# R intro: super-crash course

## Learn (base) R in hours

Axel Thieffry, PhD

October 2022

# Agenda

- Data types
- Variables & Assignment
- Data structures
- Naming
- Slicing
- Working directory, reading files, saving objects
- Functions & libraries
- Apply family
- Useful functions
- Good R coding practices

# Here: base R

- base R = core R functions and utilities
- Doesn’t require any library
- Pretty low-level
- Can (and will) get difficult to read
- Later we will see R on steroid: ***The Tidyverse***

# Pre-requisites

- R installation
  => https://ftp.belnet.be/mirror/CRAN/ (🇧🇪 mirror)

- RStudio
  => https://www.rstudio.com/products/rstudio/download/

# Data types

# Basic Data types

- **numeric** (including **double**)
  `24.3`, `5`, `-12`
- **integer** (L is old notation)
  `1L`, `20L`, `32L`
- **character** (n=1), and **string** (n>1)
  `'John'`, `"John"`, `“FALSE”`
- **logical**, aka **boolean** (always capital letters)
  `TRUE`, `FALSE`, `T`, `F`
- **factors**: ordered strings (more on that later)

RStudio Syntax highlight (Cobalt theme)

```
1  'John' "John"
2  24.3
3  5
4  5L
5  T TRUE F FALSE
6  variable
7  + - / *
8  # this is a comment
9  this_is_a_function()
```

Unsure?
=> Use the functions `type()` and `class()`

# Variables & assignment

# Variables & Assignment

**No need to declare a variable first:**
**it gets created when you assign a value.**

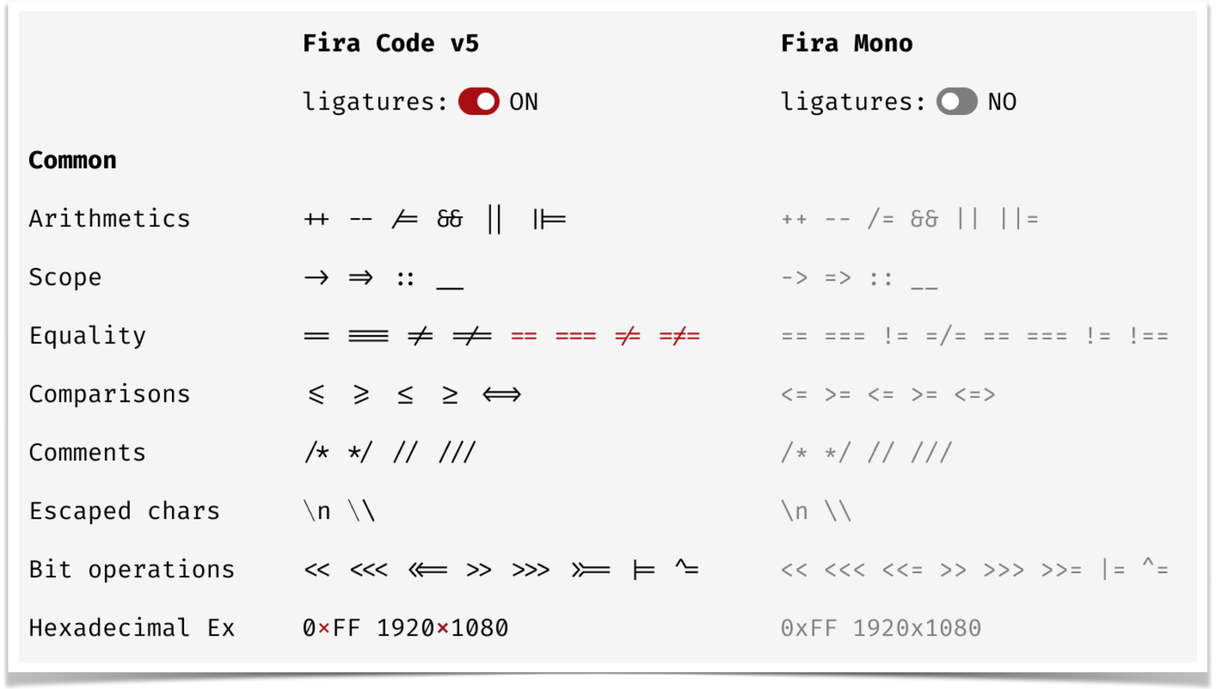
- **Python** uses the equal sign: **=**
  *ex:*
  `a=3`

- **R** uses an arrow: "**<-**"
  This better reflects the information flow and its direction
  *ex:*
  `a <- 3`

  - **Note:** `a=3` will also work in R, but is bad practice!
    We keep = and == signs to assign names (more on that later) or test equality

# PRO-tip: change font & get a theme!

- Let's install **FiraCode:** github.com/tonsky/FiraCode



| | Fira Code v5 | Fira Mono |
|---|---|---|
| | ligatures: ON | ligatures: NO |
| **Common** | | |
| Arithmetics | ++ -- /= && \|\| \|= | ++ -- /= && \|\| \|\|= |
| Scope | -> => :: __ | -> => :: __ |
| Equality | == === != =/= <= >= | == === != =/= <= >= |
| Comparisons | <= >= <= >= <=> | <= >= <= >= <=> |
| Comments | /* */ // /// | /* */ // /// |
| Escaped chars | \n \\ | \n \\ |
| Bit operations | << <<< <<= >> >>> >>= \|= ^= | << <<< <<= >> >>> >>= \|= ^= |
| Hexadecimal Ex | 0xFF 1920x1080 | 0xFF 1920x1080 |

- Get a nice theme: ***RStudio > Preferences > Appearance > Editor theme***
  My favourite is 'Cobalt': great syntax highlight & easy on the eyes.

# Variables & Assignment

**No need to declare a variable first:**
**it gets created when you assign a value.**

- **Python** uses the equal sign: **=**
  *ex:*
  `a=3`

Previous slide with
**FiraCode** On! 💪

- **R** uses an arrow: "←"
  This better reflects the information flow and its direction
  *ex:*
  `a ← 3`

- Note: `a=3` will also work in R, but is bad practice!
  We keep `=` and `==` signs to assign names (more on that later) or test equality

# Variables

## As usual….

- Use meaningful and non-cryptic variable names
- Chose a variable style (and stick to it!)
  ex:
  - camelcase: `MyDataFrame`
  - underscores: **`my_data_frame`**
  - dots: `my.data.frame`

**Probably the best for biology**

- Don’t use special characters, no spaces, don’t start with a number, no − (will be interpreted as the mathematical operator ‘minus’)

# Data structures

# (atomic) Vector

- One-dimension
- Homogeneous: One data type
- Most often created by concatenation: `c()`

```
c(1, 5, 12)
c("ORF42", 'ORF43', "noORF")
```

- Always flat:

```
    c(1, 2, c(32, -4))
```

is the same as

```
    c(1, 2, 32, -4)
```

# Coercion

- To ***coerce*** = fancy word for “trying to changing the type”
- All elements of an atomic vector must be the same. So if you combine different types, they will be *coerced* to the **most flexible type**.
- Types from least to most flexible: *logical*, *integer*, (numeric), *double*, *character*

```
> class( c(1, 2) )
[1] "numeric"
> class( c(1L, 2L) )
[1] "integer"
> class( c(1L, 2L, 3) )
[1] "numeric"
> class( c(1, '2') )
[1] "character"
```

# Coercion

- Some functions will try to coerce the data if they are fed with inadequate types.
- This can be very useful when working with logicals:

```
x ← c(F, F, T)
as.numeric(x)
#> [1] 0 0 1
```

FALSE will be 0
TRUE will be 1

```
sum(x)
#> [1] 1
```

```
mean(x)
#> [1] 0.3333333
```

Calculate proportions
super fast!

# Quickly create a vector of integers

- Sadly, this doesn't work for letters…

  `D:F`

  won't give you `[1] "D" "E" "F"`

- But! Try:

  `letters`

  `LETTERS`

```
> 1:5
[1] 1 2 3 4 5
> 5:1
[1] 5 4 3 2 1
> -4:-1
[1] -4 -3 -2 -1
> -1:-4
[1] -1 -2 -3 -4
> -2:2
[1] -2 -1  0  1  2
> 1.5:2.7
[1] 1.5 2.5
> 1.3:2.7
[1] 1.3 2.3
```

# List

- Different from atomic vectors: can **mix data types**
- Create a list using `list()`

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

# List

- Lists are sometimes called "***recursive vectors***" because they can contain other lists.
- c() will combine several lists into one
- Ways to ingress and egress from lists:
  `as.list(x)`
  `unlist(mylist)`

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

# Matrix

- 2-D object with homogeneous data type
- Create a matrix with `matrix()`
- *Coerce* to a matrix with `as.matrix()`
- Can be created from **vectors** when specifying the structure of the matrix
- Note: create a ***diagonal*** matrix with `diag()`

```
1  d ← diag(nrow=3, ncol=3)
2  names(d) ← letters[1:9]
3  colnames(d) ← letters[1:3]
4  rownames(d) ← LETTERS[1:3]
```

Try me !

```
1  matrix(1:20, nrow=5)
2  matrix(1:20, nrow=5, byrow=T)
```

5:1 (Top Level)

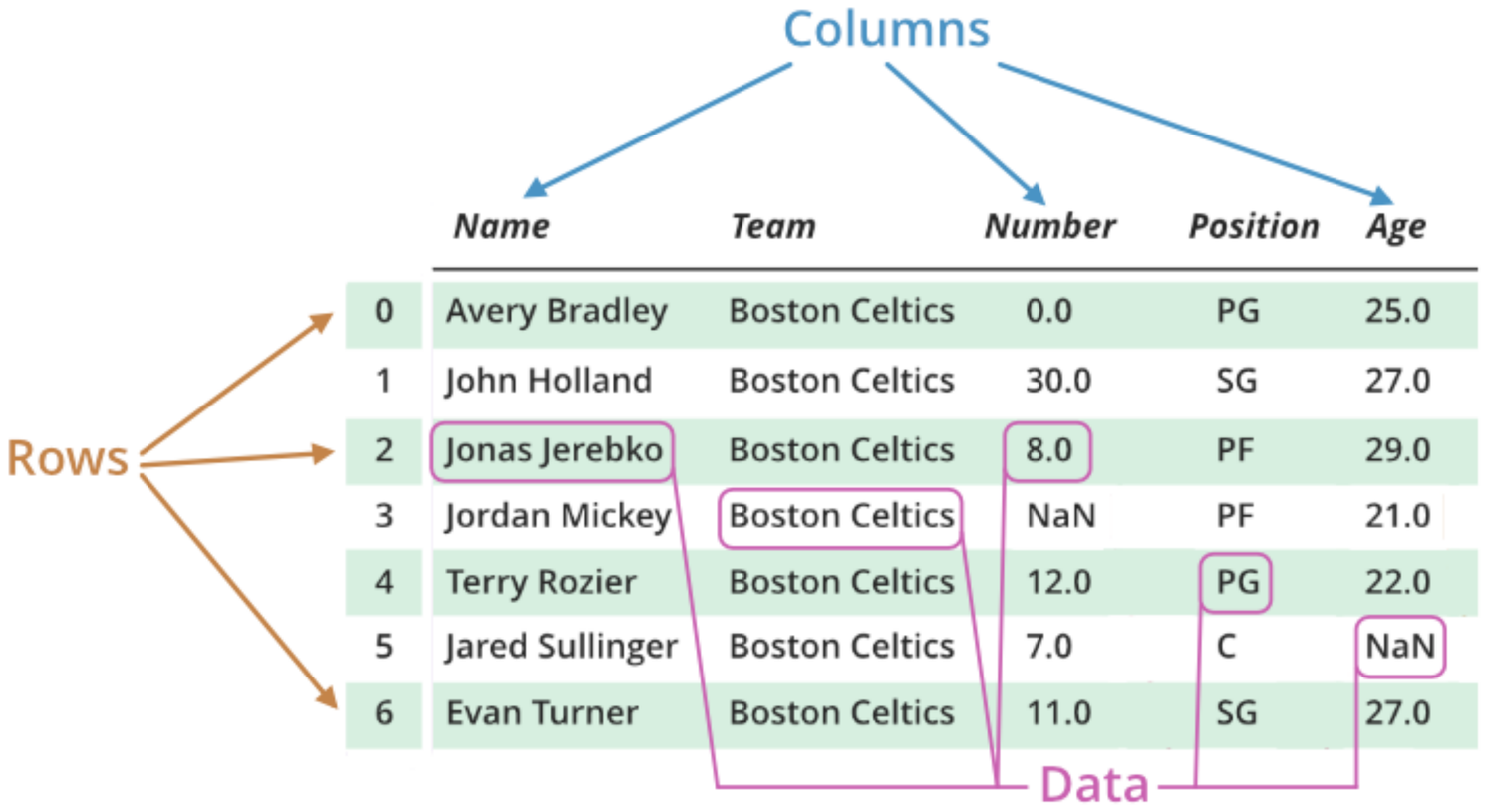
R 4.1.0 · ~/

```
> matrix(1:20, nrow=5)
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
> matrix(1:20, nrow=5, byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
[5,]   17   18   19   20
```

Notice the ‘`byrow`’

# Data frame

- 2D object with heterogeneous data types




| | Name | Team | Number | Position | Age |
|---|---|---|---|---|---|
| 0 | Avery Bradley | Boston Celtics | 0.0 | PG | 25.0 |
| 1 | John Holland | Boston Celtics | 30.0 | SG | 27.0 |
| 2 | Jonas Jerebko | Boston Celtics | 8.0 | PF | 29.0 |
| 3 | Jordan Mickey | Boston Celtics | NaN | PF | 21.0 |
| 4 | Terry Rozier | Boston Celtics | 12.0 | PG | 22.0 |
| 5 | Jared Sullinger | Boston Celtics | 7.0 | C | NaN |
| 6 | Evan Turner | Boston Celtics | 11.0 | SG | 27.0 |

# Data frame creation & basic methods

- By reading in Excel, text, csv, tsv, files (more on that later)
- Manually with `data.frame()`
- By coercing an object with `as.data.frame()`
- 

```r
mydf ← data.frame('letters'=c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'),
                  'numbers'=1:10)
rownames(mydf) ← c('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J')
dim(mydf)
colnames(mydf)
rownames(mydf)
ncol(mydf)
nrow(mydf)
```

Try me !

- Interesting methods with data frames:
  - `dim()`
  - `ncol()`
  - `nrow()`
  - `colnames()`
  - `rownames()`
  - `summary()`

# Naming

# Naming

- Names can be attributed to the elements of **lists** and **vectors**
- Names can be set with `names()` or `set_names()` or the `=` sign

```
wt_counts ← c(5, 12, 3, 0, 14, 13)
mut_counts ← c(5, 14, 3, 20, 7, 10)
gene_names ← c('P53', 'HSF2', 'CDF4', 'UGA4', 'DAL5', 'FLS2')
samples_names ← c('wt', 'mut')

names(wt_counts) ← names(mut_counts) ← gene_names
wt_counts ; mut_counts

list(wt_counts, mut_counts)
setNames(list(wt_counts, mut_counts), samples_names)
list('wt'=wt_counts, 'mut'=mut_counts)
```

Reproduce the following commands and explain what happens.

# Slicing

# Slicing

- ***Slicing*** is the fancy R word for “subsetting”
- Can be done on vectors, data.frames, and lists
- Use `[ ]` and `[[ ]]` if based on position
- Use `$` if based on names

# Slicing vectors

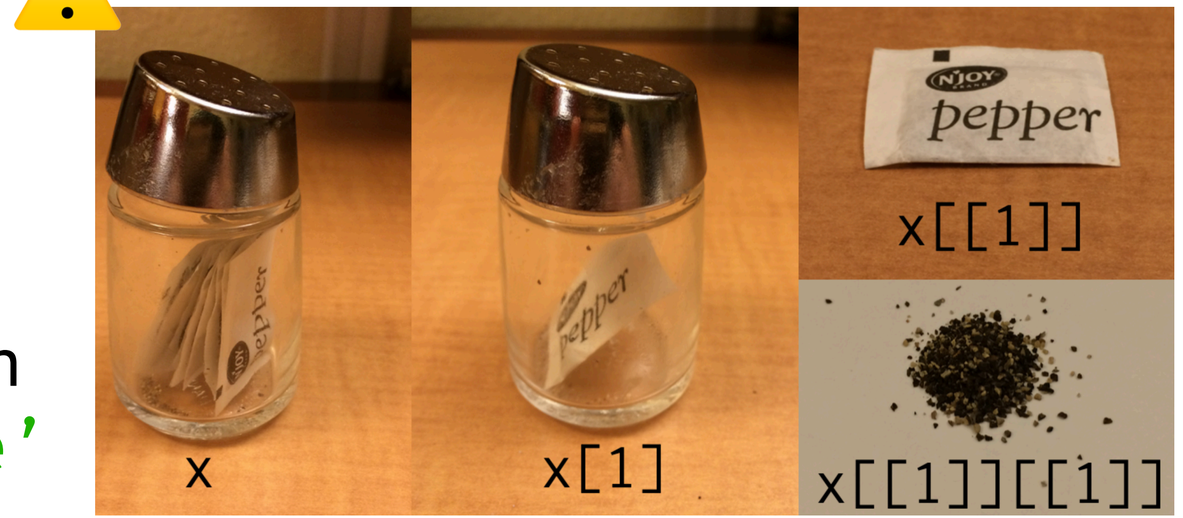
```r
vec ← c(1:10)
vec[-1]
vec[-1:-3]
vec[5]

named_vec ← c('first'=1, 'second'=2, 'third'=3)
named_vec['second']
```

`vec ← 1:10` would actually do the trick

Reproduce the following commands and explain what happens.

# Slicing lists

- Using `[ ]` and `[[ ]]`
- Note that if your list is named you can access items using: `mylist$'name'`




```
1  test ← list(c('a', 'b', 'c'),
2              1:3,
3              paste0(1:3, c('a', 'b', 'c')))
4
5  test[3]
6  test[[1]]
7
8  names(test) ← c('letters', 'numbers', 'both')
9  test$numbers
10 test['numbers']
```

Reproduce the following commands and explain what happens.
Can you understand the differences?

# Slicing dataframes

- Using `[ ]`, `[[ ]]`, and `$`
- `my_data_frame[…, …]`

For columns

For rows

*Rainbow parenthesis. Enable them in RStudio preferences!*

```
1  iris[1, ]
2  iris[, 1]
3  iris[[1, 5]]
4  iris[1:5, 1:5]
5  iris[1:5, 1:7]
6  iris$Sepal.Width
```

Let's try by doing.

# Slicing using logicals!

```r
iris[iris$Sepal.Length > 7, ]
subset(iris, Sepal.Length > 7)

iris$Sepal.Length[iris$Sepal.Length ⩽ 7.0] ← 0
```

Reproduce the following commands and explain what happens.
Can you understand the differences?

# Recycling

```
1   rec ← 1:20
2   logicals ← c(T, F)
3   rec[logicals]
```

What happened here?

**Tips:** try

```
length(rec)
length(logicals)
```

# Working directory, files, and saving objects

# Working directory

- `getwd()`

  tells you the path from which your script is executed

- `setwd("path/to/directory")`

  changes the working directory

# Listing files

- `list.files()`

# Reading files

- `read.table()`
- `read.csv()`
- `readxl::read_xlsx()`

# Writing files

- `write.table()`
- `WriteXLS::WriteXLS()`

# Saving objects

- Sometimes interesting to save R-object(s). For example:
  i) Output of an intense parsing or analysis, that will be used for many other R analyses. This is a way to compartmentalise R analyses into smaller, more manageable scripts.
  ii) Output of a large R analysis that might be needed in many (yet unforeseen) formats (Export for Excel, export for mySQL, export for the web, etc…)

- **RDS**: one object, can be renamed when imported at a later stage

```
saveRDS(cleaned_data_df, 'cleaned_data_for_R.rds')
uORFs ← readRDS('cleaned_data_for_R.rds')
```

- **RDA** (R data): can be a collection of objects, cannot be renamed when imported

```
saveRDS(list(uORF_list, DEGs_mut_vs_ctrl), 'data_21102021.Rdata')
load('data_21102021.Rdata')
# uORF_list and DEGs_mut_vs_ctrl will now be in your R environment
```

# Functions & Libraries

# Base R functions

- Typical syntax:

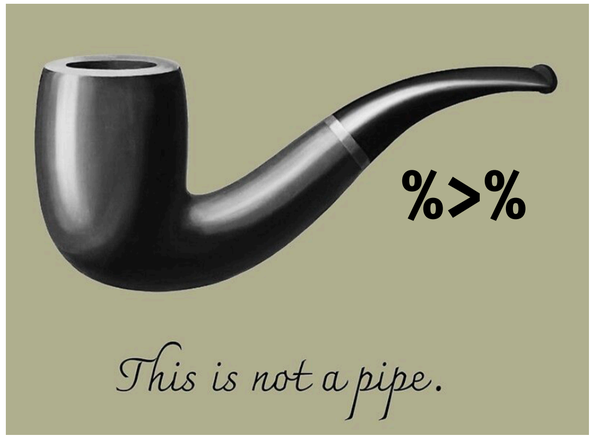
```
function(argument=value, …)
```

- They can be nested, but reading rapidly becomes difficult

```
length(unique(round(iris$Sepal.Length)))
```

- Later on we will see how to use **pipes** with the `magrittr` package. This improves readability **a lot:**

```
iris %>%
    select(Sepal.Length) %>%
    round() %>%
    unique() %>%
    length()
```



# GET-ter and SET-ters

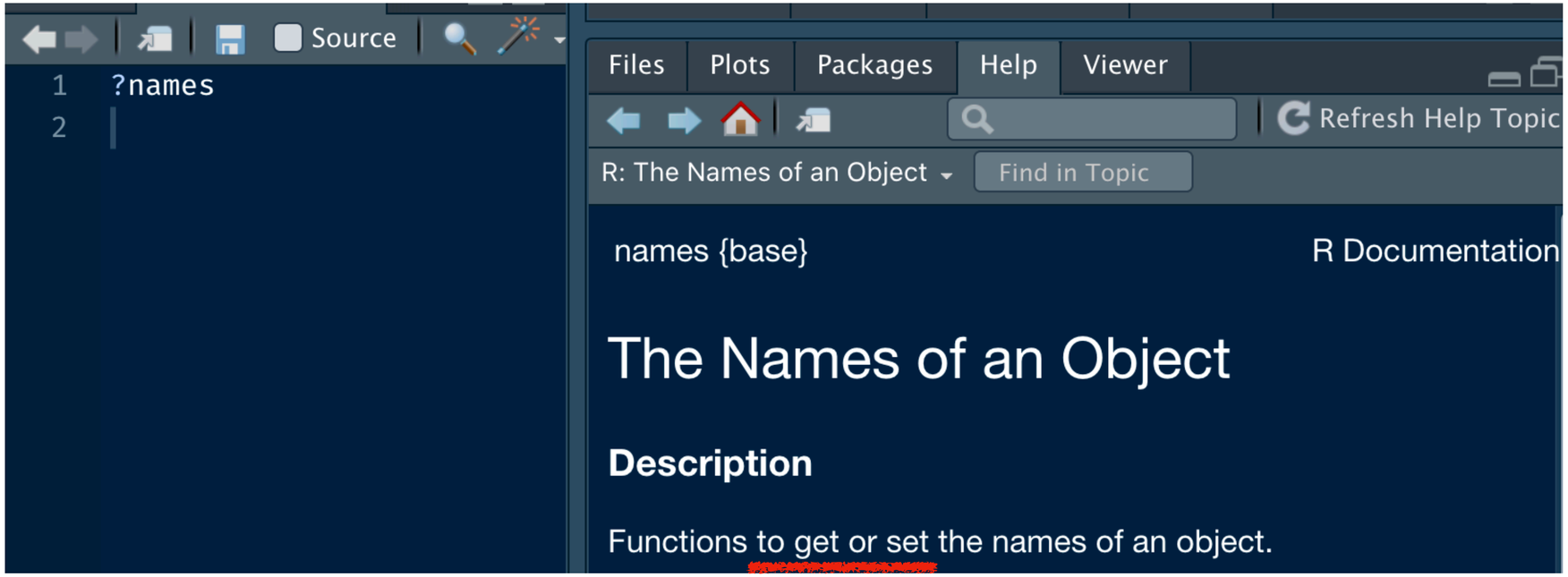
- Some functions can be use both to **GET** info or to **SET** data
- Examples: `names()`, `colnames()`, `rownames()`

```r
# create vector
read_counts ← c(1, 50, 2, 0, 34, 24, 32)
# assign names to elements of vector
names(read_counts) ← letters[1:length(read_counts)]
# get names of vector elements
names(read_counts)
```

Try it !

# Get help

- type the name of the function preceded by "**?**"
- **No parenthesis!**
- Ex: `?names`




This help page is called the "Vignette".
Basically the manual for the 'names' function.

# Custom functions

- Create you own function:

```
my_custom_fun ← function(x, y) {
                                result ← x + y
                                return(result)
                               }
```

- Use you own function:

```
my_custom_fun(x=input1, y=input2)
```

- You can set some default values for your input variables
- Sadly, no “`docstring`” as can be done in Python…

# Libraries

- Libraries are a collection of high-level functions

- Installation of a library (only once!) depends on which repository it is hosted:
  - **CRAN**:
  ```
  install.packages('library_name')
  ```
  - **Bioconductor**:
  ```
  install.packages('BiocManager')
  BiocManager::install('GenomicRanges')
  ```
  - **GitHub**:
  ```
  install.packages('devtools')
  devtools::install_github('athieffry/TxDb.Athaliana.BioMart.plantsmart51')
  ```

See how Bioconductor is a package manager that needs to be installed from CRAN first?

- Loading a library is accomplished by:
  ```
  library(library_name)
  ```

# Conflicts & masked functions

```
> library(tidyverse)
── Attaching packages ──────────────────────────────────────
✓ ggplot2 3.3.5     ✓ purrr   0.3.4
✓ tibble  3.1.4     ✓ dplyr   1.0.7
✓ tidyr   1.1.3     ✓ stringr 1.4.0
✓ readr   2.0.1     ✓ forcats 0.5.1
── Conflicts ──────────────────────────────────────────────
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
> library(reshape2)

Attaching package: ‘reshape2’

The following object is masked from ‘package:tidyr’:

    smiths
```

- If you want to use `smiths()` from the `tidyr` library rather than from `reshape2`: `tidyr::smiths()`

# Apply family

# If you use a loop (`for` or `while`) in R, you are doing it wrong in 99% of cases.

This is because most of R's functions are **vectorised:**
the function will operate on all elements of a vector without needing to loop through and act on each element one at a time.

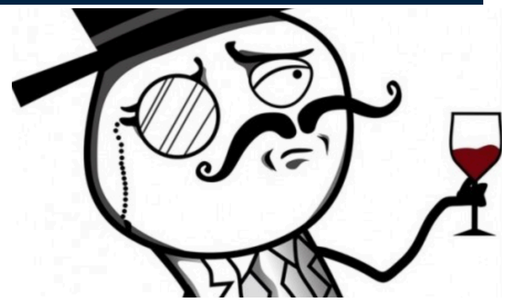
# Vectorisation example:

```
> nums ← 1:5
> nums
[1] 1 2 3 4 5
```

Multiply each element by 2:

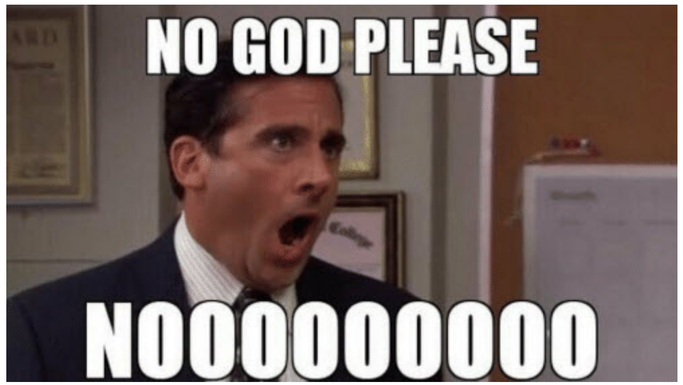
```
nums ← nums * 2
```

✅



```
for (i in 1:length(nums)) {
  nums[i] ← nums[i] * 2
  }
```

🚫



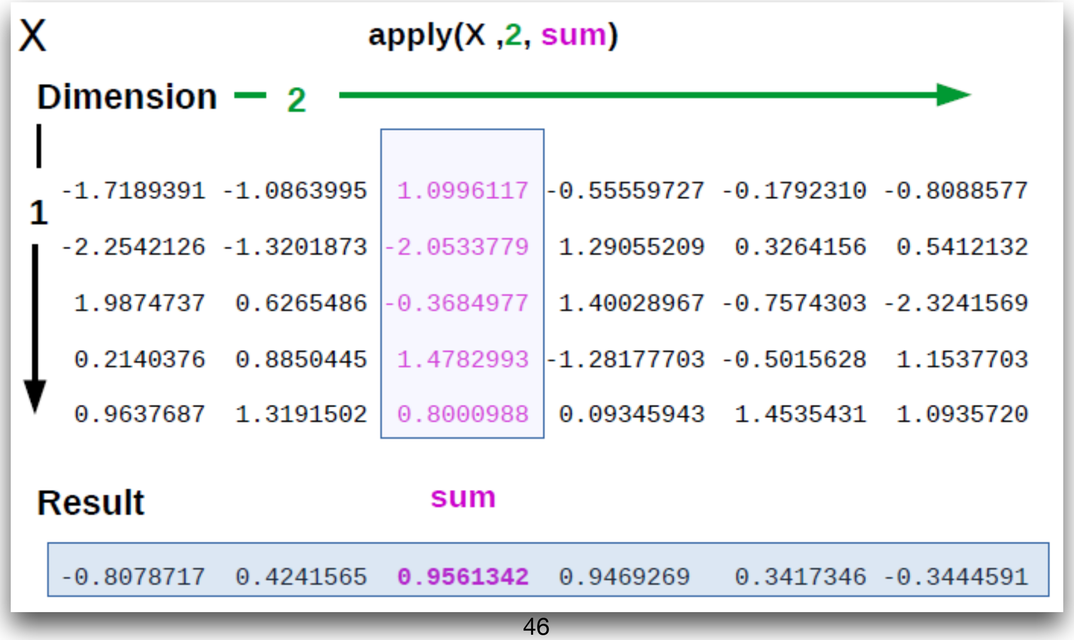
# Apply family

- Collection of functions to execute repetitive tasks on data structures
- Intuitively: a R-way of doing `for` loops
- Many of them, depending on the data structure and what you want out:
  `apply()`
  `lapply()`
  `sapply()`
  `mapply()`
  and some other exotic ones…

# Apply family

`apply(X, MARGIN, FUN, …)`

- X: a **dataframe**, **matrix**, or **array**
- `MARGIN`: how the function is applied: 1 = on rows, 2 = on columns
- `FUN`: the function you want to apply over `MARGIN`




**Note:** this summing of columns/ rows are so often needed that base R functions readily exist:

`rowSums(X)`

`colSums(X)`

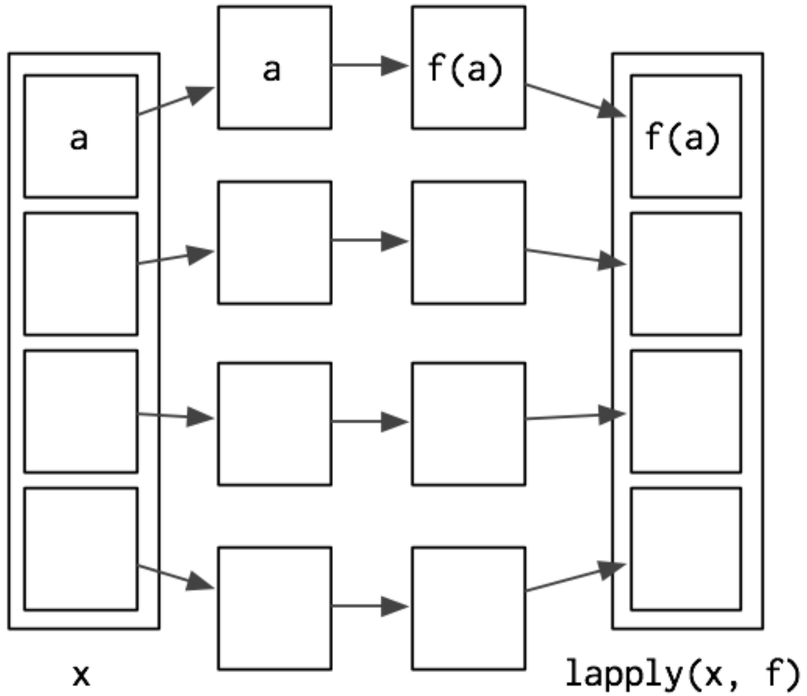
`rowMeans(X)`

`colMeans(X)`

# Apply family

`lapply(X, FUN, …)`

- `X`: a **vector** (atomic or list)
- `FUN`: function to be applied to each element of `X`




**What if I simply want a vector for result?**

# Apply family

`sapply(X, FUN, …, simplify=T, USE.NAMES=T)`

- Same as `lapply()` but with a **s**implified response: a **vector** is returned instead of a list
- Just meant to save time by not having to coerce the list into an atomic vector
- Note that if X was named, the resulting atomic vector will keep the names

```
1  iris
2  iris_dfl ← split(iris, iris$Species)
3  lapply(iris_dfl, nrow)
4  sapply(iris_dfl, nrow)
5  names(iris_dfl) ← NULL
6  sapply(iris_dfl, nrow)
```

Reproduce the following commands and explain what happens

# Apply family

```
mapply(X, Y, FUN, …, simplify=T, USE.NAMES=T)
```

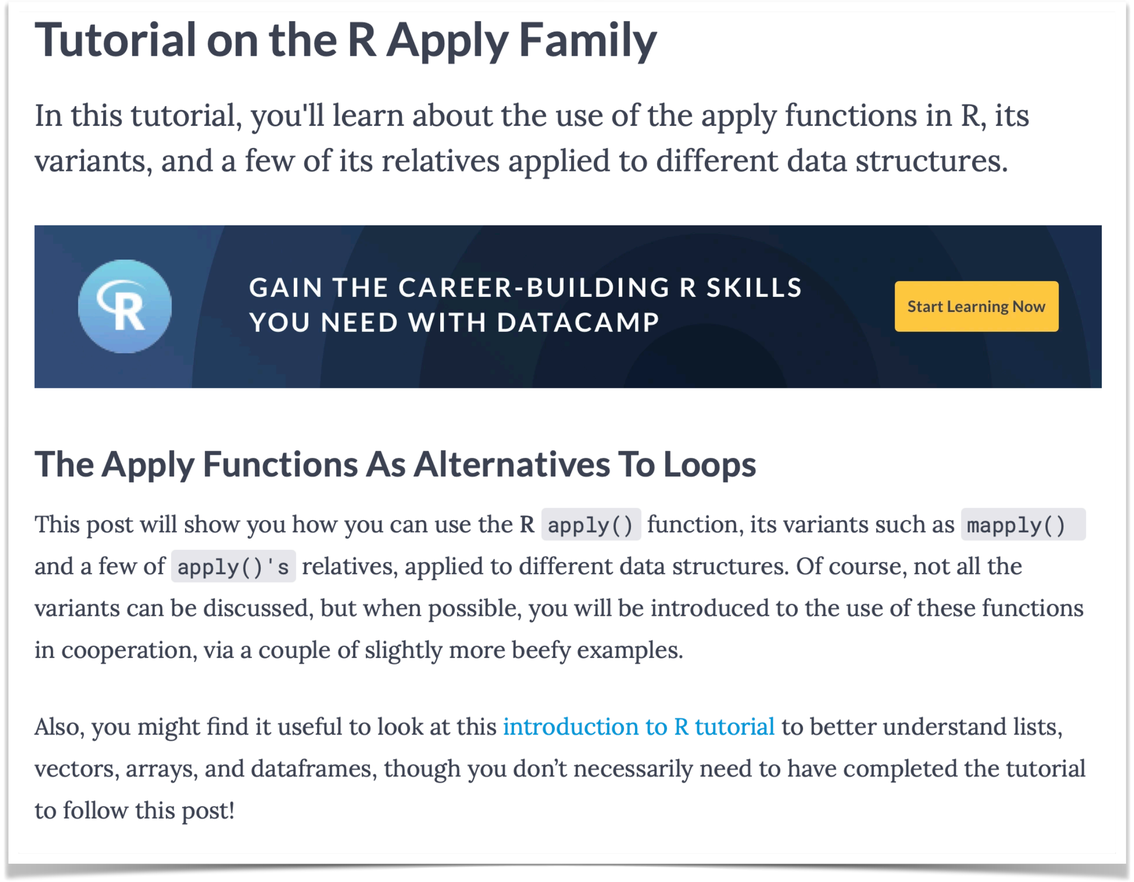
- Multivariate apply
- Vectorize arguments to a function that would otherwise not accept vectors as argument
- Intuitively:
  a) two vectors are passed to a function
  b) the function is executed while going through the vectors in a “parallel”, incremented, and ordered way
- Easier with an example:

Reproduce the following commands and explain what happens

```
iris_dfl ← split(iris, iris$Species)
filenames ← paste0(names(iris_dfl), '.csv')

mapply(function(df, fn) write.table(df, fn, sep=',', row.names=F), iris_dfl, filenames)
```

# Read more:

- More details on the apply family at Datacamp.com



## Tutorial on the R Apply Family

In this tutorial, you'll learn about the use of the apply functions in R, its variants, and a few of its relatives applied to different data structures.



### The Apply Functions As Alternatives To Loops

This post will show you how you can use the R `apply()` function, its variants such as `mapply()` and a few of `apply()'s` relatives, applied to different data structures. Of course, not all the variants can be discussed, but when possible, you will be introduced to the use of these functions in cooperation, via a couple of slightly more beefy examples.

Also, you might find it useful to look at this introduction to R tutorial to better understand lists, vectors, arrays, and dataframes, though you don't necessarily need to have completed the tutorial to follow this post!

# Useful functions

# A couple of useful functions

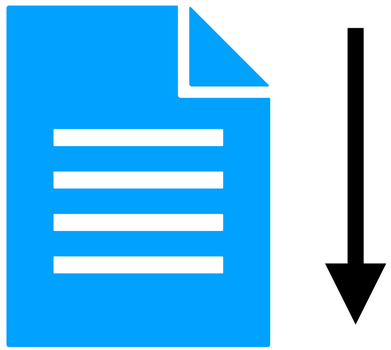
- `head()`
- `tail()`
- `summary()`
- `length()`
- `class()`
- `typeof()`
- `rowSums()`
- `colSums()`
- `rowMeans()`
- `colMeans()`
- `all()`
- `names()`
- `colnames()`
- `rownames()`
- `basename()`
- `c()`
- `data.frame()`
- `list()`
- `matrix()`
- `paste()`
- `paste0()`
- `rep()`
- `sample()`
- `cut()`
- `rm()`
- `dim()`
- `nrow()`
- `ncols()`
- `floor()`
- `ceiling()`
- `toupper()`
- `round()`
- `table()`
- `list.files()`
- `read.table()`
- `write.table()`
- `unique()`
- `duplicated()`
- etc…

**Don’t know how to do something?**
**In 95% of cases there is already a function for what you are trying to accomplish, Google it!**

# Good R coding practices

# 1. STRUCTURE YOUR SCRIPT

**A widely adopted top-to-bottom structure is:**



1. `#Script title, brief description, coder, date`
2. Load libraries
3. Set working directory (if needed)
4. Code

# 2. STRUCTURE YOUR CODE

- `# Use comments!`
- Enumerate bigger & smaller steps (whatever your favourite style)
- Indentation is not interpreted by R, use if for readability!

# Ex:

```r
#### CAGE TFs 2.0 - uORF analysis
#### Axel Thieffry - December 2020
set.seed(42)
library(tidyverse)
library(tidylog)
library(magrittr)
library(reshape2)
library(CAGEfightR)
library(GenomicRanges)

'%!in%' <- function(x,y)!('%in%'(x,y))
'select' <- dplyr::select
'rename' <- dplyr::rename
'count' <- dplyr::count
'l' <- length
'h' <- head

setwd('~/Dropbox/CAGE PTI 2.0/scripts')


# 0. RAD INPUT DATA ####
# ———————————————————
myseqinfo <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/myseqinfo.rds')
universe <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/universe_geneID.rds')$geneID
uorf_gr <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/uORFs_GR_7238.rds')

# 1. POTENTIAL FOR uORF EXCLUSION ####
# ———————————————————————————————————
# 1a. number of genes with an uORF
n_distinct(uorf_gr$geneID) # 3,433 genes

# 1b. length distribution of uORFs
uorf_gr %>%
  as.data.frame() %>%
  ggplot(aes(x=width)) +
        geom_density(fill='grey80') +
        scale_x_log10(expand=c(0, 0)) +
        cowplot::theme_cowplot() + theme(aspect.ratio=.35) +
        scale_y_continuous(expand=c(0, 0)) +
        labs(x='uORF length (bp)', title='uORF length distribution')
```

# Ex:

```
#### CAGE TFs 2.0 - uORF analysis
#### Axel Thieffry - December 2020
```

Title, description, author, date

```
set.seed(42)
library(tidyverse)
library(tidylog)
library(magrittr)
library(reshape2)
library(CAGEfightR)
library(GenomicRanges)

'%!in%' <- function(x,y)!('%in%'(x,y))
'select' <- dplyr::select
'rename' <- dplyr::rename
'count' <- dplyr::count
'l' <- length
'h' <- head
```

Libraries and bits of custom script-wide stuffs

```
setwd('~/Dropbox/CAGE PTI 2.0/scripts')
```

Working directory

```
# 0. RAD INPUT DATA ####
# ____________________
```

Big step

Code

```
myseqinfo <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/myseqinfo.rds')
universe <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/universe_geneID.rds')$geneID
uorf_gr <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/uORFs_GR_7238.rds')
```

```
# 1. POTENTIAL FOR uORF EXCLUSION ####
# ___________________________________
# 1a. number of genes with an uORF
```

Big step
small step

Code

```
n_distinct(uorf_gr$geneID) # 3,433 genes
```

```
# 1b. length distribution of uORFs
```

small step

Code

```
uorf_gr %>%
  as.data.frame() %>%
  ggplot(aes(x=width)) +
        geom_density(fill='grey80') +
        scale_x_log10(expand=c(0, 0)) +
        cowplot::theme_cowplot() + theme(aspect.ratio=.35) +
        scale_y_continuous(expand=c(0, 0)) +
        labs(x='uORF length (bp)', title='uORF length distribution')
```

# 3. BE CONSISTENT

- Use single ‘ or double “ quotes, but stick to one when possible

- Spaces around characters and operators: , = ≠ + ⩽
  Like them or hate them, but be consistent.

- <u>Widely adopted styles for best readability:</u>
  Space after a comma: `c(17, 4, 88)`
  Spaces around assignment: `foo ← 32`
  Spaces around operators: `sum(a) / 3.14`
  No spaces for function arguments: `read.table('data.txt', header=T)`

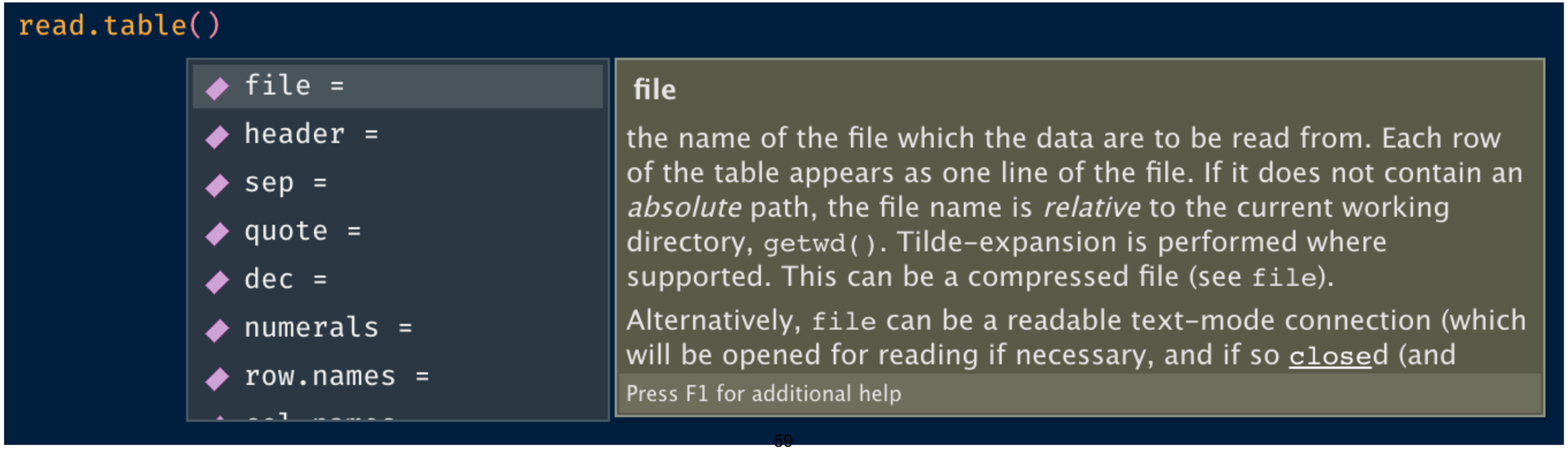
# 4. TABULATION

- Let R help you, use TAB ➡️ !

  - autocomplete **variable names** (known to your Global env.)

  - autocomplete **function names** (known to your Global env.)

  - discover and autocomplete **function parameters**



# Congratulations 🎖️

# You now know more R than 75% of Scientists.

# Let’s play with some data

## The iris dataset

- `# load the data (they are included in base R)`
  `data(iris)`

- Inspect the iris dataset, try:
  `iris`, `type()`, `str()`, `dim()`, `head()`, `tail()`, `summary()`, `colnames()`, `rownames()`, `View()`

- Coerce to a matrix:
  `as.matrix(iris)`

- Get the “Petal.Width” column:
  - still in a dataframe
  - as a simple vector
  *tips: remember the simple and double brackets!*

- For “Petal.Width” column, calculate:
  `sum()`, `mean()`, `max()`, and `min()`

# Datacamp assignment

## Introduction to R

- Complete the Datacamp course *“Introduction to R”* that has been assigned to you
- Deadline: Friday 28th, no later than 18:00
- You can use any resource available to you (slides, Google, etc)
  **Note:** nice base R cheatsheet available here