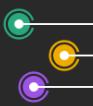


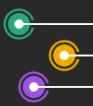
# *Reproducible Data Analysis with R*

— Introduction to R —

Cédric Scherer // R Course TU Dresden // Feb 27-Mar 3, 2023



# *Introduction to R (and RStudio)*



# What is R?

R is a programming language and free software environment for statistical computing and graphics.

R was conceived in 1992 by **Ross Ihaka** and **Robert Gentleman** as an open source implementation of the **S** programming language and released in 1995.

Since then, R has outgrown its original purpose and is used to:

- run statistical analyses and data-science workflows
- design high-level, publication-ready visualizations
- generate automated reports
- develop stand-alone web applications
- create presentation slides, books, and web pages



# Why (not) R?

## Pros:

- free, open source, and platform-independent
- wide range of dedicated packages provide extra functionality
- highly compatible with many other programming languages
- strong capabilities for data analysis and visualization
- huge (and open) online community
- often experienced as *easy to code* (from a non-programmer perspective)

## Cons:

- performance: scalability, memory and speed
- steep learning curve (as likely any programming language)
- potential security issues (relevant for web applications)
- often experienced as *strange to code* (from a programmer perspective)

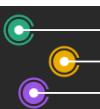


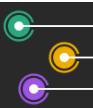


Illustration by [Allison Horst](#)

Cédric Scherer // R Course TU Dresden // Introduction to R



# R *versus* RStudio



# R Engine



# R Studio Dashboard



Modified from [ModernDive](#)

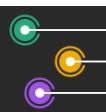
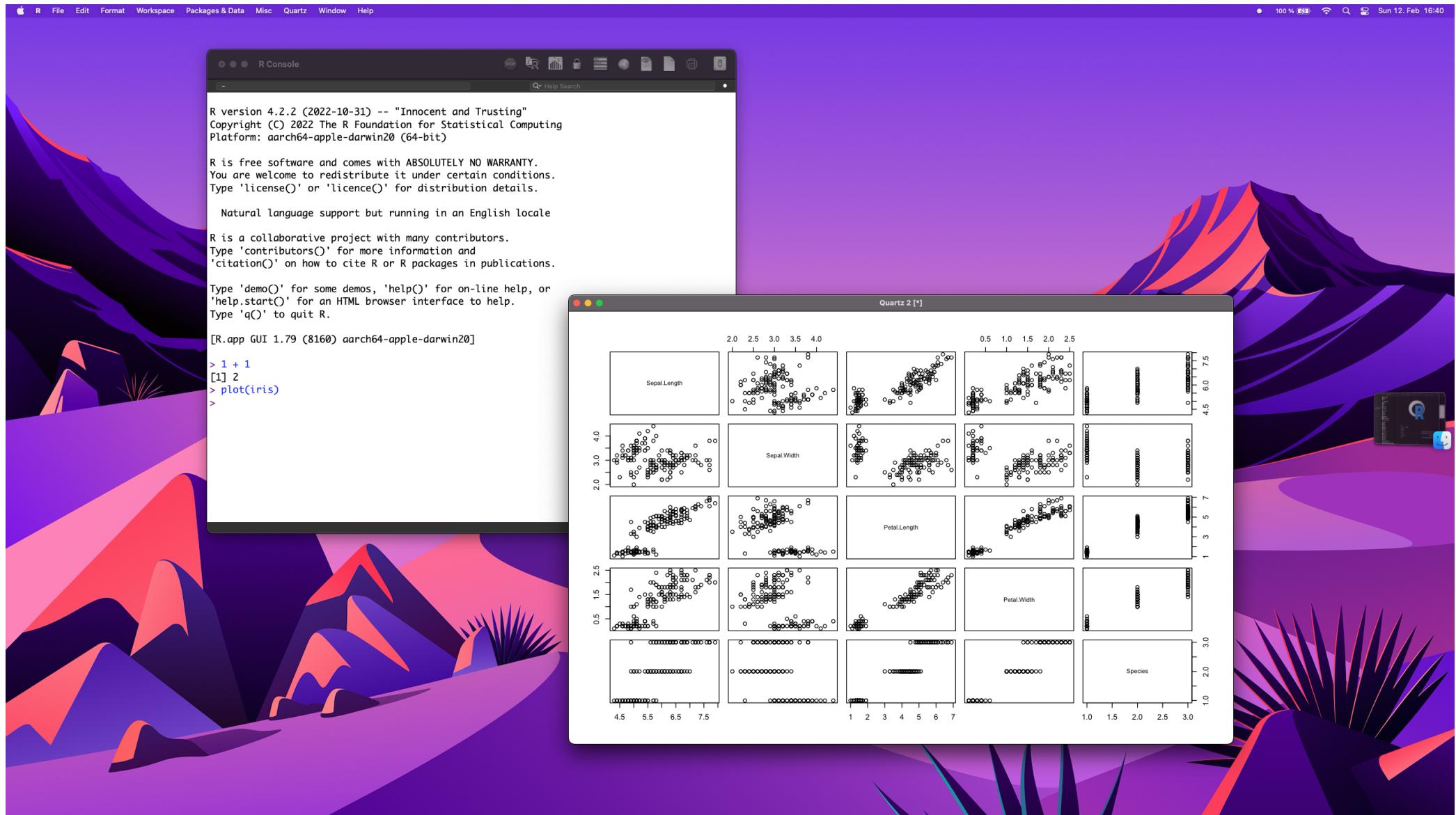
- **RStudio** is an open-source IDE (integrated development environment) for R
- most popular R IDE since several years (what is tinnR?!)
- many features + extensions to facilitate workflows (version control, toc, add-ins, ...)
- availability of R projects and Rmarkdown/Quarto (later more)

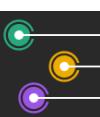
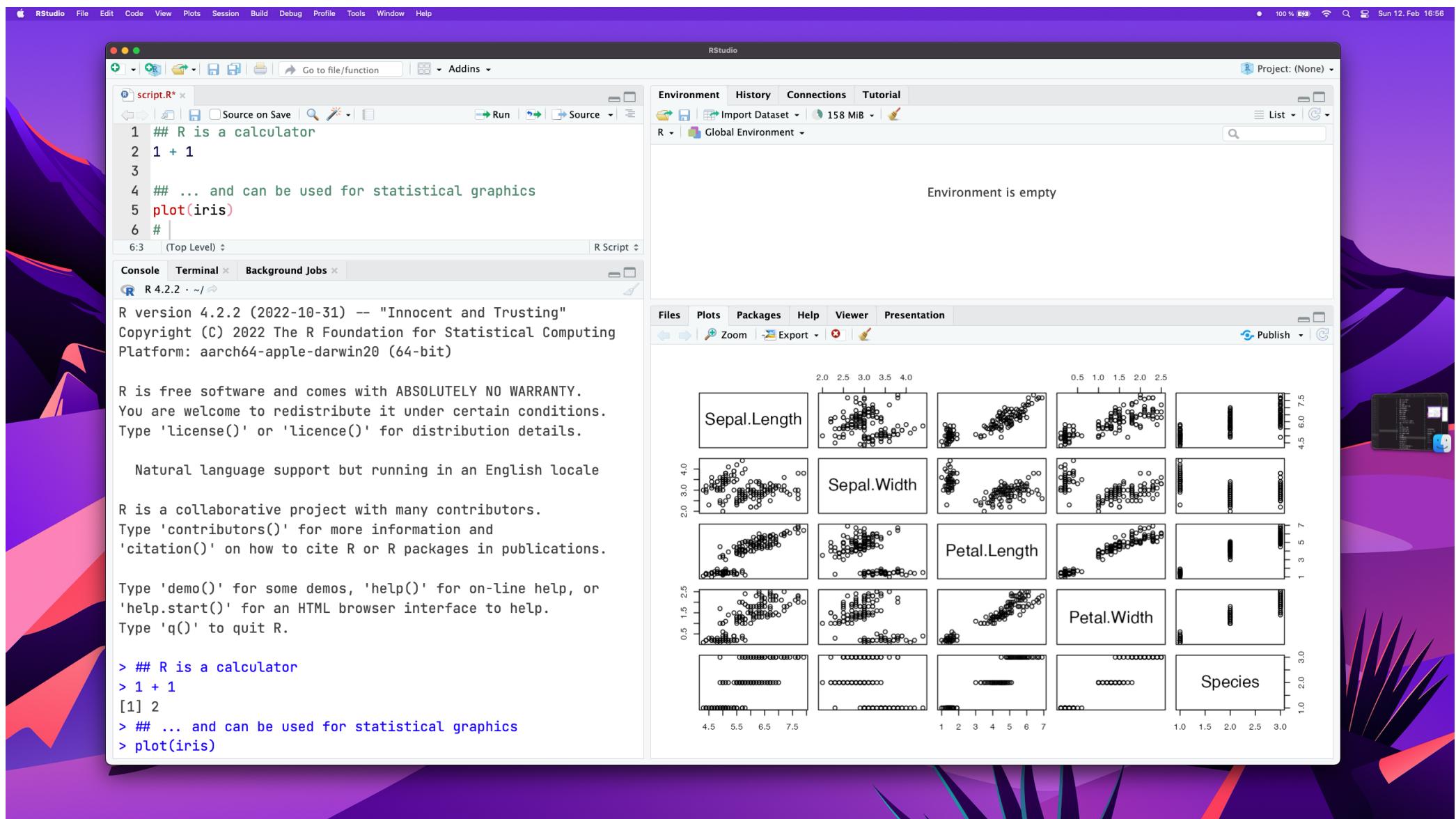


# Your Turn: Install R & RStudio

- Download and install R via  
[cloud.r-project.org](http://cloud.r-project.org)
- Download and install RStudio Desktop via  
[posit.co/download/rstudio-desktop](http://posit.co/download/rstudio-desktop)







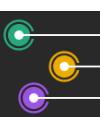
The image shows a screenshot of the RStudio interface on a Mac OS X desktop. The desktop background is a purple and pink abstract design. The RStudio window is open, displaying several panels:

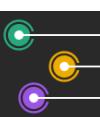
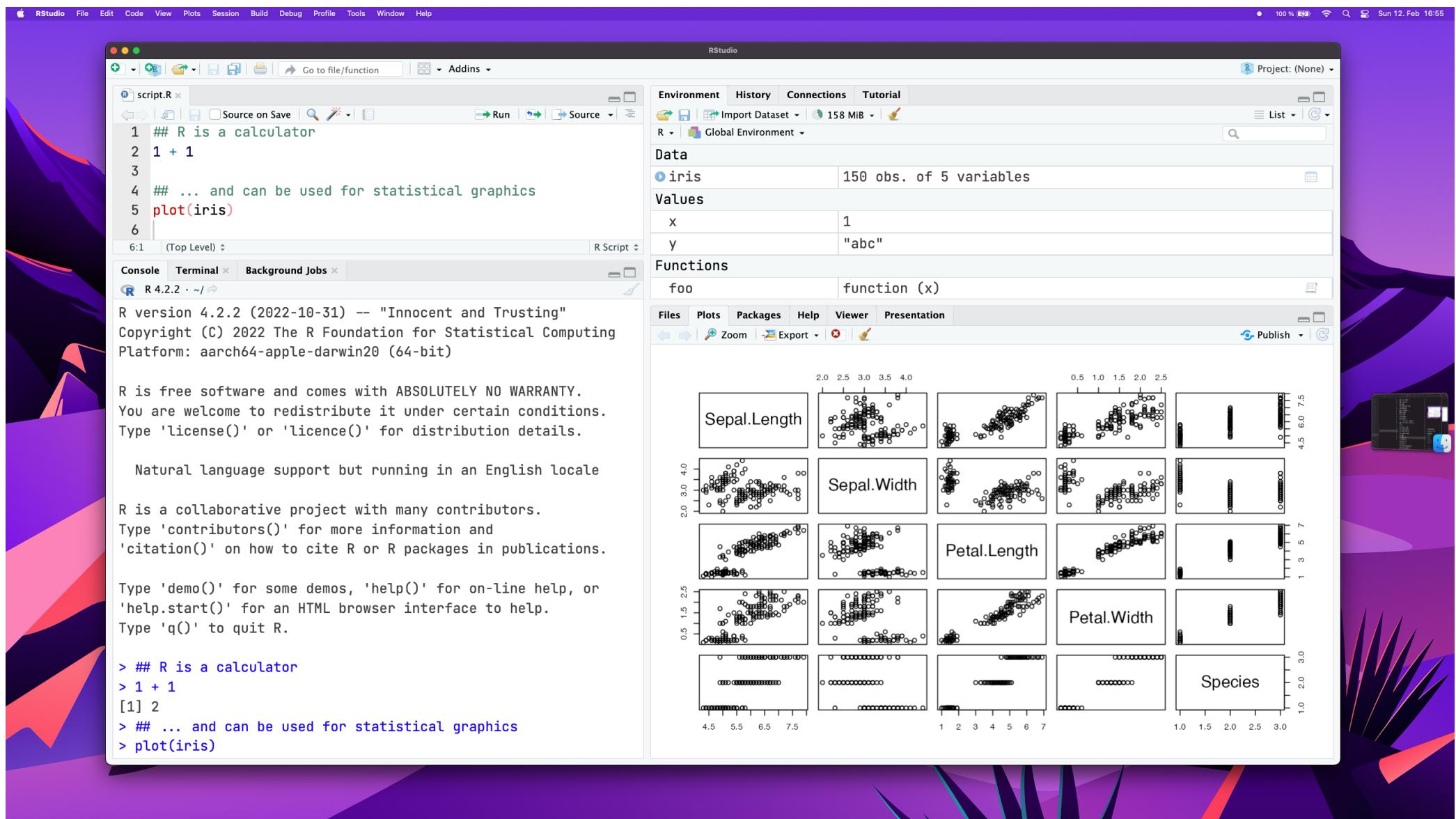
- Script Editor:** Shows a script named "script.R" with the following code:

```
1 ## ... and can be used for statistical graphics
2 1 +
3
4 ## ... and can be used for statistical graphics
5 plot(iris)
6
```
- Console:** Displays the R startup message and a help message about the Sepal.Length variable.
- Data View:** Shows the "iris" dataset with columns "Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width", and "Species".
- Environment View:** Shows the global environment with a function named "foo".
- Plots View:** Displays a 4x4 grid of plots for the "iris" dataset, including scatter plots and density plots.

Overlaid on the RStudio interface are large, bold text labels:

- Script + Environment + History, Build, Version Control** (over the Script and Environment panes)
- Console + Terminal, Background Jobs** (over the Console and Data panes)
- Plots + Files, Packages, Help, Viewer, Presentations** (over the Plots pane)





# Your Turn: “Hello World!”

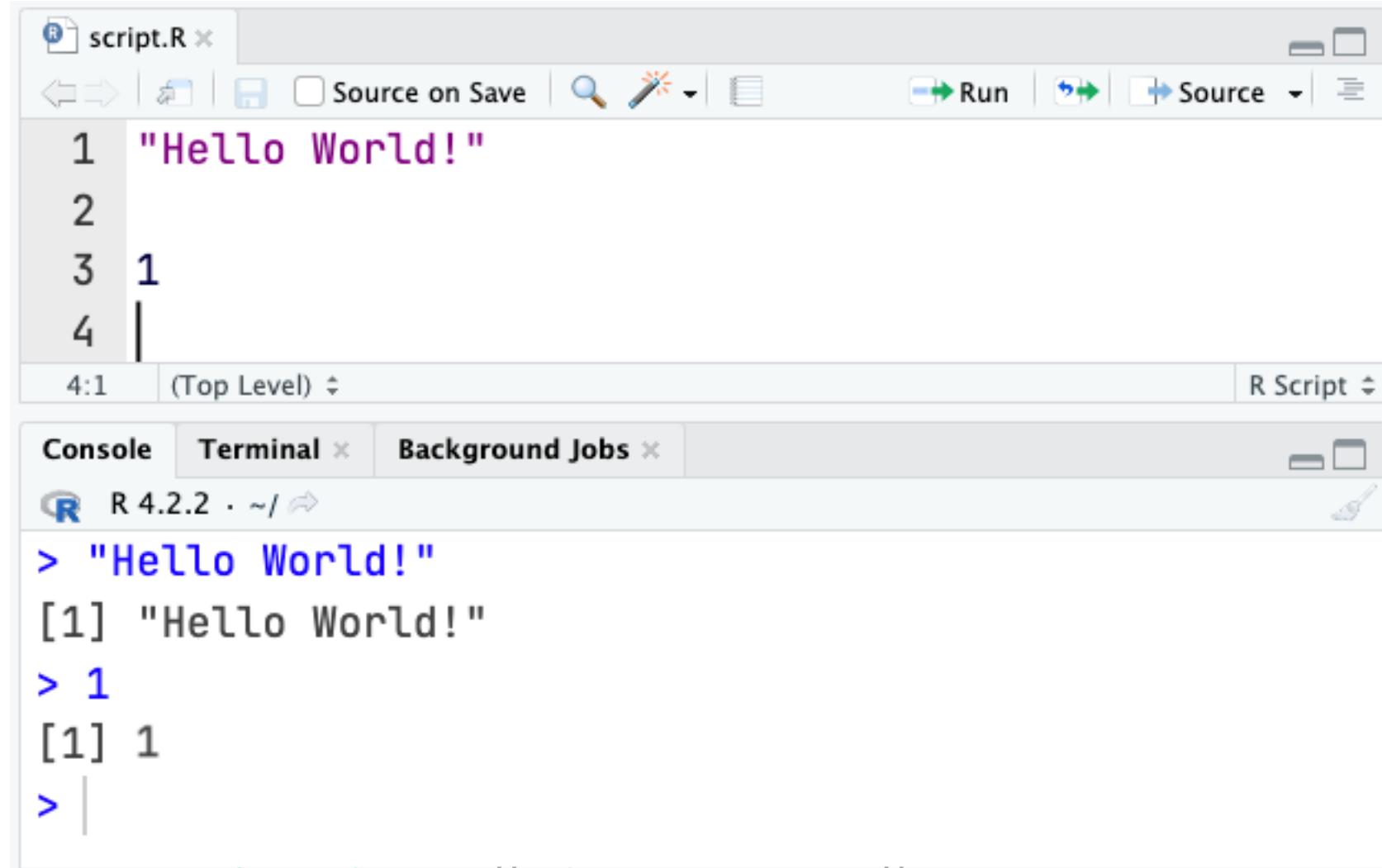
- Open **R** and run the following:
  - "Hello World!"
  - 1
- Open **RStudio** and get familiar with its environment.
  - Go to **Help > Cheatsheets > RStudio IDE Cheat Sheet** and study the document.
  - Go to **View > Panes > Pane Layout** and arrange the panes as you prefer.
  - Open a script via **File > New File... > R Script**, add the three commands from above and save the script.
  - Run the commands by placing your cursor in the first line of your script and clicking the **Run** button or via the shortcut **Cmd + Enter**.
  - Save the script.



# The Console

Code you are running appears here—the line starts with `>`.

It also prints the output—the line starts with `[1]`.



The screenshot shows the RStudio interface. The top panel is the 'script.R' editor, displaying the following R code:

```
1 "Hello World!"  
2  
3 1  
4 |
```

The bottom panel is the 'Console' tab, showing the output of the script:

```
R 4.2.2 · ~/Documents  
> "Hello World!"  
[1] "Hello World!"  
> 1  
[1] 1  
> |
```



# Comments

Comments are useful to

1. inactivate code that you don't want to run
2. add comments explaining your thoughts and reasoning

In R, comments are encoded by a hash **#**.

Everything on the same line will not be treated as actual code.

My personal convention is to use **#** to comment code and **##** to add comments:

```
1 # "Hello World!" ## I don't need this anymore but prefer to keep it
```

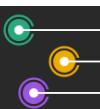


# R Projects

**R projects (or RStudio projects)** give you a solid workflow that will serve you well in the future:

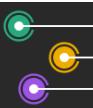
- it builds on the idea that all files associated with your project should be stored in the same folder
  - helps to find files
  - increases reproducibility
  - facilitates collaboration
- it sets the working directory to the folder the `.Rproj` files is placed in
  - ensures a correct working directory
  - independent of the operating setup and general folder structure
  - reduces hurdles when collaborating

Please re-open RStudio by double-clicking the `.Rproj` file.



# *Introduction to R*

## — Values —



# Values

1

“Hello World! ”

“2023-02-27 09:00:00”



# Values

```
1 1
```

```
[1] 1
```

```
1 "Hello World!"
```

```
[1] "Hello World!"
```

```
1 "2023-02-27 09:00:00"
```

```
[1] "2023-02-27 09:00:00"
```

```
1 x
```

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

```
1 "x"
```

```
[1] "x"
```

```
1 pi
```

```
[1] 3.141593
```



# *Introduction to R*

## — Functions —



# Your Turn: Arithmetic Operations

- Type `2 + 3` and run it.
  - Play around with other calculus operators such as `-`, `*`, `/`, or `^`.
  - Calculate the square root of a number with `sqrt()`.
- 
- What is the difference between the first 5 and the `sqrt` calculation?



# R is a Calculator!

```
1 2 + 3
```

```
[1] 5
```

```
1 (59 + 73 + 2) * 5
```

```
[1] 670
```

```
1 1 / 200 * 30
```

```
[1] 0.15
```

```
1 sin(pi / 2)
```

```
[1] 1
```

```
1 10^12 * sqrt(4312)
```

```
[1] 6.566582e+13
```

```
1 log(exp(5))
```

```
[1] 5
```



# Functions

`sqrt(x = 25)`

function name    argument name    argument value

The diagram illustrates the structure of the R code `sqrt(x = 25)`. At the top, the code is shown in a large, stylized font. Below it, three words—`function name`, `argument name`, and `argument value`—are aligned horizontally. Colored arrows point from each word to its corresponding part in the code: a purple arrow from `function name` to `sqrt`, a green arrow from `argument name` to `x`, and a yellow arrow from `argument value` to `25`.



# Functions

```
sqrt(x = 25)  
[1] 5
```

↑  
**return  
value**



# Functions

R has a large collection of built-in functions that are called like this:

```
1 function_name(arg1 = val1, arg2 = val2, ...)
```

We already have seen some functions in the exercise before:

`+, -, *, ^, sqrt(), log()` all are functions.



# Functions

```
1 sqrt()  
2 log()  
3 sin()  
4 exp()  
5 mean()
```

```
1 sqrt(x = 25)
```

```
[1] 5
```



# Infix Functions

```
1 +  
2 -  
3 ^  
4 %%  
5 %in%
```

## Unary Operator:

```
1 -1  
[1] -1
```

## Binary Operator:

```
1 12 %% 5  
[1] 2
```



# Function Body

```
1 log
```

```
function (x, base = exp(1)) .Primitive("log")
```

```
1 mean
```

```
function (x, ...)  
UseMethod("mean")  
<bytecode: 0x13cccf950>  
<environment: namespace:base>
```

```
1 methods(mean)
```

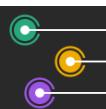
```
[1] mean.Date      mean.default    mean.difftime   mean.POSIXct    mean.POSIXlt    mean.quosure*  
[7] mean.vctrs_vctr*  
see '?methods' for accessing help and source code
```

```
1 `+`
```

```
function (e1, e2) .Primitive("+")
```

```
1 read.csv
```

```
function (file, header = TRUE, sep = ",", quote = "\"\"", dec = ".",
         fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
           dec = dec, fill = fill, comment.char = comment.char, ...)
<bytecode: 0x13ec81150>
<environment: namespace:utils>
```



# Function Body

```
1 read.table

function (file, header = FALSE, sep = "", quote = "\'\"", dec = ".",
  numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
  col.names, as.is = !stringsAsFactors, na.strings = "NA",
  colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
  fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
  comment.char = "#", allowEscapes = FALSE, flush = FALSE,
  stringsAsFactors = FALSE, fileEncoding = "", encoding = "unknown",
  text, skipNul = FALSE)
{
  if (missing(file) && !missing(text)) {
    file <- textConnection(text, encoding = "UTF-8")
    encoding <- "UTF-8"
    on.exit(close(file))
  }
  if (is.character(file)) {
    file <- if (nzchar(fileEncoding))
      file(file, "rt", encoding = fileEncoding)
    else file(file, "rt")
    on.exit(close(file))
  }
}
```



# Function Arguments

```
1 log(x = 25) ## `base = exp(1)` as default
```

```
[1] 3.218876
```

```
1 log(x = 25, base = 10)
```

```
[1] 1.39794
```

```
1 log10(x = 25) ## wrapper for `log(x, base = 10)`
```

```
[1] 1.39794
```

```
1 log2(x = 25) ## wrapper for `log(x, base = 2)`
```

```
[1] 4.643856
```



# Learn How to Use a Function

```
1 help(log)  
2 ?log  
3 # cursor + F1
```

## Description

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.



# Learn How to Use a Function

```
1 help(log)
2 ?log
3 # cursor + F1
```

## Usage

```
1 log(x, base = exp(1))
2 logb(x, base = exp(1))
3 log10(x)
4 log2(x)
5
6 log1p(x)
7
8 exp(x)
9 expm1(x)
```



# Learn How to Use a Function

```
1 help(log)
2 ?log
3 # cursor + F1
```

## Arguments

- x** a numeric or complex vector.
- base** the base with respect to which logarithms are computed.



# Function Arguments: Implicit Matching

```
1 log(x = 25, base = 5)
```

```
[1] 2
```

```
1 log(25, 5) ## the 1st argument is `x`, the 2nd `base`
```

```
[1] 2
```

```
1 log(5, 25) ## the 1st argument is `x`, the 2nd `base`
```

```
[1] 0.5
```

```
1 log(25, base = 5) ## the 1st argument is `x`
```

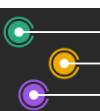
```
[1] 2
```

```
1 log(base = 5, 25) ## it works but don't do this
```

```
[1] 2
```

```
1 log(base = 5, x = 25) ## be explicit if you want to switch argument positions
```

```
[1] 2
```



# Implicit Matching: Be Careful!

Explicit argument names:

```
1 ## example uses vectors (sequences of numbers wrapped into `c()`) which we will discuss later
2 quantile(x = c(5, 1, 3), probs = c(0.25, 0.5, 0.75))
```

```
25% 50% 75%
2     3     4
```

Implicit matching by position:

```
1 quantile(c(5, 1, 3), c(0.25, 0.5, 0.75))
```

```
25% 50% 75%
2     3     4
```

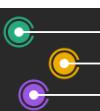
Erroneously switching positions can lead to all kinds of trouble:

```
1 quantile(c(0.25, 0.5, 0.75), c(5, 1, 3))
```

```
Error in quantile.default(c(0.25, 0.5, 0.75), c(5, 1, 3)): 'probs' outside [0,1]
```

```
1 quantile(c(0.25, 0.5, 0.75), c(0.95, 0.345, 1))
```

```
95% 34.5% 100%
0.7250 0.4225 0.7500
```



# Your Turn: Function Arguments

Which of these will work?

`log(x = 1)`

`log(x = "1")`

`log(x)`

`log(value = 1)`

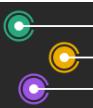
`log(`1`)`

`log(1)`



# *Introduction to R*

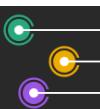
## *— Assignments & Objects —*



# Your Turn: Assignments & Objects

- Run `x <- 1` and afterwards `x`.
- Next, run `x + 2`.
- Now run `x <- 5` and again `x + 2`.

- Why is there no output when you run the first command?
- What is the arrow symbol `<-` doing?
- What is the value of `x` in the end?
- What happens if you type `y <- x` and what if `y <- x <- 2`?
- And what if you type `y <- 2 <- x`?
- Can you assign a value and show the object output at the same time?



# Assignments & Objects

```
name <- "Cedric"
```

object name      assignment arrow      object value

The diagram illustrates the components of an R assignment statement. It features the text 'name <- "Cedric"' in a large, bold font. Below this, three labels are positioned: 'object name' under 'name', 'assignment arrow' under the '<-' symbol, and 'object value' under the string '"Cedric"'. Each label is connected to its corresponding part in the text above by a curved arrow of a matching color: a purple arrow for 'object name', a green arrow for 'assignment arrow', and an orange arrow for 'object value'.



# Assignments & Objects

```
1 x <- 1  
2 x
```

```
[1] 1
```

```
1 x + 2
```

```
[1] 3
```

```
1 x <- 5  
2 x + 2
```

```
[1] 7
```

```
1 x
```

```
[1] 5
```



# Assignments & Objects

```
1 y <- x  
2 y
```

```
[1] 5
```

```
1 y <- x  
2 y
```

```
[1] 5
```

```
1 y <- x <- 2  
2 x
```

```
[1] 2
```

```
1 y
```

```
[1] 2
```

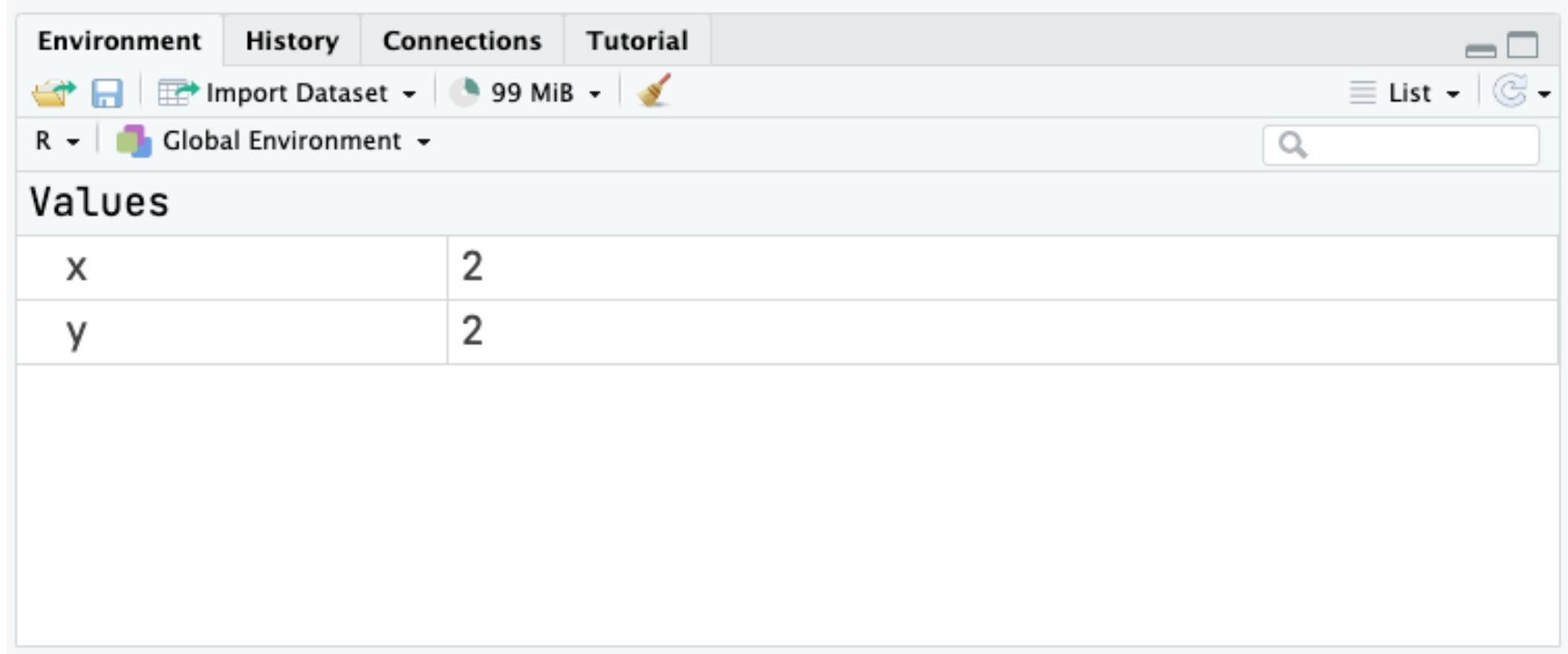
```
1 y <- 2 <- x
```

Error in 2 <- x : invalid (do\_set) left-hand side to assignment



# The Environment

By default, assigned objects exist in the **global environment** called `R_GlobalEnv`. You can find an overview of everything you have defined in the **environment pane**.



The screenshot shows the RStudio interface with the 'Environment' tab selected in the top navigation bar. Below the tabs, there are several icons: a folder for 'Import Dataset', a file for 'File', a disk for 'Memory', and a paintbrush for 'Plots'. The 'Memory' icon shows '99 MiB'. To the right of these are buttons for 'List' (selected) and 'Grid', and a search bar. Underneath the tabs, there are dropdown menus for 'R' (set to 'R') and 'Global Environment' (also set to 'Global Environment'). The main area is titled 'Values' and contains a table with two rows:

x	2
y	2



# Assign & Print Objects

```
1 print(a <- 123)
```

```
[1] 123
```

```
1 show(a <- 123)
```

```
[1] 123
```

```
1 (a <- 123)
```

```
[1] 123
```



# Syntactic Object Names

- **should be descriptive**
  - trade-off between effort to type and being precise
- **must begin with a letter**
  - `val1` works – `1val` and `.val1` will not work
- **can contain other symbols** inside the name
  - such as numbers, periods, and/or underscores, e.g. `val_1`
  - hyphens do **not** work (do not use “kebab-style”, see next slide)
- **are case-sensitive**
  - `myvalues`, `myValues`, and `MyValues` are not the same objects!
- **ideally follow a coherent convention**
  - do not mix different styles, e.g. avoid `val1`, `val_2`, and `val.3`
- **cannot use names that are reserved** and have a certain function in R
  - e.g. `if`, `in`, `function`, `TRUE`, `FALSE`, `NA`, `Nan`



# in that case...

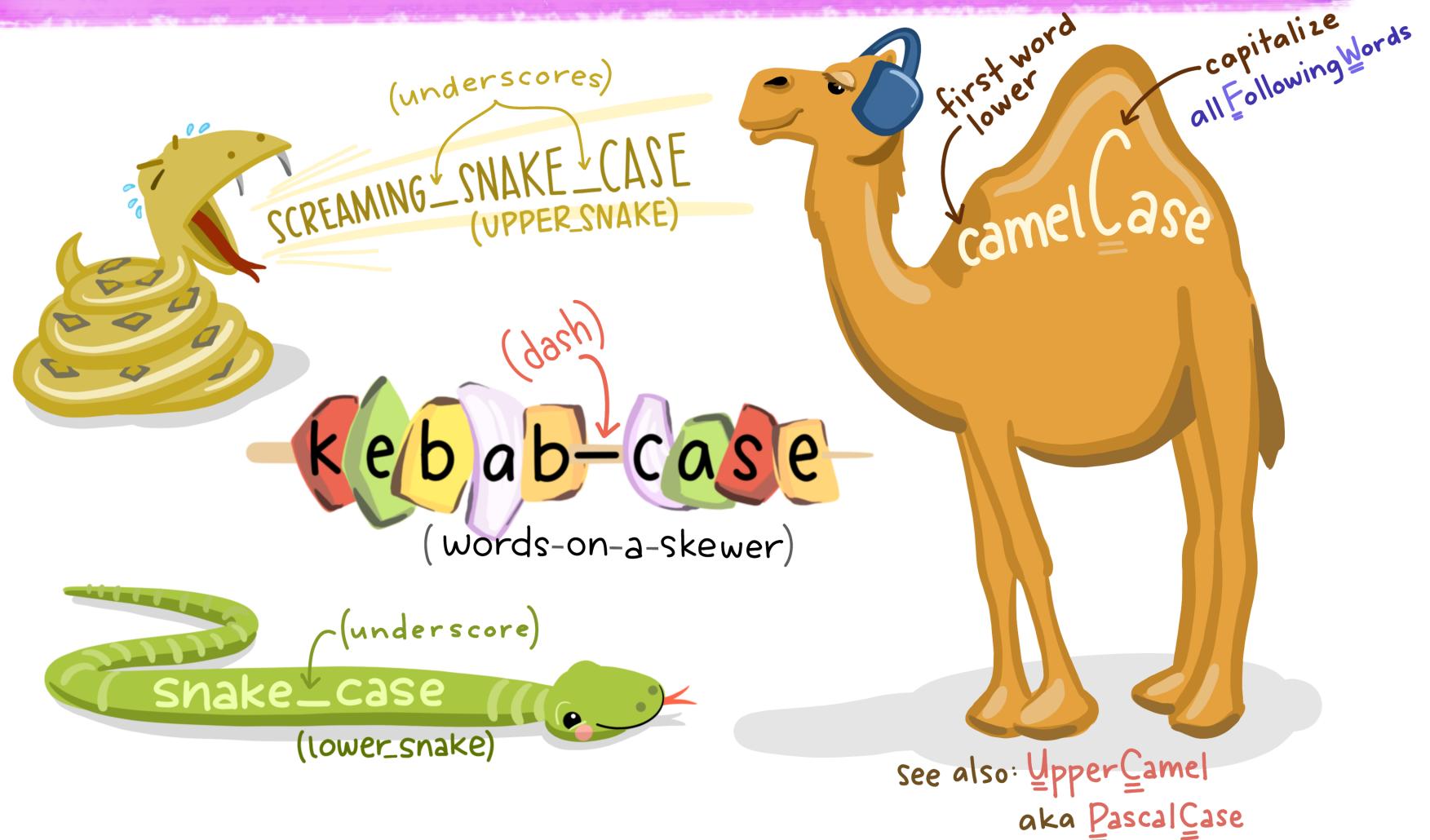
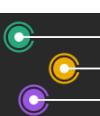


Illustration by Allison Horst

Cédric Scherer // R Course TU Dresden // Introduction to R



# Syntactic Object Names

It's all about readability—decide yourself:

```
1 i_prefer_snake_case
```

```
1 otherPeopleLikeToUseCamelCase
```

```
1 some.people.use.periods
```

```
1 And_aFew.PeopleRENUENCEconvention
```

```
1 df_population_us
```

```
1 dfPopulationUs
```

```
1 df.population.us
```

```
1 df_Population.US
```



# Syntactic Object Names

Potential object names:

- `myvalue1`
- `myValue1`
- `MyValue1`
- `my_value_1`
- `my.value.1`
- ... and all combinations of cases and symbols from above

**Most importantly:**

**Use your preferred style but be consistent!**



# Assignment Syntax

```
1 name <- "Cedric"
```

```
2 name
```

```
[1] "Cedric"
```

```
1 "Dresden" -> city ## right-hand assignment
```

```
2 city
```

```
[1] "Dresden"
```

```
1 country = "Germany" ## a single '=' can also be used for assignment
```

```
2 country
```

```
[1] "Germany"
```

Using `=` is controversially discussed as it can cause confusion but is much more common in other languages.

```
1 country == "Germany" ## a double '=' compares two values!
```

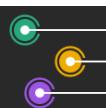
```
[1] TRUE
```

```
1 language == "English" ## a double '=' compares two values!
```

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

```
1 log(25, base = 5) ## inside functions a single '=' defines argument inputs!
```

```
[1] 2
```



# Using Objects in Functions

```
one <- 1  
log(one)
```



# Using Objects in Functions

```
1 one <- 1
```

```
2 one
```

```
[1] 1
```

```
1 log(one)
```

```
[1] 0
```

```
1 one <- "1"
```

```
2 one
```

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

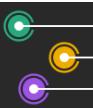
```
1 log(one)
```

```
Error in log(one): non-numeric argument to mathematical function
```



# *Introduction to R*

## *— Data Types —*



# Data Types

There are several basic data types in R including character, double, integer, complex, and logical.

```
1 typeof("Hello World!")
```

```
[1] "character"
```

```
1 typeof(1)
```

```
[1] "double"
```

```
1 typeof(1L)
```

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

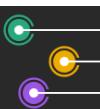
```
1 typeof(1i)
```

```
[1] "complex"
```

```
1 typeof(TRUE)
```

```
[1] "logical"
```

Each of these data types has its own characteristics and use cases.



# Data Types & Atomic Classes

```
1 typeof(1)
```

```
[1] "double"
```

```
1 class(1)
```

```
[1] "numeric"
```

```
1 typeof(1L)
```

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

```
1 class(1L)
```

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

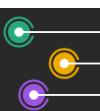
```
1 typeof(1i)
```

```
[1] "complex"
```

```
1 class(1i)
```

```
[1] "complex"
```

However, even though “integer” and “complex” are specific classes in R, they are still considered subtypes of the more general “numeric” data type.



# Coercion into Other Data Types

The `as.*()` functions allow you to change the data type of a value:

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

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

```
1 as.integer(1)
```

```
[1] 1
```

```
1 as.complex(1)
```

```
[1] 1+0i
```

```
1 as.logical(1)
```

```
[1] TRUE
```

```
1 as.logical(0)
```

```
[1] FALSE
```



# Dates & Timestamps

```
1 typeof("2023-02-27 09:00:00")
```

```
[1] "character"
```

## Date Objects

```
1 as.Date("2023-02-27 09:00:00")
```

```
[1] "2023-02-27"
```

```
1 typeof(as.Date("2023-02-27 09:00:00"))
```

```
[1] "double"
```

## Datetime Objects

```
1 as.POSIXct("2023-02-27 09:00:00")
```

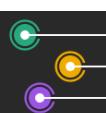
```
[1] "2023-02-27 09:00:00 CET"
```

```
1 typeof(as.POSIXct("2023-02-27 09:00:00"))
```

```
[1] "double"
```

```
1 as.POSIXlt("2023-02-27 09:00:00")
```

```
[1] "2023-02-27 09:00:00 CET"
```



# Dates & Timestamps

```
1 date <- "27.02.2023 09:00"
```

```
1 as.POSIXct(date)
```

```
Error in as.POSIXlt.character(x, tz, ...): character string is not in a standard unambiguous format
```

```
1 as.POSIXct(date, format = "%d.%m.%Y %H:%M")
```

```
[1] "2023-02-27 09:00:00 CET"
```

```
1 as.POSIXct(date, format = "%d.%m.%Y %H:%M", tz = "EST")
```

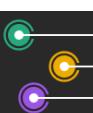
```
[1] "2023-02-27 09:00:00 EST"
```

```
1 as.POSIXct(date, format = "%d.%m.%Y %H:%M", tz = "America/New_York")
```

```
[1] "2023-02-27 09:00:00 EST"
```

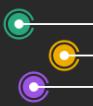
```
1 as.POSIXct(date, format = "%d.%m.%Y %H:%M", tz = "US/Eastern")
```

```
[1] "2023-02-27 09:00:00 EST"
```



# *Introduction to R*

## *— Unknown Values —*



# “Not Available”

`NA` is a special value that is used to represent missing or undefined data.

```
1 typeof(NA)
```

```
[1] "logical"
```

```
1 typeof(NA_character_)
```

```
[1] "character"
```

```
1 typeof(NA_integer_)
```

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

```
1 typeof(NA_real_)
```

```
[1] "double"
```

```
1 typeof(NA_complex_)
```

```
[1] "complex"
```



# “Not a Number”

The `NaN` value is another special value that is used to represent undefined values in mathematical operations:

```
1  NaN
```

```
[1] NaN
```

... such as the result of dividing zero by zero:

```
1  0 / 0
```

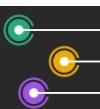
```
[1] NaN
```

```
1  is.nan(0 / 0)
```

```
[1] TRUE
```

```
1  is.na(0 / 0)
```

```
[1] TRUE
```



# Unknown Values

As they are unknown, the special values `NA` and `NaN` are not equal to any other value, including themselves:

```
1 NA == "Cedric"
```

```
[1] NA
```

```
1 NaN == 1
```

```
[1] NA
```

```
1 NA == NA
```

```
[1] NA
```

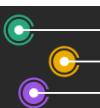
```
1 NaN == (0 / 0)
```

```
[1] NA
```

Also, you can't assign any value to the `NA` and `NaN` special values:

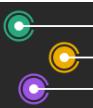
```
1 NA <- 1
```

```
Error in NA <- 1 : invalid (do_set) left-hand side to assignment
```



# *Introduction to R*

## — Vectors —

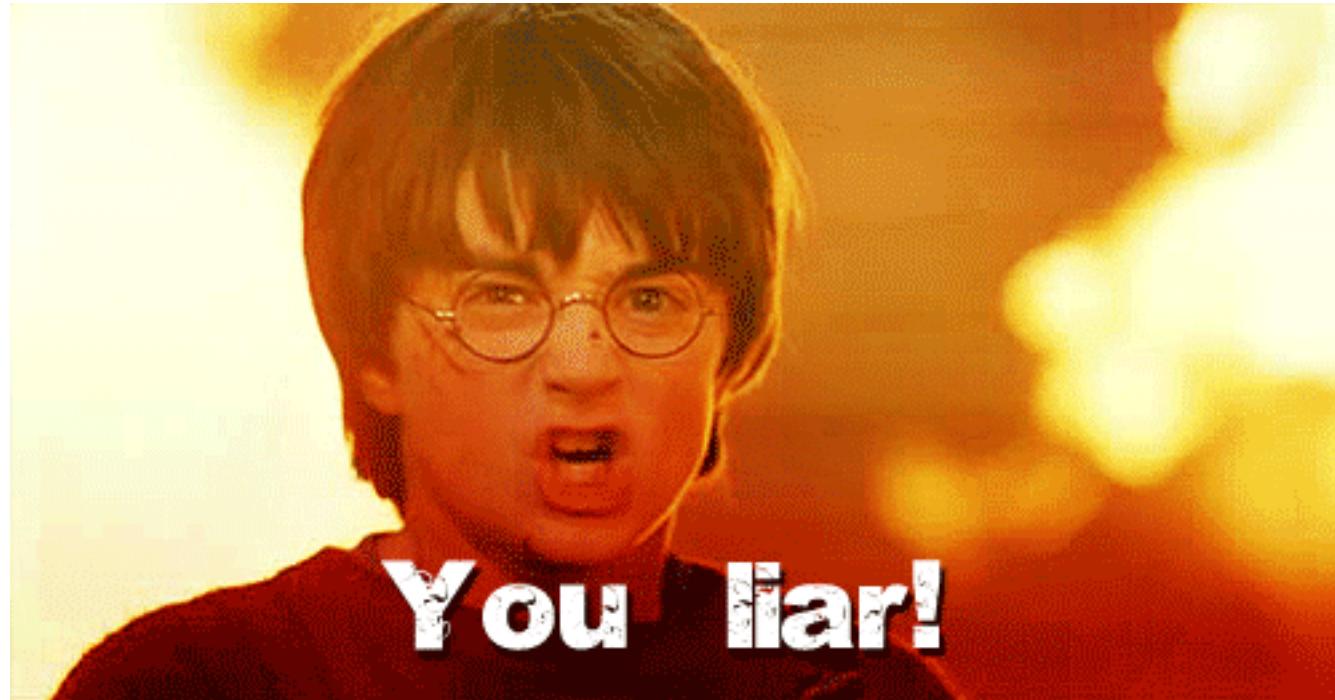


# Your Turn: Calculate Averages

- Calculate the average of these numeric values: 140, 97, and 222

```
1 mean(140, 97, 222)
```

```
[1] 140
```



# Vectors

c(135, 97, 222)

vector  
function

vector  
elements

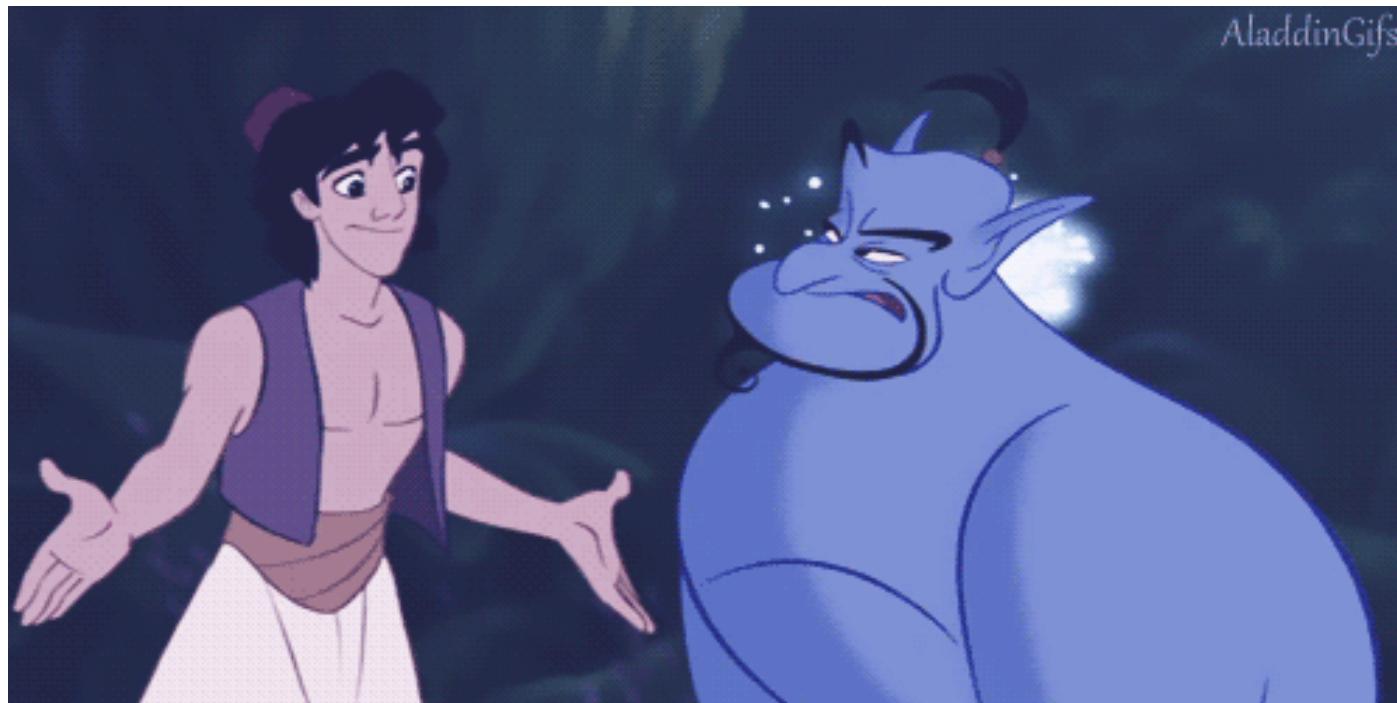


# Your Turn: Calculate Averages

- Calculate the average of these numeric values: 140, 97, and 222

```
1 mean(c(140, 97, 222))
```

```
[1] 153
```



# Your Turn: Calculate Averages

- Calculate the average of these numeric values: 140, 97, and 222

```
1 values <- c(140, 97, 222)
2 mean(values)
```

```
[1] 153
```

```
1 values <- c(140L, 97L, 222L)
2 mean(values)
```

```
[1] 153
```



# But... How Am I Supposed to Know?

```
1 help(mean)  
2 ?mean  
3 # cursor + F1
```

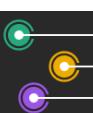
## Usage

```
1 mean(x, ...)  
2  
3 ## Default S3 method:  
4 mean(x, trim = 0, na.rm = FALSE, ...)
```

## Examples

```
1 x <- c(0:10, 50)  
2 x  
[1] 0 1 2 3 4 5 6 7 8 9 10 50
```

```
1 xm <- mean(x)  
2 xm  
[1] 8.75
```



# Vectors

Just like values, vectors can be of different types:

```
1 ids <- c(1:5)
```

```
2 ids
```

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

```
1 typeof(ids)
```

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

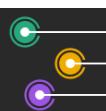
```
1 names <- c("Manisha", "Julien", "Cornelia", "Tatyana", "Eman")
```

```
2 names
```

```
[1] "Manisha"   "Julien"     "Cornelia"   "Tatyana"    "Eman"
```

```
1 typeof(names)
```

```
[1] "character"
```



# Named Vectors

```
1 colors <- c(green = "#28A87D", purple = "#9C55E3", yellow = "#EFAC00"))
```

```
green    purple    yellow  
"#28A87D" "#9C55E3" "#EFAC00"
```

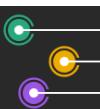
Alternatively, you can create an unnamed vector and add names via `names(x)`:

```
1 colors <- c("#28A87D", "#9C55E3", "#EFAC00")  
2 names(colors) <- c("green", "purple", "yellow")  
3 colors
```

```
green    purple    yellow  
"#28A87D" "#9C55E3" "#EFAC00"
```

If you want to get rid of the names, use `unname()`

```
1 unname(colors)  
[1] "#28A87D" "#9C55E3" "#EFAC00"
```



# Accessing Elements of Vectors

```
1 names
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```

```
1 names[1]
```

```
[1] "Manisha"
```

```
1 names[3:5]
```

```
[1] "Cornelia" "Tatyana"  "Eman"
```

```
1 names[c(1:3, 5)]
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Eman"
```

```
1 names[-2]
```

```
[1] "Manisha"  "Cornelia" "Tatyana"  "Eman"
```



# Accessing Elements of Named Vectors

```
1 colors
```

```
green     purple    yellow  
"#28A87D" "#9C55E3" "#EFAC00"
```

```
1 colors[c(1, 3)]
```

```
green     yellow  
"#28A87D" "#EFAC00"
```

Named vectors can also be assessed by element names:

```
1 colors["purple"]
```

```
purple  
"#9C55E3"
```

```
1 colors[c("green", "yellow")]
```

```
green     yellow  
"#28A87D" "#EFAC00"
```



# NA and NaN in Vectors

When used inside a vector, the type of `NA` is determined by the data type of the other values–no need to specify e.g. `NA_character_`:

```
1 x <- c(1, 2, 3, NA, 5)
2 y <- c("A", "B", NA, "D", "E")
```

```
1 typeof(x)
```

```
[1] "double"
```

```
1 typeof(y)
```

```
[1] "character"
```



# NA and NaN in Vectors

As they are unknown, any mathematical operation will return **NA** or **NaN**:

```
1 x <- c(5, 12, NA, 2, 18)
2 mean(x)
```

```
[1] NA
```

```
1 y <- c(5, 12, NaN, 2, 18)
2 mean(y)
```

```
[1] NaN
```

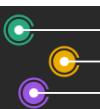
To ignore unknown values, you can set the argument **na.rm** to **TRUE**:

```
1 mean(x, na.rm = TRUE) ## `na.rm = FALSE` by default
```

```
[1] 9.25
```

```
1 mean(y, na.rm = TRUE)
```

```
[1] 9.25
```



# Vector Extension & Combination

```
1 colors_ext <- c(colors, c("#000000", "#ffffff"))
```

```
1 colors_ext
```

```
green    purple    yellow  
"#28A87D" "#9C55E3" "#EFAC00" "#000000" "#ffffff"
```

```
1 colors_ext <- c(colors, c(black = "#000000", white = "#ffffff"))
```

```
1 colors_ext
```

```
green    purple    yellow    black    white  
"#28A87D" "#9C55E3" "#EFAC00" "#000000" "#ffffff"
```

```
1 comb_vec <- c(ids, names)
```

```
1 comb_vec
```

```
[1] "1"        "2"        "3"        "4"        "5"        "Manisha"  "Julien"   "Cornelia" "Tatyana"  
[10] "Eman"
```

Note that in the last example **all values are coerced to the same data type** as atomic vectors can only contain a single type.



# Coercion

Atomic vectors only contain one data type and  
**all values will be coerced implicitly to the same data type:**

```
1 v1 <- c("ggplot2", 2023, TRUE)  
2 v1
```

```
[1] "ggplot2" "2023"    "TRUE"
```

```
1 typeof(v1)
```

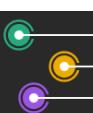
```
[1] "character"
```

```
1 v2 <- c(0.2, 1L, 3 + 0i)  
2 v2
```

```
[1] 0.2+0i 1.0+0i 3.0+0i
```

```
1 typeof(v2)
```

```
[1] "complex"
```



# Coercion

The coercion rule goes:

logical → integer → double → complex → character

```
1 typeof(c(TRUE))
```

```
[1] "logical"
```

```
1 typeof(c(TRUE, 1L))
```

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

```
1 typeof(c(TRUE, 1L, 0.2))
```

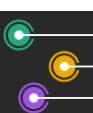
```
[1] "double"
```

```
1 typeof(c(TRUE, 1L, 0.2, 1 + 0i))
```

```
[1] "complex"
```

```
1 typeof(c(TRUE, 1L, 0.2, 1 + 0i, "a"))
```

```
[1] "character"
```



# Explicit Coercion

One can **explicitly coerce** values of an atomic vectors to a particular data type:

```
1 v1
```

```
[1] "ggplot2" "2023"    "TRUE"
```

```
1 as.numeric(v1)
```

```
[1] NA 2023 NA
```

```
1 as.logical(v1)
```

```
[1] NA NA TRUE
```



# Explicit Coercion

One can **explicitly coerce** values of an atomic vectors to a particular data type:

```
1 v1
```

```
[1] "ggplot2" "2023"    "TRUE"
```

```
1 as.numeric(v1)
```

```
[1] NA 2023 NA
```

```
1 as.logical(v1)
```

```
[1] NA NA TRUE
```

Note that all numbers except **0** are coerced to **TRUE**:

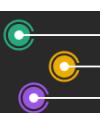
```
1 as.logical(2023)
```

```
[1] TRUE
```

... and all strings except "**TRUE**", "**FALSE**", "**T**", and "**F**" get coerced to **NA**:

```
1 as.logical("2023")
```

```
[1] NA
```



# Explicit Coercion

One can **explicitly coerce** values of an atomic vectors to a particular data type:

```
1 v2
```

```
[1] 0.2+0i 1.0+0i 3.0+0i
```

```
1 as.integer(v2)
```

```
[1] 0 1 3
```

```
1 as.character(v2)
```

```
[1] "0.2+0i" "1+0i"    "3+0i"
```

```
1 as.logical(v2)
```

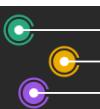
```
[1] TRUE TRUE TRUE
```

```
1 as.logical(as.character(v2))
```

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

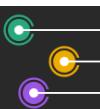
```
1 as.logical(as.integer(v2))
```

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



# Your Turn: Vector Coercion

- Discuss what the following vectors look like without running them first:
  - `a <- c(1.7, "1.7")` and `as.numeric(a)`
  - `b <- c(TRUE, 2022, 1.375)` and `as.logical(as.complex(b))`
  - `c <- c(TRUE, "TRUE", "T", FALSE, "False", "f")` and `as.logical(c)`
  - `as.integer(c(45L, 0.237, 4.9))`
- What does `as.factor(c("red", "blue"))` do?



# Your Turn: Vector Coercion

```
1 (a <- c(1.7, "1.7"))
```

```
[1] "1.7" "1.7"
```

```
1 as.numeric(a)
```

```
[1] 1.7 1.7
```

```
1 (b <- c(FALSE, 2022, 1.375))
```

```
[1] 0.000 2022.000 1.375
```

```
1 as.logical(as.complex(b))
```

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

```
1 (c <- c(TRUE, "t", "TRUE", "T"))
```

```
[1] "TRUE" "t"    "TRUE" "T"
```

```
1 as.logical(c)
```

```
[1] TRUE NA TRUE TRUE
```

```
1 as.integer(c(45L, 0.237, 4.9))
```

```
[1] 45 0 4
```



# Your Turn: Vector Coercion

```
1 c("red", "blue")
```

```
[1] "red"  "blue"
```

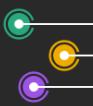
```
1 as.factor(c("red", "blue"))
```

```
[1] red  blue  
Levels: blue red
```



# *Introduction to R*

## — Factors —



# Factors

Factors represent categorical data in a **specific order**:

```
1 v <- c("good", "neutral", "bad")
```

```
[1] "good"    "neutral" "bad"
```

```
1 as.factor(v)
```

```
[1] good    neutral bad  
Levels: bad good neutral
```

```
1 factor(v)
```

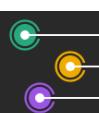
```
[1] good    neutral bad  
Levels: bad good neutral
```

```
1 factor(v, levels = c("good", "neutral", "bad"))
```

```
[1] good    neutral bad  
Levels: good neutral bad
```

```
1 factor(v, levels = rev(v))
```

```
[1] good    neutral bad  
Levels: bad neutral good
```



# Factors

... even if the variable is encoded by numbers:

```
1 (n <- 0:3)
```

```
[1] 0 1 2 3
```

```
1 (f <- factor(n))
```

```
[1] 0 1 2 3
```

```
Levels: 0 1 2 3
```

```
1 (f <- factor(n, levels = c(1:3, 0)))
```

```
[1] 0 1 2 3
```

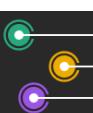
```
Levels: 1 2 3 0
```

```
1 mean(n)
```

```
[1] 1.5
```

```
1 mean(f)
```

```
[1] NA
```



# Factor Levels & Labels

`levels()` just returns unique levels in the specified order:

```
1 levels(f)  
[1] "1" "2" "3" "0"
```

You can also set labels which basically overwrite the original encodings:

```
1 (ff <- factor(n, levels = c(1:3, 0), labels = c("bad", "neutral", "good", "n/a")))  
[1] n/a      bad      neutral good  
Levels: bad neutral good n/a  
  
1 levels(ff)  
[1] "bad"     "neutral"  "good"    "n/a"
```



# Coercion of Factors

```
1 ff
```

```
[1] n/a     bad     neutral good  
Levels: bad neutral good n/a
```

If you want to turn a factor into numbers, use `as.numeric()`:

```
1 as.numeric(ff)
```

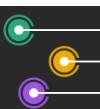
```
[1] 4 1 2 3
```

Note that this returns the factor elements as numbers based on the level position!

To turn it back into strings, use `as.character()`:

```
1 as.character(ff)
```

```
[1] "n/a"     "bad"      "neutral"   "good"
```



# Coercion of Factors: Be Careful!

```
1 f
```

```
[1] 0 1 2 3  
Levels: 1 2 3 0
```

Returns numbers based on levels:

```
1 as.numeric(f)
```

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

Returns numbers based on elements:

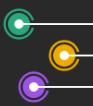
```
1 as.numeric(as.character(f))
```

```
[1] 0 1 2 3
```



# *Introduction to R*

## — Packages —



# What are Packages?

A package is a **collection of functions, data, and other objects** that are designed to **perform specific tasks or solve certain problems**.

These packages are created and maintained by the R community, and are often shared on the [Comprehensive R Archive Network \(CRAN\)](#) or other repositories such as [BioConductor](#).



# Installing Additional Packages

To use the functionality from a dedicated package, you need to install the package on your machine:

```
1 install.packages("forcats")
2 ## provides functions to work with factors
```

```
1 install.packages("palmerpenguins")
2 ## contains data sets for educational use
```



# Using Packages: Load Libraries

To use the functionality from installed packages, you usually load the package:

```
1 library(forcats)
```

```
1 f <- factor(0:10)
2 levels(f)
```

```
[1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```
1 fct_rev(f)
```

```
[1] 0  1  2  3  4  5  6  7  8  9  10
Levels: 10 9 8 7 6 5 4 3 2 1 0
```

```
1 fct_relevel(f, "0", after = Inf)
```

```
[1] 0  1  2  3  4  5  6  7  8  9  10
Levels: 1 2 3 4 5 6 7 8 9 10 0
```



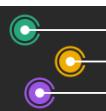
# Using Packages: Load Libraries

To use the functionality from installed packages, you usually load the package:

```
1 library(palmerpenguins)
```

```
1 penguins
```

```
# A tibble: 344 × 8
  species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g sex   year
  <fct>     <fct>        <dbl>          <dbl>            <int>       <int> <fct> <int>
1 Adelie    Torgersen      39.1           18.7             181        3750 male   2007
2 Adelie    Torgersen      39.5           17.4             186        3800 female 2007
3 Adelie    Torgersen      40.3           18               195        3250 female 2007
4 Adelie    Torgersen       NA             NA                NA         NA <NA>  2007
5 Adelie    Torgersen      36.7           19.3             193        3450 female 2007
6 Adelie    Torgersen      39.3           20.6             190        3650 male   2007
7 Adelie    Torgersen      38.9           17.8             181        3625 female 2007
8 Adelie    Torgersen      39.2           19.6             195        4675 male   2007
9 Adelie    Torgersen      34.1           18.1             193        3475 <NA>  2007
10 Adelie   Torgersen       42              20.2            190        4250 <NA>  2007
# ... with 334 more rows
```



# Using Packages: Name Conflicts

Note that in some cases, multiple packages may contain functions (or data sets) with the same name.

```
1 install.packages("dplyr")
2 ## provides functions to wrangle tabular data
```

```
1 library(dplyr)
```

Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

filter, lag

The following objects are masked from 'package:base':

intersect, setdiff, setequal, union

**In case you load two functions with the same name,  
the function of the package that is loaded most recently is used!**



# Using Packages: Namespace

`tibble::tibble()`

The diagram illustrates the structure of the function name `tibble::tibble()`. It is composed of three main parts: **package name** (purple), **namespace** (green), and **function name** (yellow). Three arrows point upwards from the corresponding labels to the respective parts of the function name: a purple arrow points to the first `tibble`, a green arrow points to the `::` separator, and a yellow arrow points to the second `tibble`.

package name    namespace    function name



# Using Packages: Namespace

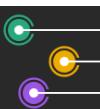
To avoid conflicts, one can explicitly refer to specific functions with namespaces:

```
1 install.packages("stringr")
2 ## provides functions to work with character strings
```

```
1 stringr::str_to_sentence(c("coffee", "latte machiatto", "espresso"))
[1] "Coffee"           "Latte machiatto" "Espresso"
```

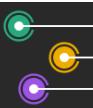
Namespaces are helpful in several situations:

- solving name conflicts
- using functions without loading the full package functionality
- being explicit about non-base functions



# *Introduction to R*

## *— Tabular Data —*



# Data Frames

You can store several vectors, also of different data types, as tabular data in so-called **data frames**:

```
1 (df_participants <- data.frame(ids, names))
```

```
ids      names
1   1    Manisha
2   2     Julien
3   3 Cornelia
4   4   Tatyana
5   5      Eman
```

Explicitly declaring column names:

```
1 (df_participants <- data.frame(id = ids, participant_name = names))
```

```
id participant_name
1 1           Manisha
2 2           Julien
3 3        Cornelia
4 4       Tatyana
5 5          Eman
```



# Accessing Data Frame Elements

## Rows:

```
1 df_participants[2,]
```

```
id participant_name
2 2 Julien
```

## Columns:

```
1 df_participants[,2]
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```

```
1 df_participants$participant_name
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```

```
1 df_participants[[2]]
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```



# Accessing Data Frame Elements

## Cells:

```
1 df_participants$participant_name[2]
```

```
[1] "Julien"
```

```
1 df_participants[[2]][2]
```

```
[1] "Julien"
```

```
1 df_participants[,2][2]
```

```
[1] "Julien"
```



# Accessing Data Frame Elements

Data frames support partial matching...

```
1 df_participants
```

```
id participant_name
1 1 Manisha
2 2 Julien
3 3 Cornelia
4 4 Tatyana
5 5 Eman
```

`participant` is **not** a column of `df_participants` but matches by name with `participant_name`:

```
1 df_participants$participant
```

```
[1] "Manisha"  "Julien"    "Cornelia"  "Tatyana"   "Eman"
```



# The Modern Data Frame: Meet the Tibble

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

```
1 install.packages("tibble")
```

The name comes from the way you originally created these objects: `tbl_df()`, which was most easily pronounced as “*tibble diff*” or “*tibble d. f.*”.

```
1 (tbl_participants <- tibble::tibble(id = ids, participant_name = names))
```

```
# A tibble: 5 × 2
  id participant_name
  <int> <chr>
1     1 Manisha
2     2 Julien
3     3 Cornelia
4     4 Tatyana
5     5 Eman
```

Similarly as the functions we have used before, `tibble()` is a function. While `mean()` and `+` are **base functions**, `tibble()` belongs to an **add-on package** which is named `{tibble}` as well.



# Data Frame vs Tibble

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

Main differences in the usage of a data frame vs a tibble:

- **tibbles have a refined print method**
  - show only the first 10 rows, and all the columns that fit on screen
  - in addition to its name, each column reports its type
  - also, the type of the data and its dimensions are shown



# Data Frame vs Tibble

Tibbles have a refined print method.

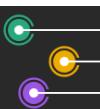
```
1 df_long <- data.frame(id = 1:1000 %% 10, value = runif(n = 1000, min = 0, max = 1000))
2 tbl_long <- tibble::tibble(id = 1:1000 %% 10, value = runif(n = 1000, min = 0, max = 1000))
```

```
1 df_long
```

```
  id      value
1 1 466.6139435
2 2 335.1909537
3 3 162.8175615
4 4 396.1200174
5 5 30.3917308
6 6 120.8848702
7 7 426.1656571
8 8 617.8578807
9 9 263.2082594
10 0 476.3238712
11 1 862.3189987
12 2 148.8786321
13 3 180.4300796
14 4 999.2733120
15 5 841.7411824
16 6 142.6490599
17 7 344.8483865
18 8 889.6254539
```

```
1 tbl_long
```

```
# A tibble: 1,000 × 2
  id      value
  <dbl> <dbl>
1 1     881.
2 2     691.
3 3     787.
4 4     79.0 
5 5     206.
6 6     923.
7 7     987.
8 8     134.
9 9     940.
10 0    165.
# ... with 990 more rows
```



# Data Frame vs Tibble

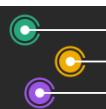
Tibbles have a refined print method.

```
1 as.data.frame(penguins)
```

	species	island	bill_length_mm	bill_depth_mm
1	Adelie	Torgersen	39.1	18.7
2	Adelie	Torgersen	39.5	17.4
3	Adelie	Torgersen	40.3	18.0
4	Adelie	Torgersen	NA	NA
5	Adelie	Torgersen	36.7	19.3
6	Adelie	Torgersen	39.3	20.6
7	Adelie	Torgersen	38.9	17.8
8	Adelie	Torgersen	39.2	19.6
9	Adelie	Torgersen	34.1	18.1
10	Adelie	Torgersen	42.0	20.2
11	Adelie	Torgersen	37.8	17.1
12	Adelie	Torgersen	37.8	17.3
13	Adelie	Torgersen	41.1	17.6
14	Adelie	Torgersen	38.6	21.2
15	Adelie	Torgersen	34.6	21.1
16	Adelie	Torgersen	36.6	17.8
17	Adelie	Torgersen	38.7	19.0
18	Adelie	Torgersen	42.5	20.7

```
1 tibble::as_tibble(penguins)
```

	species	island	bill_... <sup>1</sup>	bill_... <sup>2</sup>	flipp... <sup>3</sup>	body_... <sup>4</sup>	sex
	<fct>	<fct>	<dbl>	<dbl>	<int>	<int>	<fct>
1	Adelie	Torge...	39.1	18.7	181	3750	male
2	Adelie	Torge...	39.5	17.4	186	3800	fema...
3	Adelie	Torge...	40.3	18	195	3250	fema...
4	Adelie	Torge...	NA	NA	NA	NA	<NA>
5	Adelie	Torge...	36.7	19.3	193	3450	fema...
6	Adelie	Torge...	39.3	20.6	190	3650	male
7	Adelie	Torge...	38.9	17.8	181	3625	fema...
8	Adelie	Torge...	39.2	19.6	195	4675	male
9	Adelie	Torge...	34.1	18.1	193	3475	<NA>
10	Adelie	Torge...	42	20.2	190	4250	<NA>
	# ... with 334 more rows, 1 more variable: year <int>,						
	# and abbreviated variable names <sup>1</sup> bill_length_mm,						
	# <sup>2</sup> bill_depth_mm, <sup>3</sup> flipper_length_mm, <sup>4</sup> body_mass_g						



# Data Frame vs Tibble

Tibbles are a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

Main differences in the usage of a data frame vs a tibble:

- **tibbles have a refined print method**
  - show only the first 10 rows, and all the columns that fit on screen
  - in addition to its name, each column reports its type
  - also, the type of the data and its dimensions are shown
- **tibbles are strict about subsetting**



# Data Frame vs Tibble

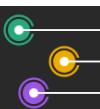
Tibbles are strict about subsetting.

```
1 df_long$val
```

```
[1] 466.6139435 335.1909537 162.8175615 396.1200174  
[5] 30.3917308 120.8848702 426.1656571 617.8578807  
[9] 263.2082594 476.3238712 862.3189987 148.8786321  
[13] 180.4300796 999.2733120 841.7411824 142.6490599  
[17] 344.8483865 889.6254539 319.8482955 709.0158337  
[21] 628.1517190 438.1725660 339.8960282 933.0925872  
[25] 712.9544122 630.8518730 746.4494368 111.4502875  
[29] 273.2885152 98.5180892 757.5294827 663.5921039  
[33] 724.3610939 458.9465314 674.3985806 700.5136148  
[37] 815.0722187 691.1814879 716.4123438 890.5798914  
[41] 340.2968822 668.5628453 384.2539703 235.1133667  
[45] 888.4970285 801.4516588 596.4277368 977.0177414  
[49] 328.1049675 805.1800283 32.3153806 448.2898777  
[53] 264.7317180 409.2675177 194.5968003 543.4875141  
[57] 935.1037010 774.0376103 996.8832501 822.6258415  
[61] 391.7053440 385.2494801 899.1378001 318.8530235  
[65] 208.6884824 564.1771706 482.0844352 65.1146327  
[69] 261.2773522 330.3356417 330.5641250 92.3532958  
[73] 693.9935274 116.4896260 870.5973122 128.6706002
```

```
1 tbl_long$val
```

```
NULL
```



# Inspect a Data Set

```
1 View(tbl_long)
```



# Inspect a Data Set

```
1 dim(tbl_long)
```

```
[1] 1000    2
```

```
1 str(tbl_long)
```

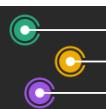
```
tibble [1,000 × 2] (S3: tbl_df/tbl/data.frame)
$ id    : num [1:1000] 1 2 3 4 5 6 7 8 9 0 ...
$ value: num [1:1000] 881 691 787 79 206 ...
```

```
1 tibble::glimpse(tbl_long)
```

```
Rows: 1,000
Columns: 2
$ id      <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2...
$ value   <dbl> 880.849571, 690.568330, 786.657729, 78.981368, 206.086510, 923.498811, 987.393297, 133.562790,...
```

```
1 summary(tbl_long)
```

	id	value
Min.	:0.0	Min. : 1.341
1st Qu.	:2.0	1st Qu.:230.005
Median	:4.5	Median :475.317
Mean	:4.5	Mean   :482.438
3rd Qu.	:7.0	3rd Qu.:739.410
Max.	:9.0	Max.   :999.169



# Inspect a Data Set

```
1 mean(tbl_long$value)
```

```
[1] 482.4375
```

```
1 range(tbl_long$value)
```

```
[1] 1.34083 999.16861
```

```
1 unique(tbl_long$id)
```

```
[1] 1 2 3 4 5 6 7 8 9 0
```

```
1 sort(unique(tbl_long$id))
```

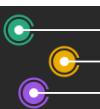
```
[1] 0 1 2 3 4 5 6 7 8 9
```

```
1 length(tbl_long$id)
```

```
[1] 1000
```

```
1 length(unique(tbl_long$id))
```

```
[1] 10
```



# Your Turn: Working with Tabular Data

- If not done yet: install and load the `{palmerpenguins}` package
- Explore the `penguins` data set:
  - Inspect the number of unique observations.
  - Count the number of observations per sex.
  - Check the years the penguins were monitored.
  - Calculate the average bill length.
  - Report the quartiles of measured flipper lengths.
  - Store the body mass column in a vector called `penguin_weights`.



# Your Turn: Working with Tabular Data

```
1 dim(penguins) ## or: `nrow(penguins)`, `length(penguins$species)`, ...
```

```
[1] 344    8
```

```
1 summary(penguins$sex)
```

```
female    male    NA's  
 165      168      11
```

```
1 unique(penguins$year)
```

```
[1] 2007 2008 2009
```

```
1 mean(penguins$bill_length_mm)
```

```
[1] NA
```

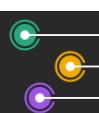
```
1 mean(penguins$bill_length_mm, na.rm = TRUE)
```

```
[1] 43.92193
```

```
1 quantile(penguins$flipper_length_mm, na.rm = TRUE)
```

```
0%  25%  50%  75% 100%  
172  190  197  213  231
```

```
1 penguin_weights <- penguins$body_mass_g
```



# Lists

Lists are by far the most flexible data structure in R. They can be seen as a collection of elements without any restriction on the class, length or structure of each element.

```
1 my_list <- list(ids = 1:5, names = c("Manisha", "Julien", "Cornelia", "Tatyana", "Eman"))
```

```
1 my_list
```

```
$ids  
[1] 1 2 3 4 5  
  
$names  
[1] "Manisha"   "Julien"     "Cornelia"  "Tatyana"   "Eman"
```

Note that data frames can be seen as simple tabular lists as well.



# Accessing List Elements

```
1 my_list$names
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```

```
1 my_list$names[1]
```

```
[1] "Manisha"
```

```
1 my_list[[2]]
```

```
[1] "Manisha"  "Julien"   "Cornelia" "Tatyana"  "Eman"
```

```
1 my_list[[2]][1]
```

```
[1] "Manisha"
```



# Nested Lists

Usually, when we talk about lists we often mean nested data structure:

```
1 my_nested_list <- list(  
2   ids = list(1:5, 1),  
3   names = list(c("Manisha", "Julien", "Cornelia", "Tatyana", "Eman"), "Cedric"),  
4   type = list(rep("participant", 5), "instructor"))  
5 )
```

```
1 my_nested_list
```

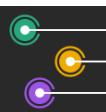
```
$ids  
$ids[[1]]  
[1] 1 2 3 4 5
```

```
$ids[[2]]  
[1] 1
```

```
$names  
$names[[1]]  
[1] "Manisha"    "Julien"      "Cornelia"    "Tatyana"    "Eman"
```

```
$names[[2]]  
[1] "Cedric"
```

```
$type  
$type[[1]]  
[1] "participant" "participant" "participant" "participant" "participant"
```



# Nested Lists

```
1 my_nested_list$names  
[[1]]  
[1] "Manisha"   "Julien"     "Cornelia"  "Tatyana"   "Eman"  
  
[[2]]  
[1] "Cedric"
```

```
1 my_nested_list$names[[1]]  
[1] "Manisha"   "Julien"     "Cornelia"  "Tatyana"   "Eman"
```

```
1 my_nested_list$names[[1]][1]  
[1] "Manisha"
```

```
1 my_nested_list[[2]][[1]][1]  
[1] "Manisha"
```



# Nested Lists as Tibbles

```
1 my_tbl_list <- tibble:::as_tibble(my_nested_list)
```

```
1 my_tbl_list
```

```
# A tibble: 2 × 3
  ids      names     type
  <list>    <list>    <list>
1 <int [5]> <chr [5]> <chr [5]>
2 <dbl [1]> <chr [1]> <chr [1]>
```



# Unnesting Nested Lists

```
1 install.packages("tidyverse")
```

```
1 my_tbl <- tidyverse::unnest(my_tbl_list)
```

Warning: `cols` is now required when using `unnest()`.  
i Please use `cols = c(ids, names, type)`.

```
1 my_tbl <- tidyverse::unnest(my_tbl_list, cols = c(ids, names, type))
```

```
1 my_tbl
```

```
# A tibble: 6 × 3
  ids names    type
  <dbl> <chr>   <chr>
1     1 Manisha participant
2     2 Julien   participant
3     3 Cornelia participant
4     4 Tatyana  participant
5     5 Eman     participant
6     1 Cedric   instructor
```



# Nesting Lists

```
1 my_nested_tbl <- tidyverse::nest(my_tbl, data = !type)
```

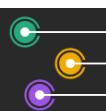
```
1 my_nested_tbl
```

```
# A tibble: 2 × 2
  type      data
  <chr>    <list>
1 participant <tibble [5 × 2]>
2 instructor  <tibble [1 × 2]>
```

```
1 my_nested_tbl$data
```

```
[[1]]
# A tibble: 5 × 2
  ids names
  <dbl> <chr>
1     1 Manisha
2     2 Julien
3     3 Cornelia
4     4 Tatyana
5     5 Eman
```

```
[[2]]
# A tibble: 1 × 2
  ids names
  <dbl> <chr>
1     1 Cedric
```



# *Introduction to R*

## *— Generating Data Sets —*



# Generate Data Sets Vectors in R

You can combine vectors to a data frame or tibble. Helpful functions to create vectors are `:`, `seq()`, and `rep()`:

```
1 1:10
```

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

```
1 seq(from = 0, to = 150, by = 25)
```

```
[1] 0 25 50 75 100 125 150
```

```
1 0:6*25
```

```
[1] 0 25 50 75 100 125 150
```

```
1 seq(from = 0, to = 150, length.out = 5)
```

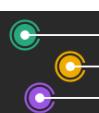
```
[1] 0.0 37.5 75.0 112.5 150.0
```

```
1 rep(1:3, times = 3)
```

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

```
1 rep(1:3, each = 3)
```

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



# Generate Data Sets in R

You can combine vectors to a data frame or tibble. Helpful functions to create vectors are `:`, `seq()`, and `rep()`:

```
1 name <- c("Brian", "Jason", "Tyler", "Sam")
2 age <- seq(23, 29, by = 2) ## or `c(23, 25, 27, 29)`
3 sex <- rep("male", length(name)) ## or `c("male", "male", "male", "male")`
4 id <- 1:length(name) ## or `c(1, 2, 3, 4)` or `1:4`
```

```
1 tibble::tibble(id, name, age, sex) ## or `data.frame()`

# A tibble: 4 × 4
  id   name    age  sex
  <int> <chr> <dbl> <chr>
1     1 Brian    23 male
2     2 Jason    25 male
3     3 Tyler    27 male
4     4 Sam      29 male
```



# Generate Data Sets in R

From scratch, you would create these columns within the `data.frame()` or `tibble()` call:

```
1 tibble::tibble(  
2   name = c("Brian", "Jason", "Tyler", "Sam"),  
3   age = seq(23, 29, by = 2),  
4   sex = rep("male", 4),  
5   id = 1:4  
6 )
```

```
# A tibble: 4 × 4  
  name    age  sex     id  
  <chr> <dbl> <chr> <int>  
1 Brian     23 male     1  
2 Jason     25 male     2  
3 Tyler     27 male     3  
4 Sam       29 male     4
```

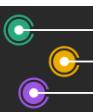


# Generate Data Sets in R

You can also create tibbles using an easier to read row-by-row layout—a **tribble**:

```
1 tibble::tribble(  
2   ~id, ~name,     ~age,   ~sex,  
3   1,   "Brian",  23,    "male",  
4   2,   "Jason",  25,    "male",  
5   3,   "Tyler",  27,    "male",  
6   4,   "Sam",    29,    "male"  
7 )
```

```
# A tibble: 4 × 4  
  id name    age sex  
  <dbl> <chr> <dbl> <chr>  
1     1 Brian    23 male  
2     2 Jason    25 male  
3     3 Tyler    27 male  
4     4 Sam      29 male
```

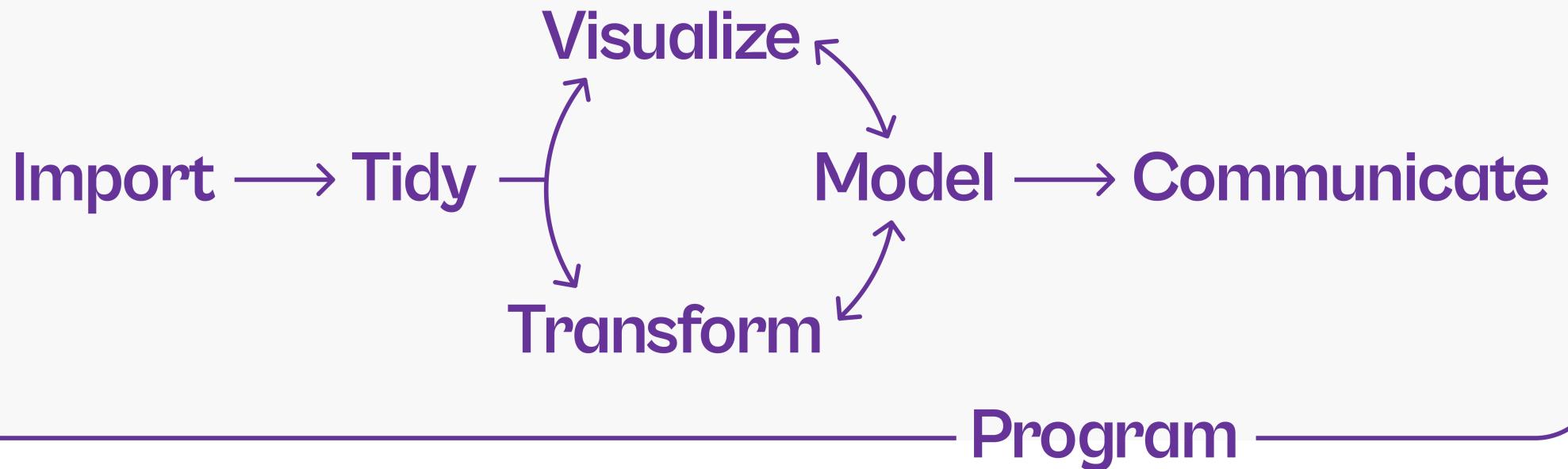


# *Introduction to R*

## *— Data Import —*



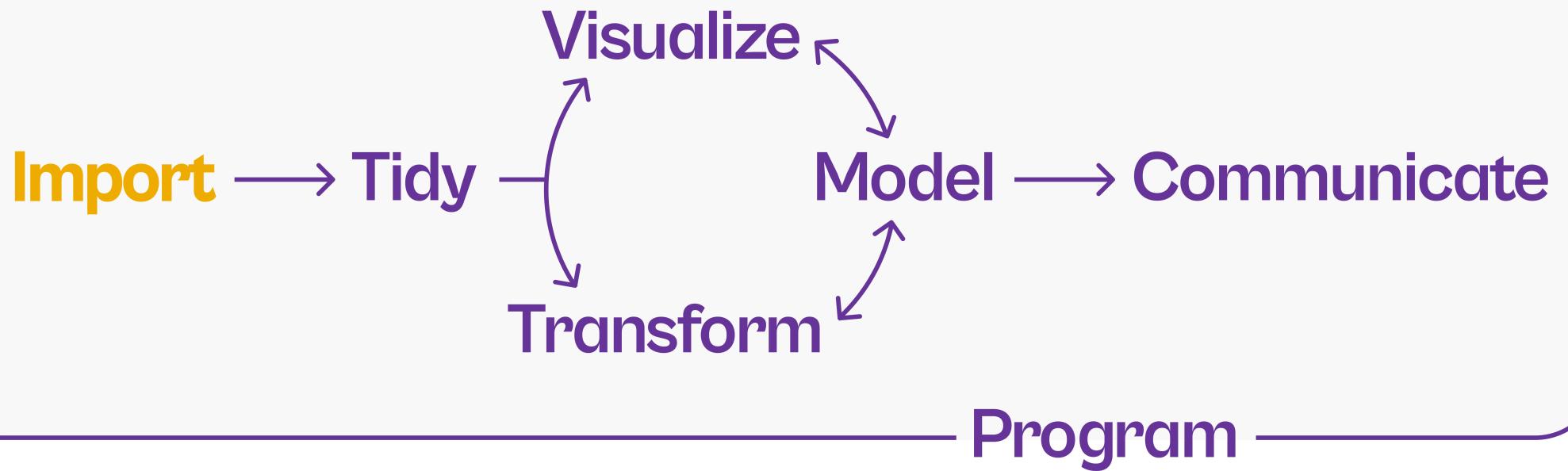
# The Data Science Workflow



The data science workflow, modified from "R for Data Science"



# The Data Science Workflow



The data science workflow, modified from "R for Data Science"



# Data Import

Usually, you create data frames by loading data files such as **.txt**, **.csv**, ...

```
1 my_data <- read.delim("./data/cool_data.txt", sep = "\t")
```

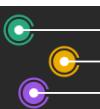
```
1 my_data <- read.csv("./data/cool_data.csv")
```

The **./** refers to a relative path, starting at your **working directory**.

You can check your “wd” by running **getwd()**.

You can also import data from a web resource by passing a valid URL:

```
1 my_data <- read.csv("https://some-webpage.com/online-data.csv")
```



# Data Import: Tibbles

The `{readr}` package provides a fast and friendly way to read rectangular data which are stored as tibbles:

```
1 my_data <- readr::read_delim("./data/cool_data.txt", delim = "\t")
```

```
1 my_data <- readr::read_csv("./data/cool_data.csv")
```

```
1 my_data <- readr::read_csv2("./data/actually_not_a_csv_but_okay.csv") ## ';' as sep
```

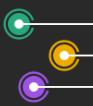


# Data Import: Excel Files (sigh)

The `{readxl}` package provides functionality to import Excel sheets:

```
1 my_data <- readxl::read_xls("./data/oh_no_excel_data.xls", sheet = 1)
```

```
1 my_data <- readxl::read_xlsx("./data/oh_no_excel_data.xlsx", sheet = 1)
```



# Data Import: R-Specific Formats

There are also R-specific formats such as `.Rds` and `.Rdata/.Rda`

```
1 ## base R:  
2 my_data <- readRDS("./data/one_of_Rs_data_formats.Rds")
```

```
1 ## readr package:  
2 my_data <- readr::read_rds("./data/one_of_Rs_data_formats.Rds")
```



# Data Import: Too Slow?

The `{data.table}` package provides functionality for data I/O, wrangling and more. The function to import and export data are known to be very fast:

```
1 my_data <- data.table::fread("./data/cool_data.csv", sep = ",")
```



# Data Import: The I/O Swiss-Army Knife

The `{rio}` package aims to unify data importing and exporting into two functions. Based on the file format, `{rio}` will pick a suitable import/export function.

```
1 my_data <- rio::import("./data/cool_data.txt")
```

```
1 my_data <- rio::import("./data/cool_data.csv")
```

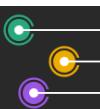
```
1 my_data <- rio::import("./data/one_of_Rs_data_formats.Rds")
```

```
1 my_data <- rio::import("./data/spatial_data_formats_can_be_annoying.shp")
```

```
1 my_data <- rio::import("./data/spatial_data_formats_can_be_annoying.geojson")
```

```
1 my_data <- rio::import("./data/spatial_data_formats_can_be_annoying.gpkg")
```

```
1 my_data <- rio::import("./data/spatial_data_formats_can_be_annoying.tif")
```



# Data Export

In general, the same logic applies when exporting data sets:

```
1 write(my_data, file = "./data/cool_data.txt", sep = "\t")
```

```
1 write.csv(my_data, file = "./data/cool_data.csv")
```

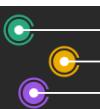
```
1 readr::write_csv(my_data, file = "./data/cool_data.csv")
```

```
1 data.table::fwrite(my_data, file = "./data/cool_data.csv", sep = ",")
```

```
1 saveRDS(my_data, file = "./data/one_of_Rs_data_formats.Rds")
```

```
1 readr::write_rds(my_data, file = "./data/one_of_Rs_data_formats.Rds")
```

```
1 rio::export(my_data, "./data/spatial_data_formats_can_be_annoying.tif")
```



# Your Turn: Data Import & Export

:: exercise ::: {.incremental} \* Import the TfL bikes data set (`london-bikes-custom.csv`), stored in the `data` folder or load it directly from the following URL: <https://cedricscherer.com/data/london-bikes-custom.csv> \* Check the data set—do the values and data types look fine? \* Save the file as a Rds file. \* Bonus: import your own data set or other data sets and formats. :::



# Your Turn: Data Import

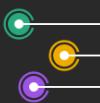
```
1 df_bikes <- read.delim("./data/london-bikes-custom.csv", sep = ",")  
2 tibble::glimpse(df_bikes)
```



# Your Turn: Data Import

```
1 df_bikes <- read.csv("./data/london-bikes-custom.csv")
2 tibble::glimpse(df_bikes)
```

```
Rows: 1,454
Columns: 14
$ date              <chr> "2015-01-04", "2015-01-04", "2015-01-05", "2015-01-05", "2015-01-06", "2015-01-06", "20...
$ day_night        <chr> "day", "night", "day", "night", "day", "night", "day", "night", "day", ...
$ year              <int> 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 201...
$ month             <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ season            <chr> "winter", "winter", "winter", "winter", "winter", "winter", "winter", "winter", ...
$ count              <int> 6830, 2404, 14763, 5609, 14501, 6112, 16358, 4706, 9971, 5630, 16568, 5536, 10413, 4296...
$ is_workday        <int> 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, ...
$ is_weekend         <int> 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, ...
$ is_holiday         <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ temp               <dbl> 2.166667, 2.791667, 8.958333, 7.125000, 9.000000, 6.708333, 8.166667, 6.681818, 9.45833...
$ temp_feel          <dbl> -0.75000000, 2.04166667, 7.70833333, 5.70833333, 6.45833333, 4.20833333, 5.08333333, 3...
$ humidity            <dbl> 95.16667, 93.37500, 81.08333, 79.54167, 80.20833, 77.58333, 75.20833, 81.27273, 79.4166...
$ wind_speed          <dbl> 10.416667, 4.583333, 8.666667, 9.041667, 19.208333, 12.791667, 21.250000, 18.136364, 18...
$ weather_type        <chr> "broken clouds", "clear", "broken clouds", "cloudy", "broken clouds", "clear", "scatter...
```



# Your Turn: Data Import

```
1 df_bikes <- rio::import("./data/london-bikes-custom.csv")
2 tibble::glimpse(df_bikes)
```



# Your Turn: Data Import

```
1 df_bikes <- rio::import("./data/london-bikes-custom.csv", setclass = "tbl_df")
2 tibble::glimpse(df_bikes)
```



# Your Turn: Data Import

```
1 df_bikes <- readr::read_csv("./data/london-bikes-custom.csv")
2 tibble::glimpse(df_bikes)
```

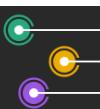
```
Rows: 1,454
Columns: 14
$ date              <date> 2015-01-04, 2015-01-04, 2015-01-05, 2015-01-05, 2015-01-06, 2015-01-06, 2015-01-07, 20...
$ day_night        <chr> "day", "night", "day", "night", "day", "night", "day", "night", "day", ...
$ year              <dbl> 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 201...
$ month             <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ season            <chr> "winter", "winter", "winter", "winter", "winter", "winter", "winter", "winter", ...
$ count              <dbl> 6830, 2404, 14763, 5609, 14501, 6112, 16358, 4706, 9971, 5630, 16568, 5536, 10413, 4296...
$ is_workday        <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, ...
$ is_weekend         <dbl> 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
$ is_holiday         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ temp               <dbl> 2.166667, 2.791667, 8.958333, 7.125000, 9.000000, 6.708333, 8.166667, 6.681818, 9.4583...
$ temp_feel          <dbl> -0.75000000, 2.04166667, 7.70833333, 5.70833333, 6.45833333, 4.20833333, 5.08333333, 3...
$ humidity            <dbl> 95.16667, 93.37500, 81.08333, 79.54167, 80.20833, 77.58333, 75.20833, 81.27273, 79.4166...
$ wind_speed          <dbl> 10.416667, 4.583333, 8.666667, 9.041667, 19.208333, 12.791667, 21.250000, 18.136364, 18...
$ weather_type        <chr> "broken clouds", "clear", "broken clouds", "cloudy", "broken clouds", "clear", "scatter...
```



# Your Turn: Data Import

```
1 df_bikes <- readr::read_csv("./data/london-bikes-custom.csv",
2                               col_types = "Dcficillllddddf")
3 tibble::glimpse(df_bikes)
```

```
Rows: 1,454
Columns: 14
$ date            <date> 2015-01-04, 2015-01-04, 2015-01-05, 2015-01-05, 2015-01-06, 2015-01-06, 2015-01-07, 20...
$ day_night       <chr> "day", "night", "day", "night", "day", "night", "day", "night", "day", ...
$ year            <fct> 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 2015, 201...
$ month           <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ season          <chr> "winter", "winter", "winter", "winter", "winter", "winter", "winter", "winter", "winter...
$ count           <int> 6830, 2404, 14763, 5609, 14501, 6112, 16358, 4706, 9971, 5630, 16568, 5536, 10413, 4296...
$ is_workday      <lgl> FALSE, FALSE, TRUE, FALSE, FALSE, ...
$ is_weekend      <lgl> TRUE, TRUE, FALSE, TRUE, ...
$ is_holiday       <lgl> FALSE, FAL...
$ temp             <dbl> 2.166667, 2.791667, 8.958333, 7.125000, 9.000000, 6.708333, 8.166667, 6.681818, 9.45833...
$ temp_feel       <dbl> -0.75000000, 2.04166667, 7.70833333, 5.70833333, 6.45833333, 4.20833333, 5.08333333, 3....
$ humidity         <dbl> 95.16667, 93.37500, 81.08333, 79.54167, 80.20833, 77.58333, 75.20833, 81.27273, 79.4166...
$ wind_speed       <dbl> 10.416667, 4.583333, 8.666667, 9.041667, 19.208333, 12.791667, 21.250000, 18.136364, 18...
$ weather_type     <fct> broken clouds, clear, broken clouds, cloudy, broken clouds, clear, scattered clouds, cl...
```



# Your Turn: Data Export

```
1 saveRDS(df_bikes, file = "./data/london-bikes-custom.Rds")
```

```
1 readr::write_rds(df_bikes, file = "./data/london-bikes-custom.Rds")
```

```
1 rio::export(df_bikes, file = "./data/london-bikes-custom.Rds")
```



# *Introduction to R*

## *— Getting Help —*

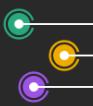


# Getting Help with R

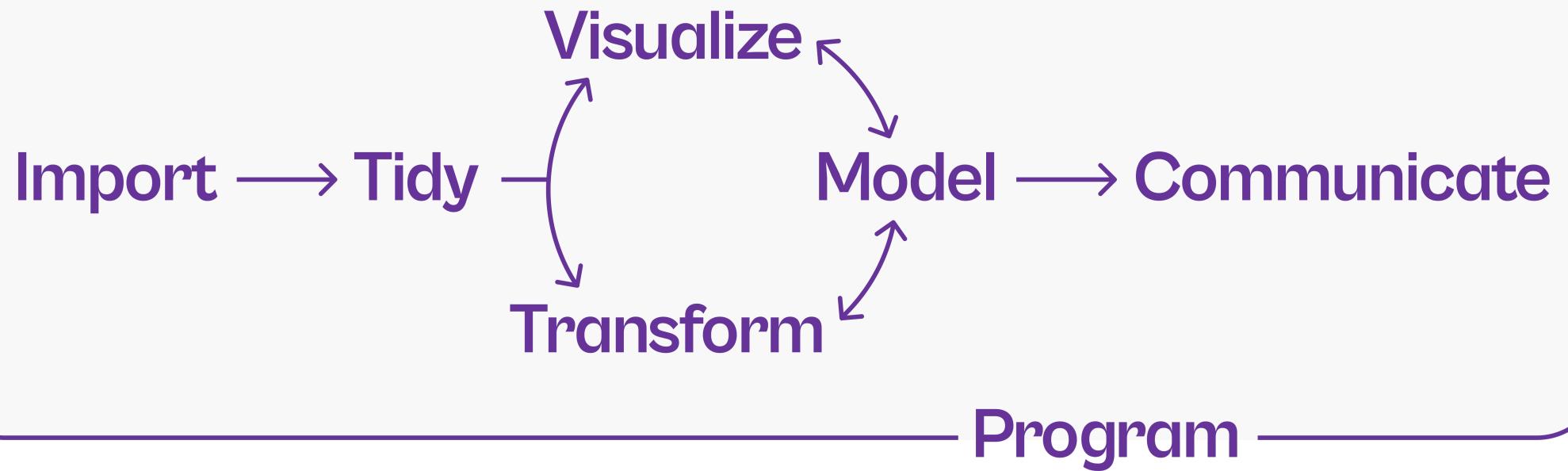
- R Foundation Help Page: [r-project.org/help.html](http://r-project.org/help.html)
- Posit / RStudio Community: [community.rstudio.com](https://community.rstudio.com)
- R For Data Science Online Learning Community: [rfor datasci.com](https://rfor datasci.com)
- Stack Overflow: [stackoverflow.com](https://stackoverflow.com)
- R User Groups (local): [benubah.github.io/r-community-explorer/rugs.html](https://benubah.github.io/r-community-explorer/rugs.html)
- R Ladies (global & local): [rladies.org](https://rladies.org)
- Twitter: [#rstats](#), [#tidyverse](#), [#ggplot2](#)
- Books:
  - [Modern Dive](#)
  - [R for Data Science](#)
  - [The Big Book of R](#)



# *What's Next?*



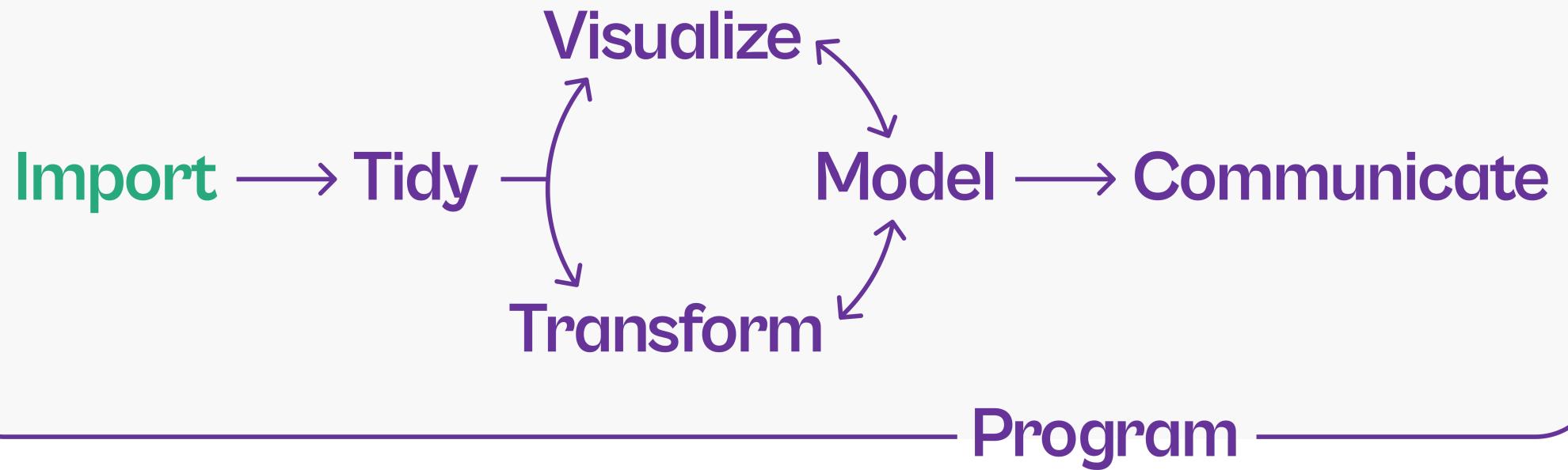
# The Data Science Workflow



The data science workflow, modified from "[R for Data Science](#)"



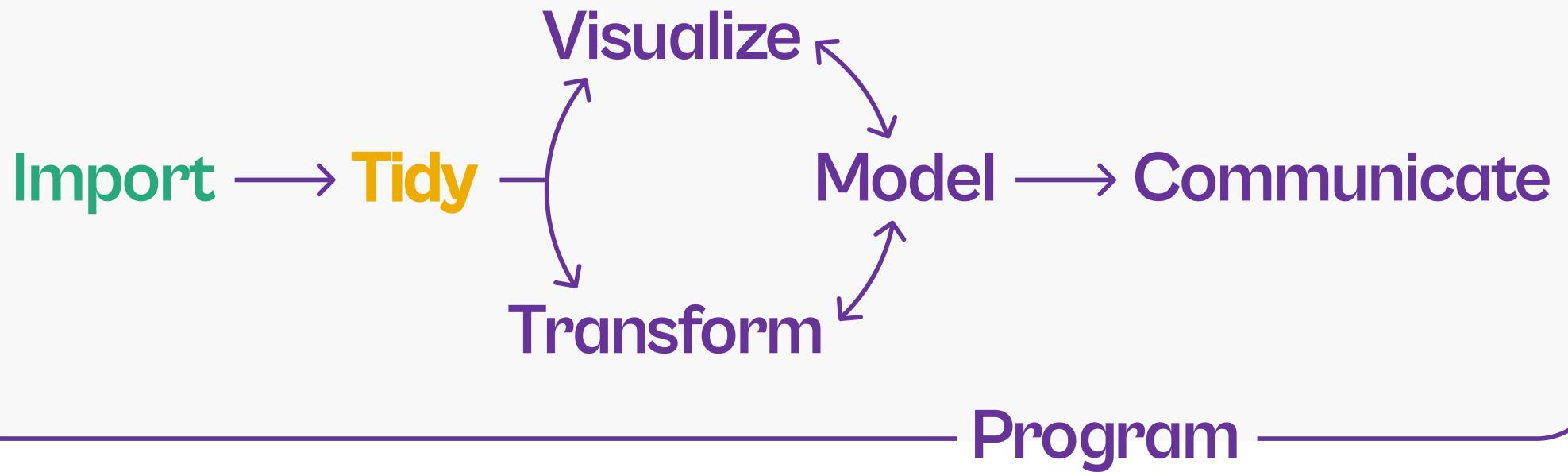
# The Data Science Workflow



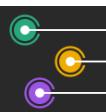
The data science workflow, modified from "R for Data Science"



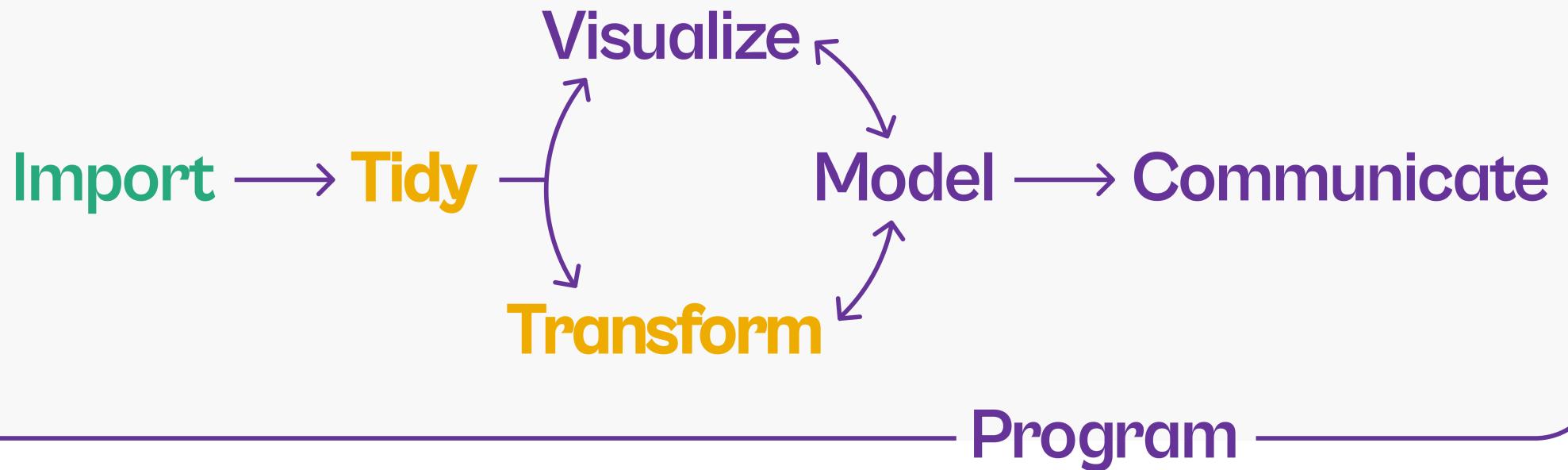
# The Data Science Workflow



The data science workflow, modified from "R for Data Science"



# The Data Science Workflow



The data science workflow, modified from "R for Data Science"



# That's it Folks... — Thank you! —

