ">" indicates that R is waiting for you to type a command

After each command, hit the return key.

This causes R to execute it.

Try it out...

Type the following at the command prompt:

Simple calculations

```
> 1 + 1
```

Pre-defined functions

```
> abs(-11)
```

Assign values to a variable names

```
> x  <- 128.5
```

Display content of variables

```
> x
```

After each command, hit the return key.

R Key Concepts

- **Variable: A storage space in memory that has a name and can hold data.**

  temp <- -5.5 # Create a variable named temp, holding value -5.5

- **Numeric constant:** a number, such as 128.5
- **Character constant:** a text sequence, such a "Hello" (enclosed in quotes)

- **Function: pre-written code that performs a specific task and can be executed by "calling" the function.**

  Write the function's name, followed by parentheses. Inside the parenthesis, pass variables or literal values to the function code (function arguments).

  abs(temp) # the absolute value of temp

  log2(16) # the base 2-logarithm of 16

  q() # quit R (no function arguments necessary)

- **Operator:** a special function for arithmetic, logical or other operations.

  Examples of arithmetic operators: +, -, *, /, ^, …

Working Directory

R can read and write files. The working directory is the folder on your computer where it will look for files.

See the current working directory:

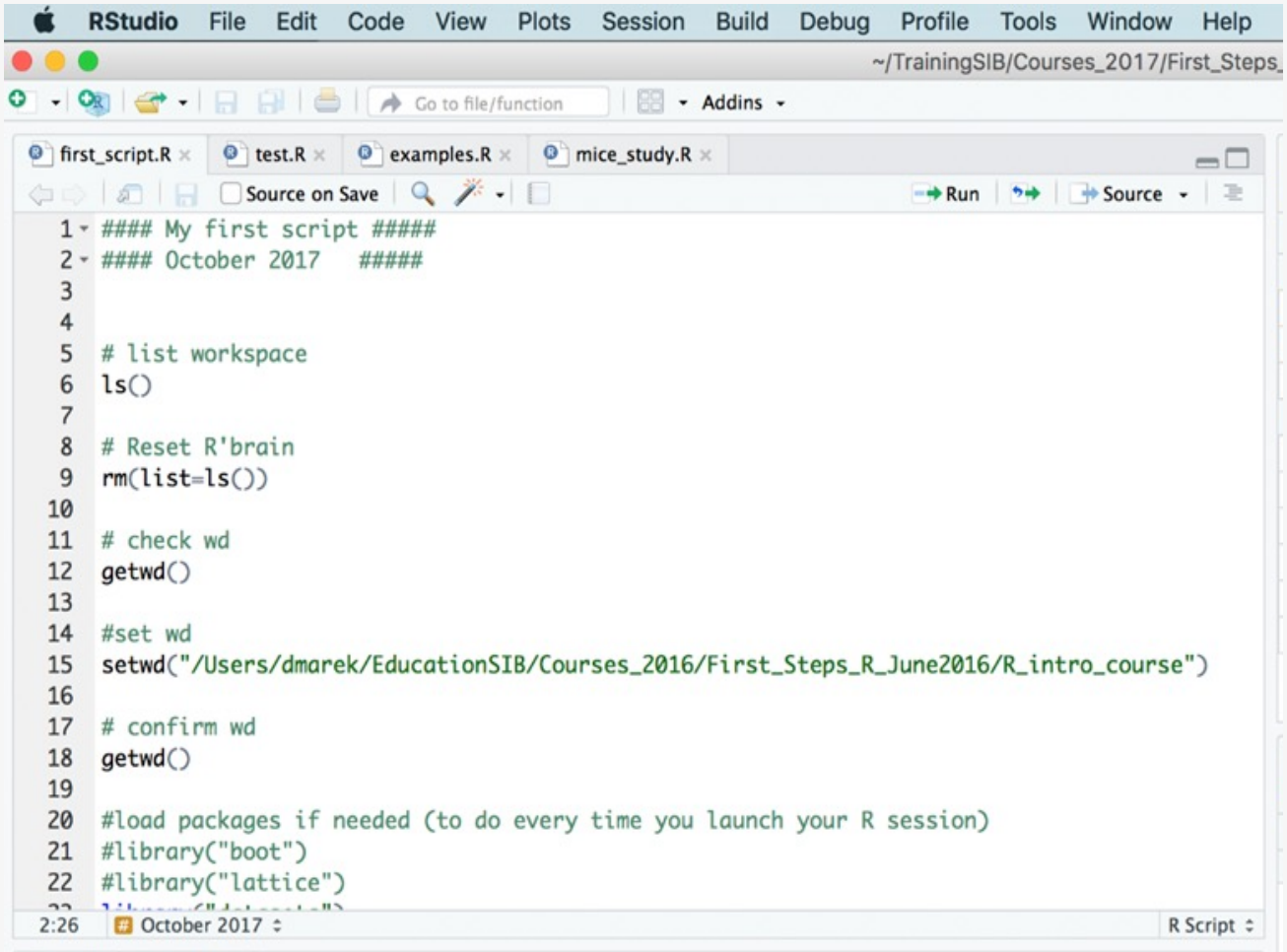
```
> getwd()
[1] "C:/Users/lwigger/Documents/Rcourse2022"
```

Change the working directory to any existing folder on your hard drive or system using setwd() and the file path, e.g.

```
> setwd("D:/R_exercises/")
```

In an Rstudio project, we usually do not need to change the working directory

Editor: Write Code to a File



Notice the syntax highlighting

R Scripts

A script is a file that contains commands to be executed in succession. Write your code into a script and save it

- to have documentation later of what you did
- to be able to re-use the code and create variations
- for easy execution

Writing Scripts (.R file)

- Create a new script using File > New File > R script. **Don't forget to save your script often.**
- By default, scripts are saved into the working directory.
- Files can be saved to other locations (File -> Save As…)

R scripts

Tips:

- Most of your code should be developed and saved in scripts.
  - You can execute individual lines of code interactively while you are writing it.
  - You can run the entire script after it is finished and debugged.

Send Code From a Script to the Console

Run individual lines, one by one:

- In RStudio: put the cursor anywhere in a line, hit

Ctrl + enter (Windows)

Cmd + return (Mac)

or click the "Run" button

Tip: Run part of a line or multiple lines: Highlight the code, then proceed as above

Workspace

The workspace is the internal memory where R stores the objects you created during the session.

Explore your workspace using the command line:

- To list what is in your workspace, type

```
> ls()
```

- To remove (delete) an object from the workspace, use function rm:

```
> rm(x)
```

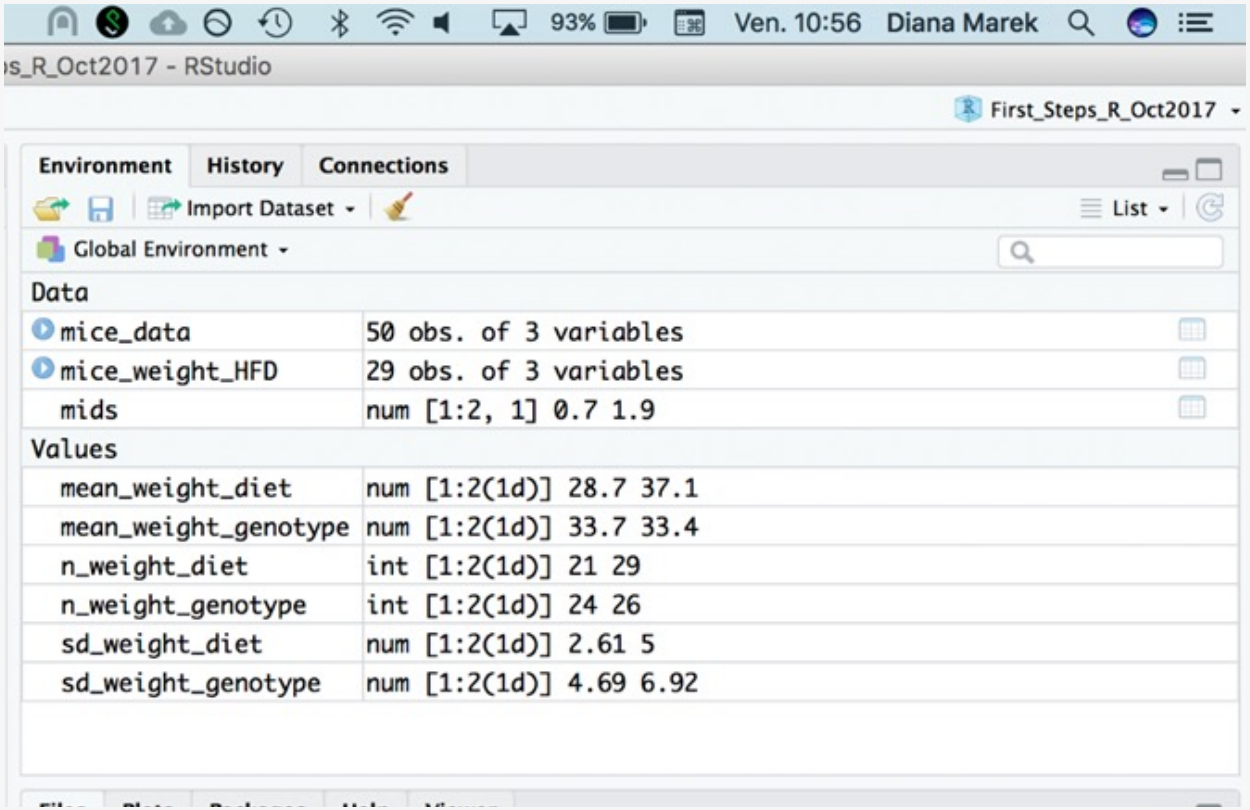
- To remove (delete) all objects from the workspace, type

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

Workspace in RStudio

Explore your workspace using Rstudio's GUI:

- See the upper right quadrant, tab "Environment": all objects are listed
- To remove all objects form the workspace, click the broom icon.



Let's practice – 2

1) Prepare your first script

- Open a script file and save it with file name "`ex1.R`"
- Comment it (# symbol at the beginning of the line).
- Type or paste the following code:

```
# First Steps, ex 1
w <- 3
h <- 0.5
area <- w * h
area
```

2) Look at the script (before trying to run it)

- Can you understand each line? What do you expect it to print to the console?

3) Run the script and explore RStudio features

- Run the script line by line. Try both the "Run" button and the keyboard shortcut. Watch variables appear in the Environment panel (top right).
- Watch what is printed to the console (bottom left). Does it match your expectation?

Closing or Switching Projects

- Close a project:

  File -> Close

- Switch to another project, which will close the current one:

  File -> Open Project...

- Open another project and keep the current one open as well:

  File -> Open Project in New Session...

Reopening an R Project from a File

You can access your R project directly from your hard drive

- Find the .Rproj file and open it (double click on many systems).
- Rstudio will automatically start if it is not already running.

Workspace (.Rdata) and History (.Rhistory) Options

- When closing or switching projects, the workspace and the history are automatically cleared.

Your RStudio project can be configured to save your workspace and history to a file upon closing a project - or not.

Menu:

Tools -> Global Options -> General (set default for all projects)

Tools -> Project Options -> General (settings for one project)

CAUTION: .Rdata files can be very large.
Save only when you have space on your hard drive!

Let's practice – 2bis

4) Look at the project options (RStudio's *Tools* menu). If needed, modify them to save your workspace and history and to restore them at startup.

Check if this works:

- Close RStudio.
- In your course folder, (double-)click the .rproj file.
- Does your project open? Are your variables still in the Environment?

Check if this works, too:

- Close RStudio.
- Open RStudio again.
- Verify that your project is currently closed. How do you see this?
- From inside RStudio, open your project. Are your variables now in the Environment?

Packages

- Sets of related functions (and sometimes data sets)
- A small number of packages are part of the basic R installation.
- Many, many packages are developed by the user community and can be installed later as needed.

- There are two main repositories that provide R packages of interest in the life sciences. Their content can be browsed on the web :
  - CRAN (Comprehensive R Archive Network, http://cran.r-project.org/) : the main R repository, with over 18600 packages (June 2022).
  - Bioconductor (http://www.bioconductor.org) : a separate project specialized in the analysis of high-throughput genomic and transcriptomic data, with 2040 packages (June 2022).

Installing packages from CRAN

Install packages from CRAN with the install.packages() function

```
> install.packages("stringi") # character string
manipulations
```

Installation necessary only once until you upgrade R to a new version.

Loading functions from packages

Once a package is installed, its content needs to be made accessible to R.

`library()` loads the package for the current session.

It is good practice to load all needed packages at the top of a script.

```
# My Script

library(limma)
library(DESeq2)
library(MASS)
library(ggplot2)

# Here my data analysis begins
...
```

Session information

- **R.version.string** prints the currently used R version.
- **sessionInfo()** prints version information about R and attached or loaded packages.

Tip: Run sessionInfo() at the end of your data analysis and save the output.

This information is useful when you want to redo an analysis later, generate a report, or post a question on an online forum, …

Working at the command prompt in RStudio

Shortcuts for both R console and RStudio:

- TAB key for command auto completion.
- Up and down arrows to scroll through the command history.
- Ctrl-l to clear console window.

Shortcuts specific to Rstudio:

- Ctrl-1 and Ctrl-2 to jump between the script and the console windows (shift focus).

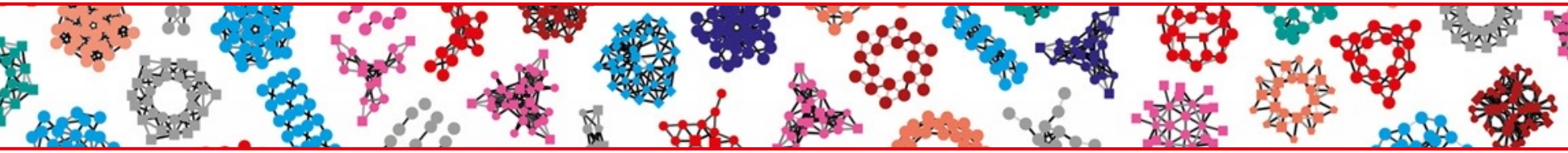
Support for incomplete statements (R console and Rstudio):

- If you hit return while your statement is incomplete, the command prompt (>) will change to +. R is waiting for you to finish writing it.
  - Keep typing and hit return again when done
  - OR hit “Esc” to abandon the unfinished command

In a Nutshell

- R and RStudio environments
  - Use of R project to gather all documents related to a project together.
  - The working directory becomes the directory of your project.
  - Possibility to save/load workspace (.Rdata file) containing your objects.
  - Possibility to save/load history of commands (.Rhistory file).
  - Help and packages available.
  - More info on how to set up your R environment (if not using R projects) in the extra slides.

- Now let's get familiar with R syntax and objects.



03 Getting started with R syntax and objects

R Basic Data Types

- Variables we have seen so far can hold one value. This value can be of different types. Use mode() to display it.

The three most common data types:

- **Numeric**:
  - A number stored with decimal point. (Decimal point need not be displayed).

    Examples: 0, 5, 55.2, -11.111

    - in some contexts this data type is labeled "double“
    - integers, stored without decimal points, exist but are rarely used.

- **Character:**

  A text sequence. Must be enclosed in quotes " ". (Single quotes work, too).

  Examples: "1a++", "Hello World", "s", "99"

- **Logical:**

  TRUE or FALSE (This is binary. No other possible values).

R Syntax

Syntax refers to the spelling and "grammar" rules of a programming language.

A few important points :

- Case sensitive: R differentiates between small letters and capitals.

- Statements can be separated by a newline or by a semicolon ";" (for better readability, a newline after each statement is almost always preferable)

- Long statements can be written on multiple lines

- R has no strict rules about including or omitting blank spaces between elements, as long as the code is unambiguous. Make your spacing consistent and think of readability.

The # character stands for comments. Anything after a # on a line is ignored by R. Write comments into your code to explain what it does.

R Objects

An object is a storage space that takes (or contains) a value, a data structure or a section of code.

- All elements of an R statement can be thought of as objects.

  Variables are objects containing data.

  Functions are objects containing code.

Allowed Names for Objects

Object names can consist of letters, numbers, dots and underscores.

- Cannot start with a number.
- Cannot contain operators (including hyphen).
- Best to start with letter

Examples:

x

mydata1

mydata.normalized

n_times

The Assignment Operator "<-" (or equivalent: "=")

We can use either the symbol "<-" or "=" to assign values to objects. Stick to one for consistency.

- Create an object:

```
> x <- 10    # Create object x, assign the value 10 to it
             # NB: This does the same as x = 10
```

- Change the value of an existing object:

```
> x <- 25    # x has the value 10; overwrite it
```

- Set one object to equal the value of another object:

```
> myNumber <- 15
> x <- myNumber  # Both x and myNumber now contain 15
```

- Modify the content of an object:

```
> x <- x + sqrt(16)  # add the square root of 16 to x
```

Using Functions (I)

- Functions are called with parentheses () after the function name

- Arguments are the input to functions, passed inside the ()

```
> ls()       # no argument – list objects in workspace
> sqrt(81)   # one argument – square root of the input
> rep(1,5)   # two arguments – repeat the number 1, 5 times
```

- Arguments have names (specified in the function definition). Function calls can be made with unnamed or named arguments or a mix of both. Use "=" for named arguments.

```
> rep(x=1, times=5)    # Named args. Equivalent to rep(1,5)
> rep(1, times=5)      # Mixed. Equivalent to rep(1,5)
```

Check R help (?function_name) to see which arguments are expected by a function.

Using Functions (II)

Many functions take more than one argument

- If unnamed, arguments must be listed in correct order (association by position).
- If named, arguments can be passed in arbitrary order (association by name).

```
> write.table(object, "outfile.txt", TRUE)

> write.table(object,  append=TRUE,  file="outfile.txt")
```

Unnamed arguments: must appear in their correct position
Named arguments: their position does not matter

Using Functions (III)

Some functions have arguments with default values.

Example: function **round()**


```
Usage (from R Help):  round(x, digits = 0)
```

Arguments with default values can be omitted in the function call; the default value is then used. Arguments without default values cannot be omitted.

```
> round(2.011)     #rounding to 0 digits after decimal point
[1] 2              # (default value)

> round(2.011, 2) #rounding to 2 digits after decimal point
[1] 2.01
```

Using Functions (III)

Using and understanding the help/documentation is 50% of what makes a programmer!

- Look up the help page, try the examples, experiment

```
?paste
?"^"
```

Also, internet is your friend:

Google "R paste function"

"R how to …"

https://stackoverflow.com/questions/tagged/r

Let's practice – 3

For all exercises, feel free to use

- cheat sheets provided

- R help (? at command prompt)

Open a new script file and save it as ex3.R

1) Assign the values 6.7 and 56.3 to variables **a** and **b**, respectively.

2) Calculate (2*a)/b + (a*b) and assign the result to variable **x**. Display the content of **x**.

3) Find out how to compute the square root of variables. Compute the square roots of **a** and **b** and of the ratio **a/b**.

4) a) Calculate the logarithm to the base 2 of **x** (i.e., $\log_2 x$).

b) Calculate the natural logarithm of **x** (i.e., $\log_e x$).

Common Object Classes

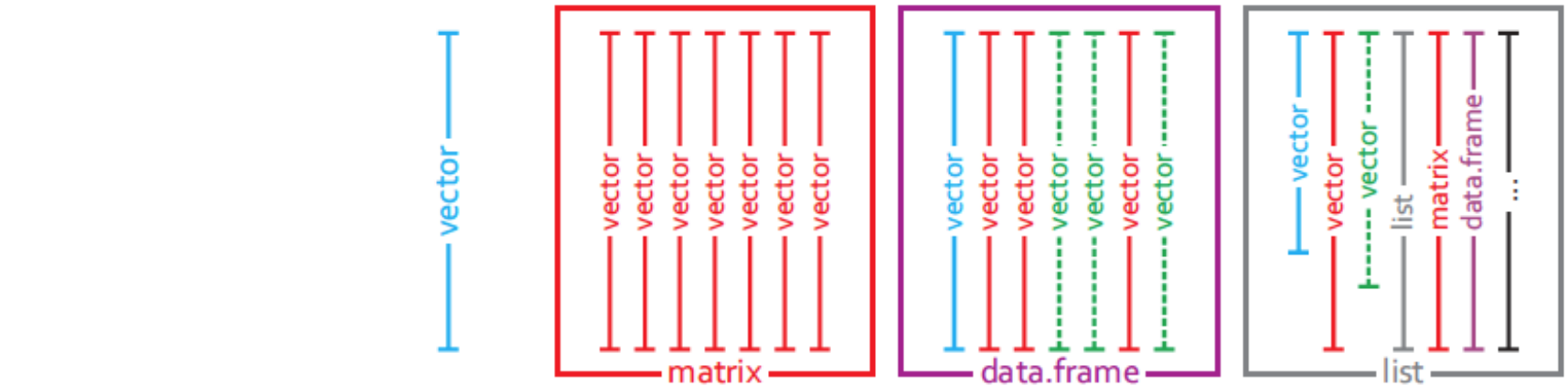
- **vector** – a series of data, all of the same type
- **matrix** – multiple columns of same length, all must have the same type of data
- **data frame** – multiple columns of same length, can be mix of data types
- **list** – a collection of objects; can be of different classes and different sizes

- **function** – a command to perform a specific task

Data objects

Function objects

Graphical View on Data Object Classes



dimension	1	n	2	1
element data type	single	single	multiple	multiple
element data structure	atomic	atomic	vector	any
subsetting	x[i] x["name"] x[1:3]	x[i,j,...] x["row","col",...] x[,1:3,...]	x[i,j] x["row","col"] x$colname	x[[i]] x$colname

From M. Stadler

Creating Objects: Vectors

Vector: A series of data, all of the same type

- Create a vector using c()

```
height_in_cm <- c(180, 167, 199)                    # c() stands for concatenate
```

- Create a vector using c() where each element has a name

```
height_in_cm <- c(Mia=180, Paul=167, Ed=199)
```

- Access elements of a vector using []

```
height_in_cm[1]          # get the first element
height_in_cm[c(1,3)]     # get the 1st and 3rd element
height_in_cm["Paul"]     # get the element named "Paul"
```

Scalars in R (the simple variables we have seen so far) can be thought of as vectors of length 1.

Creating Objects: More Ways to Generate Vectors

- Vectors of defined length with default value

```
> numeric(4); character(4); logical(4)
```

- **:** (colon operator)

```
> a <- 1:10
```

- **seq()** for sequences with any step size

```
> s <- seq(4,10,2) #start at 4, end at 10, step by 2
```

- **rep()** for vectors with repeating elements

```
> genotypes <- c(rep("WT",3), rep("KO",3))
```

- **Trick :** Use **[]** to extract repeated elements

```
> tplayer <- c("Federer", "Nadal")
> tplayer[c(1,1,1,2,2,1)] #3x Federer, 2x Nadal, 1x Federer
```

Vector Manipulation (I)

Applying operators to vector results in element-wise operations

```
> a
[1] 1 2 3 4

> a * 2                    # multiply each element of a by 2
[1] 2 4 6 8

> a + c(12,10,12,10) # add the elements in 2 vectors
[1] 13 12 15 14
```

Vector Manipulation (II)

Many functions take a vector as argument.

Some perform an element-wise operation. Example:

```
> log2(a) # compute the logarithm in base 2 of each element
    [1] 0.000000 1.000000 1.584963 2.000000
```

Some return a single value. Example:

```
> mean(a) # compute mean of the elements
    [1] 2.5
```

Coercion

- All elements of a vector must be the same type
- If combining different types, they will be coerced to the most flexible type
  - least to most flexible are: logical < numeric < character

Example:

```
> vec <- c(12, "twelve", TRUE) #combine 3 data types
> vec                          #all are coerced to character
[1] "12" "twelve" "TRUE"

> class(vec)
[1] "character"
```

Coercion

- We can coerce an existing vector to another type using the functions **as.logical(), as.numeric(), as.character()**.

- Example: Coerce a logical vector to numeric

  Values are converted to 1 (for TRUE) and 0 (for FALSE)

  - we can use **as.numeric()** for explicit coercion
  - we can use mathematical functions on logical vectors, coercion to numeric happens automatically

```
> x <- c(FALSE, FALSE, TRUE)
> as.numeric(x)
> sum(x)      # number that are true
> mean(x)     # proportion that are true
```

Factors

- A **factor** is a vector containing values from a limited set; used for storing categorical data.

- Example: Genotype of mouse individuals

```
> genotype <- factor(c("WT", "WT", "Mut2", "Mut1", "Mut2"))
```

The available values in a factor are called levels. Extract them:

```
> levels(genotype)
[1] "Mut1" "Mut2" "WT"
```

- Convert the factor back to a character vector:

```
> geno <- as.character(genotype)
```

Factors with Custom Sorted Levels

- By default, factor levels are sorted alphabetically.
- We can specify a different sorting with the argument levels.

- Example: Genotype of mouse individuals

```
> genotype <- factor(c("WT", "WT", "Mut2", "Mut1", "Mut2"),
                     levels = c("WT", "Mut1", "Mut2"))
```

```
> levels(genotype)
[1] "WT"   "Mut1" "Mut2"
```

Levels are sorted the way we wanted

Let’s practice - 4

1) Create two vectors, **vector_a** and **vector_b**, containing the values from −5 to 5 and from 10 down to 0, respectively.

2) Calculate the (element-wise) sum, difference and product between the elements of **vector_a** and **vector_b**.

3) a) Calculate the sum of elements in **vector_a**.

b) Calculate the overall sum of elements in both **vector_a** and **vector_b**.

4) a) Identify the smallest and the largest value in **vector_a**

b) among both **vector_a** and **vector_b**.

5) Compute the overall mean of the values among both **vector_a** and **vector_b**.

Hint: Each task in exercises 1-5 can be performed in a single statement per vector (the minimum and maximum count as 2 tasks)

Operators (Most Commonly Used Ones)

- **Arithmetic**

+, -, *, /, ^

- **Comparison**

>, <, <=, >=, == (equal to), != (not equal to)

- **Logical**

! (negation), & (AND), | (OR)

- **Other**

%in% (in operator)

Comparisons, logical operators and %in% always return logical values! (TRUE, FALSE)

Operators returning logical values: examples

```
> c(1,3,2) == 2
[1] FALSE FALSE TRUE

> !(c(1,3,2) < 2)
[1] FALSE  TRUE  TRUE

> table(!(c(1,3,2) < 2))
#FALSE  TRUE
#2      1

> c("Fred", "Marc", "Dan", "Ali") %in%
  c("Dan", "Geoff", "Ali")
[1] FALSE FALSE TRUE  TRUE
```

Missing Values

- R distinguishes between
  - NA (not available)
  - NaN (not a number, e.g. result of 0/0)

- Use the functions **is.na()** and **is.nan()** to detect them.

Missing Values: Examples (I) NA

Missing values are usually represented by NA:

```
> y <- c(1,2,3,4,5,NA,NA)
```

NA's interfer with many functions:

```
> mean(y)
[1] NA
```

Arguments often exist to remove NA's before calculation

```
> mean(y, na.rm=TRUE)
[1] 3
```

Alternatively, use **na.omit()** to remove NAs from the data

```
> y_cleaned <- na.omit(y)
> mean(y_cleaned)
[1] 3
```

Missing Values: Examples (II)

```
> x <- c(1, NA, 0/0) ; x # a vector to play with
[1] 1 NA NaN

> is.na(x) #detects NAs and NaNs from x
[1] FALSE TRUE TRUE

> is.nan(x) # detects only NaNs from x
[1] FALSE FALSE TRUE

> x > 2  # what if we try to compare NA and NaN to a number?
[1] FALSE NA NA

> x[!is.na(x)]  # removes NAs and NaNs from x
[1] 1
```

Creating Objects: Data Frames

data frame: multiple columns of same length, can be mix of data types

```
>name          <- c("Joyce", "Chaucer", "Homer")
>status        <- c("dead", "deader", "deadest")
>reader_rating <- c(55, 22, 100)
```

Create a data frame using the function **data.frame()**

```
>poets <- data.frame(name, status, reader_rating)
>poets
```

```
     name   status reader_rating
1   Joyce     dead            55
2 Chaucer   deader            22
3   Homer deadest           100
```

Creating Objects: Lists

List: a collection of objects; can be of different classes and different sizes

Create a few objects:

```
> vec <- c(0.4, 0.9, 0.6)
> mat <- cbind(c(1,1), c(2,1))
> df <- data.frame(name=c("Ed", "Lisa"), age=c(61, 71))
```

Unnamed list - collect these objects in a list, using the function **list()**:

```
> l <- list(vec, mat, df)
```

Named list - collect these objects in a list with named elements:

```
> l_with_names <- list(myvector=vec, mymatrix=mat, mydata=df)
```

Detecting Data Types and Object Classes

The function **class()** is useful when we are not sure what kind of object we are dealing with.

- for vectors, returns the basic data type of its elements ("numeric", "character", "logical", …)

  similar to mode() but slightly more fine-grained
  - recognizes "integer" as different from "numeric"
  - recognizes factors (categorical variables)

- for all other objects covered on previous slide, returns their class ("matrix", "data.frame", "list", "function", ...)

Accessing Data Elements

matrix:

```
>m[2, 2]      # gets the element on row 2 in column 2
>m[1:3, ]     # gets rows 1,2,3
>m[, c(1,4)] # gets columns 1 and 4
```

data frame:

```
>poets[2, 2]       # gets the element on row 2 in column 2
>poets[, c(1,3)]  # gets columns 1 and 3
>poets[, c("name", "reader_rating")] # gets columns "name"
                                     #  and "reader_rating"
>poets$name        # gets column "name"
```

list:

```
>l[[1]]                    # gets the first object
>l_with_names[["myvec"]]# gets the object named "myvec"
>l_with_names$myvec      # gets the object named "myvec", too
```

Accessing Names of Data Elements

matrix and data frame:

```
>rownames(poets)    # gets the row names
>colnames(poets)    # gets the column names

>rownames(poets) <- c("J", "C", "H") # overwrites row names
```

list:

```
>names(l_with_names) # gets the list elements' names
>names(l_with_names) <- c("A", "B", "C") # overwrites names
```

Let’s practice – 5

Open a new script and save it as "Ex5.R". Comment it.

1) In your script, write commands to install and load the package "MASS".

2) Write the following command to load the bacteria data set from the package MASS:

```
data(bacteria) # loads the bacteria data set (from MASS)
```

Execute the command. Check: You should have a variable named "bacteria" in your Environment.

3) What are the names of the columns of the **bacteria** data.frame ?

4) Use **[]** to select rows 100 to 119 of the column “ap” .

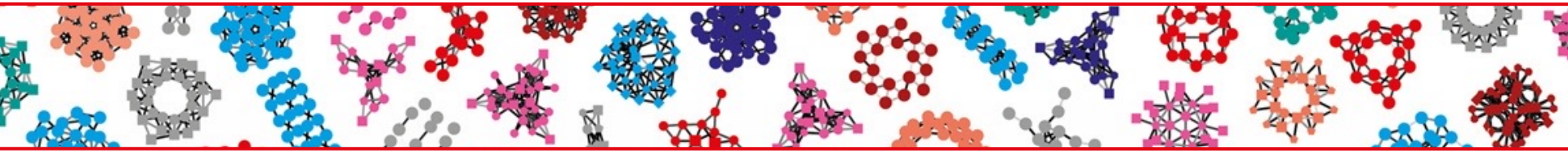
5) Use $ to get the column "week" and check how many missing values it has.

Optional : 6) Count how many rows correspond to a “placebo” treatment (“trt” column) using the comparison operator "==".

In a Nutshell

- Everything in R is an object.
- Using R is all about creating and manipulating data objects using functions (which are themselves objects).
  - Objects can be assigned to a name
  - Objects have a class (data frame, matrix, list etc)
  - Data values inside objects have different data storage modes (numeric, character, logical)
- We covered many ways to generate data (create objects).

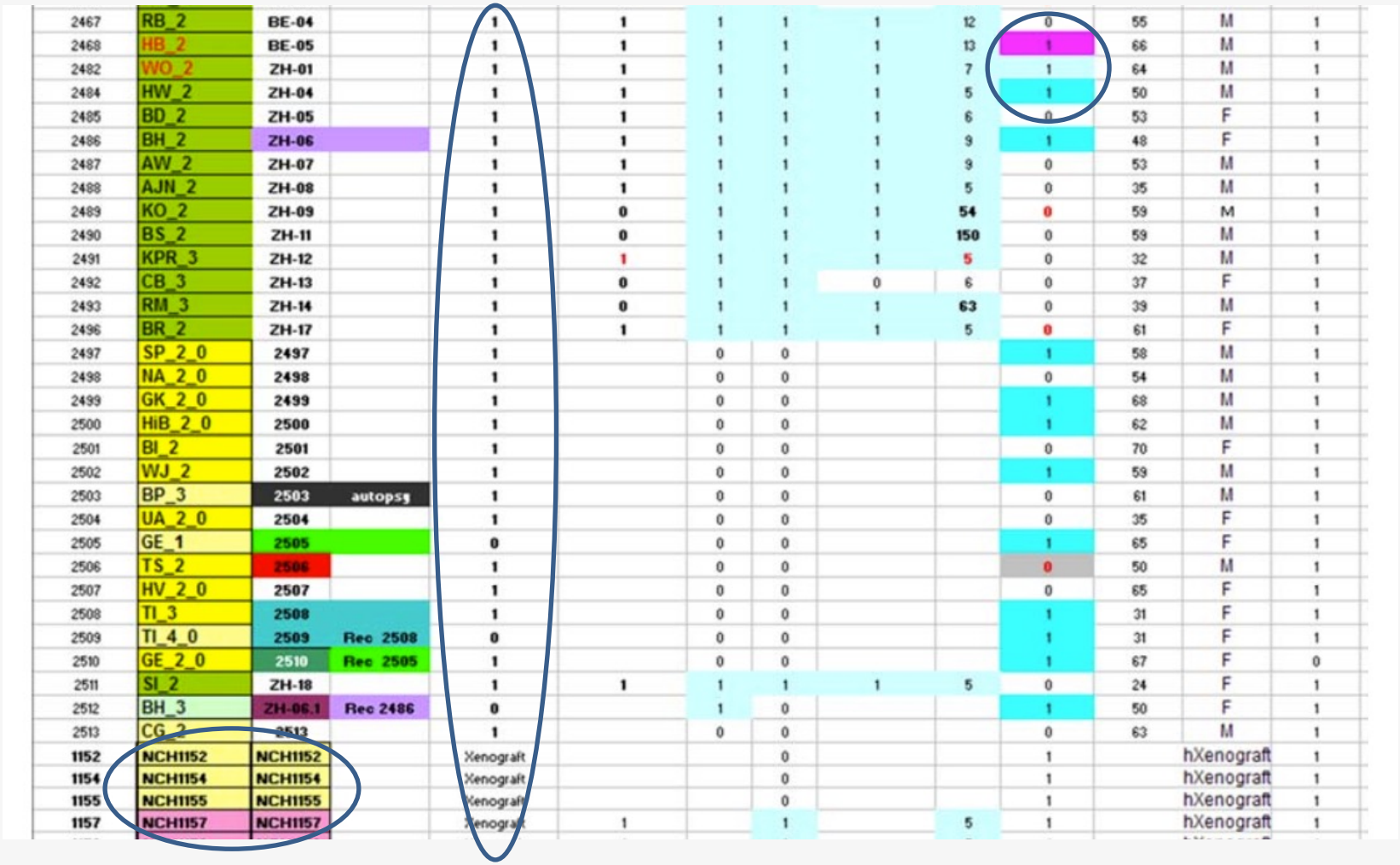
- Now, let's import some data !



04

Formatting Your Data

Example of a dataset



2467	RB_2	BE-04		1	1	1	1	1	12	0	55	M	1
2468	HB_2	BE-05		1	1	1	1	1	13	1	66	M	1
2482	WO_2	ZH-01		1	1	1	1	1	7	1	64	M	1
2484	HW_2	ZH-04		1	1	1	1	1	5	1	50	M	1
2485	BD_2	ZH-05		1	1	1	1	1	6	0	53	F	1
2486	BH_2	ZH-06		1	1	1	1	1	9	1	48	F	1
2487	AW_2	ZH-07		1	1	1	1	1	9	0	53	M	1
2488	AJN_2	ZH-08		1	1	1	1	1	5	0	35	M	1
2489	KO_2	ZH-09		1	0	1	1	1	54	0	59	M	1
2490	BS_2	ZH-11		1	0	1	1	1	150	0	59	M	1
2491	KPR_3	ZH-12		1	1	1	1	1	5	0	32	M	1
2492	CB_3	ZH-13		1	0	1	1	0	6	0	37	F	1
2493	RM_3	ZH-14		1	0	1	1	1	63	0	39	M	1
2496	BR_2	ZH-17		1	1	1	1	1	5	0	61	F	1
2497	SP_2_0	2497		1		0	0			1	58	M	1
2498	NA_2_0	2498		1		0	0			0	54	M	1
2499	GK_2_0	2499		1		0	0			1	68	M	1
2500	HiB_2_0	2500		1		0	0			1	62	M	1
2501	BI_2	2501		1		0	0			0	70	F	1
2502	WJ_2	2502		1		0	0			1	59	M	1
2503	BP_3	2503	autopsy	1		0	0			0	61	M	1
2504	UA_2_0	2504		1		0	0			0	35	F	1
2505	GE_1	2505		0		0	0			1	65	F	1
2506	TS_2	2506		1		0	0			0	50	M	1
2507	HV_2_0	2507		1		0	0			0	65	F	1
2508	TI_3	2508		1		0	0			1	31	F	1
2509	TI_4_0	2509	Rec 2508	0		0	0			1	31	F	1
2510	GE_2_0	2510	Rec 2505	1		0	0			1	67	F	0
2511	SI_2	ZH-18		1	1	1	1	1	5	0	24	F	1
2512	BH_3	ZH-06.1	Rec 2486	0		1	0			1	50	F	1
2513	CG_2	2513		1		0	0			0	63	M	1
1152	NCH1152	NCH1152		Xenograft			0			1		hXenograft	1
1154	NCH1154	NCH1154		Xenograft			0			1		hXenograft	1
1155	NCH1155	NCH1155		Xenograft			0			1		hXenograft	1
1157	NCH1157	NCH1157		Xenograft	1		1		5	1		hXenograft	1

Difficult to use with any statistical program

Courtesy of Frédéric Schütz

Prepare Your Data Outside of R

Before using R and importing the dataset you collected from an experiment, you need to know how to format it properly, so R can read it.

A spreadsheet program such as Excel or OpenOffice can be used for data entry and simple manipulation.

Three precepts of tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

http://www.ucd.ie/ecomodel/pdf/TidyData.pdf

Example of Well-Formatted Dataset

	A	B	C	D
1	chr	pos	minor	major
2	1	123369	A	C
3	1	138369	G	T
4	1	153369	T	C
5	1	168369	C	T
6	1	183369	G	A
7	1	198369	T	A
8	5	228369	G	A
9	5	258369	G	A
10	5	288369	A	G
11	5	318369	C	A
12	5	348369	A	T

- A header line with variable names
- 4 variables, one in each column
- One observation per row, here SNP info

Example of Well-Formatted Dataset

	A	B	C	D
1	chr	pos	minor	major
2	1	123369	A	C
3	1	138369	G	T
4	1	15336		
5	1	16836		
6	1	18336		
7	1	19836		
8	5	22836		
9	5	25836		
10	5	28836		
11	5	31836		
12	5	34836		

Glossary for the snp data

snp single nucleotide polymorphism

chr chromosome

pos position

minor minor allele (a minority of individuals have this letter at this position)

major major allele (a majority of individuals have this letter at this position)

Formatting Recommendations – Checklist

- If you work with spreadsheets, the first row is usually reserved for the header.
- The first column may or may not be an ID column.
- Remove blank spaces from column names and in fields. If you want to concatenate words, insert a "_" between words.
- Avoid column names containing symbols other than "_".
- Short names are preferred over longer names.
- Delete any comments or other content in the spreadsheet that are not part of the data table but are above, below or beside the data table.
- Make sure that any missing values in your data set are indicated with NA. (Check spelling! N.A. or n.a. does not work.)

Other Recommendations

- If you're using a spreadsheet, keep a copy of the original data as it was provided to you. Prepare a new, "cleaned" version for your data analysis.
- Do not include columns that you do not need for your analysis.
- Have data backups!

Saving Your Data

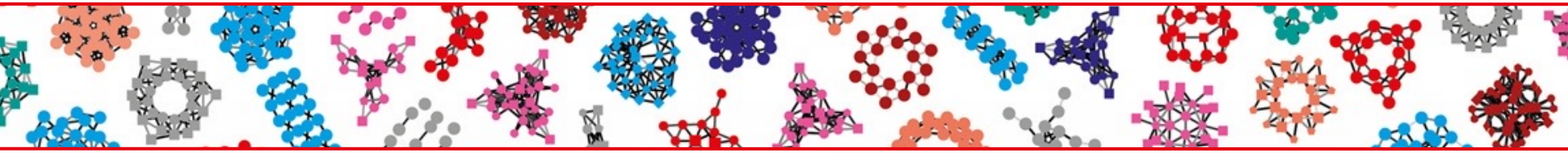
- Export the spreadsheet to your computer in a text file format:
  - csv (comma separated values) format, with file extension .csv OR
  - tsv (tab separated values) format, with file extension .txt or .tsv

snp.csv

```
chr,pos,minor,major
1,123369,A,C
1,138369,G,T
1,153369,T,C
1,168369,C,T
1,183369,G,A
1,198369,T,A
5,228369,G,A
5,258369,G,A
5,288369,A,G
5,318369,C,A
5,348369,A,T
5,378369,G,A
5,408369,A,C
5,438369,C,A
5,468369,G,C
16,513369,C,A
```

Now you are ready to start with the analysis

Keep your data save:
Have a back up!



05

Importing/exporting data into R

Importing Data

- Most widely used R base function for data import: **read.table()**
  - reads a formatted text file
  - imports it as a data frame
  - **many** options, to accommodate most text files (e.g., csv, tsv).

- To read an entire data frame from a file, it should have:
  1. a header line containing the names of all variables

     -> (not obligatory but preferable)
  2. one line per row, with values for each variable

     -> (missing values should be indicated using NA)
  3. Items must be separated by the same separator symbol

     -> (most common: , ; \t)

Importing Data – Two Questions

- Where is the file I want to import?
  - Look for your file in the file system.
  - Note its path: the succession of folders to access it

- Where is my working directory?
  - use **getwd()** **(recommendation: should be the project directory)**

File Paths in R

File paths can be specified as a string with **'/'** as separator:

```
"C:/Users/Leo/courses/data/snp.csv"
```

Or with a little help from the function **file.path():**

```
file.path("C:", "Users", "Leo", "courses", "data", "snp.csv")
```

WARNINGS:

On Windows : replace '\' by '/'

File Paths in R - Relative

R understands "." and ".." for relative file paths

. is the current directory (=working directory)

.. is the parent directory

```
"./data/snp.csv"  # file "snps.csv" in subfolder "data" of current directory
"../../snp.csv"   # file "snps.csv", 2 levels up from current directory
```

Also works with file.path():

```
file.path("..", "..", "snp.csv")
```

Importing Data – File Paths

read.table() needs to know where the file is located.

- **Data file is in the working directory:** file name suffices.

```
read.csv("snp.csv“)
```

- **Data file is in a sub-folder of working directory:** It's easy to use a relative path. (Great option for projects shared with others).

```
read.csv("data/snp.csv“) or
read.csv(file.path("data", "snp.csv"))
```

- **Data file is somewhere else or you are not working inside a project:** it's safest to use an absolute path (but can be more painful to specify!).

```
read.csv("C:/Users/Leo/courses/data/snp.csv”)
```

Importing Data

Important optional arguments of **read.table(), read.csv(), read.delim()**

- **header** (TRUE/FALSE): specifies whether the first line contains column names

  Default in read.table() is FALSE.

  Default in read.csv() and read.delim() is TRUE.

- **sep:** specifies the field separator character (e.g. "," or tab "\t").

  Default in read.table() is any white space characters (space, tab, newline and carriage return).

  Default in read.csv() is comma. Default in read.delim() is tab.

- **colClasses**: manually setting each variable data type

When in doubt, use **help(read.table)**

snp.csv

```
chr,pos,minor,major
1,123369,A,C
1,138369,G,T
1,153369,T,C
1,168369,C,T
1,183369,G,A
1,198369,T,A
5,228369,G,A
5,258369,G,A
5,288369,A,G
5,318369,C,A
5,348369,A,T
5,378369,G,A
5,408369,A,C
5,438369,C,A
5,468369,G,C
16,513369,C,A
16,558369,A,C
16,603369,G,A
16,648369,G,A
16,693369,C,G
16,738369,G,A
16,783369,A,T
16,828369,C,T
16,873369,A,C
22,923369,G,A
22,973369,T,A
22,1023369,C,T
22,1073369,G,A
22,1123369,C,T
```

The file can be imported as a data frame using the functions **read.table() or read.csv()**

Use read.table()

```
> snps <- read.table("snp.csv", header=TRUE, sep=",")
# we need to supply certain arguments
```

Use read.csv()

```
> snps <- read.csv("snp.csv") # arguments can be
omitted since defaults are adapted to reading .csv
```

Checking the Imported Data

- It is very important to check that data you asked R to import is the data you wanted.
- **head()** returns the first 6 lines of the data frame
- **dim()** returns the dimension of the data frame
- **nrow(), ncol()** returns the number of row and columns
- **colnames()** and **rownames()** functions return the column and row names of the data frame
- **str()** returns the structure of the data frame
- **summary()** is a generic function that can be applied to many types of objects. For data frames, it returns:
  - Numeric columns: min, max, median, mean, 1st and 3rd quantiles.
  - Factors columns: counts of each factor level

```
> head(snps) # shows first 6 rows (tail(snps) - shows
last 6 rows)
  chr    pos minor major
1   1 123369     A     C
2   1 138369     G     T
3   1 153369     T     C
4   1 168369     C     T
5   1 183369     G     A
6   1 198369     T     A

> dim(snps)
[1] 40  4

> nrow(snps); ncol(snps)
[1] 40
[1] 4
```

```
> colnames(snps)  # column names
[1] "chr"   "pos"   "minor" "major"

> str(snps)  # structure of the data frame
'data.frame':     40 obs. of  4 variables:
 $ chr  : int  1 1 1 1 1 1 5 5 5 5 ...
 $ pos  : int  123369 138369 153369 168369 183369
198369 228369 258369 288369 318369 ...
 $ minor: chr  "A" "G" "T" "C" ...
 $ major: chr  "C" "T" "C" "T" ...
```

R made its best guess for data types.

- Are they what we need?
- Do we wish to convert any variables to factors?

Setting factor variables

Convert categorical variables to factors as needed.

```
> snps$chr   <- factor(snps$chr, levels=c("1","5","16","22"))
> snps$minor <- factor(snps$minor)
> snps$major <- factor(snps$major)
```

```
> str(snps)  # structure of the data frame
 'data.frame':      40 obs. of  4 variables:
 $ chr  : Factor w/ 4 levels "1","5","16","22": 1 1 1 1 1 1 2
2 2 2 ...
 $ pos  : int  123369 138369 153369 168369 183369 198369
228369 258369 288369 318369 ...
 $ minor: Factor w/ 4 levels "A","C","G","T": 1 3 4 2 3 4 3 3
1 2 ...
 $ major: Factor w/ 4 levels "A","C","G","T": 2 4 2 4 1 1 1 1
3 1 ...
```

```
> summary(snps)
 chr           pos           minor  major
 1 : 6   Min.   : 123369   A: 9   A:17
 5 : 9   1st Qu.: 340869   C:10   C:10
 16: 9   Median : 715869   G:15   G: 4
 22:16   Mean   : 777869   T: 6   T: 9
         3rd Qu.:1185869
         Max.   :1673369
```

Accessing Parts of the Data

```
> snps[2,]  # 2nd row
   chr    pos minor major
2    1 138369     A     T

> snps[, "minor"]  # column named "minor"
[1] A A T C G T G G A C A G A C G C A G G C G A C A G T C G
C C T A G G T G T C G A
Levels: A C G T

> snps[1:3, c(1,3)] # 3 first rows, 1st and 3rd column
    chr minor
1    1     A
2    1     G
3    1     T
```

```
> snps$chr # vector of chromosomes, equivalent to snps[, 1]
 [1]  1  1  1  1  1  1  5  5  5  5  5  5  5  5  5 16 16 16
16 16 16 16 16 16 22 22 22
[28] 22 22 22 22 22 22 22 22 22 22 22 22 22
Levels: 1 5 16 22

> snps$chr[40] # chromosome of the last row
[1] 22
```

Accessing Parts of the Data

- subset() is a powerful function which allows you to subset your data by specific columns and values in those columns. Logical operators can be used within the subset.

```
> subset(snps, chr==1) # keeps only the snps where chr is 1
chr    pos minor major
1   1 123369     A     C
2   1 138369     G     T
3   1 153369     T     C
4   1 168369     C     T
5   1 183369     G     A
6   1 198369     T     A
```

```
> subset(snps, chr==1 & major=="A") # keeps only the
snps in chr 1 with an "A" as major allele
  chr    pos minor major
5   1 183369     G     A
6   1 198369     T     A
```

```
> subset(snps, chr==1 & (major=="A" | major=="T"))
# keeps only the snps in chr 1 with an "A" or "T" as
major allele
 chr    pos minor major
2   1 138369     G     T
4   1 168369     C     T
5   1 183369     G     A
6   1 198369     T     A
```

Customising Summaries of Data

tapply() generates custom summaries of your data using :

- X: a column you want to aggregate (of any data type)
- INDEX: a factor column, or list of factor columns, for grouping
- FUN: a function to be applied to X (mean, sd, min, max, length, median, range, quantiles…), separately for each grouping indicated by INDEX

```
> tapply(X=snps$pos, INDEX=snps$chr, FUN=min)

1      5     16     22
123369 228369 513369 923369
```

In each chromosome, find the smallest (min) position number where a SNP is located.

Data Reshaping : Adding Rows and Columns

- Rows and columns of data can be added using the functions **rbind()** and **cbind()**, respectively.

- Add a row to the SNP data:

```
> snps_updated <- rbind(snps,
      data.frame(chr=22, pos=1723369, minor="A", major="T"))
```

- Add a column to the SNP data:

```
> tested <- rep( c("yes","no") ,  nrow(snps)/2)
> snps_mod <- cbind(snps, idx)
```

Always check that your new dataset is what you expect, the same way you did after you imported the original one

Data Reshaping : Removing a Column

- Remove the new column of indexes, using exclusion (-) or column extraction

```
> snps_orig <- snps_mod[,-1] # remove the first column
> head(snps_orig)  # check resulting data
```

or

```
> snps_orig <- snps_mod[,2:dim(snps_mod)[2]] # extract all
columns that you want to keep (from the 2nd to the last)

> head(snps_orig) # check resulting data
```

Exporting Data to a File

The functions **write.table()** and **write.csv()** allow to write a data frame to a file.

Example:

```
> write.table(snps_updated, file="snps_updated.csv",
   quote=FALSE, sep=",",row.names=FALSE)
```

- Important optional arguments (check ?write.table for more):
  - **file** is the file path for the output file (if file name without a path is given, will be stored in current working directory).
  - **append** allows to append to an existing file (default is FALSE).
  - **quote** specifies whether the elements of character or factor columns should be surrounded by double quotes in the printed output (default is TRUE).
  - **sep** specifies the field separator to be used, e.g., comma (",") or tab ("\t").
  - **row.names** specifies whether or not the row names are written (default is TRUE). Alternatively, accepts a character vector with new row names to be written.
  - **col.names** specifies whether the column names are written (default is TRUE).

In a Nutshell

- How to import data into data frames (R's typical container for data)
- How to check the imported data, summarize it , access part of it, and manipulate it.
- How to export data to files

- Next step tomorrow: How to represent data graphically?

Let’s practice - 6

A dataset from mouse experiments at 18 weeks is available in the file ***mice_data.csv*** *(courtesy of F Schutz and F. Preitner)*. Let's explore the dataset to see what it contains.

1) Open a new script file in R studio, comment it and save it.
2) Have look at the csv file in R studio's file explorer. What do you need to check in order to be able to read in the file correctly?
3) Read the file into R, assign its content to object "mice_data". Examine the object.
4) How many observations and variables does the dataset have?
5) What is the structure of the dataset? What are the names and classes of the variables?
6) Which variables appear to be categorical? Convert them to factors.
7) Get the summary statistics of "mice_data"

Let's practice – 6bis

8) Use the function table() to compute the number of observations in different mouse groups. a) How many mice are included of each genotype (WT, KO)? b) How many mice are included per diet (HFD, CHOW)? c) Make a 2x2 table by genotype and diet crossed.

   Hint : try some of the example in the help(table) page.

9) Isolate the observations for the mice on high fat diet (HFD) using subset(). Compute a summary statistics just for the weights of the subset. Then do the same for the mice on regular chow diet (CHOW). Export the data of each subgroup to a csv file.

10) Look at the results from the two previous exercises. What does this initial exploration of the data suggest about mouse weights?

11) Optional: Compute the means and standard deviations for WT and KO mouse weights using tapply(). Then do the same for CHOW and HFD groups.

R Style: Google's R Style Guide

Different authorities have different style recommendations for naming things, spacing, operator symbols, layout, commenting etc.

Example:
https://web.stanford.edu/class/cs109l/unrestricted/resources/google-style.html

Summary of selected styles from above guide (relevant to course content):

File names: Use meaningful names, ending with file extension .R `(predict_ad_revenue.R)`

Identifiers: Variable names should have all lower case letters, words separated with dots `(avg.clicks)`

Line length: maximum 80 characters

Indentation: two spaces, no tabs

Assignment: use <-, not =

Semicolon: don't use them

R Style: Google's R Style Guide (II)

Spacing:

Place spaces around all binary operators (=, +, -, <, etc.)

Do not place a space before a comma, but always place one after a comma.

Otherwise, do not place spaces around code in parentheses or square brackets

Good:

```
Total <- sum(x[, 1])      # spaces around <- and after comma
```

Bad:

```
Total<-sum(x[,1])         # no spaces
Total <- sum ( x[ , 1] ) # too many spaces
```

R Style: Google's R Style Guide (III)

Spacing - Exceptions:

Spaces around ='s are optional when passing parameters in a **function call**.

```
write.table(snps_updated, file="snps_updated.csv",
quote=FALSE, sep=",",row.names=FALSE)
```

Extra spacing is okay if it improves alignment of equal signs (=) or arrows (<-).

```
write.table(x         = snps_updated,
            file      = "snps_updated.csv",
            quote     = FALSE,
            sep       = ",",
            row.names = FALSE)
```

This is twice the same function call:
styled for brevity and styled for readability.
Both versions conform to Google R style.