

# Introduction to Data Analysis with R

Instructor: [Selina Baldauf](#)

Freie Universität Berlin - Theoretical Ecology

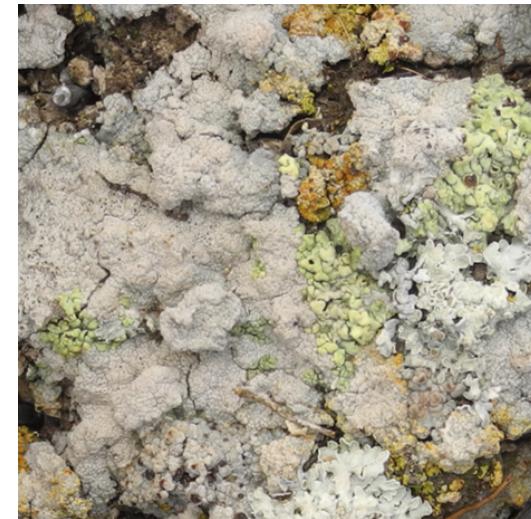


2021-06-15 (updated: 2021-08-01)

Artwork by [Allison Horst](#)

# Who am I?

- scientific modeller with ecology background
- working in the **theoretical ecology group** at Freie Universität
- PhD student at Freie Universität Berlin
  - Topic: Modelling the impact of biological soil crusts on dryland hydrology



## Teaching

- Statistics with R for Biology Master students
- Workshops on R packages, R development, ...



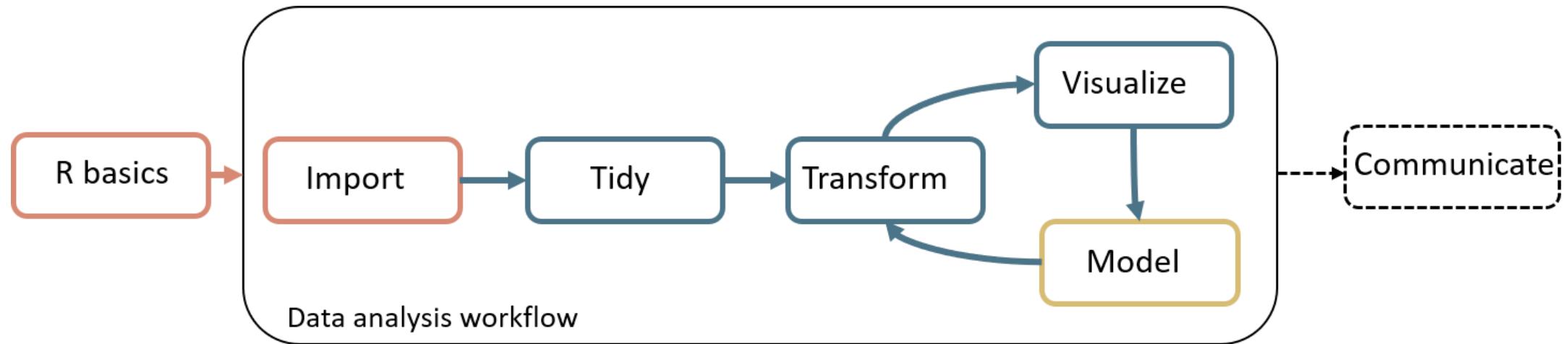
# For what do I use R?

- Short answer: everything
- Long answer:
  - modelling biocrusts
  - **cleaning data**
  - **visualizing data**
  - writing documents
  - presentations
  - **performing statistical analyses**
  - **wrangling data**
  - workflow automation
  - ...

# Who are you?

Results of questionnaire

# The Workshop: Topics



Day 1: Introduction to R and RStudio and data import

Day 2: Data analysis with the tidyverse

Day 3: Statistical models with R

Day 4: Bring your own data

# The Workshop: Schedule and Organization

⌚ 9 a.m. - 4 p.m. (💻 ~ 12 a.m. - 1 p.m.)

🔑 We will meet in the General meeting on Webex

## Organization

- input sessions
  - presentation and demonstration of a topic
  - some examples
- tasks regarding this topic
  - solve them in small groups
- joint discussion of tasks and additional questions

# The Workshop: Material

- All material can be found on the [workshop's website](#)
  - presentations
  - tasks
  - solutions
  - additional resources
- The complete material can be downloaded from the website after the workshop
- [Joint document](#) for all participants
  - collection of additional resources and helpful links that you found
  - things we could look at if we have enough time in the end
  - feedback and notes for me
  - collection of data for bring your own data day

# Bring your own data

## Learning by doing

- work on your own research data in small groups (or alone if you prefer)
  - I will also provide some real life data sets from different topics

## Idea

- first define a question/goal together and then work on it
- use any of the methods from the course or try new things, ...
- present your results at the end of the day

# Bring your own data - preparation

- keep the last workshop day in mind during the next days
    - remember if you learn something that might be applicable to your data
    - think of questions that you would like to answer for your data set
  - during the group work
    - if there is time after the tasks: talk about your data, research questions and methods
- you might already find common interests and questions

# Bring your own data - preparation

If you want to propose a group project using your data set:

- Add a description of the project idea to the joint [joint document](#)
- Prepare your data set to work seamlessly with R
  - We will learn about that today
- Send the data set to me beforehand then I can make sure that it's easy to get started with

If you want to work alone on your data set

- Add your name and the methods that you would like to use to the [joint document](#)

If you want to joint a group project

- go through the proposals and add your name next to it

# Before we get started I

This is the first time that I am doing this workshop.

So any feedback is very welcome.

I am curious to know

- what you liked/disliked
- which topics or methods you missed
- which tasks were too easy/too difficult or just right
- ...

Add any feedback to the feedback section of our [joint document](#) whenever something comes to your mind.

# Before we get started II

Did anyone have problems installing R and RStudio?



Download and install R from <https://cran.r-project.org>



Download and install RStudio from <https://www.rstudio.com>

Let's get started

# Why learn R?

There are many reasons to learn R. Some of them are

- free and open source
- lingua franca in computational statistics and data science
- regular updates
- large and supportive community
- packages for extensions
- easy documentation
- reproducible workflows
- ...

# First things first

An introduction to RStudio for R programming

# Difference between R and RStudio



R is the **programming language** and the **program** that does the actual work

- can be used with many different programming environments (But RStudio is the best for R)



RStudio is the **integrated development environment (IDE)**

- provides an interface to R
- specifically built around R code
- execute code
- syntax highlighting
- file and project management
- ...

→ You can use R without RStudio but RStudio without R would be of very little use

# Difference between R and RStudio

R is like the engine

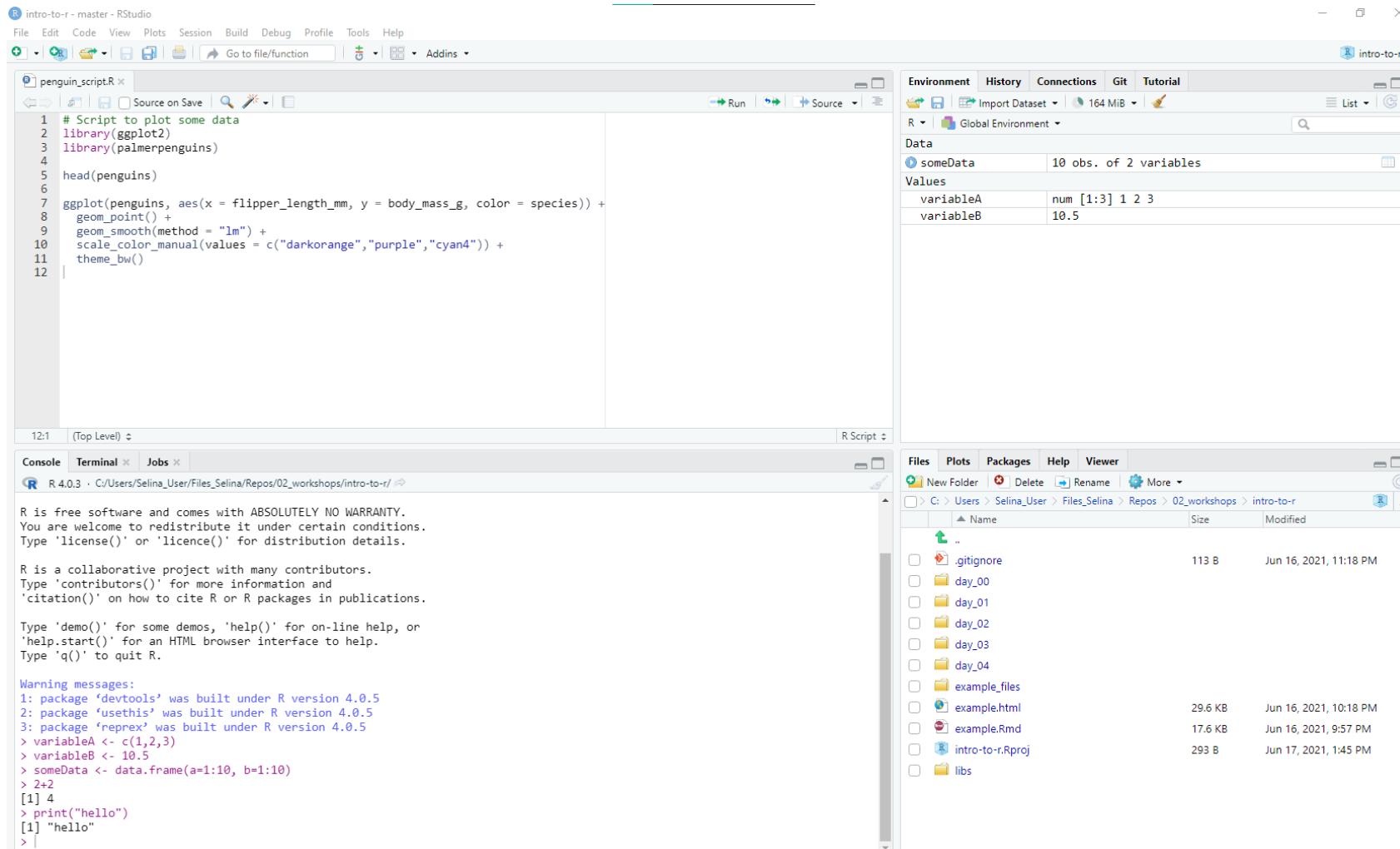


RStudio is more like the dashboard, etc.



analogy and image from [ModernDive Book](#)

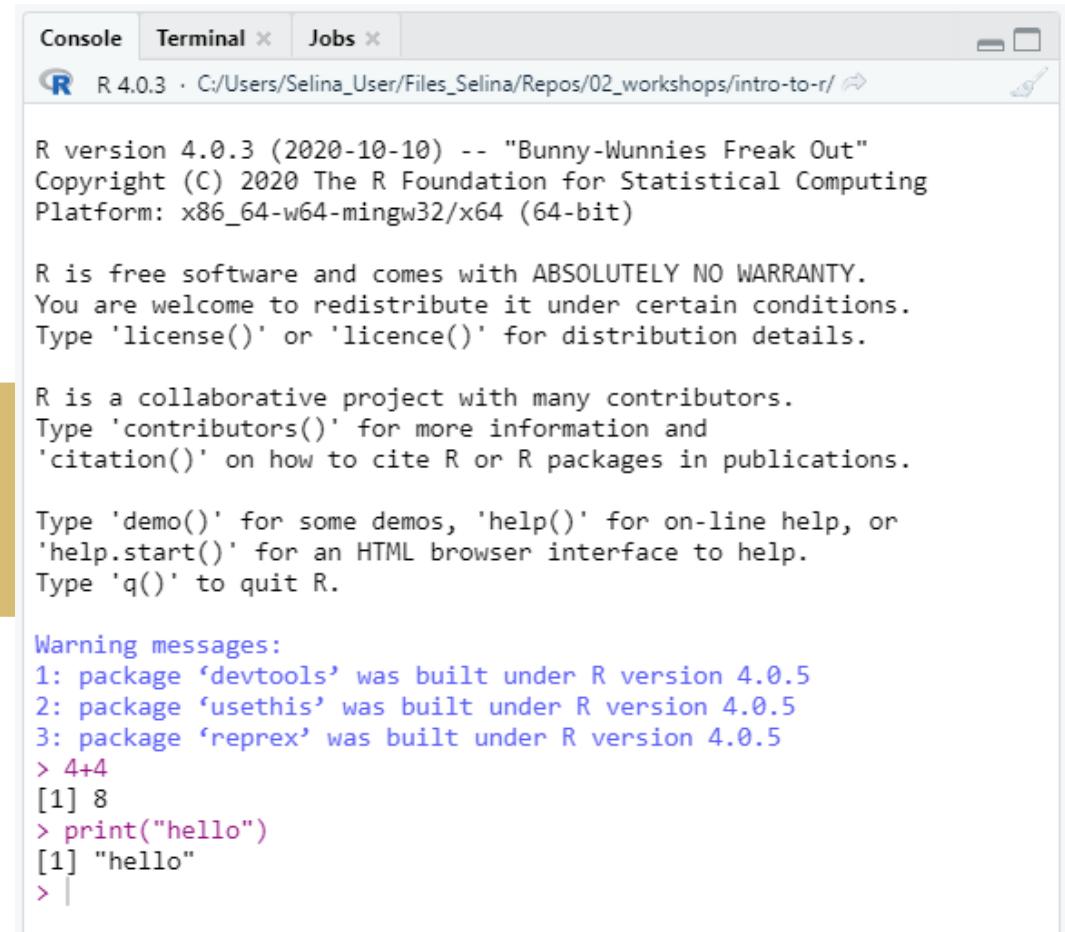
# A quick tour around RStudio



# Console pane

- execute R code
- output from R code in scripts is printed there
- type command into the console and execute with Enter/Return

💡 Use arrow keys to bring back last commands



R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"  
Copyright (C) 2020 The R Foundation for Statistical Computing  
Platform: x86\_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.

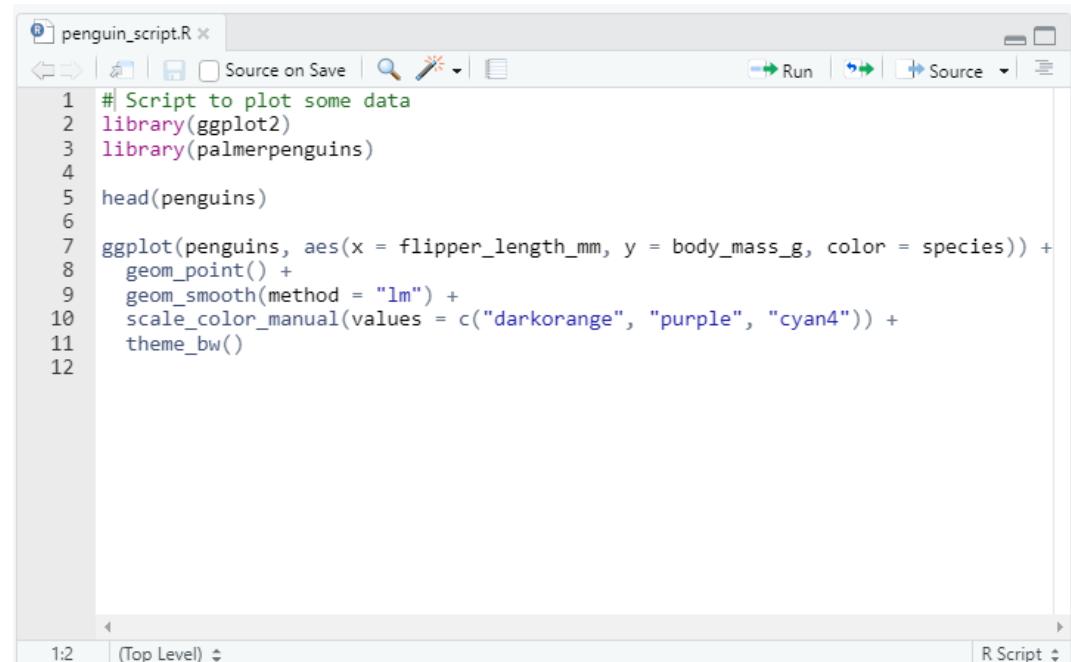
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.

Warning messages:  
1: package ‘devtools’ was built under R version 4.0.5  
2: package ‘usethis’ was built under R version 4.0.5  
3: package ‘reprex’ was built under R version 4.0.5

```
> 4+4  
[1] 8  
> print("hello")  
[1] "hello"  
> |
```

# Script pane

- write scripts with R code
  - scripts are text files with R commands (file ending `.R`)
  - use scripts to save commands for reuse



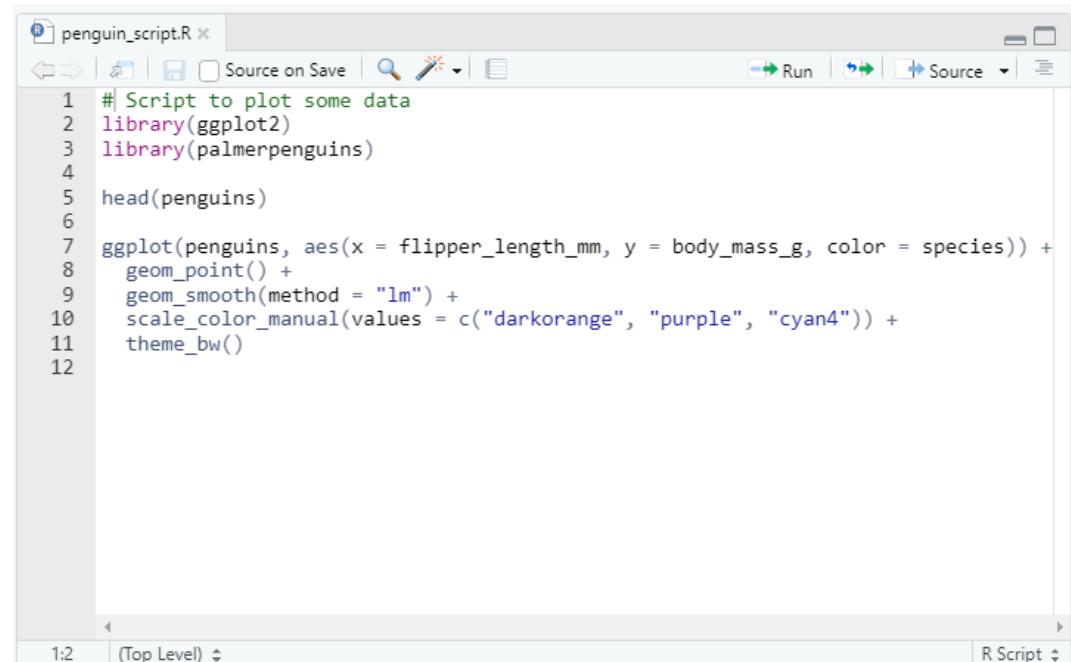
The screenshot shows an R script editor window titled "penguin\_script.R". The code in the editor is:

```
1 # Script to plot some data
2 library(ggplot2)
3 library(palmerpenguins)
4
5 head(penguins)
6
7 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
8   geom_point() +
9   geom_smooth(method = "lm") +
10  scale_color_manual(values = c("darkorange", "purple", "cyan4")) +
11  theme_bw()
12
```

The editor has a toolbar with icons for back, forward, save, run, and source. The status bar at the bottom shows "1:2 (Top Level) R Script".

# Script pane

- create a new R script:  
File -> New File -> R Script
- save an R script:  
File -> Save (Ctrl/Cmd + S)
- run code line by line with Run button (Ctrl + Enter/Cmd + Return)
- you can open multiple scripts at the same time



```
# Script to plot some data
library(ggplot2)
library(palmerpenguins)

head(penguins)

ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_color_manual(values = c("darkorange", "purple", "cyan4")) +
  theme_bw()
```

💡 Use **scripts** for all your analysis and for commands that you want to save.

💡 Use **console** for temporary commands, e.g. to test something.

# Environment pane

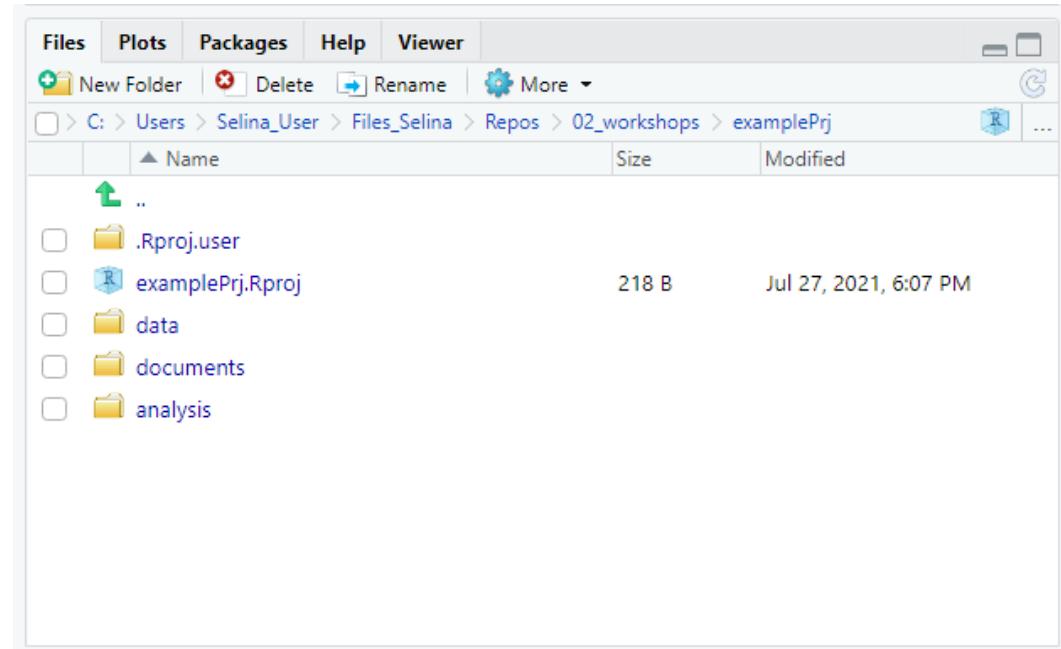
- shows objects currently present in the R session
- is empty if you start R

The screenshot shows the RStudio interface with the 'Environment' tab selected in the top navigation bar. The 'Global Environment' tab is also selected in the dropdown menu. The main area displays two objects:

someData	10 obs. of 2 variables
variableA	num [1:3] 1 2 3
variableB	10.5

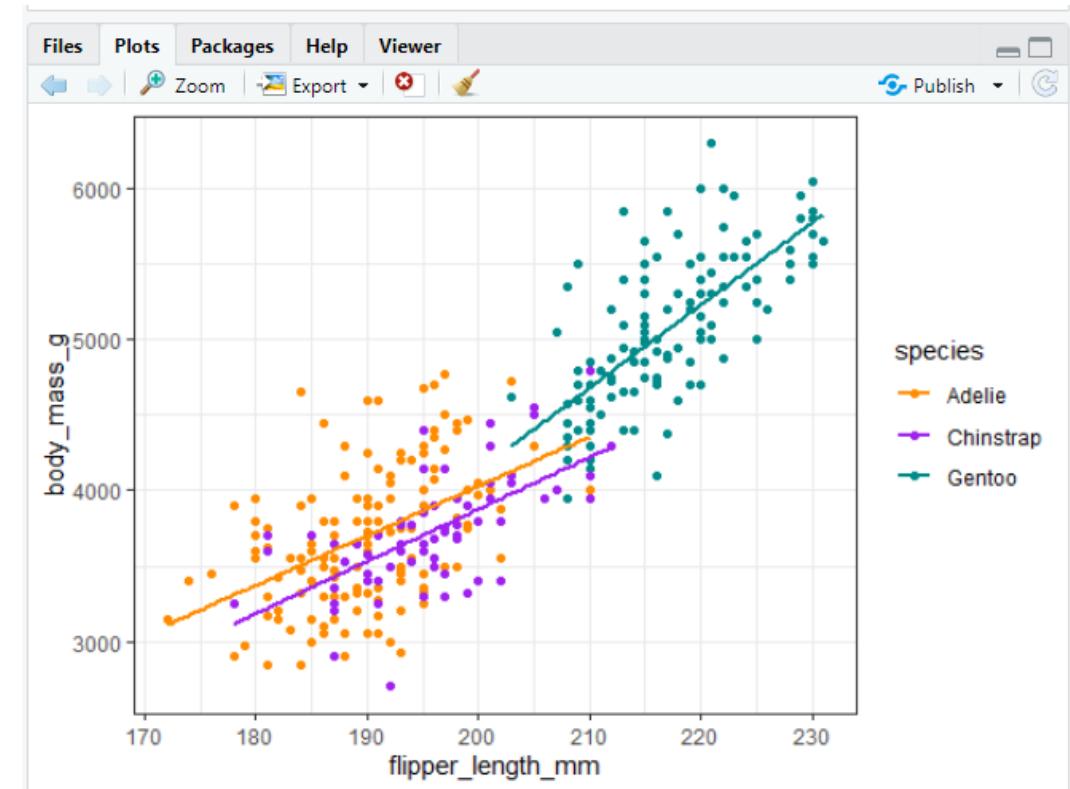
# Files pane

- similar to Explorer/Finder
- browse project structure and files
  - find and open files
  - create new folders
  - delete files
  - rename files
  - ...



# Plot pane

- Plots that are created with R will be shown here
- You can export plots by clicking on export button
  - but better to do it by code



# Project oriented workflow with RStudio

- one directory with all files relevant for project
  - scripts, data, plots, documents, ...
- an RStudio project is just a normal directory with an `*.Rproj` file
- advantages of using RStudio projects
  - easy to navigate in R Studio (File pane)
  - easy to find and access scripts in RStudio
  - project root is working directory
  - open multiple projects simultaneously in separate RStudio instances

```
Project
| - data
| - documents
|   | - notes
|   | - reports
|
| - analysis
|   | - clean_data.R
|   | - statistics.R
|
| - *.RProj
```

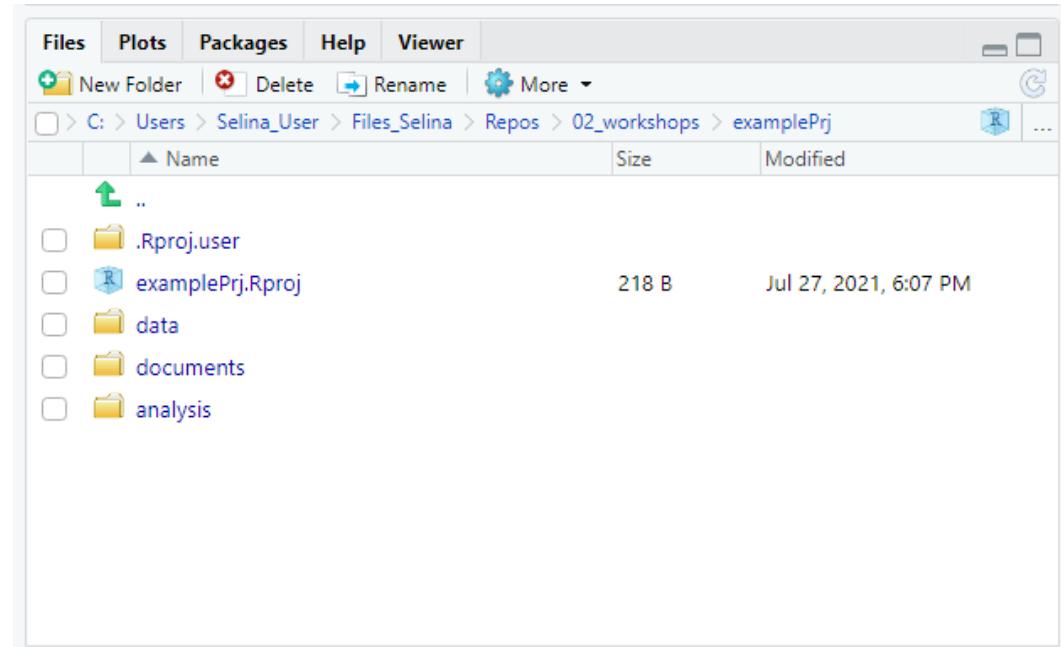
Example project structure

# Create an RStudio project

Create a project from scratch:

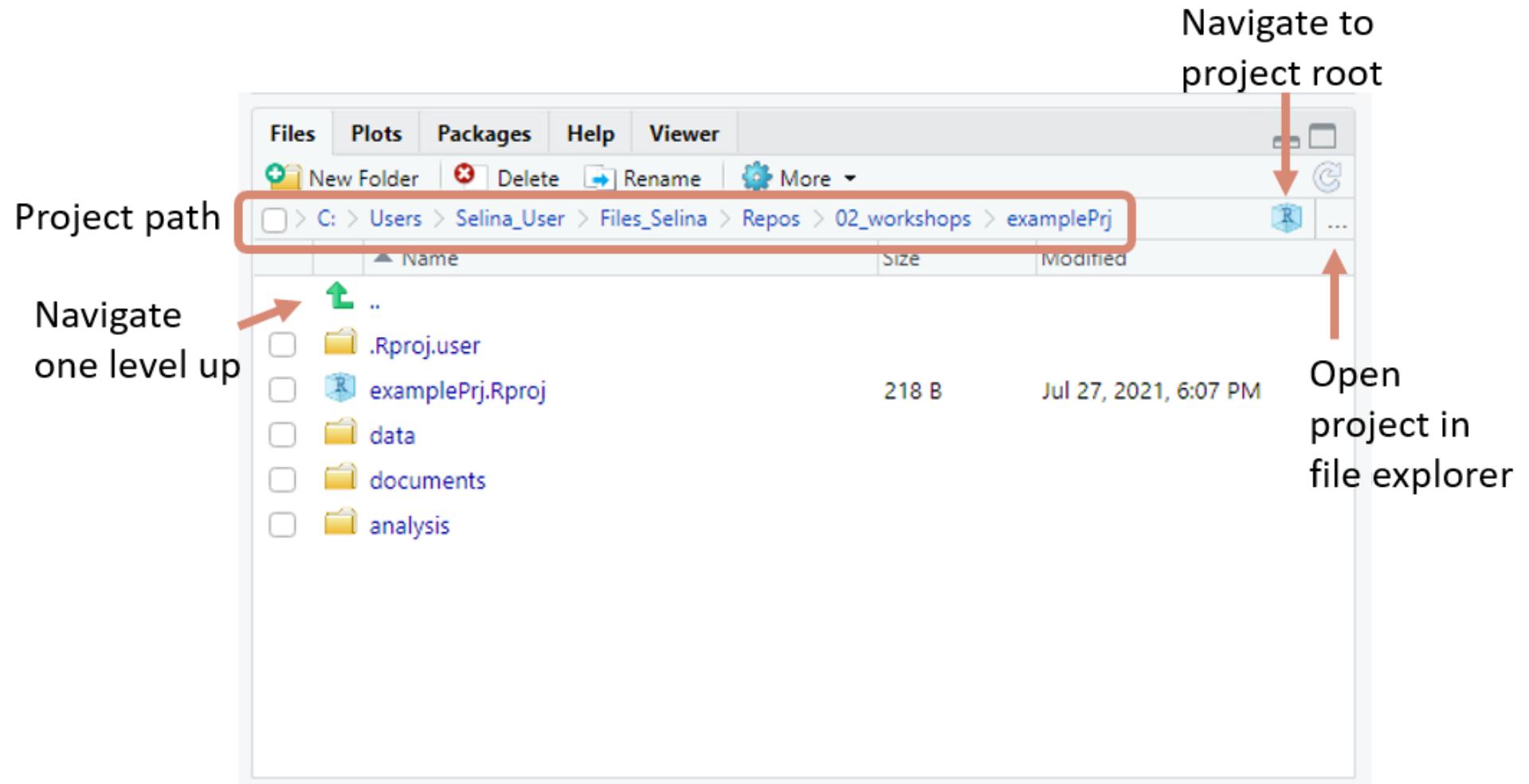
1. File -> New Project -> New Directory -> New Project
2. Enter a directory name (this will be the name of your project)
3. Choose the Directory where the project should be initiated
4. Create Project

RStudio will now create and open the project for you.



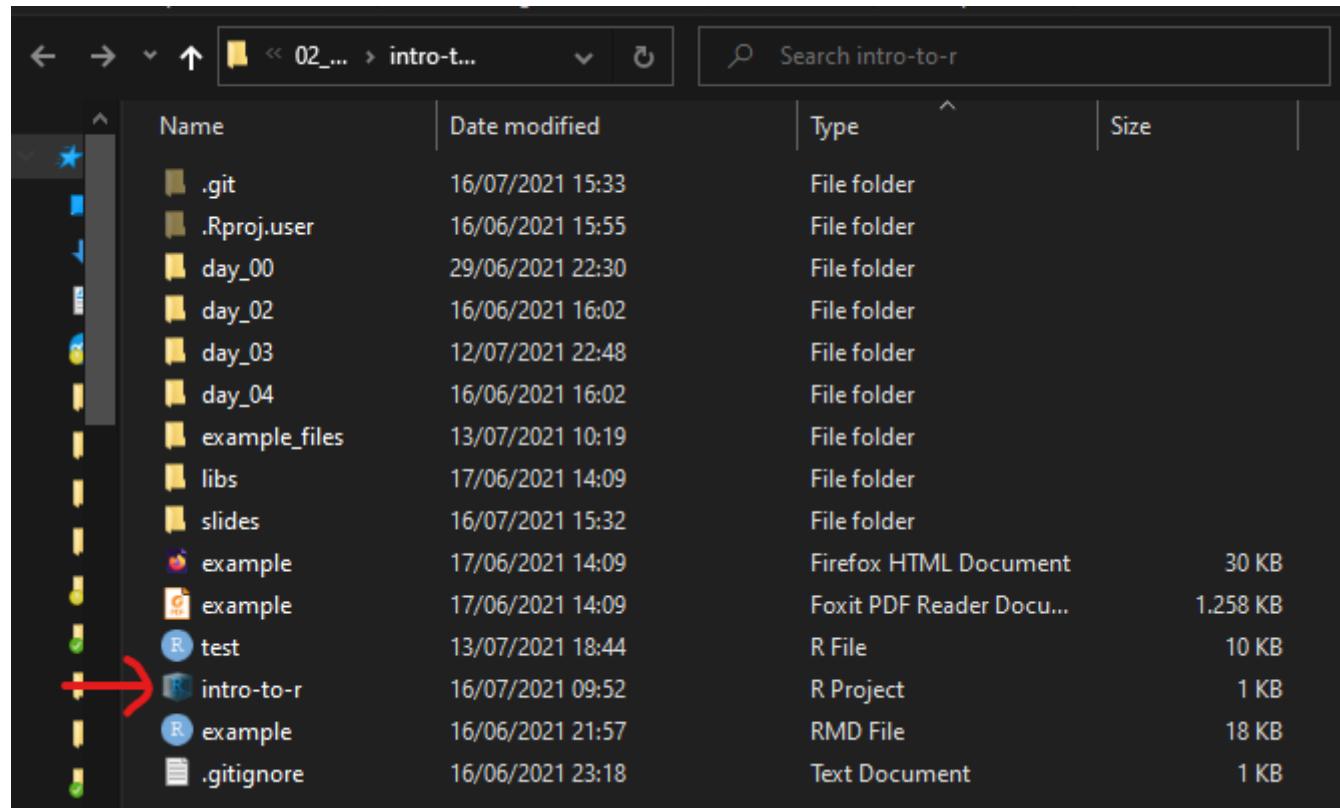
Example project structure in RStudio

# Navigate an RStudio project



# Open a project from outside RStudio

To open an RStudio project from your file explorer/finder, just double click on the \* .Rproj file



# Now you

Task 1: Set up your own RStudio project for this workshop

Find the task description [here](#)

# Introduction to

# R as a calculator

## Arithmetic operators

---

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%%
Power	^

---

```
# Addition  
2 + 2  
# Subtraction  
5.432 - 34234  
# Multiplication  
33 * 42  
# Division  
3 / 42  
# Modulo (Remainder)  
2 %% 2  
# Power  
2^2  
# Combine operations  
( (2 + 2) * 5 ) ^ (10 %% 10)
```

# R as a calculator

## Relational operators

Equal to

`==`

Not equal to

`!=`

Less than

`<`

Greater than

`>`

Less or equal than

`<=`

Greater or equal than

`>=`

```
2 == 2
```

```
## [1] TRUE
```

```
2 != 2
```

```
## [1] FALSE
```

```
33 <= 32
```

```
## [1] FALSE
```

```
20 < 20
```

```
## [1] FALSE
```

# R as a calculator

## Logical operators

---

Not

---

!

---

```
! TRUE
```

```
## [1] FALSE
```

```
!(3 < 1)
```

```
## [1] TRUE
```

# R as a calculator

## Logical operators

---

Not

!

And

---

&

```
(3 < 1) & (3 == 3) # FALSE & TRUE = FALSE
```

```
## [1] FALSE
```

```
(1 < 3) & (3 == 3) # TRUE & TRUE = TRUE
```

```
## [1] TRUE
```

```
(3 < 1) & (3 != 3) # FALSE & FALSE = FALSE
```

```
## [1] FALSE
```

# R as a calculator

## Logical operators

---

Not

!

And

&

Or

---

|

```
(3 < 1) | (3 == 3) # FALSE | TRUE = TRUE
```

```
## [1] TRUE
```

```
(1 < 3) | (3 == 3) # TRUE | TRUE = TRUE
```

```
## [1] TRUE
```

```
(3 < 1) | (3 != 3) # FALSE | FALSE = FALSE
```

```
## [1] FALSE
```

# Comments in R

```
# Reading and cleaning the data -----
data <- read_csv("data/my-data.csv")
# clean all column headers
# (found on https://stackoverflow.com/questions/68177507/)
data <- janitor::clean_names(data)

# Analysis -----
```

- notes that make code more readable or add information
- Everything that follows a `#` is a comment
- Comments are not evaluated
- Comments can be used for
  - Explanation of code (if necessary)
  - Include links, names of authors, ...
  - Mark different sections of your code (💡 try `Ctrl/Cmd + Shift + R`)

# Objects and data types in

# Variables

- store values under meaningful names to reuse them
- a variable has a **name** and **value** and is created using the **assignment operator**

```
radius <- 5
```

- variables are available in the global environment
- R is case sensitive: `radius != Radius`
- choose meaningful variable names
  - make your code easier to read
  - variable names should start with a letter

# Variables

```
# create a variable  
radius <- 5  
# use it in a calculation and save the result  
circumference <- 2 * pi * radius  
# change value of variable radius  
radius <- radius + 1
```

```
# print a variable's value -----  
radius # just use the name to print the value  
# or print it in a sentence  
print(paste0("With a radius of ", radius, " the circumference is ",  
            circumference))  
# does this print the correct circumference for radius?
```

```
# Remove variables from global environment -----  
rm(radius) # remove variable radius  
rm(list = ls()) # remove all elements
```

# Atomic data types

There are 6 so-called atomic data types in R. The 4 most important are:

**Numeric:** There are two numeric data types:

- **Double:** can be specified in decimal (1.243 or -0.2134), scientific notation (2.32e4) or hexadecimal (0xd3f1)
- **Integer:** numbers that are not represented by fraction. Must be followed by an L (1L, 2038459L, -5L)

**Logical:** only two possible values TRUE and FALSE (abbreviation: T or F - but better use non-abbreviated form)

**Character:** also called string. Sequence of characters surrounded by quotes ("hello", "sample\_1")

# Check the type of a variable

- check the type of a variable with `typeof()`
- check if a variable is of a specific data type with `is.*()`

# Check the type of a variable

```
var <- 123L  
typeof(var)
```

```
## [1] "integer"
```

```
is.double(var)
```

```
## [1] FALSE
```

```
is.integer(123)
```

```
## [1] FALSE
```

# Check the type of a variable

```
var2 <- TRUE  
is.logical(var2)
```

```
## [1] TRUE
```

```
is.character(var2)
```

```
## [1] FALSE
```

```
var3 <- "TRUE"  
is.logical(var3)
```

```
## [1] FALSE
```

```
is.character(var3)
```

```
## [1] TRUE
```

# Explicit type conversion

You can convert a value from one data type to the other using `as.*()`

```
as.character(1L)
```

```
## [1] "1"
```

`TRUE` is converted to 1 and `FALSE` is converted to 0:

```
as.numeric(TRUE)
```

```
## [1] 1
```

```
as.numeric(FALSE)
```

```
## [1] 0
```

# Explicit type conversion

Strings can be converted to numbers if possible

```
as.integer("hello")
```

```
## [1] NA
```

```
as.integer("2")
```

```
## [1] 2
```

# Implicit type conversion

Type conversion can happen implicitly

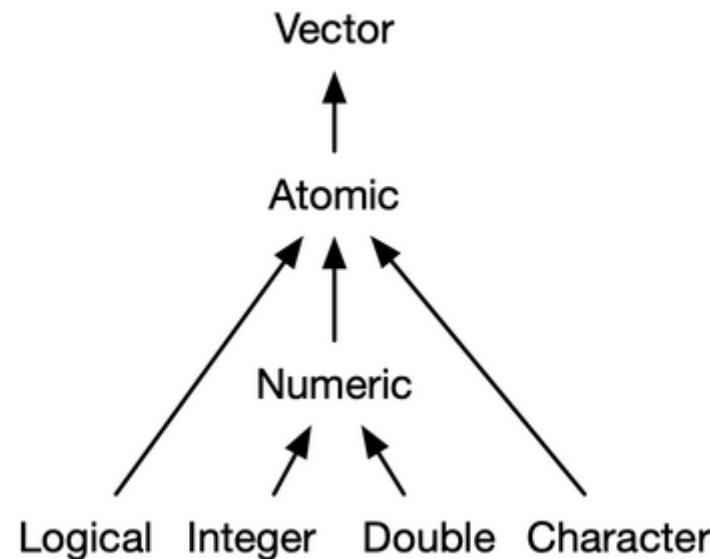
```
typeof(1L + 2.5) # integer -> double  
## [1] "double"  
  
typeof(1L + TRUE) # logical -> integer (TRUE = 1, FALSE = 0)  
## [1] "integer"  
  
typeof(1.34 + FALSE) # logical -> double  
## [1] "double"  
  
typeof("hello" + FALSE) # Error: no implicit conversion from string to other data types  
## Error in "hello" + FALSE: non-numeric argument to binary operator
```

# Vectors

# Vectors

Vectors are data structures that are built onto atomic data types.

Imagine a vector as a **collection of values** that are all of the same data type.



# Creating vectors: `c()`

Use the function `c()` to combine values into a vector

```
lg1_var <- c(TRUE, TRUE, FALSE)
dbl_var <- c(2.5, 3.4, 4.3)
int_var <- c(1L, 45L, 234L)
chr_var <- c("These are", "just", "some strings")
```

You can also combine multiple vectors into one:

```
# Combine multiple vectors
v1 <- c(1, 2, 3)
v2 <- c(800, 83, 37)
v3 <- c(v1, v2)
```

Be aware of implicit type conversion when combining vectors of different types

```
c(int_var, lg1_var)
```

```
## [1] 1 45 234 1 1 0
```

# Creating vectors: : and seq()

The `:` operator creates a sequence between two numbers with an increment of (-)1

```
1:10
```

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

The `seq()` function creates a sequence of values

```
seq(from = 1, to = 10, by = 1) # specify increment of sequence with by
```

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

```
seq(from = 1, to = 10, length.out = 10) # specify desired length with length.out
```

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

# Creating vectors: `rep()`

Repeat values multiple times with `rep()`

```
rep("hello", times = 5)
```

```
## [1] "hello" "hello" "hello" "hello" "hello"
```

You can also repeat entire vectors

```
rep(c(TRUE, FALSE, TRUE), times = 2) # repeat the whole vector twice
```

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

```
rep(c(1, 2, 3), each = 2) # repeat each element of the vector twice
```

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

# Working with vectors

Let's create some vectors to work with.

```
# list of 10 biggest cities in Europe
cities <- c("Istanbul", "Moscow", "London", "Saint Petersburg", "Berlin", "Madrid", "Kyiv",
"Rome", "Bucharest", "Paris")

population <- c(15.1e6, 12.5e6, 9e6, 5.4e6, 3.8e6, 3.2e6, 3e6, 2.8e6, 2.2e6, 2.1e6)

area_km2 <- c(2576, 2561, 1572, 1439, 891, 604, 839, 1285, 228, 105 )
```

We can check the length of a vector using the `length()` function:

```
length(cities)
```

```
## [1] 10
```

# Working with vectors

Divide the population and area vector to calculate the population density in each city:

```
population / area_km2
```

```
## [1] 5861.801 4880.906 5725.191 3752.606 4264.871 5298.013 3575.685 2178.988 9649.123
## [10] 20000.000
```

The operation is performed separately for each element of the two vectors and the result is a vector.

The same happens, if a vector is divided by vector of length 1 (i.e. a single number). The result is always a vector.

```
mean_population <- mean(population) # calculate the mean of vector population
population / mean_population
```

```
## [1] 2.5549915 2.1150592 1.5228426 0.9137056 0.6429780 0.5414552 0.5076142 0.4737733 0.3722504
## [10] 0.3553299
```

# Working with vectors

We can also work with relational and logical operators

```
population > mean_population
```

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

The result is a vector containing `TRUE` and `FALSE`, depending on whether the city's population is larger than the mean population or not.

Logical and relational operators can be combined

```
# population larger than mean population OR population larger than 3 million
population > mean_population | population > 3e6
```

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

# Working with vectors

The `%in%` operator checks whether *multiple* elements occur in a vector.

So instead of writing this:

```
cities == "Istanbul" | cities == "Berlin" | cities == "Madrid"  
## [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE
```

we can also write this:

```
to_check <- c("Istanbul", "Berlin", "Madrid")  
cities %in% to_check  
## [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE
```

For each element of `cities`, `%in%` checks whether that element is contained in `to_check`

# Indexing vectors

You can use square brackets `[]` to access specific elements from an object.

The basic structure is:

`vector [ vector of indexes to select ]`

```
cities[5]
```

```
## [1] "Berlin"
```

```
cities[1:3] # the three most populated cities
```

```
## [1] "Istanbul" "Moscow"    "London"
```

```
cities[length(cities)] # the last entry of the cities vector
```

```
## [1] "Paris"
```

# Indexing vectors

Change the values of a vector at specified indexes using the assignment operator `<-`

Imagine for example, that the population of

- Istanbul (index 1) increased to 20 Million
- Rome (index 8) changed but is unknown
- Paris (index 10) decreased by 200,000.

```
# first copy the original vector to leave it untouched
population_new <- population
# Update Istanbul (1) and Rome (8)
population_new[c(1, 8)] <- c(20e6, NA) # NA means missing value
# Update Paris (10)
population_new[10] <- population_new[10] - 200000
```

```
population_new
```

```
## [1] 20000000 12500000 9000000 5400000 3800000 3200000 3000000 NA 2200000 1900000
```

# Indexing vectors

You can also index a vector using logical tests. The basic structure is:

vector [ logical vector of same length ]

```
mega_city <- population > mean_population  
mega_city
```

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

Now extract only the cities that return `TRUE` for the comparison of their population against the mean population

```
cities[mega_city] # or short: cities[population > mean_population]
```

```
## [1] "Istanbul" "Moscow"    "London"
```

# Indexing vectors

We also use `%in%` for logical indexing.

```
population[ cities %in% c("Berlin", "Paris", "Stockholm", "Madrid") ]
```

```
## [1] 3800000 3200000 2100000
```

- returns only 3 values for population, because Stockholm is not a city in our vector
  - no city in `cities` returns `TRUE` for the comparison with `"Stockholm"`

# Summary I

- Variables have a name and a value and are created using the assignment operator `<-`, e.g.

```
radius <- 5
```

- Vectors are a collection of values of the same data type:
  - character ("hello")
  - numeric: integer (23L) and double (2.23)
  - logical (TRUE and FALSE)
- Be careful about implicit type conversion:
  - numeric to character
  - logical to character
  - logical to numeric

# Summary II

## Data types

```
# check the data type of a variable
typeof("hello")

# check if a variable is of a certain data type
is.*()
is.integer(1L)

# convert a variable into a certain data type
as.*()
as.logical("hello")
```

# Summary III

## Create vectors

```
# combine objects into vector
c(1,2,3)

# create a sequence of values
seq(from = 3, to = 6, by = 0.5)
seq(from = 3, to = 6, length.out = 10)
2:10

# repeat values from a vector
rep(c(1,2), times = 2)
rep(c("a", "b"), each = 2)
```

# Summary IV

## Indexing and subsetting vectors

```
# By index
v[3]
v[1:4]
v[c(1,5,7)]

# Logical indexing with 1 vector
v[v > 5]
v[v != "bird" | v == "rabbit"]
v[v %in% c(1,2,3)] # same as v[v == 1 | v == 2 | v == 3]

# Logical indexing with two vectors of same length
v[y == "bird"] # return the value in v for which index y == "bird"
v[y == max(y)] # return the value in v for which y is the maximum of y
```

# Summary V

## Working with vectors

```
# length  
length(v)  
# rounding numbers  
round(v, digits = 2)  
# sum  
sum(v)  
# mean  
mean(v)  
# median  
median(v)  
# standard deviation  
sd(v)  
# find the min value  
min(v)  
# find the max value
```

Now you

Task 2: Data types and vectors

Find the task description [here](#)