

Basics in R

24.03.2020, Data Science (SpSe 2022): T3

Prof. Dr. Claudius Gräbner-Radkowitz

Europa-University Flensburg, Department of Pluralist Economics

www.claudius-graebner.com | [@ClaudiusGraebner](https://twitter.com/ClaudiusGraebner) | claudius@claudius-graebner.com

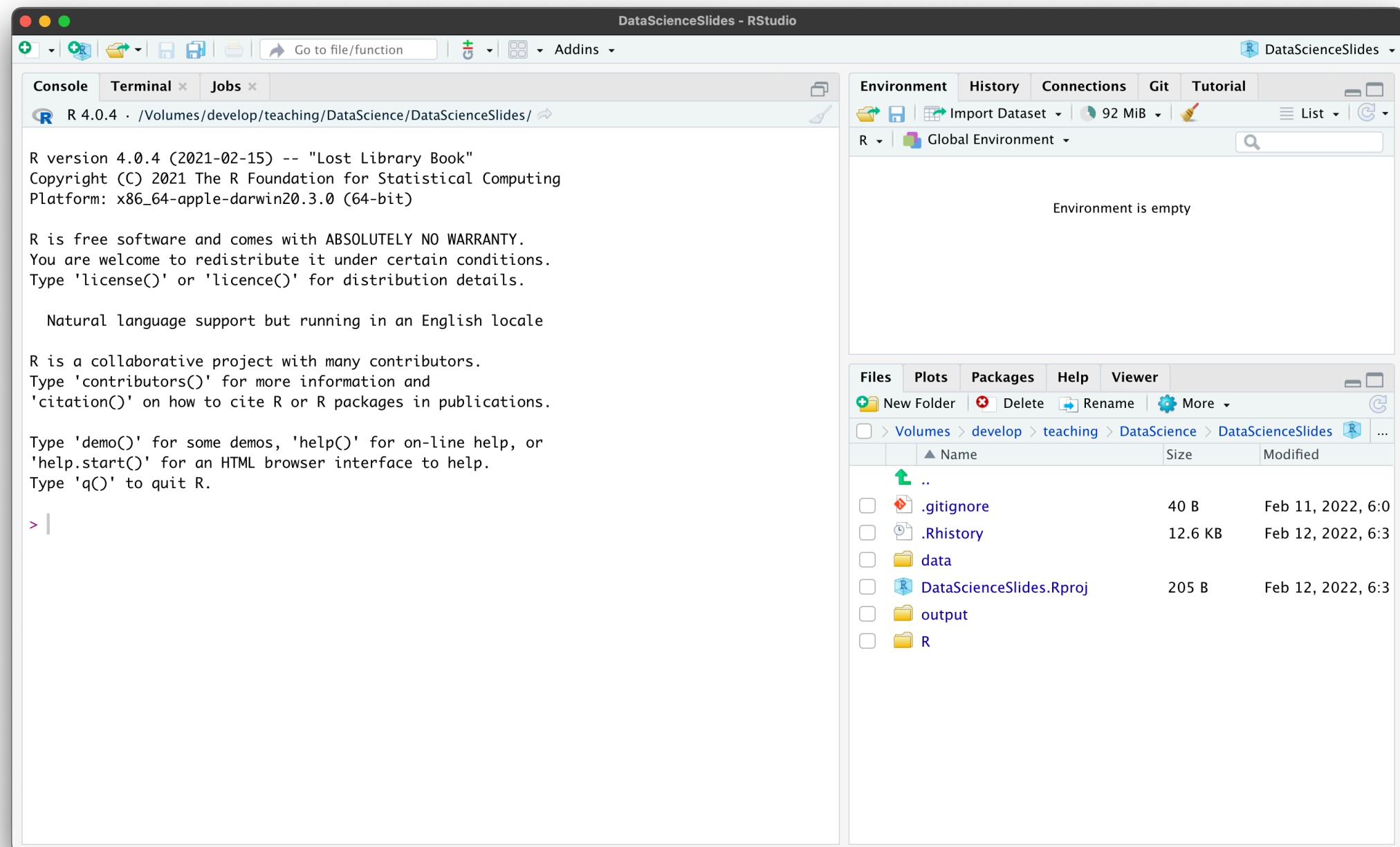
Goals for today

- I. Learn how to navigate the R-Studio interface and how to issue basic R commands
- II. Explore the concepts of objects, functions, and assignments
- III. Learn about R packages
- IV. Learn how to use and define functions

The R Studio interface

The R Studio interface

- After starting R-Studio, you will see something like this:



The R Studio interface

Some general settings

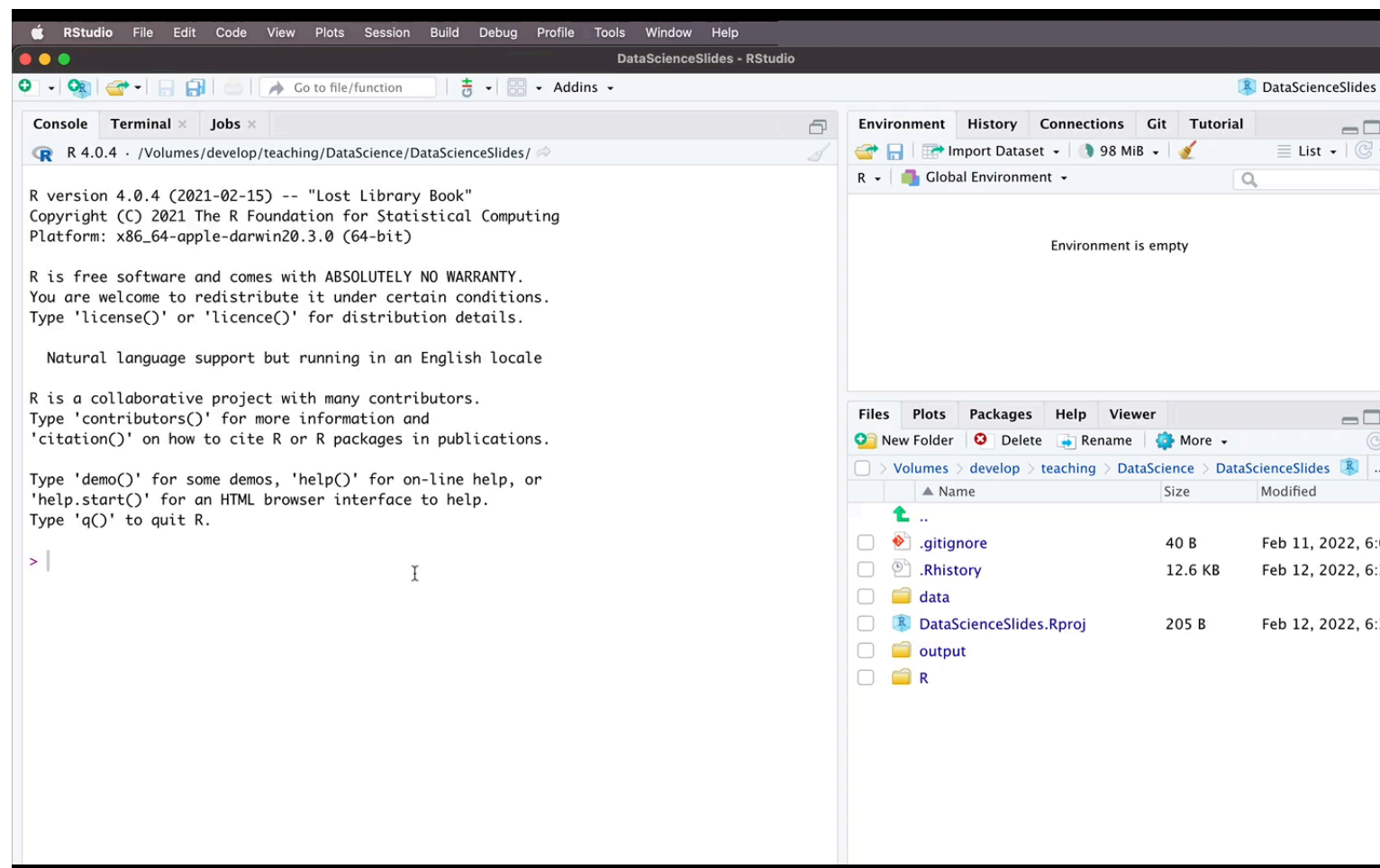
- As a first step, I recommend you to adjust some general settings:

- RStudio → Settings → General →

Workspace

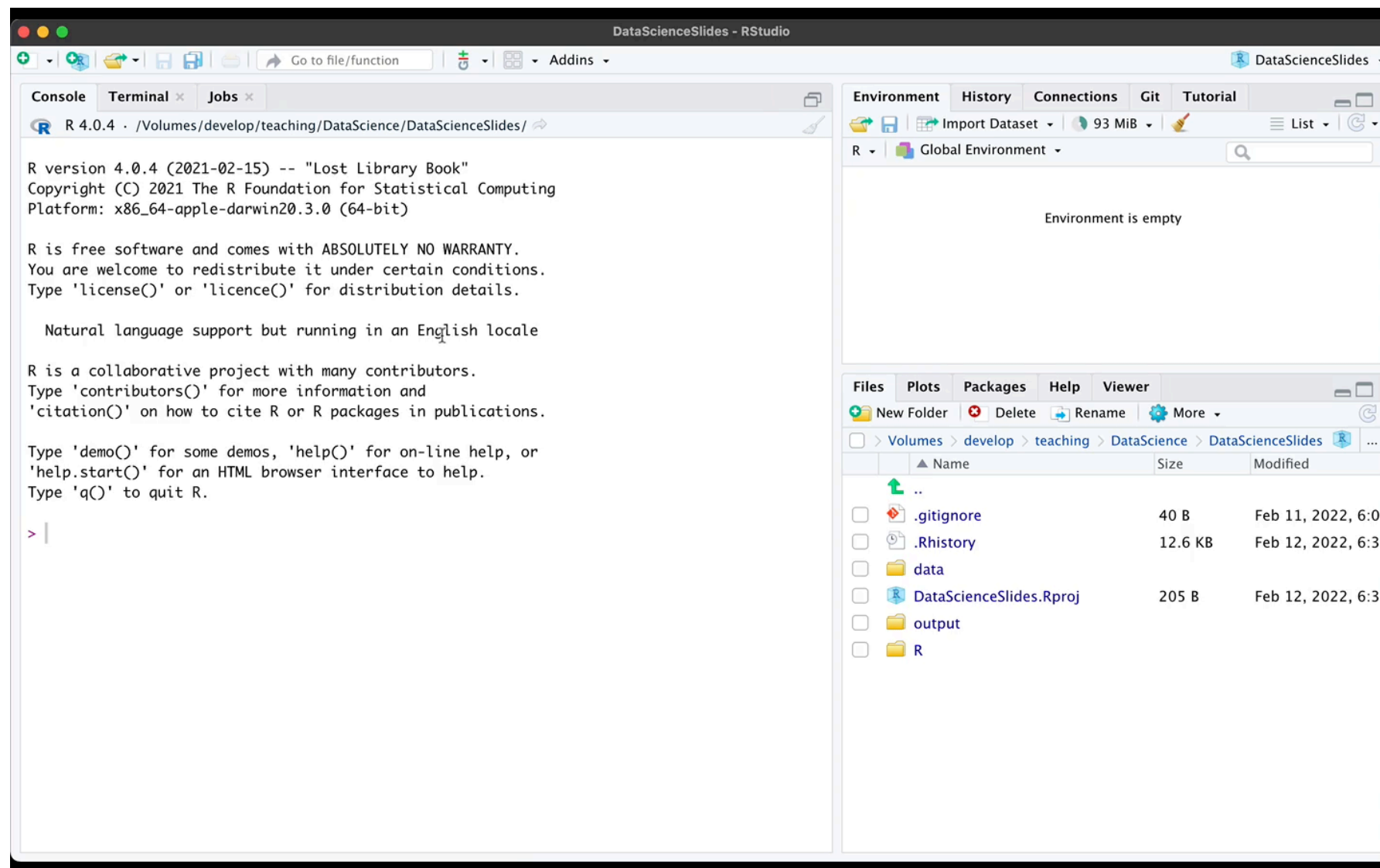
☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never ▾



The R Studio interface

- Create a new script and you will see R Studio in the way you work with it most of the time:

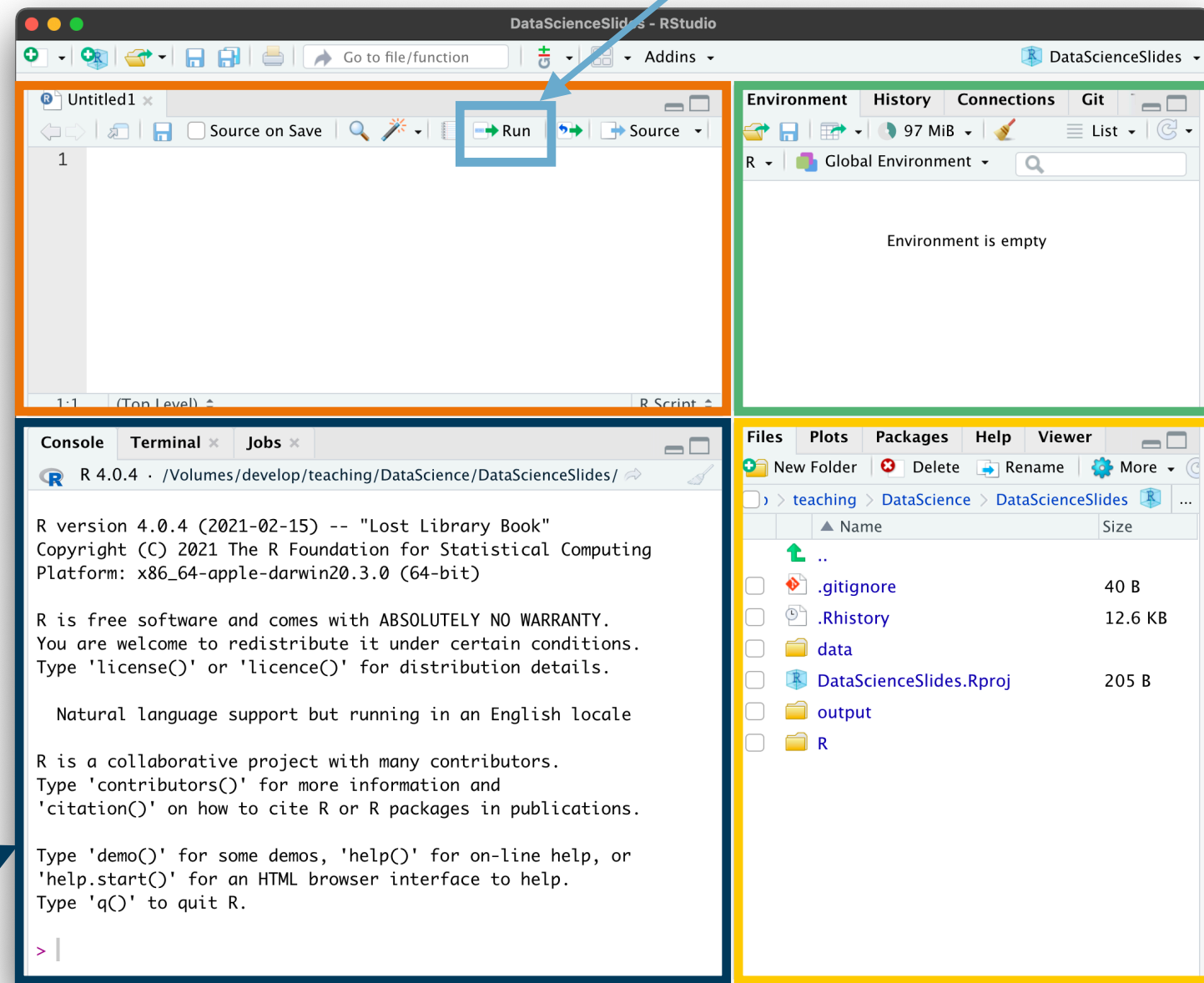


The R Studio interface

The most important elements

The run button: click here to execute marked part of a script in the console

The script editor: a 'fancy' text editor to modify R files



The environment: an overview over all objects you have defined so far

The console: used to issue commands to the computer directly

Your working directory: the files in your current project

We will learn about environments, projects, and working directories later!

Basic commands in R

- Now lets practice how to issue commands to R
- All the practical steps (and some additional information) are summarised in the section “**Issue commands to your computer**” of the tutorial R-basics on the course page

Intermediate task

- Sit together in groups of 2-3
- Execute the following mathematical computations via the console:

$$5 + 12$$

$$(2 \cdot 3)^2$$

- Write the following commands in R syntax in the script editor and execute the script:

$$2 \cdot 5.8$$

$$\frac{8^2 + 5^4}{3}$$

- What do you see?

Objects, functions, and assignments

Objects, functions, and assignments

“ To understand computations in R, two slogans are helpful:
Everything that exists is an object.
Everything that happens is a function call.

John Chambers


- Every number, function, letter, or whatever there is, is an object that is stored somewhere in the physical memory of your computer
- Whenever we tell our computer to do something via R, we are effectively calling a *function*
- The operation $2 + 3$ refers to three objects:
 - The numbers 2 and 3, as well as the function $+$ (addition)
 - It executes the addition function and produces a further object: the number 5

Assignments

- What if we wanted to keep the result of a computation for further use?

```
> 2 + 3  
[1] 5
```

This result is created after the addition has been executed and stored somewhere in your computer memory



- Since it is impossible to remember the precise location in the computer, the way to go is to give the result a **name**, and then later call it by this name
 - This process of binding an object to a name is called **assignment**
 - It is done by the function `assign()`:

```
> assign("int_results", 2 + 3)
```

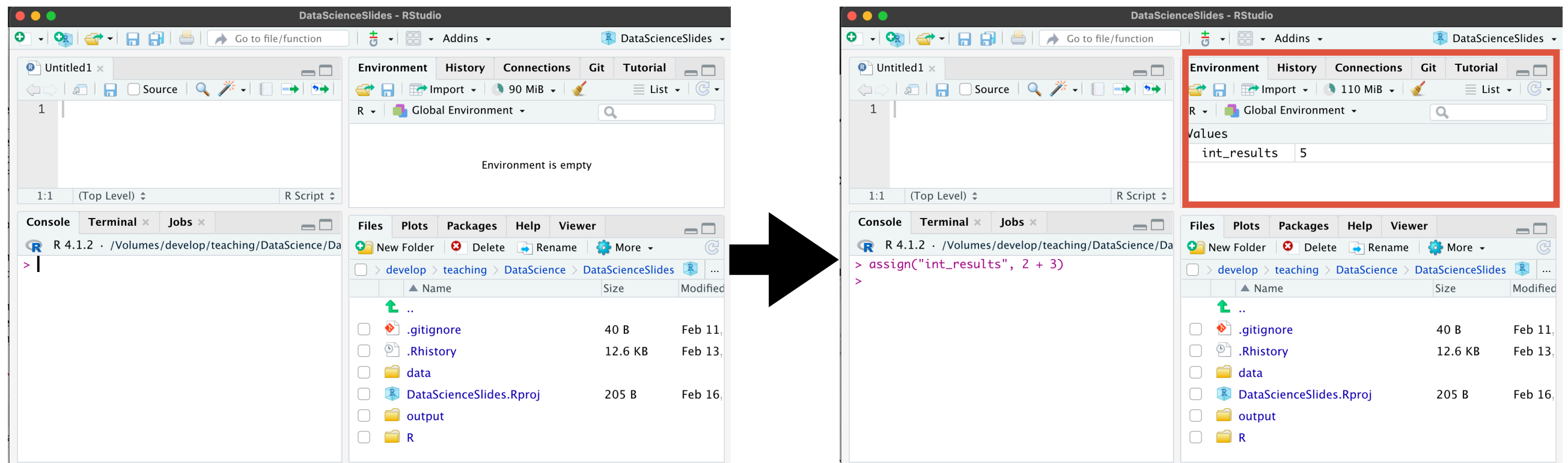
- The name `int_results` is now bound to the result of `2 + 3`!

Assignments

- You can now call the result by its name:

```
> assign("int_results", 2 + 3)
> int_results
[1] 5
```

- You see all the names currently given in the upper right pane of R-Studio:



Assignments - shortcuts, names, and removal

- Since assignments happen frequently, there is a shortcut to use `assign()`:
 - `assign("int_result", 2 + 3)` does the same as:
 - `int_result <- 2 + 3`
 - Tip: check out the keyboard shortcut for your OS (Mac: `⌘-`)
- Not all names are allowed → see the tutorial reading for more info
- You can remove an assignment by calling the function `rm()` on the name:

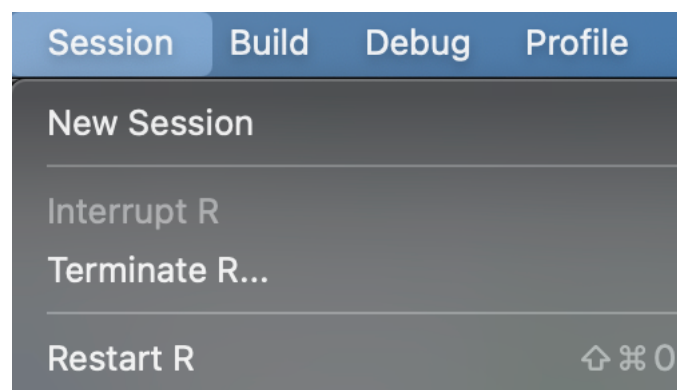
```
> x <- 2 + 2
> x
[1] 4
> rm(x)
> x
Error: object 'x' not found
```

Assignments - what about many of them?

- One object can have many names...
- ...but each name can only point to one single object:

```
> a <- 2      > a <- 2
> a <- 4      > a <- 4
> a           > a
               [1] 4
```

- Be aware not to overwrite important pre-defined assignments
- In the worst case: remove all assignment and restart R (Mac: ⌘ ⌘ 0)



Basic commands and assignments - Tasks

- Get again together in groups of 2-3
- Compute the following chain problem and assign a name to each intermediate result:

$$a = 2 + 3$$

$$b = \frac{5 \cdot a}{2}$$

$$c = (b + 1)^2$$

$$d = \sqrt{c}$$

- What is the result when you call d?

Functions

Functions

- A function is an algorithm, which takes an **input**, applies a **routine**, and returns an **output**:



- The function `log()`, for instance, computes the logarithm of a number:

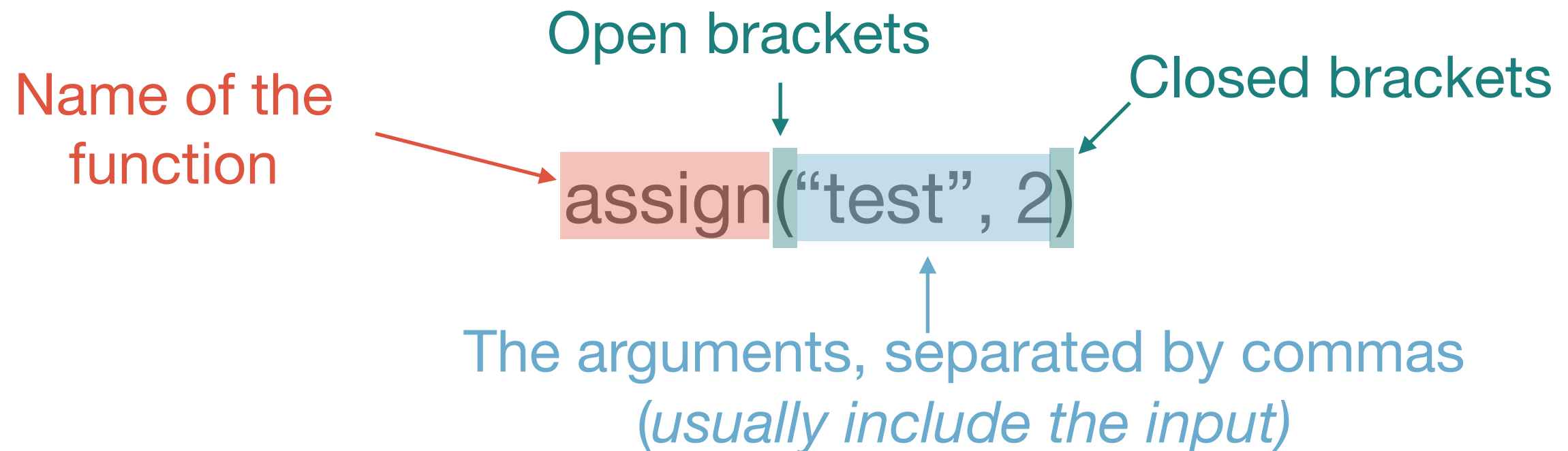


- Functions usually have **names** that we can use to call them
 - Two main ways to call a function: the *prefix* or *infix* form

Functions

Calling functions

- The most common form is the prefix form:



- Alternatively, we might use the infix form
 - Function name is written between the arguments, e.g.: `2 + 3`
 - Most common for mathematical operations → further readings

Functions

Calling functions

- There are two different types of arguments:
 - **Mandatory** arguments and **optional** arguments
- Mandatory arguments usually represent the function input
- Optional arguments allow you to specify details on how the function routine should be executed
 - While mandatory arguments *can* be specified via their name, optional arguments usually *must* be specified via their name
- Let's look at the example of `mean()`, a function that computes the mean.

Functions

Calling functions - mandatory arguments

- We first use the function `c()` – which stands for *concatenate* – to create a vector of numbers:

```
t_vec y <- c(1, 2, 3, 4)
```

- We then want to use `mean()` to compute the mean of this set of numbers:

```
mean(t_vec)
```

- The first (mandatory) argument of `mean()` is called `x` and means the set of which the mean should be computed
- Being a mandatory argument we can, but do not need to specify it:

```
mean(x=t_vec)
```

Functions

Calling functions - optional arguments

- Among others, `mean()` also accepts an optional argument called `na.rm`
 - It specifies how `mean()` should deal with missing values in the original input
 - If `na.rm` equals `TRUE`, then missing values (`NA`) are removed before the mean gets computed, if `na.rm` equals `FALSE`, then they are not
- We set this value by writing the name of the optional argument followed by `=` and the value:
- Lets add a missing value to our original vector to see the difference:

```
t_vec <- c(1, 2, 3, 4, NA)
```

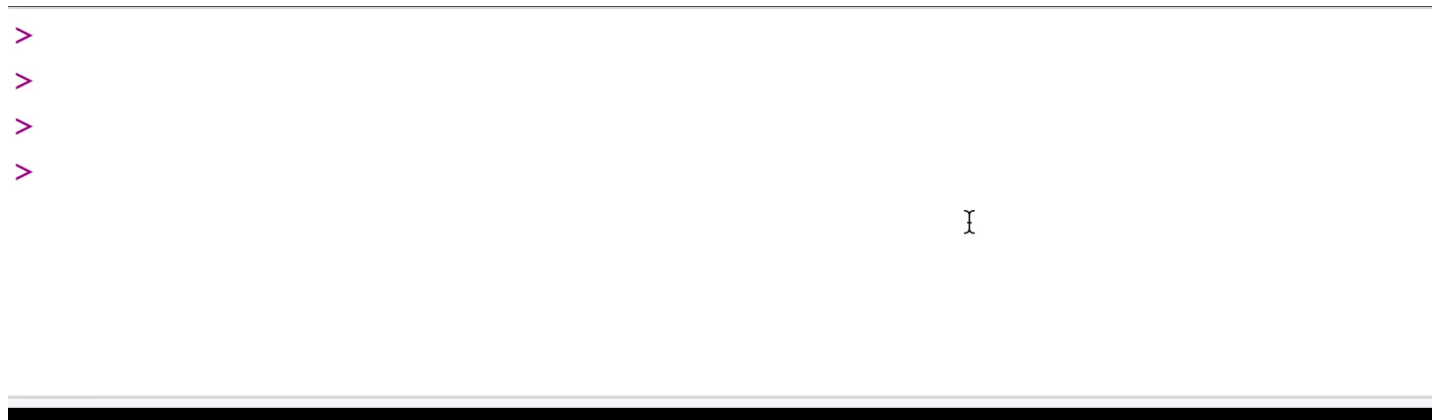
- Now test how the three applications of `mean()` differ:

```
mean(t_vec) vs. mean(t_vec, na.rm=TRUE) vs. mean(t_vec, na.rm=FALSE)
```

Functions

Calling functions - mandatory and optional arguments

- As all optional arguments, `na.rm`, has a default value that is chosen if you do not set another value explicitly
- How to know whether there are optional arguments, what are their defaults, or what the names of the arguments are?
- Use the Tab key after having written the open bracket:



- Call the function `help()`:
 - Here: `help(mean)`

Function calls - practice

- Define a vector with the elements -2, 2, 4, 6, 9 and NA
- Apply the following functions and understand what they are doing:
`median()`
`is.na()`
`anyNA()`
`sum()`
- There are two different ways to compute the variance of a vector: compute the population variance, or the sample variance. What does the function `var()` do? How can you compute the other version in R?

Defining your own functions

- Knowing how to define your own functions important for two reasons:
 - Defining own functions is super useful and often recommendable
 - It allows us to better understand how functions work in general
- We define a new function via the function `function()` 🙈
 - Let's look at the definition and go through it in practice!

Defining our own functions

The name of the new function
and the association operator

The arguments of the new
function

```
pythagoras <- function(cathetus_1, cathetus_2){  
  hypo_squared <- cathetus_1**2 + cathetus_2**2  
  hypotenuse <- sqrt(hypo_squared)  
  return(hypotenuse)  
}
```

Specifying what the function
returns as its output

The function body:
The routine the function should
apply to the input

*Note that all associations only
exist within the function!*

Final remarks about functions

- There are many reasons to use functions, e.g.:
 1. Code becomes more **concise and transparent**
 2. Functions help to **structure** your code
 3. Functions **facilitate debugging** and **help avoiding incidental mistakes**
- Before writing a function in daily life, check via Google whether it is not already written 🤔
- When developing a more complicated function, it usually a good idea to sketch your ideas with pen an paper, and then implement it
- Always document your functions → see the readings for a manual

Let's practice!

- Go together in pairs, one of you is the driver, the other the navigator
 - Only the driver writes code, the navigator tells her what to do
- After 5 minutes, exchange your work with another team. These two should try to understand what you have done
 - Then, sit together, give mutual feedback on your implementation and discuss open questions
- The task is to write a function that takes a set of numbers x as an input, and normalises them into the range of zero and one:

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- Two R functions that might come in handy are `min()` and `max()`
- **Bonus:** write a function that z-normalizes the vector!

Packages

R packages

- One cool thing about R is that there is a great community of R users that write objects and functions that perform useful purposes and makes them available to all
 - This process of ‘making available objects to others’ is done via the use of **R packages**
- You can think of an R package as a collection of assignments and documentations that people pass around
- If you install R, you can use all objects that...
 - ...you defined for yourself
 - ...are pre-defined in R
- If you want to use objects defined by someone else in her package you need to install this package

Installing packages

- The official way to distribute packages is via CRAN, the *The Comprehensive R Archive Network*
- To install a package that was deployed on CRAN you must execute the following command:

```
install.packages("NAME OF PACKAGE")
```

- To install the package `ineq`, for instance, do:

```
install.packages("ineq")
```

- To install packages that were not yet released on CRAN, other functions are available
- After having installed the `ineq` package, you can use all objects defined by it

Calling objects defined in packages

- One object defined in `ineq` is the function `Gini()`
 - Simply calling `Gini()` does, however, not work
- You need to tell R that `Gini()` is defined by the package `ineq`
- To do use, use `::`

```
ineq::Gini(c(1,2,3,4))
```

 - You may think of `::` as building a bridge between your R session and all objects defined in a package
- A sometimes more convenient way is to use the function `library()` at the beginning of your script:

```
library(ineq)
```
- This makes available all objects of `ineq` in your current R session

Packages and masking

- Packages are written by many different people
- It is not unlikely that two packages assign the same name to different objects
- If you then attach both packages, the assignment of the earlier package will be masked
 - Try this by attaching the two packages `dp1yr` and `p1m`
 - In these cases, you must use `::` to access the masked object of the first package
- As a general rule: always use `::` whenever masking is a potential problem → makes your code much easier to understand for you and others
 - Use the function `conflicts()` to see all names for which conflicts exist

Summary and outlook

- You made your first big steps into the R programming world 🦵🥳
 - We checked out the main elements of the R-Studio interface
 - We learned about how to issue commands to the computer
 - We learned that everything in R that exists is an object, and everything that happens is a function call
 - We learned about how to associate objects with names
 - We learned about how to call and define functions
- This was a lot → its a good idea to take your time to digest and repeat these topics

Outlook

Summary and outlook

- Next week we will...
 - ...learn about the different types of objects you can encounter in R
 - ...learn how to automate tasks with loops and conditionals
- Then we are finished with the general introduction and move to data visualisation the week thereafter

Tasks until next week:

1. Fill in the **quick feedback survey** on Moodle
2. Read the **tutorials** posted on the course page
3. Do the **exercises** provided on the course page and **discuss problems** and difficulties via the Moodle forum