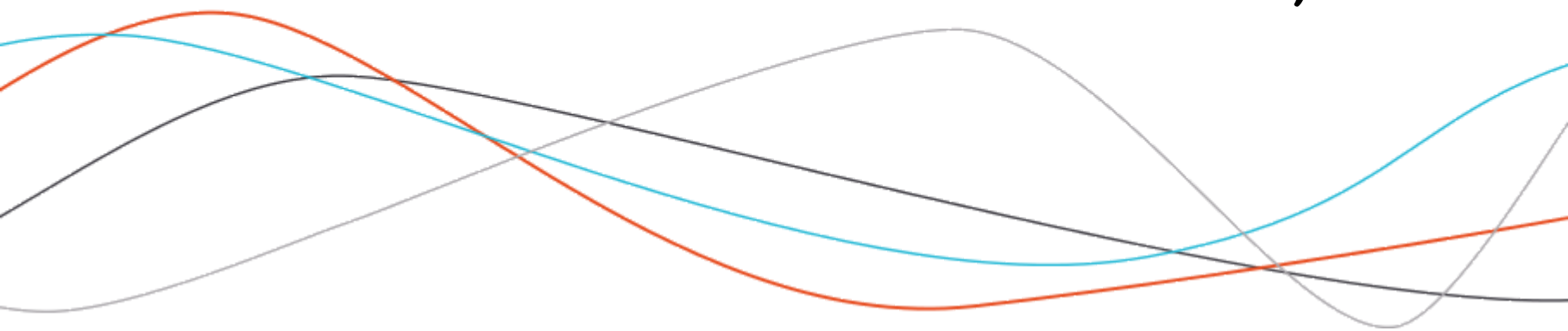
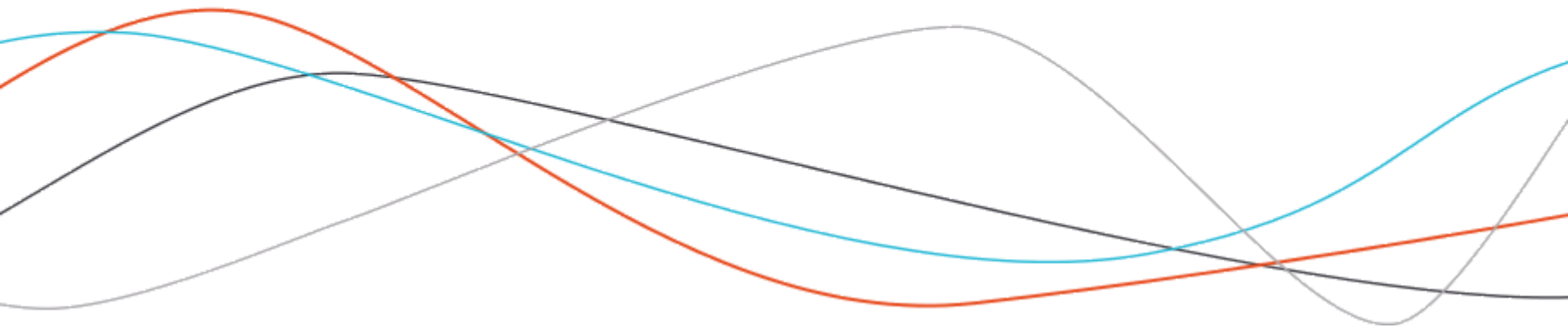
R engineering

November 12, 2018



First steps in programming with R

A dive into automation



if: conditional expression

```
if (condition) {
  # what you want

} else {
  # what you want
  }
```

For example :

```
a <- 2
if (a == 2) {
  print("The number is 2")
} else {
  print("The number is not 2")
}
```

```
#> [1] "The number is 2"
```

if: conditional expression

Notes:

- The condition are a logical: you can use the `&`, `|`, `!`... operators.
- `condition_*` **must** contain a unique value: otherwise, only the first element of the vector will be used.
- `else` is optional.
- you can use `else if` to imbricate conditions.

```
if (condition_1){
  # what you wan
} else if (condition_2) {
  # what you wan
} else {
  # what you wan
}
```

Question

What is the output of this code?

```
day <- Sys.Date()
if (day == "2017-10-04") {
  print("We are October the 4th")
} else {
  print("We are not October the 4th")
}
```

Question

What is the output of this code?

```
a <- 12
b <- 3*5
if (a > b) {
  print(a)
} else {
  print(b)
}
```

for loop

Repeat execution

```
for (variable in sequence) {
        # Some code using variable
   }
```

Notes:

- `variable` is the incremented variable.
- `sequence` is a vector containing all the values that will be taken by `variable`. Its length defines the number of steps of the for loop.
- The `for` loop is quite slow comparing to the functions of the `apply` family, even more if you put a for loop inside a for loop inside a for loop...

for loop

```
for (i in c("a", "b", "c")) {
        print(i)
        }
```

```
#> [1] "a"
#> [1] "b"
#> [1] "c"
```

- The variable is `i`
- The sequence is `a, b, c`
- The expression is `print(i)`

Functions

```
function_name <- function(arg1, arg2, ...){
  # what you wan
        result <- expression(arg1, arg2, ...)

        return(result)
}
```

Notes :

- `return` might be omitted : in that case, the output of last evaluated expression will be returned.
- But it's better to use it, just for clarity's sake.
- You can write your own function in your script, or source it from an external file.

About functions

Properties

- The arguments can be passed along unordered, as long as they are named.
- We can set a default value for any argument when defining the function: `arg = 10`.
- A function can only return a single object. Multiple output should be put into a list.

Notes

- Default values make the function clearer, and the user has to provide less arguments.
- If your function returns a list, the names of the elements of this list are kept outside the function.

About functions

Variables which are created inside a function stay inside the function.

In other words, "What happens in a function stays in a function. "

```
a <- 5
plus_one <- function(x) {
  a <- "XXXXXX"
  return(x + 1)
}

plus_one(10)
a
```

About variables

In a function, you can use variables that exist outside the function.

```
rm(list = ls(all = TRUE))# empty the environment
a <- 5
plus_a <- function(x) {
    return(x + a)
}

plus_a(10)
a <- 14
plus_a(10)
```

But in practice, this kind of behavior is dangerous:

```
rm(list = ls(all = TRUE))
plus_a <- function(x){return(x+a)}
plus_a(10) # Won't work any more
```

Vectorize functions

`apply` and friends (`sapply`, `lapply`, `tapply`) are used to save you A LOT of time.

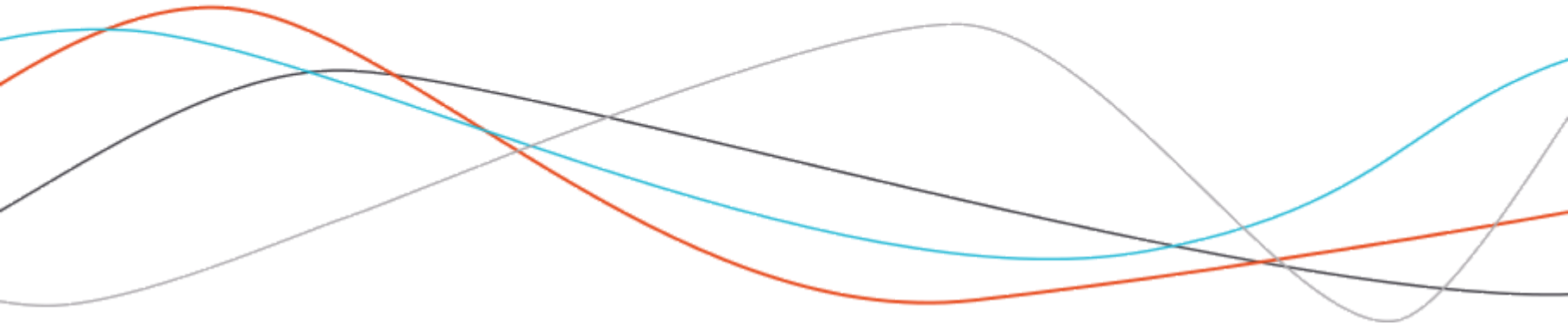
This function applies simultaneously the same transformation to all the rows / columns in a dataset.

`apply` and al. are really efficient on large dataset. This is how to use it:

```
apply(dataset, MARGIN, FUN)

# MARGIN = 1 for row, 2 for col
# FUN = the name of the function to apply

data(iris)
for (i in 1:4){print(mean(iris[,i]))}
apply(iris[,1:4], MARGIN = 2, FUN = mean)
```

tidy eval

Programming & Evaluation



Programming in the tidyverse

Programming in the tidyverse

=> use {dplyr} / {tidyr} / {ggplot2} functions

=> make functions that behave like tidyverse functions

Programming with {dplyr}

You might have tried to do this:

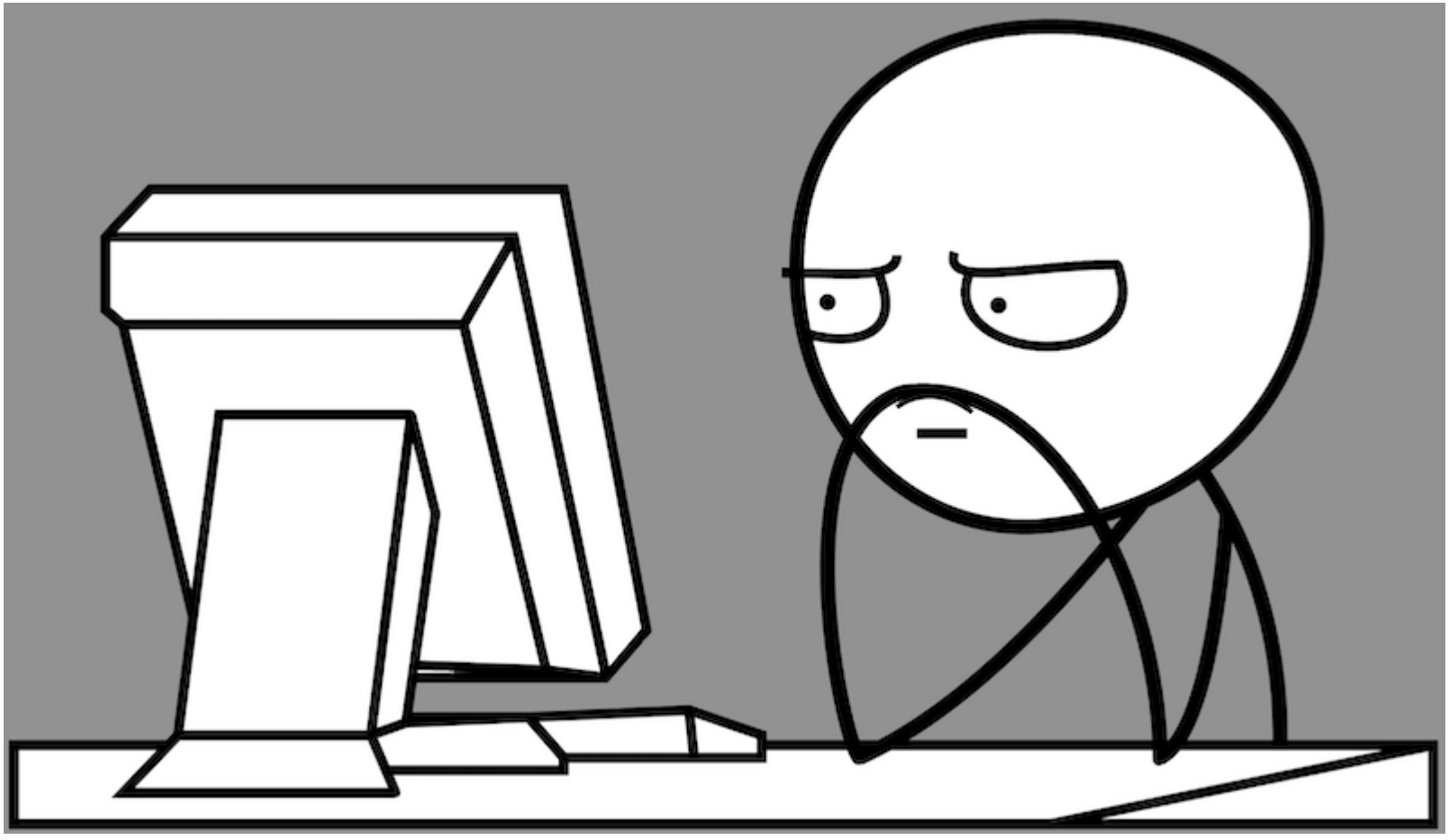
```
library(dplyr)
uber_select <- function(tbl, col){
  select(tbl, col)
}
```

But unfortunately:

```
uber_select(mtcars, carb)
```

```
#> Error in .f(.x[[i]], ...): object 'carb' not found
```

Whyyyyyyyyyyyy?



Because of tidyevaluation

{dplyr} uses a framework called **tidyevaluation**.

You'll find this framework in:

- dplyr
- tidyr
- ggplot2
- ...

tidy what?

When using {dplyr}, you might have noticed that you can refer to columns directly, without `"` or `$`, contrary to what is done in R base.

```
# In R base
mtcars[ mtcars$carb == 2 & mtcars$cyl == 8, ]
```

VS

```
mtcars %>% filter(carb == 2 & cyl == 8)
```

Yet :

```
carb
```

```
#> Error in eval(expr, envir, enclos): object 'carb' not found
```

=> **tidyevaluation** is a form of **non standard evaluation**.

Non standard what?

Non standard evaluation is opposed to... standard evaluation.

Standard evaluation is the process of **looking for the value attached to a symbol**.

```
plop <- 12
plop
```

```
#> [1] 12
```

When run in the console, these two lines:

- Binds the **value** 12 to the **symbol** `plop`.
- **Evaluate** `plop`, i.e look in **memory** for the value binded to the symbol.

NSE allows to bypass these steps.

Now, a little quizz.

What's the difference between these two?

```
library("dplyr")
```

&

```
library(dplyr)
```

Now, a little quizz.

What's the difference between these two?

```
library("dplyr")
```

&

```
library(dplyr)
```

What does this code do?

```
dplyr <- "data.table"
library(dplyr)
```

Now, a little quizz.

What's the difference between these two?

```
library("dplyr")
```

&

```
library(dplyr)
```

What does this code do?

```
dplyr <- "data.table"
library(dplyr)
```

And this one?

```
y <- "data.table"
library(y)
```

NSE

Non standard evaluation in R:

- R allows non standard evaluation: defining your own rules for evaluating a symbol.
- A strength and oddity: pretty rare for a programming language.
- Allows to build DSL (domain specific language) like {dplyr} or {data.table}

> In standard evaluation, **a symbol has to be 'referentially transparent'** => in other words, it should be clear what the symbol is refering to.

About scoping in R

Everything in R happens in an environment, and when R evaluates a symbol, it looks for its value inside this symbol environment. Each function has its own environment, just as lists (and remember data.frames are lists).

```
create_a <- function() { result <- 12 }
create_a()
result
```

```
#> Error in eval(expr, envir, enclos): object 'result' not found
```

```
e <- new.env()
e$pouet <- 12
pouet
```

```
#> <environment: 0x1034fbce0>
```

```
e$pouet
```

```
#> [1] 12
```

About scoping in R

Each environment can have its own evaluation rules. That's what happens with tidyevaluation: the functions **"capture"** the symbol typed by the user **before it is evaluated**, and evaluate this symbol in a non standard way.

```
data("starwars")
starwars %>%
  filter(species == "Human")
```

```
#> # A tibble: 35 x 13
#>    name  height  mass hair_color skin_color eye_color birth_year gender
#>    <chr>  <int> <dbl> <chr>      <chr>      <chr>           <dbl> <chr>
#>  1 Luke…    172    77 blond      fair       blue             19   male
#>  2 Dart…    202   136 none       white      yellow           41.9 male
#>  3 Leia…    150    49 brown      light      brown            19   female
#>  4 Owen…    178   120 brown, gr… light      blue             52   male
#>  5 Beru…    165    75 brown      light      blue             47   female
#>  6 Bigg…    183    84 black      light      brown            24   male
#>  7 Obi-…    182    77 auburn, w… fair       blue-gray        57   male
#>  8 Anak…    188    84 blond      fair       blue             41.9 male
#>  9 Wilh…    180    NA auburn, g… fair       blue             64   male
#> 10 Han …    180    80 brown      fair       brown            29   male
```

About scoping in R

You can evaluate symbol in various environments:

```
pouet <- new.env()
pouet$pouet <- 12

plop <- new.env()
plop$pouet <- "hey!"

eval( quote(pouet), envir = plop )
```

```
#> [1] "hey!"
```

```
eval( quote(pouet), envir = pouet )
```

```
#> [1] 12
```

About lazy evaluation

R has a feature called lazy evaluation, stating that **no symbol is evaluated until it is actually needed**.

```
plop <- function(a, b){
  a * 10
}
plop(4)
```

```
#> [1] 40
```

```
plop(a = 4, b = nothing)
```

```
#> [1] 40
```

```
plop(a = 4, b = stop())
```

```
#> [1] 40
```

```
ping <- function(a = Sys.time(),
                 b = Sys.time(),
                 c = Sys.time()){
  print(a)
  Sys.sleep(1)
  print(b)
  Sys.sleep(1)
  print(c)
}
ping()
```

```
#> [1] "2018-11-11 19:08:41 CET"
#> [1] "2018-11-11 19:08:42 CET"
#> [1] "2018-11-11 19:08:43 CET"
```

Lazy evaluation & NSE

Lazy evaluation is what makes NSE possible:

- When entering a function, the symbol is not yet evaluated
- We can **capture this symbol**
- We can **make custom evaluation**

Tidy evaluation

- Capture with `enquo()` and `quos()`
- Define custom evaluation in function with `!!` & `!!!`
    - We are telling the functions that we have already taken care of the symbol capture

Why is this function not working?

And do you now understand the error message?

```
starwars %>%
  group_by(gender) %>%
  arrange(height) %>%
  top_n(1, height) %>%
  select(gender, height)
```

```
#> # A tibble: 5 x 2
#> # Groups:   gender [5]
#>   gender        height
#>   <chr>          <int>
#> 1 <NA>             167
#> 2 hermaphrodite    175
#> 3 none             200
#> 4 female           213
#> 5 male             264
```

```
top_one_by_group <- function(tbl,
grp, var) {
tbl %>%
  group_by(grp) %>%
  arrange(var) %>%
  top_n(1, var) %>%
  select(grp, var)
}
top_one_by_group(starwars,
species, height)
```

```
#> Error in grouped_df_impl(data,
unname(vars), drop): Column `grp`
is unknown
```

Make it work

Capture the symbols

```
library(rlang)
top_one_by_group <- function(tbl, grp, var) {
   grp <- enquo(var)
   var <- enquo(var)
  tbl %>%
    group_by(grp) %>%
    arrange(var) %>%
    top_n(1, var) %>%
    select(grp, var)
}
top_one_by_group(starwars, species, height)
```

```
#> Error in grouped_df_impl(data, unname(vars), drop): Column `grp` is
unknown
```

Make it work

Custom evaluation

```
library(rlang)
top_one_by_group <- function(tbl, grp, var) {
  grp <- enquo(grp)
  var <- enquo(var)
  tbl %>%
    group_by(!!grp) %>%
    arrange(!!var) %>%
    top_n(1, !!var) %>%
    select(!!grp, !!var)
}
top_one_by_group(starwars, species, height)
```

```
#> # A tibble: 39 x 2
#> # Groups:   species [38]
#>    species        height
#>    <chr>           <int>
#>  1 Yoda's species     66
#>  2 Aleena             79
#>  3 Ewok               88
```

Example 1

`sqrt_df`, a function that compute the square root of a given df.

```
sqrt_df <- function(tbl, col){
  col <- enquo(col)
  tbl %>%
    mutate(sqrt = sqrt(!!col))
}

sqrt_df(mtcars, disp)
```

```
#>     mpg cyl  disp  hp drat    wt  qsec vs am gear carb      sqrt
#> 1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4 12.649111
#> 2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4 12.649111
#> 3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1 10.392305
#> 4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1 16.062378
#> 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2 18.973666
#> 6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1 15.000000
#> 7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4 18.973666
#> 8  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2 12.111978
#> 9  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2 11.865918
#> 10 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4 12.946042
#> 11 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4 12.946042
```

Another NSE : select and change column names

- *We can change column name when selecting with* `select`

```
my_select <- function(tbl, grp, var) {
  grp <- enquo(grp)
  var <- enquo(var)
  tbl %>%
    select( group = !!grp, var = !!var)
}
my_select(starwars, species, height)
```

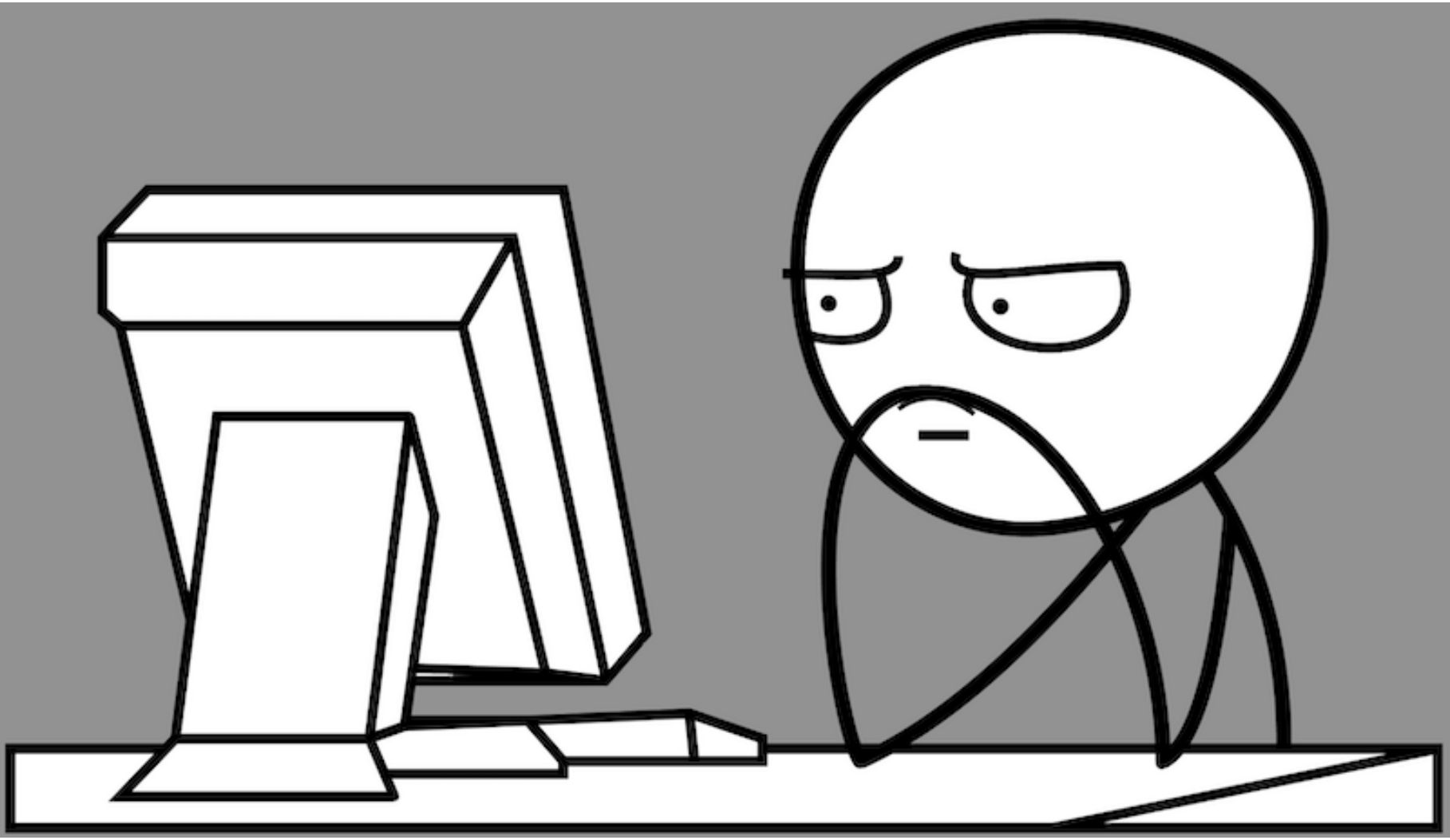
```
#> # A tibble: 87 x 2
#>    group   var
#>    <chr> <int>
#>  1 Human   172
#>  2 Droid   167
#>  3 Droid    96
#>  4 Human   202
#>  5 Human   150
#>  6 Human   178
#>  7 Human   165
#>  8 Droid    97
#>  9 Human   183
```

Another NSE : select and change column names

Naive approach

```
my_select_new <- function(tbl, grp, var, name_grp, name_var) {
  grp <- enquo(grp)
  var <- enquo(var)
  name_grp <- enquo(name_grp)
  name_var <- enquo(name_var)
  tbl %>%
    select( !!name_grp = !!grp, !!name_var = !!var)
}
```

```
Error: unexpected '=' in:
"  tbl %>%
    select( !! group ="
```



Another NSE : select and change column names

Operator `:=` is **an assignment** operator

```
my_select_new <- function(tbl, grp, var, name_grp, name_var) {
  grp <- enquo(grp)
  var <- enquo(var)
  name_grp <- enquo(name_grp)
  name_var <- enquo(name_var)
  tbl %>%
     select( !!name_grp := !!grp, !!name_var := !!var)
}
my_select_new(starwars, species, height, new_grp, new_var)
```

```
#> # A tibble: 87 x 2
#>    new_grp new_var
#>    <chr>     <int>
#>  1 Human       172
#>  2 Droid       167
#>  3 Droid        96
#>  4 Human       202
#>  5 Human       150
#>  6 Human       178
```

Another NSE : select and change column names

- Before `{rlang}` version 0.3.0, symbol side of an assignment should either be a symbol or a character vector
- `!! group` is not a symbol nor a character vector, but a **quosure**.
- We need to turn `grp` into a character or a symbol

Another NSE : select and change column names

- Before `{rlang}` version 0.3.0, `quo_name` was necessary

```
my_select_new <- function(tbl, grp, var, name_grp, name_var) {
  grp <- enquo(grp)
  var <- enquo(var)
  name_grp <- enquo(name_grp)
  name_var <- enquo(name_var)
  tbl %>%
    select( !!quo_name(name_grp) := !!grp, !!quo_name(name_var) := !!var)
}
my_select_new(starwars, species, height, new_grp, new_var)
```

```
#> # A tibble: 87 x 2
#>    new_grp new_var
#>    <chr>     <int>
#>  1 Human       172
#>  2 Droid       167
#>  3 Droid        96
#>  4 Human       202
#>  5 Human       150
#>  6 Human       178
```

Example 2

`sqrt_df` with custom column name:

```
sqrt_df <- function(tbl, col, new_name){
  col <- enquo(col)
  new_name <- enquo(new_name)
  tbl %>%
    mutate(!!new_name := sqrt(!!col))
}

sqrt_df(mtcars, disp, new_column)
```

```
#>     mpg cyl  disp  hp drat    wt  qsec vs am gear carb new_column
#> 1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4  12.649111
#> 2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4  12.649111
#> 3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1  10.392305
#> 4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1  16.062378
#> 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2  18.973666
#> 6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1  15.000000
#> 7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4  18.973666
#> 8  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2  12.111978
#> 9  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2  11.865918
```

Handling multiple input

How to pass quosures in `...`?

`quos` & `!!!`

```
overview <- function(df, ...){
  to_select <- quos(...)
  df %>%
     select(!!!to_select) %>%
    head()
}
overview(mtcars, carb, cyl)
```

```
#>                   carb cyl
#> Mazda RX4            4   6
#> Mazda RX4 Wag        4   6
#> Datsun 710           1   4
#> Hornet 4 Drive       1   6
#> Hornet Sportabout    2   8
#> Valiant              1   6
```

Example 3

`sqrt_df` on various columns:

```
sqrt_df <- function(tbl, ...){
  cols <- quos(...)
  tbl %>%
    mutate_at( vars(!!!cols), sqrt )
}

sqrt_df(mtcars, disp, drat)
```

```
#> Warning: `is_lang()` is soft-deprecated as of rlang 0.2.0.
#> Please use `is_call()` instead
#> This warning is displayed once per session.

#> Warning: `mut_node_car()` is soft-deprecated as of rlang 0.2.0.
#> This warning is displayed once per session.

#>     mpg cyl      disp  hp     drat    wt  qsec vs am gear carb
#> 1  21.0   6 12.649111 110 1.974842 2.620 16.46  0  1    4    4
#> 2  21.0   6 12.649111 110 1.974842 2.875 17.02  0  1    4    4
#> 3  22.8   4 10.392305  93 1.962142 2.320 18.61  1  1    4    1
```

To sum up

- `enquo` captures the symbol before it is evaluated, and turns it into a *quosure*. It will only be used in a function.
- `quos` captures `...` and creates a list of quosures.
- `!!` & `!!!` unquote a quosure. Will be used to signal the quotation has already been made.
- `:=` is to be used to assign a name.

A little bit of theory

`quo()` captures a symbol and its environment.

```
yiha <- function(x, y) { quo(x + y) }
a <- yiha(100, 12)
b <- yiha(200, 17)
a
```

```
#> <quosure>
#> expr: ^x + y
#> env:  0x103e933c0
```

```
b
```

```
#> <quosure>
#> expr: ^x + y
#> env:  0x103e23028
```

If we `eval_tidy()` them:

```
eval_tidy(a)
```

```
#> [1] 112
```

```
eval_tidy(b)
```

```
#> [1] 217
```

Quasiquotation

> Put simply, quasi-quotation enables one to introduce symbols that stand for a linguistic expression in a given instance and are used as that linguistic expression in a different instance. — Willard van Orman Quine

In R, an expression is flexible: it can refer to its `value` or to itself `as a symbol`. Its behavior depends on the context.

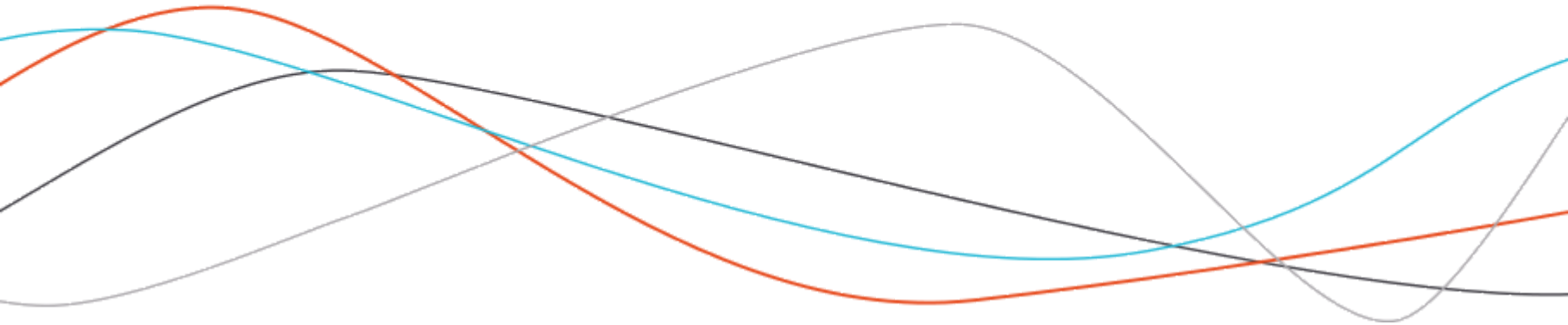
```
a <- 12
# As a symbol, then as a value
paste(quote(a), "is", a)
```

```
#> [1] "a is 12"
```

=> tidyevaluation relies on this ability to do quasiquotation: an operator is either refering to itself as a symbol or to the value it contains.

{purrr}

Make your code purr



What is {purrr}?

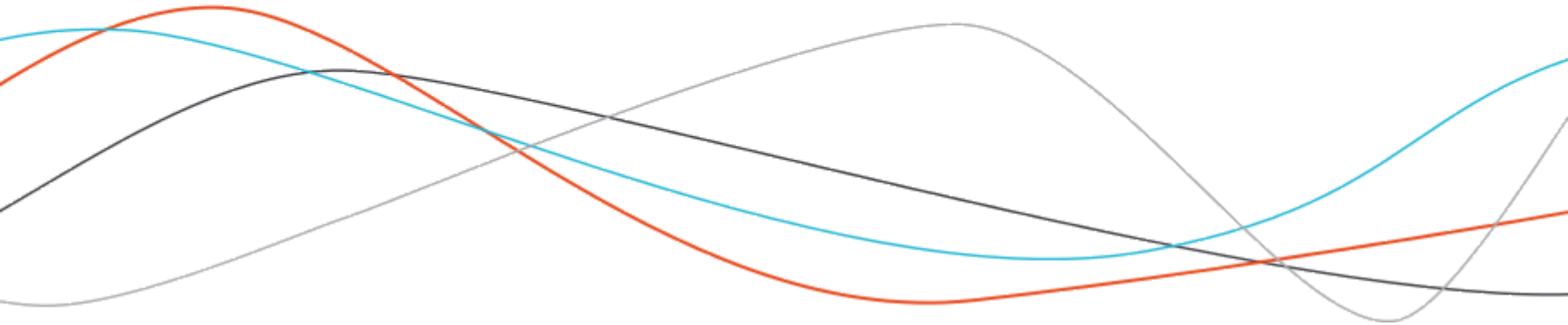
- Package by Hadley Wickham & Lionel Henry
- Toolbox for functional programming

```
library(purrr)
```

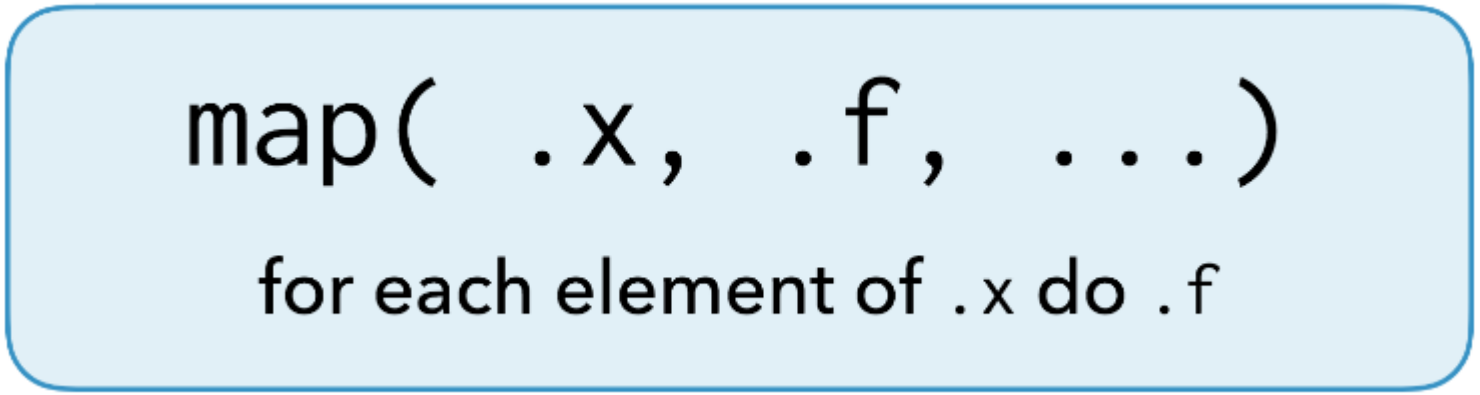
{purrr}, why?

- Consistant grammar for iteration
- Tools for function creation and manipulation
- Cleaner code

Iteration with {purrr}



{purrr} basic skeleton



`.x` can be:

- A list
- A vector
- A data.frame

```
map(.x = iris, .f = length)
iris %>% map( .f = length)
```

map and dot dot dot

`map( .x, .f, ...)`

for each element of `.x` do `.f`

- `...` is used to pass function parameters:

```
l <- list(a = sqrt(1), b = sqrt(2), c = sqrt(3))
l %>% map( round, digits = 2)
```

```
#> $a
#> [1] 1
#>
#> $b
#> [1] 1.41
#>
#> $c
#> [1] 1.73
```

map and .f

`map( .x, .f, ...)`

for each element of `.x` do `.f`

- .f can be a function, a number, or a character.

```
iris %>% map(2)
```

is the same as:

```
iris$Sepal.Length[[2]]
iris$Sepal.Width[[2]]
# etc.
```

With a character vector, `map()` looks for the named subelement in each element of the list.

map() use case

Read all CSVs from a folder:

```
my_files <- list.files("data", pattern = "csv", full.names = TRUE)
my_files <- my_files %>% map( data.table::fread, data.table = FALSE)
length(my_files)
my_files %>% map( dim)
```

About lambda functions

With {purrr}, you can use what is called lambda (or anonymous) functions:

```
iris %>% map( function(x) { sample(x, 1) })
```

```
#> $Sepal.Length
#> [1] 6.4
#>
#> $Sepal.Width
#> [1] 3.6
#>
#> $Petal.Length
#> [1] 5.8
#>
#> $Petal.Width
#> [1] 1.8
#>
#> $Species
#> [1] versicolor
#> Levels: setosa versicolor virginica
```

About mappers functions

With lambda functions, we create a function "on the fly". Lambda functions are also called "anonymous" because we don't give them any name: they are created in the context of the iteration.

We can also create mappers: anonymous functions with one-sided formula.

```
iris %>% map( function(x) { sample(x, 1) })
# To
iris %>% map( ~ sample( .x, 1) )
```

With mappers, we use `.x` to refer to the parameter of the function. You can also use a dot `.`, or `..1`.

Examples

- Compute the mean of all columns of airquality:

```
data("airquality")
airquality %>% map( mean )
```

- Get the class of each column of iris

```
iris %>% map( class )
```

- Get the mean and round it:

```
mtcars %>% map( ~round(mean(.x), 2) )
mtcars %>% map(mean) %>% map(round,digits = 2)
```

Control map_* output

```
# Simple list
map()

# Character vector
map_chr()

# Numeric
map_dbl()
map_int()

# Dataframe
map_df()
map_dfr() # Created with rbind
map_dfc() # Created with cbind

# Logical
map_lgl()
```

map2

Map on 2 elements:

```
map2(.x = iris$Sepal.Length,
     .y = iris$Sepal.Width,
     .f = sum)
```

```
#> [[1]]
#> [1] 8.6
#>
#> [[2]]
#> [1] 7.9
#>
#> [[3]]
#> [1] 7.9
#>
#> [[4]]
#> [1] 7.7
#>
#> [[5]]
#> [1] 8.6
#>
#> [[6]]
```

`map_at` and `map_if`

Map on specific places or under specific conditions

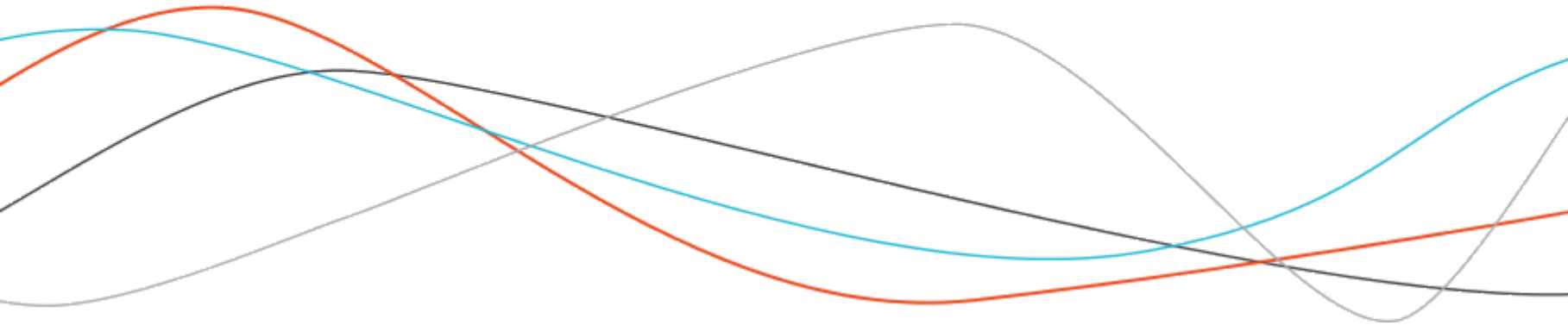
```
l <- list(thy = 1:10, art = letters, is = 1:100)

l %>% map_at( .at = c("art"), .f = toupper)

l %>% map_if( .p = is.numeric, .f = mean)
```

`walk()`: side effect only

Nothing printed to the console, just side effect (downloading, plotting...)

```
iris %>% walk( plot)
```

Cleaning with {purrr}



About predicates

- A predicate returns either `TRUE` or `FALSE`
- Exists in base R: `is.numeric()`, `%in%`, `is.character()`

About predicate functionals

- Function that takes an object and a predicate, and does something with these two inputs.

`keep()` and `discard()`

`keep()` takes a list, a vector or a data.frame, and a predicate. It keeps all the elements that return `TRUE` when tested against the predicate

`discard()` does the opposite

```
keep(iris, is.numeric) %>% head()
```

```
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1          5.1         3.5          1.4         0.2
#> 2          4.9         3.0          1.4         0.2
#> 3          4.7         3.2          1.3         0.2
#> 4          4.6         3.1          1.5         0.2
#> 5          5.0         3.6          1.4         0.2
#> 6          5.4         3.9          1.7         0.4
```

`every()` and `some()`

Do every/some element(s) satisfy a condition?

```
every(iris, is.numeric)
```

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

```
some(iris, is.numeric)
```

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

has_element(), detect(), detect_index()

has_element(): does .x contains .y?

```
has_element(iris, iris$Sepal.Length)
```

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

detect() & detect_index() returns the first element that satisfies the condition.

```
detect(23:77, ~ .x < 50)
```

```
#> [1] 23
```

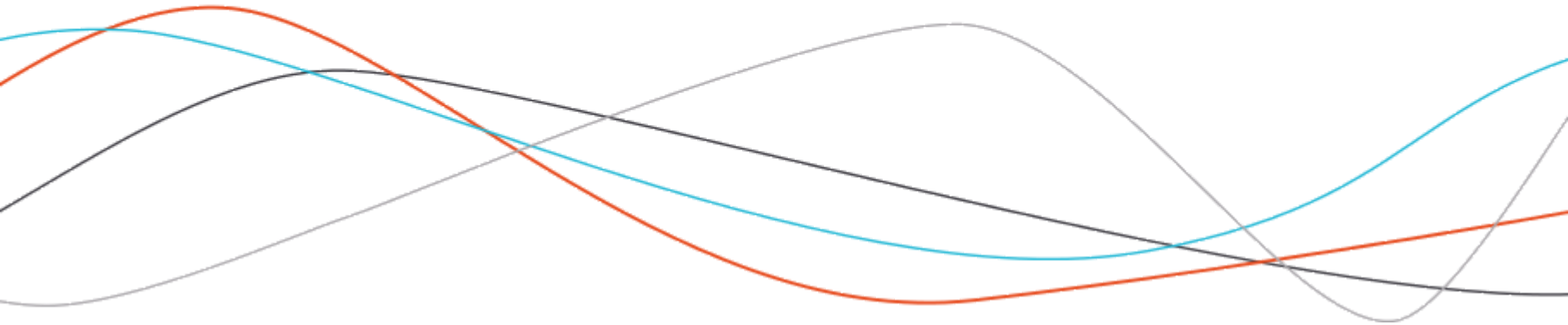
compact()

`compact()` removes the NULL

```
list( 1, 2, NULL, 3) %>%
  compact()
```

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

Programming with {purrr}



A high order function

A high order function can:

- Take one or more functions as arguments
- Return a function

```
nop_na <- function(fun){
  function(...){
    fun(..., na.rm = TRUE)
  }
}
sd_no_na <- nop_na(sd)
sd_no_na( c(NA, 1, 2, NA) )
```

```
#> [1] 0.7071068
```

Three types of high order functions

- Functionals
- Function factories
- Function operators

In \ Out	Vector	Function
Vector		Function factory
Function	Functional	Function operator

Three types of high order functions

- functionals -> take another function and return a vector: `map()` and friends
- function factories -> take a vector as input and create a function.
- function operators -> take one or more functions and return a function as output

> called "adverbs" in the tidyverse

Why adverbs?
In the tidyverse terminology, functions that take data and compute a value are called verbs. As function operators take a function as input and return this function modified (so they modify a verb), they can be considered as adverbs.

`possibly()` & `safely()`

`safely()` returns a function that will return:

- `$result`
- `$error`

One of them is NULL, depending on whether or not the function call succeeded.

```
safe_log <- safely(log)
```

```
safe_log(10)
```

```
#> $result
#> [1] 2.302585
#>
#> $error
#> NULL
```

```
safe_log("a")
```

```
#> $result
#> NULL
#>
#> $error
#> <simpleError in log(x = x, base =
base): non-numeric argument to
mathematical function>
```

`possibly()` & `safely()`

`possibly()` creates a function that returns either:

- the result
- the value of `otherwise` in case of error

```
possible_sum <- possibly(sum, otherwise = "nop")

possible_sum(1)
```

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

```
possible_sum("a")
```

```
#> [1] "nop"
```

`possibly()` & `safely()`

`possibly()` can return:

```
possibly(sum, FALSE)("a") # A logical
```

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

```
possibly(sum, NA)("a") # A NA
```

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

```
possibly(sum, "nope")("a") # A character
```

```
#> [1] "nope"
```

```
possibly(sum, 0)("a") # A number
```

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

```
possibly(sum, NULL)("a") # A NULL
```

```
#> NULL
```

To be combined with `map()`


```
safe_log <- safely(log)

map( list("a", 2), safe_log ) %>%
  map("error")
```

```
#> [[1]]
#> <simpleError in log(x = x, base = base): non-numeric argument to mathematical
function>
#>
#> [[2]]
#> NULL
```

Handling adverb results

- Cleaning `safely()` results -> Transform the result with `transpose()`:

> `transpose()` turn a list "inside-out". If you have a list of length 3 where each component is of length 2, transpose() creates a list of length 2, where each element is of length 3

```
l <- list("a", 2, 3)
map(l, safe_log)
```

```
#> [[1]]
#> [[1]]$result
#> NULL
#>
#> [[1]]$error
#> <simpleError in log(x = x, base =
base): non-numeric argument to
mathematical function>
#>
#>
#> [[2]]
#> [[2]]$result
#> [1] 0.6931472
```

```
map(l, safe_log) %>% transpose()
```

```
#> $result
#> $result[[1]]
#> NULL
#>
#> $result[[2]]
#> [1] 0.6931472
#>
#> $result[[3]]
#> [1] 1.098612
#>
#>
#> $error
#> $error[[1]]
#> <simpleError in log(x = x, base =
```

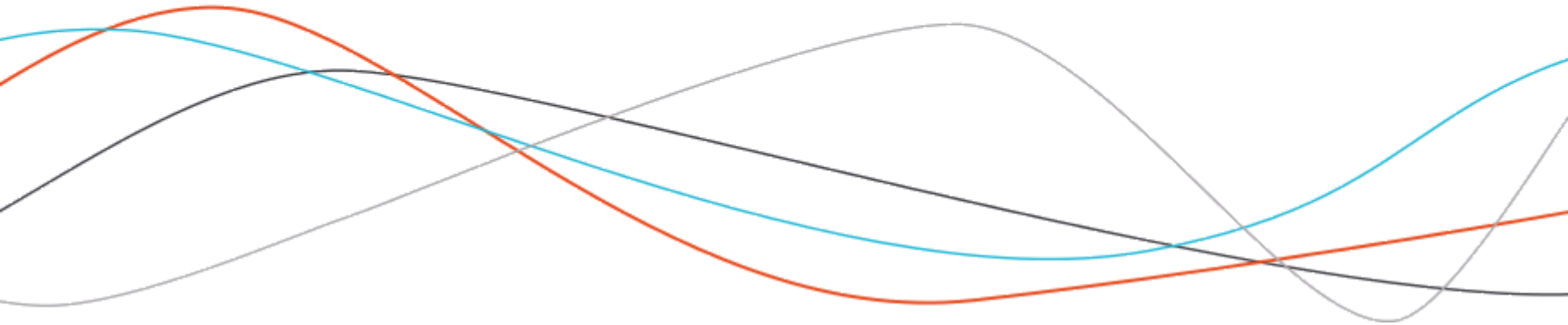
Handling adverb results

- `possibly()` and `compact()`

```
l <- list(1,2,3,"a")

possible_log <- possibly(log, otherwise = NULL)

map(l, possible_log) %>% compact()
```

```
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 0.6931472
#>
#> [[3]]
#> [1] 1.098612
```

Cleaner code with {purrr}



Why cleaner code?

Where's Waldo?

```
library(broom)
library(dplyr)

lm(Sepal.Length ~ Species, data=iris) %>%
  tidy() %>% filter(p.value < 0.01)
lm(Pepal.Length ~ Species, data=iris) %>%
  tidy() %>% filter(p.value < 0.01)
lm(Sepal.Width ~ Species, data=iris) %>%
  tidy() %>% filter(p.value < 0.01)
lm(Sepal.Length ~ Species, data=iris) %>%
  tidy() %>% filter(p.value < 0.01)
```

Why cleaner code?

Finding Waldo:

```
tidy_iris_lm <- compose(
  as_mapper(~ filter(.x, p.value < 0.01)),
  tidy,
  partial(lm, data = iris, na.action = na.fail)
)
list(
  Petal.Length ~ Petal.Width,
  Petal.Width ~ Sepal.Width,
  Sepal.Width ~ Sepal.Length
) %>% map(tidy_iris_lm)
```

About clean code

Clean code is:

- Light
- Readable
- Interpretable
- Maintainable

We aim for a code which is as compact as possible,
and with as less repetition as possible

Composing functions

```
my_stat <- compose(round, mean)
my_stat(1:2811)
```

```
#> [1] 1406
```

If you finally decide that your stat is the `median`, not the `mean`:

```
# FROM
#
round(mean(1:10))
round(mean(1:100))
round(mean(1:1000))
round(mean(1:10000))
```

> 4 changes needed.

```
# TO
my_stat <- compose(round, median)
my_stat(1:10)
my_stat(1:100)
my_stat(1:1000)
my_stat(1:10000)
```

> 1 change needed.

compose()

```
library(broom)
lm(Sepal.Length ~ Sepal.Width, data = iris) %>%
  anova() %>%
  tidy()
```

```
#> # A tibble: 2 x 6
#>   term           df  sumsq meansq statistic p.value
#>   <chr>       <int>  <dbl>  <dbl>     <dbl>   <dbl>
#> 1 Sepal.Width     1   1.41  1.41       2.07   0.152
#> 2 Residuals     148 101.    0.681     NA     NA
```

```
clean_aov <- compose(tidy, anova, lm)
clean_aov(Sepal.Length ~ Sepal.Width, data = iris)
```

```
#> # A tibble: 2 x 6
#>   term           df  sumsq meansq statistic p.value
#>   <chr>       <int>  <dbl>  <dbl>     <dbl>   <dbl>
#> 1 Sepal.Width     1   1.41  1.41       2.07   0.152
#> 2 Residuals     148 101.    0.681     NA     NA
```

negate()

Flip the logical:

```
is_not_na <- negate(is.na)

x <- c(1,2,3,4, NA)

is.na(x)
```

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

```
is_not_na(x)
```

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

`negate()` & mappers:

```
under_hundred <- as_mapper(~ mean(.x) < 100)

not_under_hundred <- negate(under_hundred)

map_lgl(98:102, under_hundred)
```

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

> Just one change to change the threshold

Prefilling functions

Prefill a function with `partial()`

```
mean_na_rm <- partial(mean, na.rm = TRUE)
mean_na_rm( c(1,2,3,NA) )
```

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

```
lm_iris <- partial(lm, data = iris)
lm_iris(Sepal.Length ~ Sepal.Width)
```

```
#>
#> Call:
#> lm(formula = ..1, data = iris)
#>
#> Coefficients:
#> (Intercept)  Sepal.Width
#>      6.5262      -0.2234
```

Prefilling functions

- If you finally decide to use `na.rm = FALSE`

```
# FROM
#
mean(mtcars$cyl, na.rm = TRUE)
```

```
#> [1] 6.1875
```

```
mean(mtcars$mpg, na.rm = TRUE)
```

```
#> [1] 20.09062
```

```
mean(mtcars$hp, na.rm = TRUE)
```

```
#> [1] 146.6875
```

> 3 changes

```
# TO
mnr <- partial(mean, na.rm =TRUE)
mnr(mtcars$cyl)
```

```
#> [1] 6.1875
```

```
mnr(mtcars$mpg)
```

```
#> [1] 20.09062
```

```
mnr(mtcars$hp)
```

```
#> [1] 146.6875
```

> 1 change

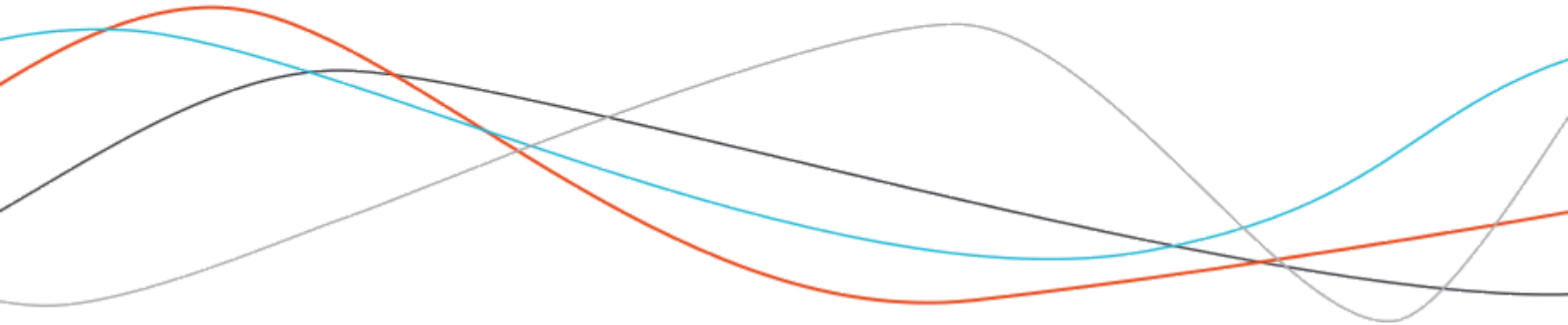
`partial()` & `compose()`

You can combine both:

```
rounded_mean <- compose(
  partial(round, digits = 2),
  partial(mean, na.rm = TRUE)
)
rounded_mean(airquality$Ozone)
```

```
#> [1] 42.13
```

List-columns



About list-columns

A data.frame with a list for a column:

```
library(tidyverse)

df <- tibble(
  classic = c("a", "b","c"),
  list = list(
    c("a", "b","c"),
    c("a", "b","c", "d"),
    c("a", "b","c", "d", "e")
  )
)
df
```

```
#> # A tibble: 3 x 2
#>   classic list
#>   <chr>   <list>
#> 1 a       <chr [3]>
#> 2 b       <chr [4]>
#> 3 c       <chr [5]>
```

About list-columns

With a list column, also called a nested data.frame, you can put any element in one "cell" of a dataframe, instead of a scalar value.

Something that you cannot do with classic data.frames. This behavior is specific to the **tibble** class, which is the tidyverse implementation of dataframes.

Why list columns?

- **Cleaner code**: everything inside the same pipe, as everything will stay inside the same data.frame.
- **More flexibility**: cells with different length -> useful when output size is not predictable.

```
library(rvest)
a_node <- partial(html_nodes, css = "a")
href <- partial(html_attr, name = "href")
get_links <- compose( href, a_node,  read_html )

urls_df <- tibble(urls = c("https://thinkr.fr", "https://colinfay.me"))
urls_df %>%  mutate(links = map(urls, get_links))
```

```
#> # A tibble: 2 x 2
#>   urls                links
#>   <chr>               <list>
#> 1 https://thinkr.fr   <chr [106]>
#> 2 https://colinfay.me <chr [33]>
```

`unnest()` a nested data.frame

- `unnest` is in package {tidyr}

```
urls_df %>%
  mutate(links = map(urls, get_links)) %>%
  unnest()
```

```
#> # A tibble: 139 x 2
#>    urls             links
#>    <chr>            <chr>
#>  1 https://thinkr… https://thinkr.fr/
#>  2 https://thinkr… https://thinkr.fr/
#>  3 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/
#>  4 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/introduction…
#>  5 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/statistique-…
#>  6 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/programmatio…
#>  7 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/r-et-le-big-…
#>  8 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/r-pour-la-fi…
#>  9 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/integration-…
#> 10 https://thinkr… https://thinkr.fr/formation-au-logiciel-r/formation-si…
#> # ... with 129 more rows
```

`nest()` a standard data.frame

- `nest` is in package {tidyr}

```
iris_n <- iris %>%
  group_by(Species) %>% nest()
iris_n
```

```
#> # A tibble: 3 x 2
#>   Species    data
#>   <fct>      <list>
#> 1 setosa     <tibble [50 × 4]>
#> 2 versicolor <tibble [50 × 4]>
#> 3 virginica  <tibble [50 × 4]>
```

```
iris_n %>%
  mutate(lm = map(data, ~ lm(Sepal.Length ~ Sepal.Width, data = .x)))
```

```
#> # A tibble: 3 x 3
#>   Species    data              lm
#>   <fct>      <list>            <list>
#> 1 setosa     <tibble [50 × 4]> <S3: lm>
#> 2 versicolor <tibble [50 × 4]> <S3: lm>
#> 3 virginica  <tibble [50 × 4]> <S3: lm>
```

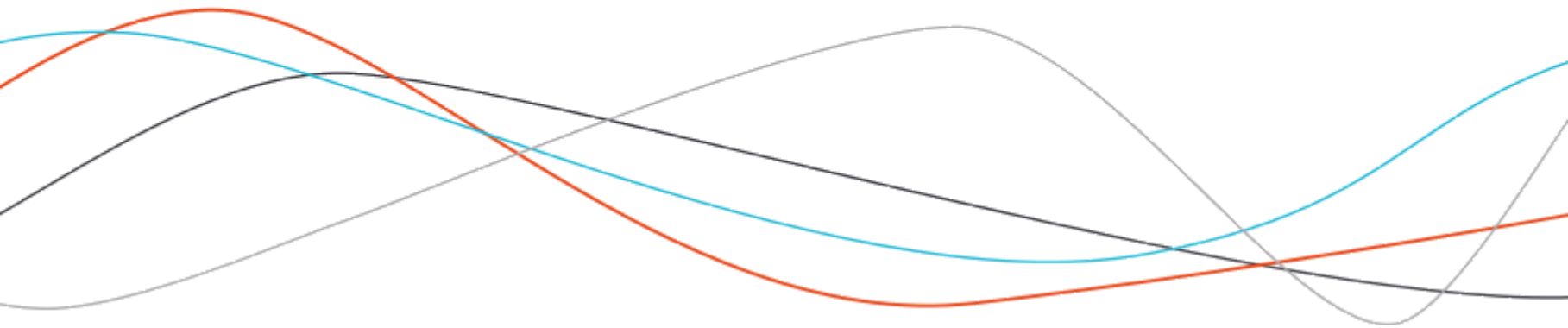
Example

```
summary_lm <- compose(summary, lm)

iris %>%
  group_by(Species) %>%
  nest() %>%
  mutate(data = map(data, ~ summary_lm(Sepal.Length ~ Sepal.Width,
                                       data = .x)),
         data = map(data, "r.squared")) %>%
  unnest()
```

```
#> # A tibble: 3 x 2
#>   Species     data
#>   <fct>      <dbl>
#> 1 setosa     0.551
#> 2 versicolor 0.277
#> 3 virginica  0.209
```

Create a package

Document, standardize, share



What is a package?

In R, the basic (complete) unity of code is a package.

A package is a combination of:

- code
- data
- doc
- tests

Why a package?

A package is designed to share code. In other words, it makes the code you've written on your machine accessible and usable to any other user, on any other machine (well, theorically).

We use a package rather than random scripts, because a good package can be installed and work on all the machines on which it is installed.

> *Anything that can be automated, should be automated. Do as little as possible by hand. Do as much as possible with functions. The goal is to spend your time thinking about what you want your package to do rather than thinking about the minutiae of package structure.*
>
> H. Wickham

Create your own packages – tools

To create your own packages, you'll need:

- RStudio.
- {devtools}
- {roxygen2} to create the documentation
- {usethis} for templating and skeletons
- {testthat}, for tests
- {knitr}, for Vignettes

Create your own packages – tools

Specific tools:

- Rtools.exe (not mandatory and only on windows) : https://cran.r-project.org/bin/windows/Rtools/ => is used to handle packages that contains compiled code.
- r-base-dev on Linux

Install the packages:

```
install.packages(c("devtools",
                   "roxygen2",
                   "usethis",
                   "testthat",
                   "knitr"))
```

Find a name

The {available} package checks on several platforms if your name is either available or not, and performs a short sentiment analysis - which can be useful if you're not a native english speaker.

```
available::available("thinkr")
```

Creation

New project > New directory > R package using devtools.

Choose a name.

You can also use `usethis::create_package("path/to/package")`.

> Name should contain only ASCII letters, numbers and dot, have at least two characters and starts with a letter and not end in a dot

A package skeleton

Any package should contains these files:

- `DESCRIPTION`: the general description of the package: purpose, developer, license ...

You can edit this file by hand, and {roxygen2} will also help later on.

- `NAMESPACE`: How does your package interact with R and with other packages?

You should not edit this file by hand - {roxygen2} will do it for us.

- The `R/` folder: contains the R scripts with our functions.

A package skeleton

While developping, we'll add:

- `man/` folder: contains help of each function: fonction1.Rd , fonction2.Rd.

This folder will be filled automatically by {roxygen2}. All packages should have this folder.

- `inst/` folder: files that will be moved as is where the package is installed.
- `data/` folder (self-explanatory).
- `data-raw/` folder: raw datasets, to be included as archive.
- `test/` folder: tests created by {testthat}.
- `vignettes/` folder: vignettes.

A package skeleton

About *.Rbuildignore*

The *.Rbuildignore* file allows to specify files and folder that will be ignored when the package is built.

You can add them as is or specify regex:

```
^.*\.Rproj$
^\.Rproj\.user$
^README\.Rmd$
^README-.*\.png$
.travis.yml
^CONDUCT\.md$
^data-raw$
^cran-comments\.md$
paper\..*
^revdep$
^docs$
```

Metadata

The DESCRIPTION file (always in CAPS) contains your package metadata.

Filling this file is the first thing to do when you start a new package. We'll fill some fields by hand, and {roxygen2} will help us doing the rest.

```
Package: package
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

DESCRIPTION

- `Title`: One line, short description of the purpose of your package. In Title Case and Does Not End with a Dot
- `Author`: Use this syntax :

```
Authors@R: c(person("First Name", "Last Name", email =
"a_valid@email.com", role = c("aut", "cre")))
```

You can add several `person`. Most common roles are `aut` (author), `cre` (maintainer), `ctb` (contributor), `cph` (copyright holder).

All possible roles: https://www.loc.gov/marc/relators/relaterm.html.

All packages should have at least one author and one maintainer (which can both be the same person).

- `Description` : One longer paragraph to describe the package. End with a dot.

DESCRIPTION

- `License`: the package license. Should met a standard license (GPL, GNU, BSD, MIT...), or name a LICENSE file, distributed along the package.
- `Version`: every package should have a version number, composed by three numbers separated with dot. This version number is used by R to solve dependencies issues and to know if a package needs to be upgraded or reinstalled.

Choose a version number

> The version number contains `major.minor.patch`

Good practice

> As long as the package is not out there in the real world (i.e stable, in production or on CRAN), the version number should stay 0.0.0.9000 and next. It allows to increment of OOO1 each time something new is implemented.

DESCRIPTION

Other fields:

- `BugReport`: link to where to report for bugs (usually, a github issue link)
- `URL`: website
- `LazyData`: should the data be lazily loaded?
- `Imports` & `Suggests`: dependencies
- `Depends`: if you rely on a specific version of R

and other...

A "package-ready" R function

In the R folder are what we can call "nice" functions, i.e. functions that :

- Do not use `library()` or `require()`: dependencies are managed in the NAMESPACE file.
- Do not change user `options()` or `par()`.
- Do not use `source()` to call code.
- Do not play with `setwd()`.
- Do not silently write in any other place than in OS temporary directory.

The R code

Organise your R files

- You can have as much .R files as you want in the R/ folder. Usually, one file contains one "big" function or a family of functions.
- Utils functions are usually put into a "utils.R".

> Note that this R/ folder should not contain any subdirectory.

NAMESPACE

The NAMESPACE file is the most crucial file of your package, and you should NEVER edit it by hand.

This file describes how your package interact with R, and with other packages. Dependencies are handled there, as are exported functions.

This NAMESPACE is what allows your package to work. You don't have to fill it by hand, thanks to {roxygen2} and the `@export`, `@import` & `@importFrom` tags (we'll see them later on).

NAMESPACE

When programming, there's a big chance you'll name a function as another function from another package (remember there are 13K+ packages on the CRAN). Namespaces are what makes names cohabitation possible.

> When doing `pkg::fun()`, you're doing "explicit namespacing" you're refering to the function `fun` from `pkg`

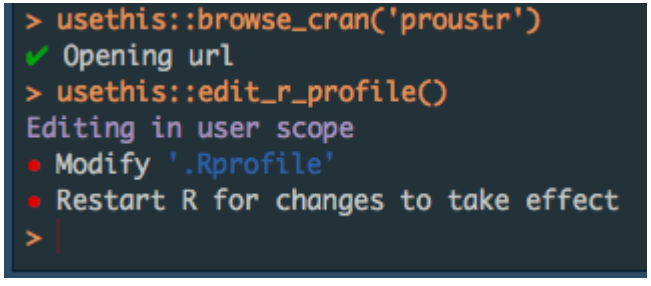
Using {usethis}

{usethis} is a package by Hadley Wickham designed to insert and create common package elements:

- license
- dependencies
- README
- Github, AppVeyor, Travis...
- NEWS
- data-raw/
- data files

When calling a function from {usethis}, the function has either :

- done all the job
- left you some things to do



Dev with {usethis}

All functions starting with `use_` allows to use a template/skeleton for common elements.

Day to day package commands

- `use_build_ignore(file)`: creates a regex from `file` and puts it into `.RBuildignore`
- `use_data`: converts a dataset in Rda and puts it into `data/`
- `use_data_raw`: creates the `data-raw` folder and adds its name into `.RBuildignore`.
- `use_package`: adds a package as a dependency. `use_dev_package` creates a `Remote` field in the DESCRIPTION, if you want to link to a GitHub package.
- `use_pipe`: imports the `%>%`
- `use_testthat`: creates the test folder
- `use_vignette`: creates a Vignette

Using {usethis}

Licensing

- use_apl2_license
- use_cc0_license
- use_gpl3_license
- use_mit_license

```
> options(usethis.full_name = "Colin FAY")
> usethis::use_mit_license()
✔ Setting License field in DESCRIPTION to 'MIT + file LICENSE'
✔ Writing 'LICENSE.md'
✔ Adding '^LICENSE\\.md$' to '.Rbuildignore'
✔ Writing 'LICENSE'
```

Where do I put my dev script?

Using {usethis}

```
file.create("devstuffs.R")
usethis::use_build_ignore("devstuffs.R")
```

-> Create a "devstuffs.R" (you can use any other name) to keep everything you did during the engineering process.

Documentation

Documentation

Documentation is crucial: anybody using your package needs it. And this anybody can, of course, be you in the future.

Documentation is found in the RStudio Help panel, or via `?a_specific_function`.

Activate roxygen2

In Build > Configure build tools, check "Generate documentation with roxygen" and all the other boxes.

Documentation is generated and found in .Rd files in the man/ folder. They are in LaTeX, and are readable in HTML, text or pdf once the package is installed. Good news: you don't have to type the LaTeX by hand, thanks to {roxygen2}.

There are several advantages to {roxygen2}:

- Don't need to write LaTeX code
- The documentation stays with the function
- The documentation fits the data type

A "UX first" approach

Building a package that lasts means that once your package is stable, you'll need **people to (be able) use it**.

"Be able" means: documentation should easily guide them.

Consider:

> "An R Package using the R6 paradigm to create an object oriented API designed to interactively and programmatically write Docker setup files inside an R session or script."

VS

> "Easy Dockerfile Creation with R6"

Documentation

Comments starting with `#'` are read by {roxygen2} as documentation. (ctrl + alt + shift + R generates this skeleton).

Example:

```
#' Title
#'
#' description of the function
#' @param x first param
#' @param y second param
#' @return what comes out of the function
#' @examples
#' plop(4,5)
#' plop(5,9)
#' @export

plop <- function(x, y){
  return(x+y)
}
```

Documentation

Every roxygen comments starts with a `#'`. Then, a tag is added with `@` (RStudio has autocompletion). If there are no `@` on the first two lines, they are read as title and description.

Most common used tags are:

- `@details`: details about the fun
- `@param param this is that`: name of the param, and its description
- `@return`: what the function returns
- `@examples`: (self-explanatory)
- `@export`: **to use if you want to export your function**
- `@import` & `@importFrom`: dependencies
- `@source` & `@references`: references

=> `browseVignettes("roxygen2")`, vignette "rd".

Documentation

Some roxygen fields make it easier to navigate in packages:

- `@seealso` : points to other resources, on the web or in the package
- `@family` : makes the function belong to a family of functions
- `@aliases` : gives "nicknames" to the function, to allow it to be found with `?my_alias`.

Documentation

Prevent an example from being executed

=> Put the example(s) between `\dontrun{}`.

Render the doc

```
roxygen2::roxygenise()
# OR
devtools::document()
```

External data

Available for the user

- It is recommended to store your raw dataset in data-raw/ folder
- Create a R code in data-raw/ to read and modify the dataset if needed
- Use `usethis::use_data` to store it in the data/ folder in the correct format

```
sampleiris <- sample_frac(iris, 0.2)

usethis::use_data(sampleiris, overwrite = TRUE)
```

`overwrite = TRUE` => overwrite the data if it already exists.

Once installed => `data(sampleiris)`

Document external data

`doc_sampleiris.R` in R/:

```
#' sample_iris
#'
#' A sample from iris
#'
#' @name sampleiris
#' @docType data
#' @author Colin \email{colin@@thinkr.fr}
#' @source \url{google.com}{unlien}
#' @format data.frame
#' @keywords data
"sampleiris"
```

The two importants tags here are `@format` and `@source`.

> Don't `@export` a dataset

Data

Internal data

This function creates a `sysdata.Rda` in R/, that allows to use datasets which won't be exported:

```
usethis::use_data(sampleiris, overwrite = TRUE, internal = TRUE)
```

Raw dataset

You can insert raw data in `inst/extdata` directory.

Then, anybody will be able to reach this dataset using:

```
system.file("extdata", "sampleiris.csv", package = "monpackage")
```

Vignettes

A vignette is an html/pdf file that accompanies a package and is more complete than help pages, as the format is free (you can include images, tables, links, html...).

```
usethis::use_vignette("mypackage")
```

...creates a "mypackage.Rmd" file in the vignettes/ folder at the root of the package, and adds the appropriate dependency in your DESCRIPTION.

The vignette is a classical RMarkdown page.

Vignettes

This is a verbatim inline expression `2`.

```
---
title: "Vignette Title"
author: "Vignette Author"
date: '`r Sys.Date()`'
output: rmarkdown::html_vignette
thumbnail: >
  %\VignetteIndexEntry{Vignette
Title}
  %
ThumbnailEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

The YAML describes your vignette

Replace the title, the author and the "VignetteIndexEntry".

Then, you can write the vignette as a classic Rmarkdown file.

To preview the vignette rendering, press the knit button, or Ctrl/Cmd + Shift + K.

How many vignettes?

There are no right answers to this question. This depends on the size of your package and of what you have implemented.

Build Vignettes

Use `devtools::build_vignettes()` to render the vignette (in /inst/doc).

The end user can see the vignettes with the following instruction:

```
browseVignettes("myPackage")
```

About the NAMESPACE

The NAMESPACE file is one of the most important files of your package. It's also the one you should not edit by hand.

This file describes **how your package interacts with R, and with other packages**. This is where, among other things, the dependencies are managed.

This file also lists the functions that are exported.

The namespace allows the package to work. This file is managed by {roxygen2}, via the tags `@export`, `@import` and `@importFrom`.

What is a dependency?

To work, your package may need external functions, i.e. contained in other packages.

R has three types of dependencies that will be contained in the DESCRIPTION :

- `Depends` & `Imports`: the packages that will be `attach()` or `load()` respectively. In practice, always list in `Imports`. `attach` means that the package is attached to the search path (and remember that a good package should not touch to the user's environment).
- `Suggests`: suggests packages to use in addition to your package. Will not be attached or loaded. May be the one used in Vignettes but not in the package.

These elements are filled automatically thanks to `{usethis}` or (in-dev) package `{attachment}`.

List dependencies

- Add dependencies to EACH function to fill the NAMESPACE file
  - You can use `import` (a whole package) or `importFrom` (a specific function).
  - The better is to use `importFrom`, for preventing namespace conflict.
  - Add to EACH function.

> It will take a lot of time, but it's better on the long run.

```
#' @import magrittr
#' @importFrom  stats na.omit

moyenne <- function(x){
  x <- x %>% na.omit()
  sum(x)/length(x)
}
```

List dependencies

The DESCRIPTION file lists all dependencies of the package.

To fill the DESCRIPTION file, for each package, use the R command:

```
usethis::use_package("attempt")
```

An easy way to fill all dependencies needed in DESCRIPTION is to use package {attachment}:

```
# remotes::install_github("ThinkR-open/attchment")
attachment::att_to_description()
```

Keep these commands in your "devstuffs.R" as a reminder

startup

A file called zzz.R (by convention) in R/ can contain specific functions:

```
#' @importFrom utils packageDescription
#' @noRd
.onAttach <- function(libname, pkgname) {
  if (interactive()) {
    pdesc <- packageDescription(pkgname)
    packageStartupMessage('')
    packageStartupMessage(pdesc$Package, " ", pdesc$Version, " by
",pdesc$Author)
    packageStartupMessage(paste0('->  To correctly start use :
help(',pkgname,')'))
    packageStartupMessage('')
  }}
```

`.onAttach` & `.onLoad` are run on package launch.

Why automate code testing?

- To save time!
- Work peacefully with your coworkers
- Transfer the project
- Guarantee the stability on the long run

Thanks to `{testthat}`, we can automate the tests.

-> Allows bugs to be detected before they happen, and guarantees the validity of the code.

```
install.packages("testthat")
usethis::use_test("myfunction")
```

Creates a test/testthat folder, adds {testthat} to the `Suggests` of the DESCRIPTION, and creates test/testthat.R (do not touch it).

Does it rings a bell?

```
my_awesome_function <- function(a, b){
  res <- a + b
  return(res)
}
# Works
my_awesome_function(1, 2)
```

```
#> [1] 3
```

```
# Doesn't work
my_awesome_function("a","b")
```

```
#> Error in a + b: non-numeric argument to binary operator
```

Test that

In the test/ folder is a testthat.R file and a testthat folder. In this folder, you'll find .R files of the following form:

```
test-my_function.R
```

- One test file per (big) function, with general contexts.
- Break down the tests in this file by type.

These tests will be performed during `devtools::check()` (which also performs other tests), or with `devtools::test()` (Ctrl/Cmd + Shift + T).

Test that

Each file is composed of a series of tests in this format:

```
context("global info")

test_that("details series 1",
          {
         test1a
         test1b
 })

test_that("details series 2",
          {
         test2a
         test2b
 })
```

Test functions

Your test functions start with `expect_*`.

- This takes two elements: (1) the actual result, (2) the expected result.
- If the test has not passed, the function returns an error. If the test passes, the function returns nothing.

```
library(testthat)

expect_equal(10, 10)

a <- sample(1:10, 1)
b <- sample(1:10, 1)
expect_equal(a+b, 200)
```

```
Erreur : a + b not equal to 200.
1/1 mismatches
[1] 11 - 200 == -189
```

Expectations

```
library(testthat)
grep("^expect", ls("package:testthat"), value = TRUE)
```

```
#>  [1] "expect"                    "expect_condition"
#>  [3] "expect_cpp_tests_pass"     "expect_equal"
#>  [5] "expect_equal_to_reference" "expect_equivalent"
#>  [7] "expect_error"              "expect_failure"
#>  [9] "expect_false"              "expect_gt"
#> [11] "expect_gte"                "expect_identical"
#> [13] "expect_is"                 "expect_known_failure"
#> [15] "expect_known_hash"         "expect_known_output"
#> [17] "expect_known_value"        "expect_length"
#> [19] "expect_less_than"          "expect_lt"
#> [21] "expect_lte"                "expect_match"
#> [23] "expect_message"            "expect_more_than"
#> [25] "expect_named"              "expect_null"
#> [27] "expect_output"             "expect_output_file"
#> [29] "expect_reference"          "expect_s3_class"
#> [31] "expect_s4_class"           "expect_setequal"
#> [33] "expect_silent"             "expect_success"
#> [35] "expect_that"               "expect_true"
```

"Skip" a test

If you want to skip a test (if the code depends on a web connection, an API, etc...), use the `skip_if_not()` function.

```
library(httr)
url <- "http://numbersapi.com/42"
test_that("API test", {
  skip_if_not(curl::has_internet(), "No internet connection")
  res < content(GET(url))
  expect_is(url, "character")
})
```

`testthat:::skip_if_not_installed()` skips a test if a package is not installed.

There are other functions to skip a test under particular conditions, such as `skip_on_os()`, to prevent from testing on specific operating systems.

Create your own test

You can also create your own tests in `test_that()`:

```
plop <- function(class) {
  structure(1:10, class = class)
}

expect_plop <- function(object, class){
  expect_is(object, class)
}

test_that("Class well assigned", {
  a <- plop("ma_class")
  expect_plop(a, "ma_classe")
})
```

Launch tests

```
grep("^test", ls("package:testthat"), value = TRUE)
```

```
#>  [1] "test_check"      "test_dir"        "test_env"
#>  [4] "test_example"    "test_examples"   "test_file"
#>  [7] "test_package"    "test_path"       "test_rd"
#> [10] "test_that"       "testing_package"
```

Launch tests

```
devtools::check()

...
✔ |  12       | adverbs
✔ |   7       | test-utils.R
✔ |  22       | test-warn.R
✔ |  18       | test-any-all-none.R

══ Results ═════════════════════════════════════════════════════════
Duration: 0.8 s

OK:       192
Failed:   3
Warnings: 0
Skipped:  0
```

R CMD check

To test the code more globally, in the command line (i.e. in the terminal): R CMD check.

Or simply the `devtools::check()` function in your R session.

More tests are performed with `check` than with `devtools::test()`, which "only" performs the tests in the test folder.

This command runs around 50 different tests.

Is performed when you click the "Check" button on the Build tab of RStudio.

R CMD check

Three types of errors:

ERROR

Fix that before creating the package.

WARNING

Alerts about the package.

NOTES

Small problems.

Test with rhub

{rhub} is a package that allows you to test for several OS:

```
library(rhub)
ls("package:rhub")
```

```
#>  [1] "check"                 "check_for_cran"
#>  [3] "check_on_centos"       "check_on_debian"
#>  [5] "check_on_fedora"       "check_on_linux"
#>  [7] "check_on_macos"        "check_on_ubuntu"
#>  [9] "check_on_windows"      "check_with_rdevel"
#> [11] "check_with_roldrel"    "check_with_rpatched"
#> [13] "check_with_rrelease"   "check_with_sanitizers"
#> [15] "check_with_valgrind"   "last_check"
#> [17] "list_my_checks"        "list_package_checks"
#> [19] "list_validated_emails" "platforms"
#> [21] "rhub_check"            "rhub_check_for_cran"
#> [23] "rhub_check_list"       "validate_email"
```

Test with rhub

```
devtools::install_github("r-hub/rhub")
# or
install.packages("rhub")
# verify your email
library(rhub)
validate_email()
```

Test with rhub

```
rhub::check()
```

Which platforms are supported?

```
rhub::platforms()
```

```
> rhub::platforms()
debian-gcc-devel:
  Debian Linux, R-devel, GCC
debian-gcc-patched:
  Debian Linux, R-patched, GCC
debian-gcc-release:
  Debian Linux, R-release, GCC
fedora-clang-devel:
  Fedora Linux, R-devel, clang, gfortran
fedora-gcc-devel:
  Fedora Linux, R-devel, GCC
linux-x86_64-centos6-epel:
  CentOS 6, stock R from EPEL
linux-x86_64-centos6-epel-rdt:
  CentOS 6 with Redhat Developer Toolset, R from EPEL
linux-x86_64-rocker-gcc-san:
```

Colin Fay

colin@thinkr.fr

+33 (0)6.24.21.32.18