

# 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**

TRUE, FALSE, T, F

- **factors**: ordered characters (more on that later)

Unsure?

=> Use the functions **type()** and **class()**

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()
```

# 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:

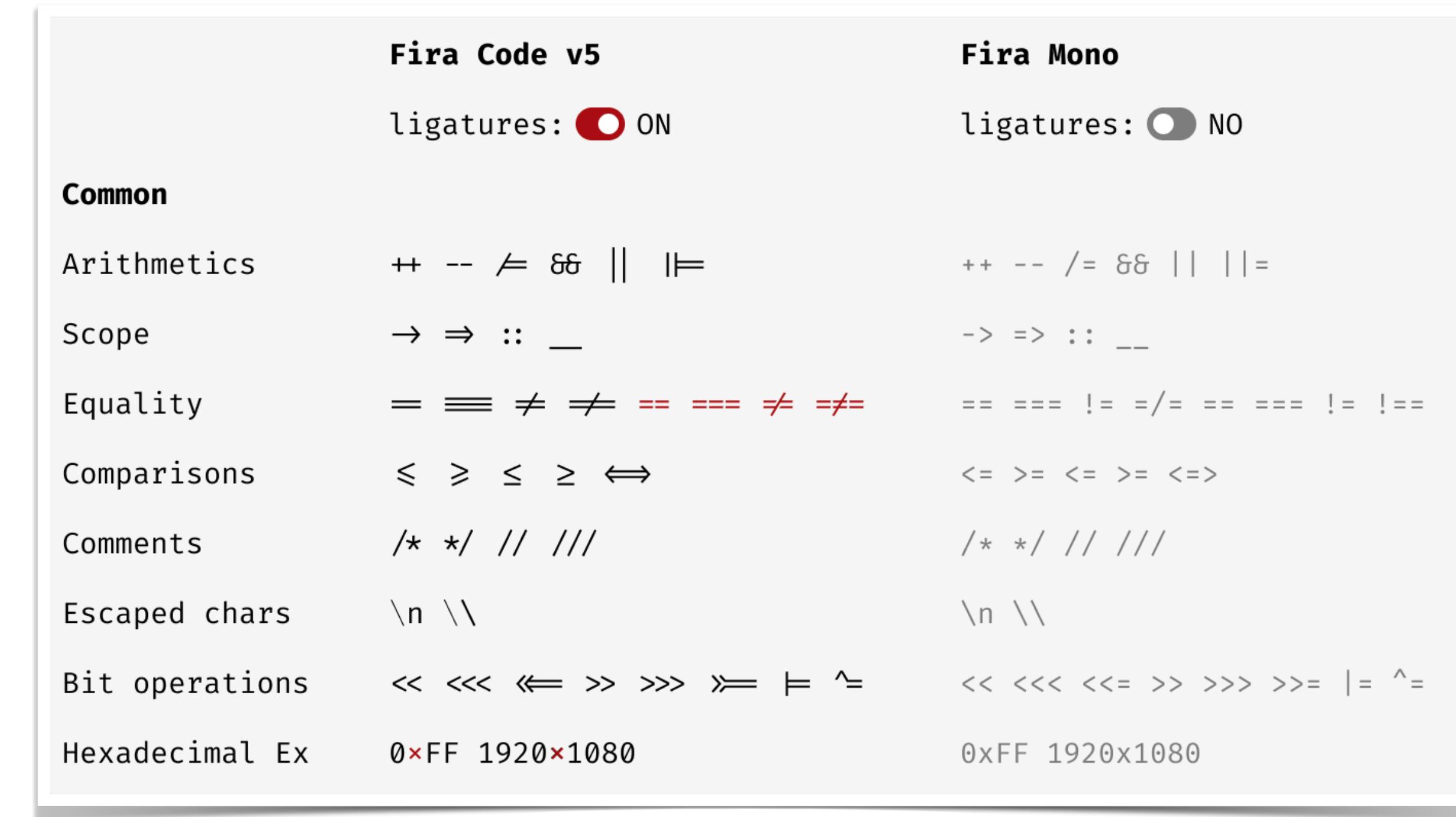
a <- 3

- **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/tomsky/FiraCode](https://github.com/tomsky/FiraCode)



- Get a nice theme: **RStudio > Preferences > Appearance > Editor theme**  
My favourite is 'Cobalt

# 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
- Choose 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 change 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, 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  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
```

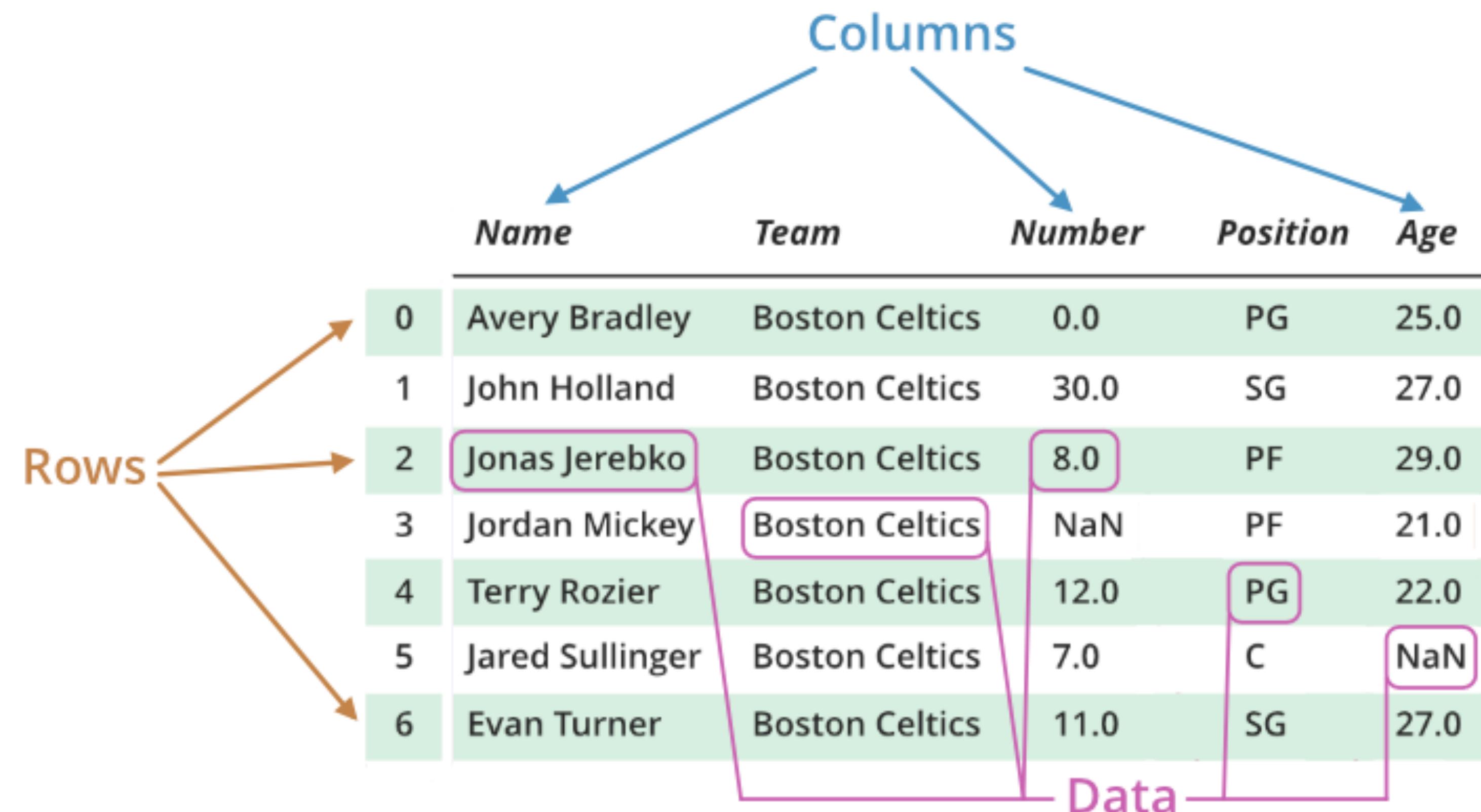
Notice the ‘byrow’

```
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 !

# Data frame

- 2D object with heterogeneous data types



# 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()`

```
1 mydf <- data.frame('letters'=c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'),  
2   |   |   |   |   'numbers'=1:10)  
3 rownames(mydf) <- c('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J')  
4 dim(mydf)  
5 colnames(mydf)  
6 rownames(mydf)  
7 ncol(mydf)  
8 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

```
1 wt_counts <- c(5, 12, 3, 0, 14, 13)
2 mut_counts <- c(5, 14, 3, 20, 7, 10)
3 gene_names <- c('P53', 'HSF2', 'CDF4', 'UGA4', 'DAL5', 'FLS2')
4 samples_names <- c('wt', 'mut')
5
6 names(wt_counts) <- names(mut_counts) <- gene_names
7 wt_counts ; mut_counts
8
9 list(wt_counts, mut_counts)
10 setNames(list(wt_counts, mut_counts), samples_names)
11 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

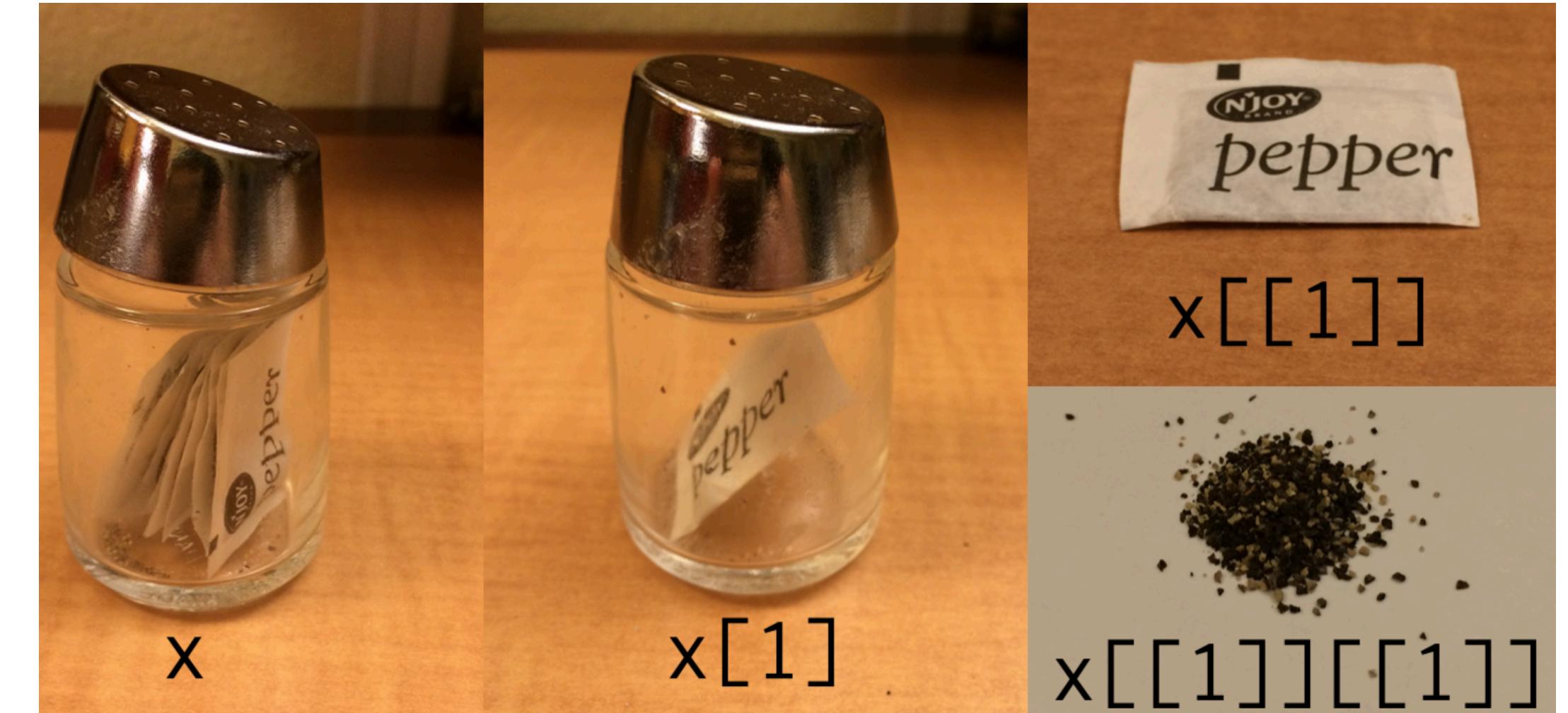
# Slicing vectors

```
1  vec ← c(1:10)                                vec ← 1:10 would actually do the trick
2  vec[-1]
3  vec[-1:-3]
4  vec[5]
5
6  named_vec ← c('first'=1, 'second'=2, 'third'=3)
7  named_vec['second']
```

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'

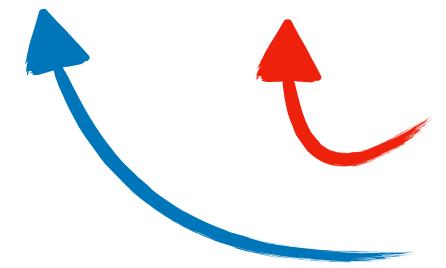


```
1 test <- list(c('a', 'b', 'c'),
2                   1:3,
3                   paste0(1:3, c('a', 'b', 'c')))
4
5 test[3]
6 test[[1]]
7
8 names(test) <- c('letters', 'numbers', 'both')
9 test$numbers
10 test['numbers']
```

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!

```
1 iris[iris$Sepal.Length > 7, ]  
2 subset(iris, Sepal.Length > 7)  
3  
4 iris$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(uORF_list, 'uORF_list_21102021.rds')`  
`uORFs ← readRDS('path/file.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

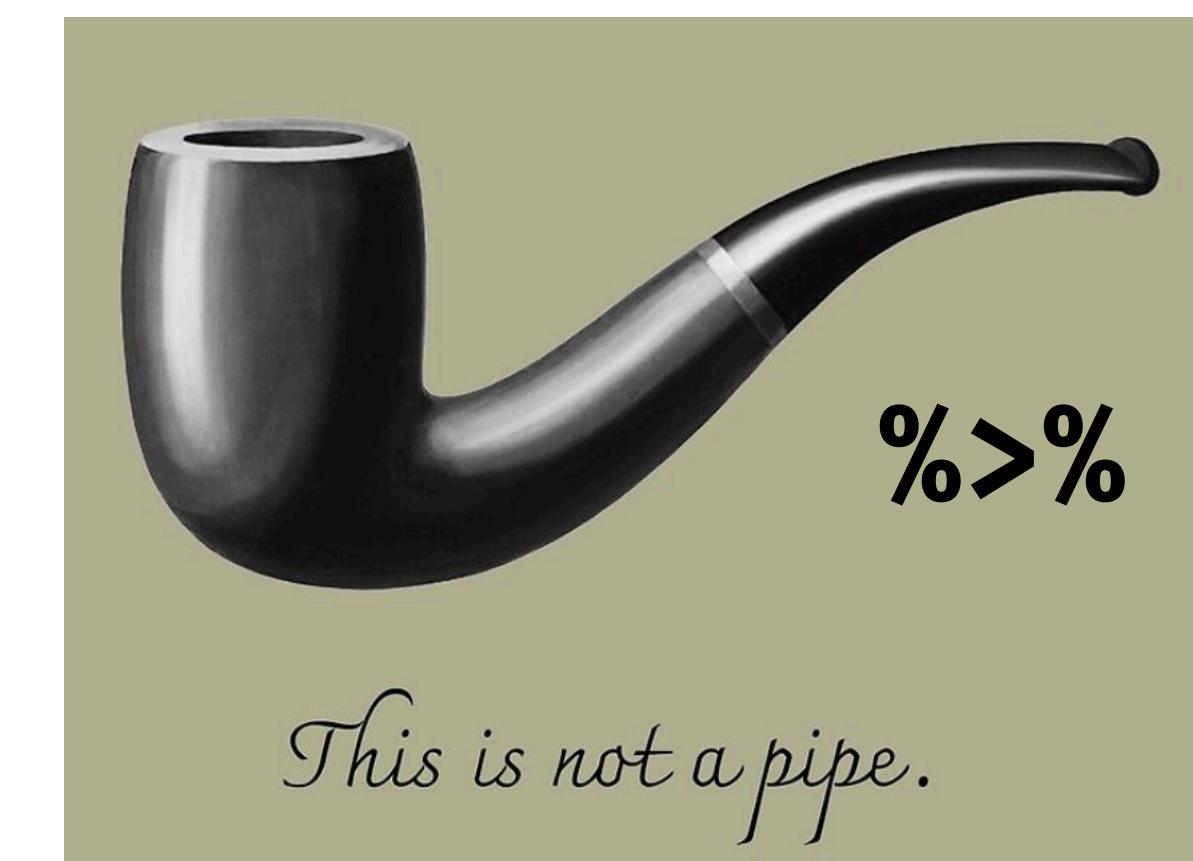
- Typical syntax:

```
function(argument=value, ...)
```

- They can be nested, but reading rapidly becomes difficult

```
length(unique(round(mydataframe$Sepal.length)))
```

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



# Getter and Setters

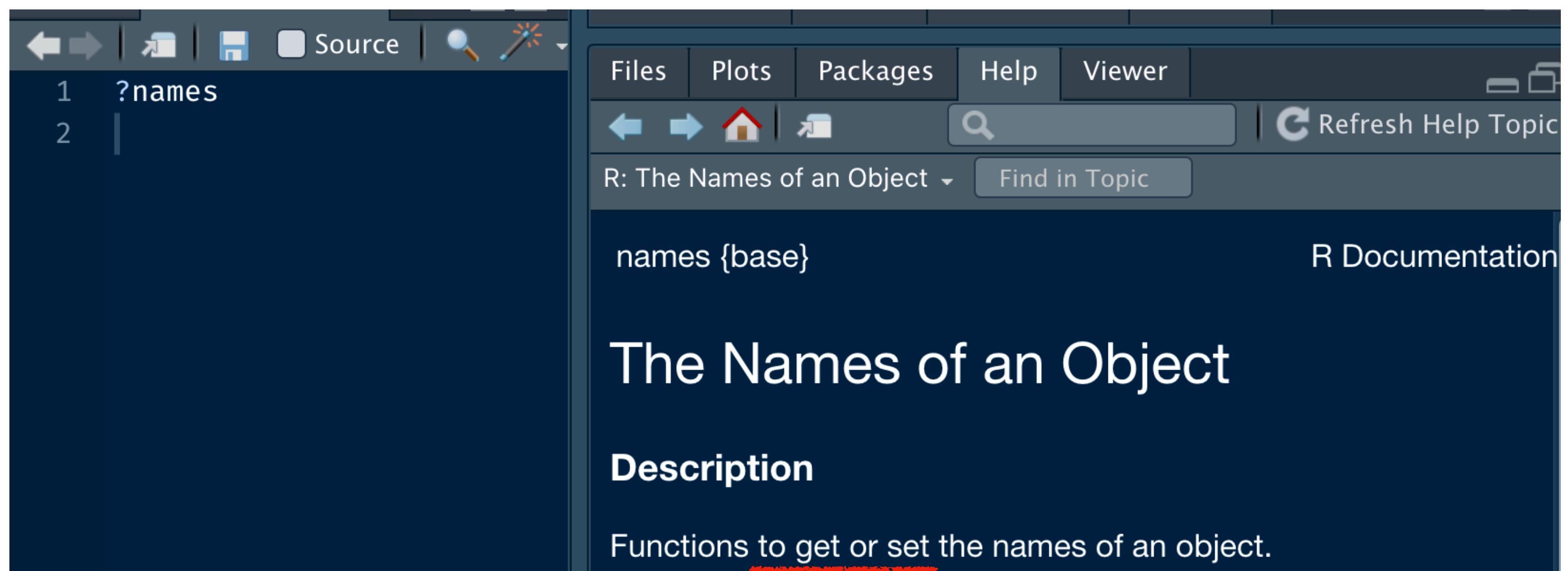
- Some functions can be used both to **GET** info or to **SET** data
- Examples: `names()`, `colnames()`, `rownames()`

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

Try it !

# Get help

- type the name of the function followed 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 <- <your code here>  
  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')
```

- Loading a library is accomplished by:

```
library(library_name)
```

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

# 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 —
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()   masks stats::lag()
> library(reshape2)

Attaching package: 'reshape2'

The following object is masked from 'package:tidyverse':

  smiths
```

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

# Apply family

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

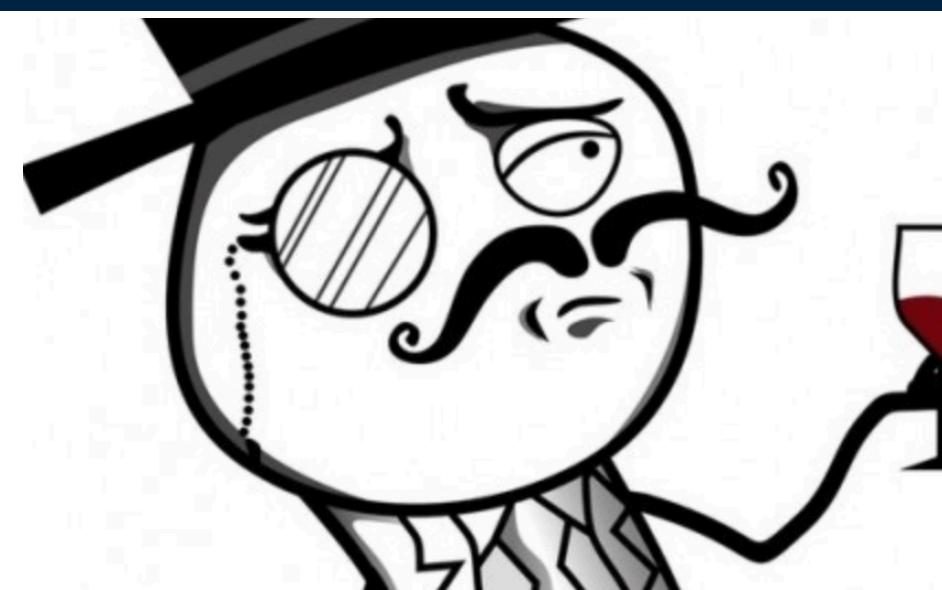
This is because most of R's functions are vectorised:  
the function will operate on all elements of a vector without needing to loop  
through and act on each element one at a time.

# Vectorisation example:

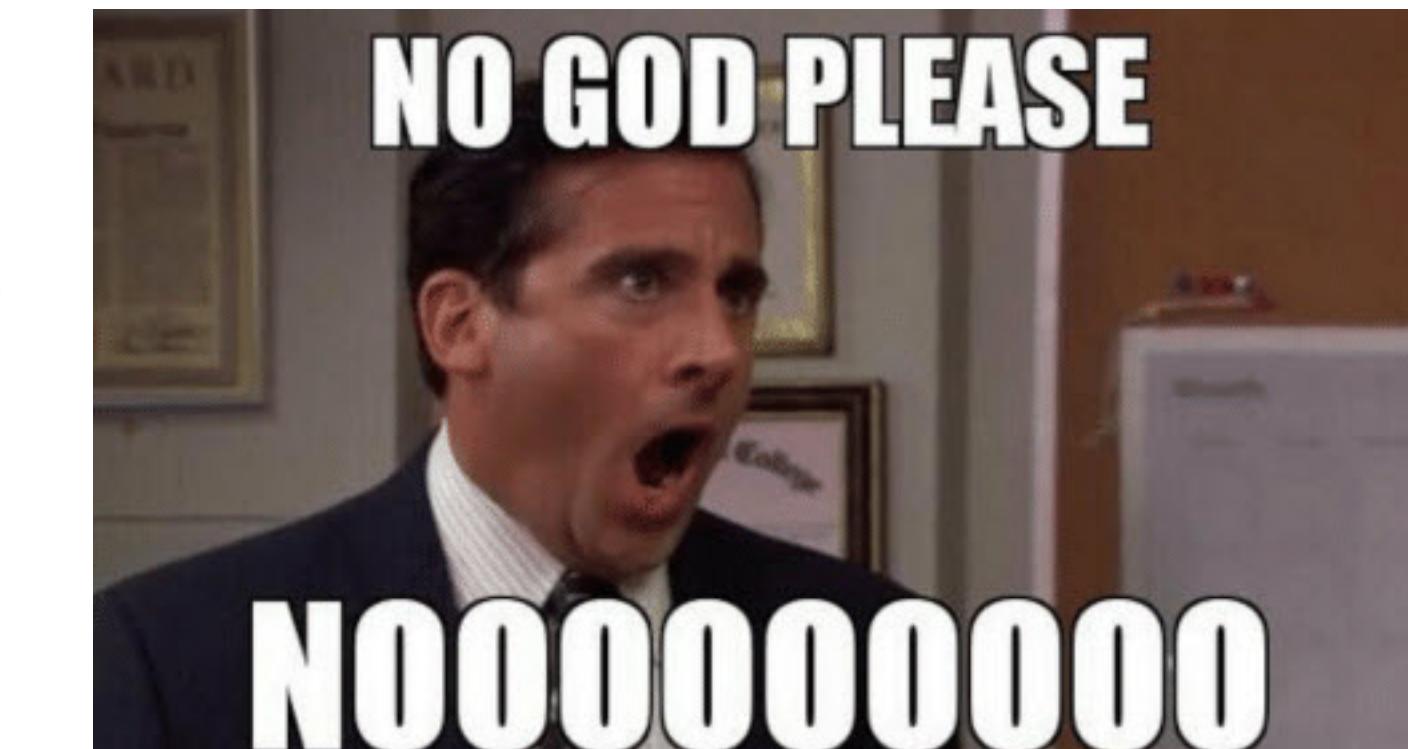
```
> nums ← 1:5  
> nums  
[1] 1 2 3 4 5
```

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

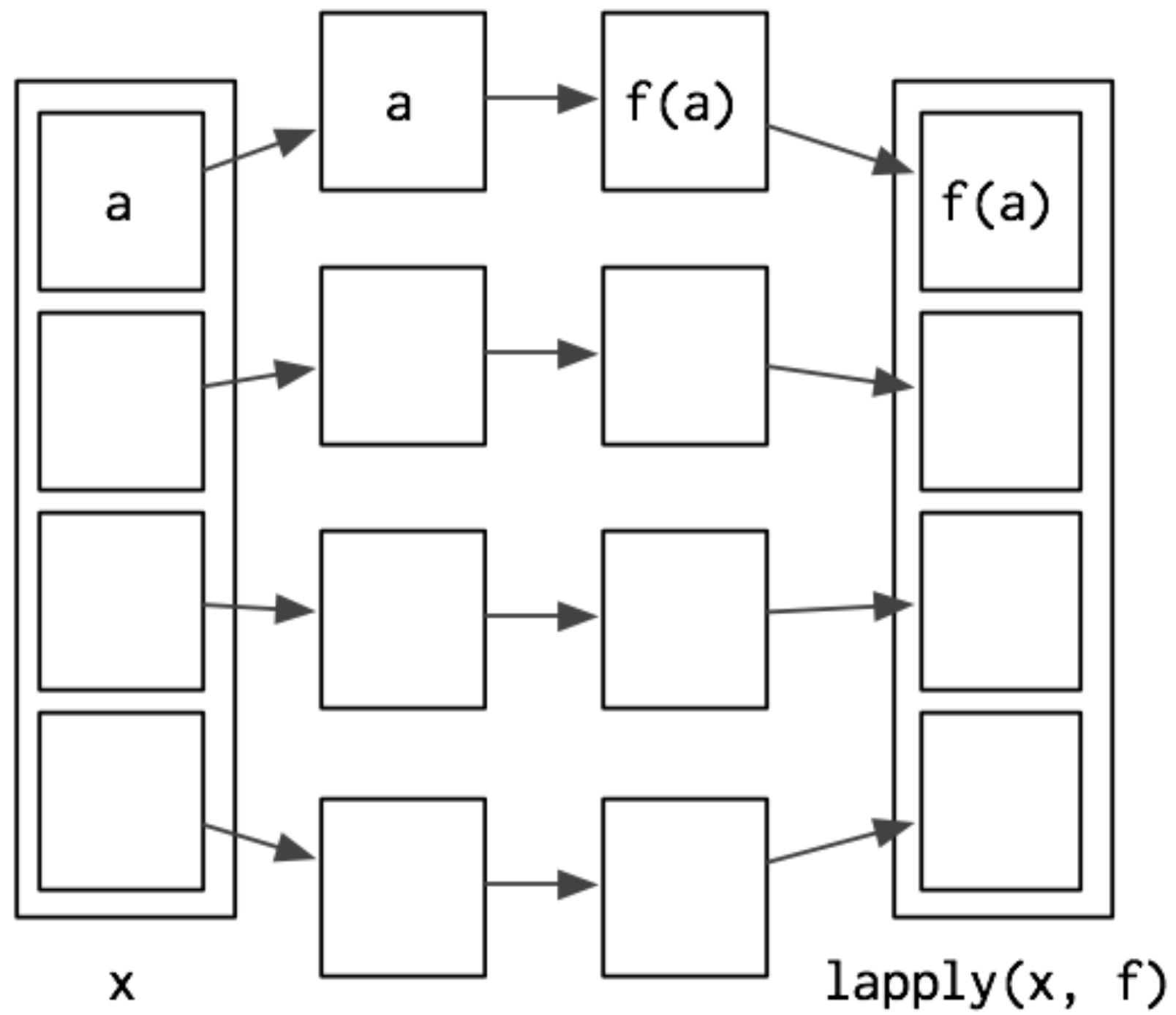
X	apply(X ,2, sum)
	<b>Dimension — 2</b> →
1	-1.7189391 -1.0863995 -2.2542126 -1.3201873 1.9874737 0.6265486 0.2140376 0.8850445 0.9637687 1.3191502
	1.0996117 -2.0533779 -0.3684977 1.4782993 0.8000988
	-0.55559727 -0.1792310 -0.8088577 1.29055209 0.3264156 0.5412132 1.40028967 -0.7574303 -2.3241569 -1.28177703 -0.5015628 1.1537703 0.09345943 1.4535431 1.0935720
Result	sum
	-0.8078717 0.4241565 0.9561342 0.9469269 0.3417346 -0.3444591

**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 simplified 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, 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

```
1 iris_dfl <- split(iris, iris$Species)
2 filenames <- paste0(names(iris_dfl), '.csv')
3
4 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...

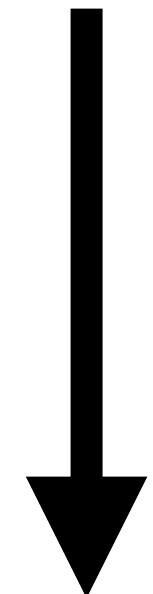
Don't know how to do something?

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:

```
1 ##### CAGE TFs 2.0 - uORF analysis
2 ##### Axel Thieffry - December 2020
3 set.seed(42)
4 library(tidyverse)
5 library(tidylog)
6 library(magrittr)
7 library(reshape2)
8 library(CAGEfightR)
9 library(GenomicRanges)
10
11 '%!in%' <- function(x,y)!('%in%'(x,y))
12 'select' <- dplyr::select
13 'rename' <- dplyr::rename
14 'count' <- dplyr::count
15 'l' <- length
16 'h' <- head
17
18 setwd('~/Dropbox/CAGE PTI 2.0/scripts')
19
20
21 # 0. RAD INPUT DATA #####
22 #
23 myseqinfo <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/myseqinfo.rds')
24 universe <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/universe_geneID.rds')$geneID
25 uorf_gr <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/uORFs_GR_7238.rds')
26
27 # 1. POTENTIAL FOR uORF EXCLUSION #####
28 #
29 # 1a. number of genes with an uORF
30 n_distinct(uorf_gr$geneID) # 3,433 genes
31
32 # 1b. length distribution of uORFs
33 uorf_gr %>%
34   as.data.frame() %>%
35   ggplot(aes(x=width)) +
36     geom_density(fill='grey80') +
37     scale_x_log10(expand=c(0, 0)) +
38     cowplot::theme_cowplot() + theme(aspect.ratio=.35) +
39     scale_y_continuous(expand=c(0, 0)) +
40     labs(x='uORF length (bp)', titleuORF length distribution')
```

# Ex:

```
1 ##### CAGE TFs 2.0 - uORF analysis
2 ##### Axel Thieffry - December 2020
3 set.seed(42)
4 library(tidyverse)
5 library(tidylog)
6 library(magrittr)
7 library(reshape2)
8 library(CAGEfightR)
9 library(GenomicRanges)
10
11 '%!in%' <- function(x,y)!('%in%')(x,y))
12 'select' <- dplyr::select
13 'rename' <- dplyr::rename
14 'count' <- dplyr::count
15 'l' <- length
16 'h' <- head
17
18 setwd('~/Dropbox/CAGE PTI 2.0/scripts')
19
20
21 # 0. RAD INPUT DATA #####
22 #
23 myseqinfo <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/myseqinfo.rds')
24 universe <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/universe_geneID.rds')$geneID
25 uorf_gr <- readRDS('~/Dropbox/CAGE PTI 2.0/data/RDS files/uORFs_GR_7238.rds')
26
27 # 1. POTENTIAL FOR uORF EXCLUSION #####
28 #
29 # 1a. number of genes with an uORF
30 n_distinct(uorf_gr$geneID) # 3,433 genes
31
32 # 1b. length distribution of uORFs
33 uorf_gr %>%
34   as.data.frame() %>%
35   ggplot(aes(x=width)) +
36     geom_density(fill='grey80') +
37     scale_x_log10(expand=c(0, 0)) +
38     cowplot::theme_cowplot() + theme(aspect.ratio=.35) +
39     scale_y_continuous(expand=c(0, 0)) +
40     labs(x='uORF length (bp)', titleuORF length distribution')
```

**Title, description, author, date**

**Libraries and bits of custom script-wide stuffs**

**Working directory**

**Big step**      **Code**

**Big step small step**      **Code**

**small step**

**Code**

# 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.
- Widely adopted styles for best readability:

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**

```
read.table()  
  
◆ file =  
◆ header =  
◆ sep =  
◆ quote =  
◆ dec =  
◆ numerals =  
◆ row.names =  
◆ ...  
  
file  
the name of the file which the data are to be read from. Each row  
of the table appears as one line of the file. If it does not contain an  
absolute path, the file name is relative to the current working  
directory, getwd(). Tilde-expansion is performed where  
supported. This can be a compressed file (see file).  
Alternatively, file can be a readable text-mode connection (which  
will be opened for reading if necessary, and if so closed (and  
Press F1 for additional help
```

Congratulations



You now know more R than 75% of  
Scientists.