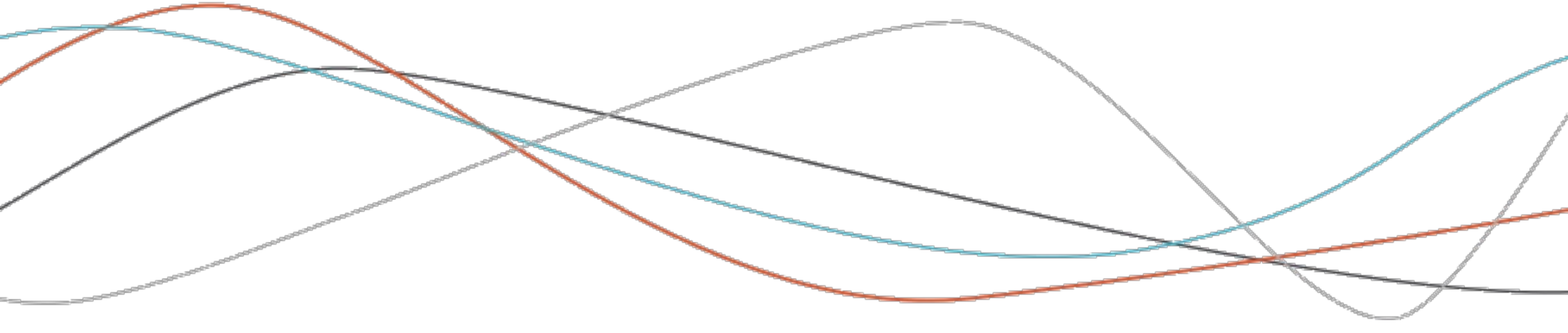
THINKR

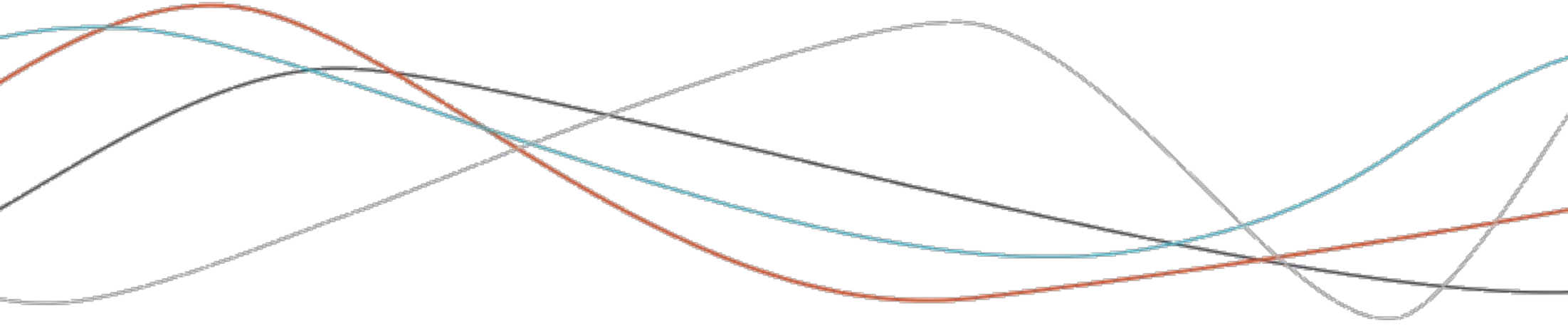
Getting data from the web

November 26, 2018



Git setup

Version control your R projects

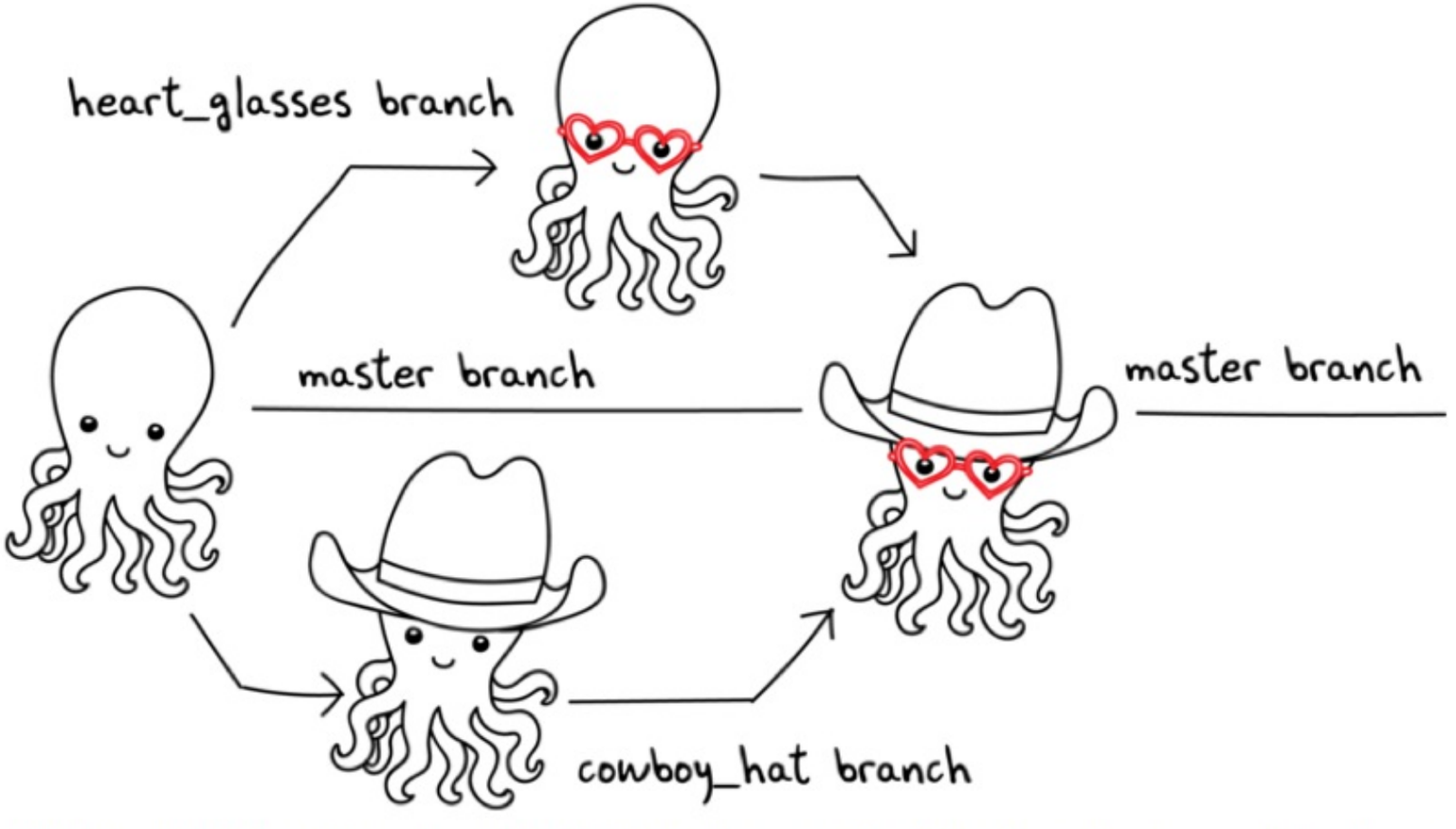


How does it work?

Git is a tool to detect and track changes in files. It keeps a history of what happened to files and is designed for collaborative work.

merge it back into the original project.

Here's a simplified version of that workflow:



Each feature is added back into master individually. So if the glasses are finished before the cowboy hat, no problem: those can be

Git setup

Install

On windows (other platform, see the email)

- Download for windows at https ://git-scm.com/downloads.
- Install Git in the default folder.
- Choose "Use Git from the Windows Command Prompt"
- Choose "Checkout as is, commit as-is“.

Setup RStudio

- Tools > Global Option > Git/SVNs.
- Check if Git is turned on and that the path to the executable is the right one.

Git setup

Start an R project with git - GitHub side

You need to remember that each project in R is associated with a project in GIT.

The name of the R project will be the very same as the GIT project.

Create a github account first

1. Create a github project
2. Name the repository (this will be the name of you future RStudio project)
3. Make a short description of the project
4. Click on "Create project"
5. Copy to clipboard the URL of the project (Ex: https://github.com/DianeBeldame/test_x.git)

Git setup

Start an R project with git - RStudio side

- Menu File > New Project
- Choose : Version Control > Git
- Repository URL : paste what's in your clipboard (here : https://github.com/DianeBeldame/test_x.git)
- Project Directory Name is filled with test_x
- Create project as subdirectory : Path to your project

Git setup

If your project already exists -

This process is done AT YOUR OWN RISK (prefer the previous method)

- Menu Tools > Project Options > Git/SVN
- Choose Git, and confirm git init.
- With the "ignore" menu, you can add the files you don't want to keep a track of (this is optional).
- Use the "commit" button to save your files.
- In the "Git" windows in RStudio, use the `Shell` (often under the `More` menu).
- In the shell, type:

```
git remote add origin 'https://github.com/<username>/<projectname>.git'
git push -u origin master
```

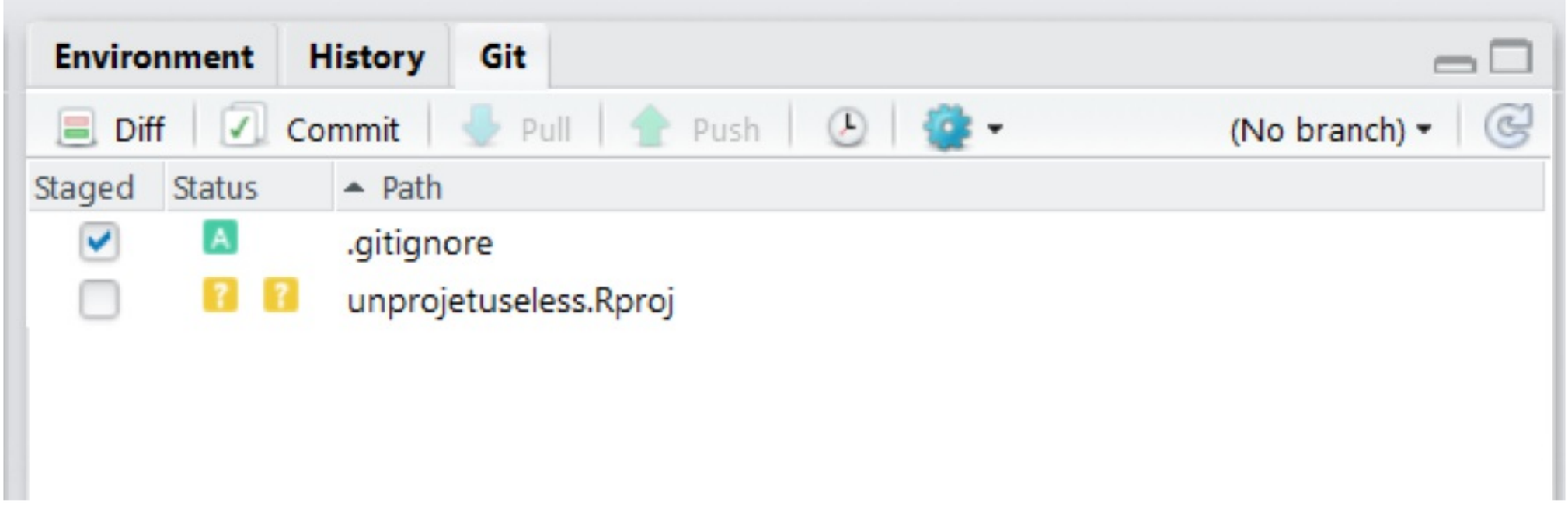
- Close the terminal

Use git

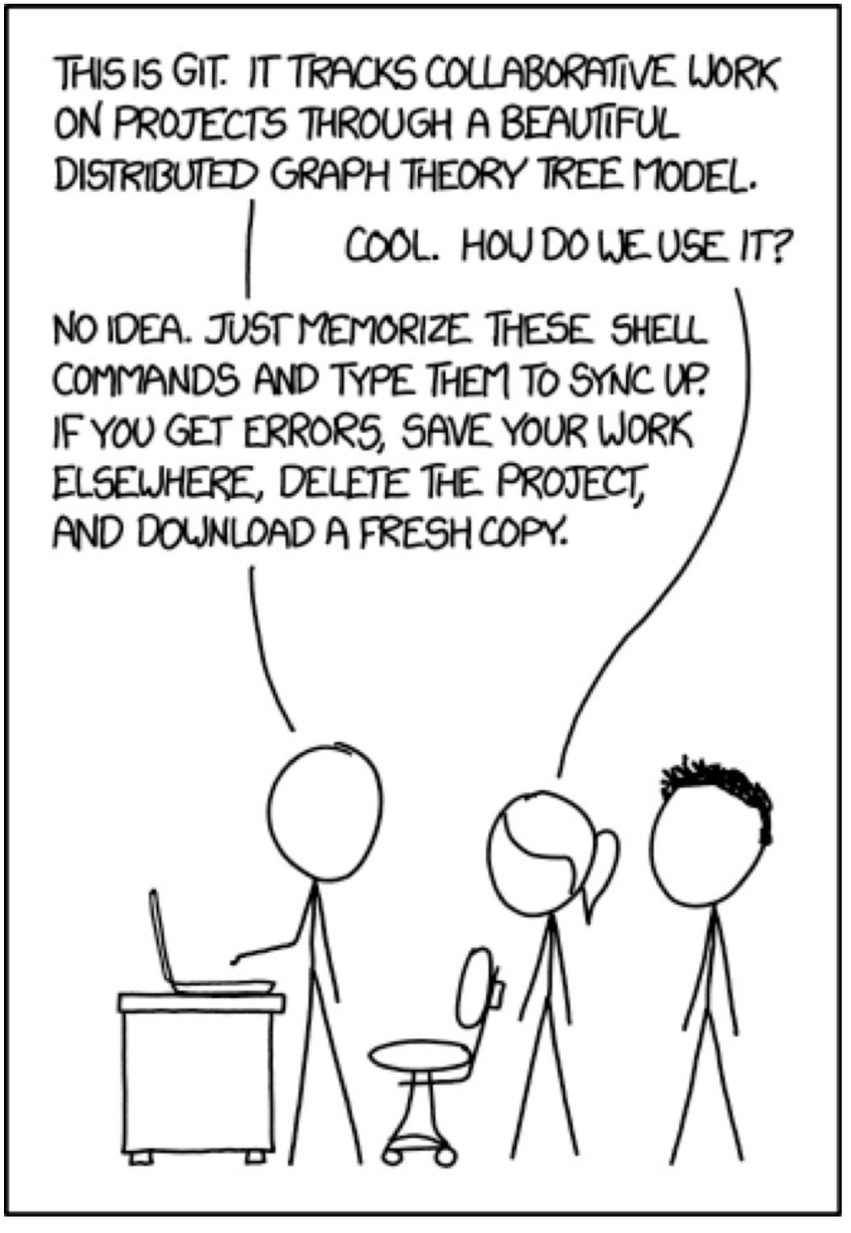
How to ?

There are 3 main actions.

- Save your changes locally: the `commit`.
- `push`: save a batch of changes on the git central server.
- Bring the remote changes from the central git server to your local session: `pull`.



Use git


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Use git

Commit

Once you've modified a file, click on the git tab and press the commit button. You'll need to write a commit message explaining the changes you have made.

Push

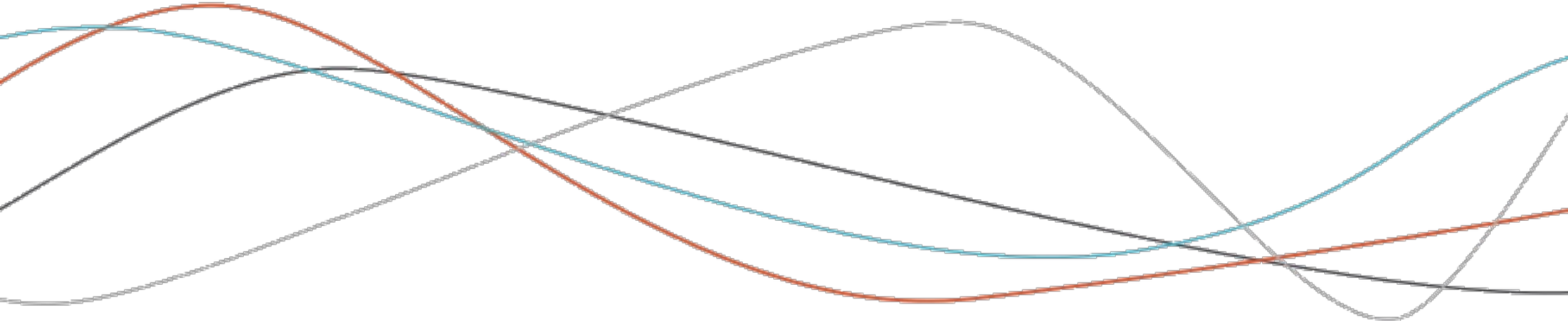
Once you've made several commits, you have to push them to the main server. Use the push button from the git tab. Before making a push, you need to pull first, to ensure there are no changes made on the remote git server. If changes have been made, you'll receive them.

Pull

When you want to receive the changes that have been made on the remote server (most of the time made by someone else), you need to use the pull button, which is sometime found under the "More" button.

{purrr}

Make your code purr



What is {purrr}?

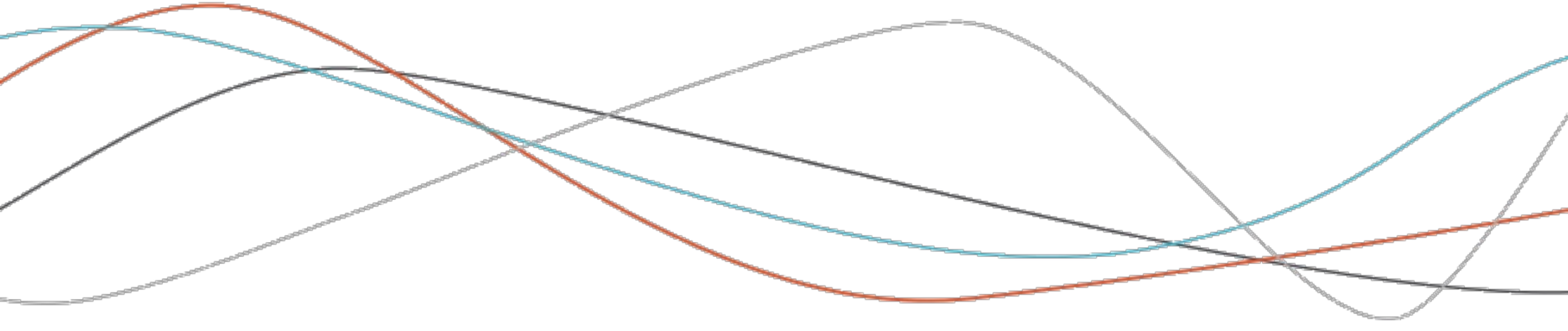
- Package by Hadley Wickham & Lionel Henry
- Toolbox for functional programming

```
library(purrr)
```

{purrr}, why?

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

Iteration with {purrr}



{purrr} basic skeleton


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

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

`.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:

```
mtcars %>% map( function(x) { mean(x * 100) })
```

```
#> $mpg
#> [1] 2009.062
#>
#> $cyl
#> [1] 618.75
#>
#> $disp
#> [1] 23072.19
#>
#> $hp
#> [1] 14668.75
#>
#> $drat
#> [1] 359.6562
#>
#> $wt
#> [1] 321.725
#>
```

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.

```
mtcars %>% map( function(x) { mean(x * 100) })
# To
mtcars %>% map( ~ mean(.x * 100) )
```

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:

```
l <- list(iris = iris,
          airquality = airquality,
          mtcars = mtcars)
names(l)
```

```
#> [1] "iris"       "airquality" "mtcars"
```

```
map2(l, names(l), ~ write.csv(x = .x, files = paste0(.y, ".csv")))
```

imap

Shortcut for `map2(l, names(l), ...)`.

- `.x` == element
- `.y` == names(element)

```
l %>% imap(~ write.csv(x = .x, files = paste0(.y, ".csv")))
```

`map_at` and `map_if`

Map on specific places or under specific conditions

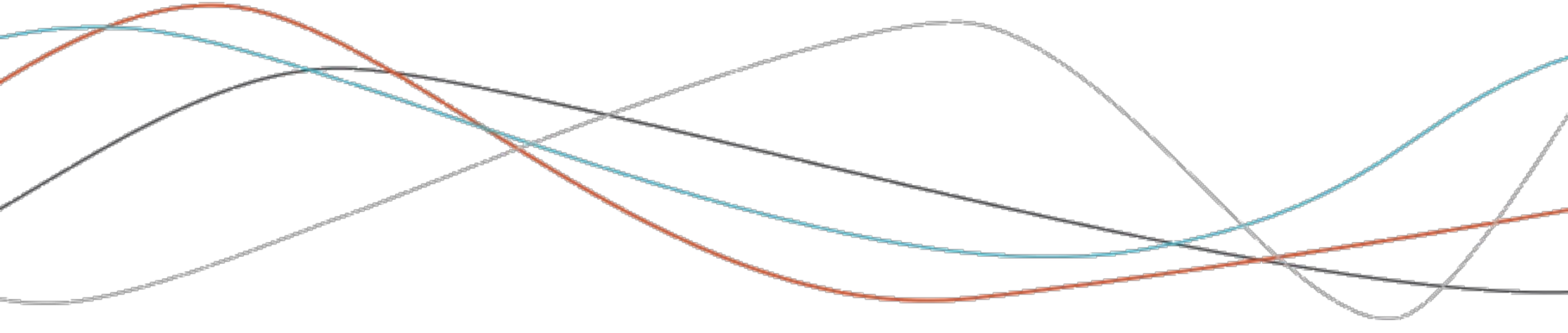
```
l <- list(thy = 1:10, art = letters, is = 1:100)

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

```
iris %>% keep(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?

```
iris %>% every(is.numeric)
```

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

```
iris %>%  some(is.numeric)
```

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

has_element(), detect(), detect_index()

has_element(): does .x contains .y?

```
iris %>%  has_element(iris$Sepal.Length)
```

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

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

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

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

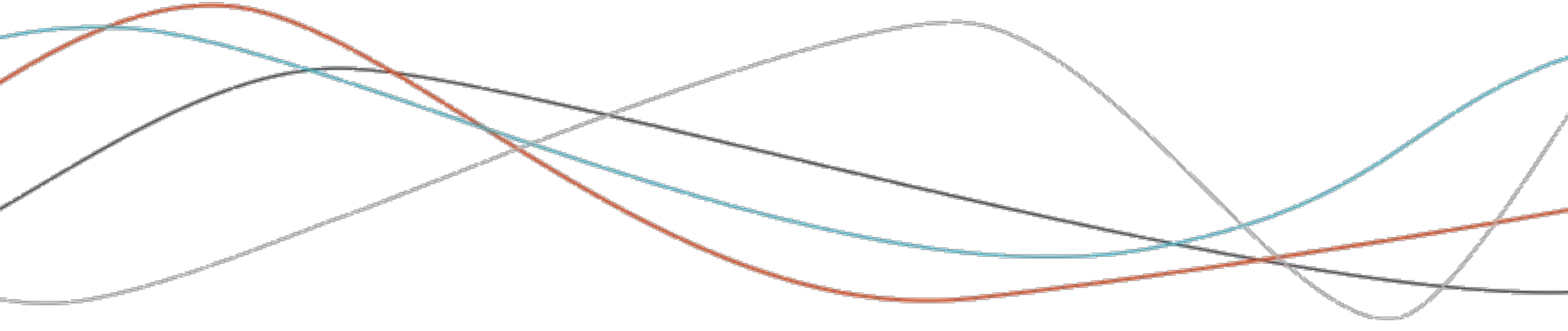
compact()

`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)

list("a", 2) %>%
  map(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)
l %>% map(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
```

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

```
#> $result
#> $result[[1]]
#> NULL
#>
#> $result[[2]]
#> [1] 0.6931472
#>
#> $result[[3]]
#> [1] 1.098612
#>
#>
#> $error
#> $error[[1]]
```

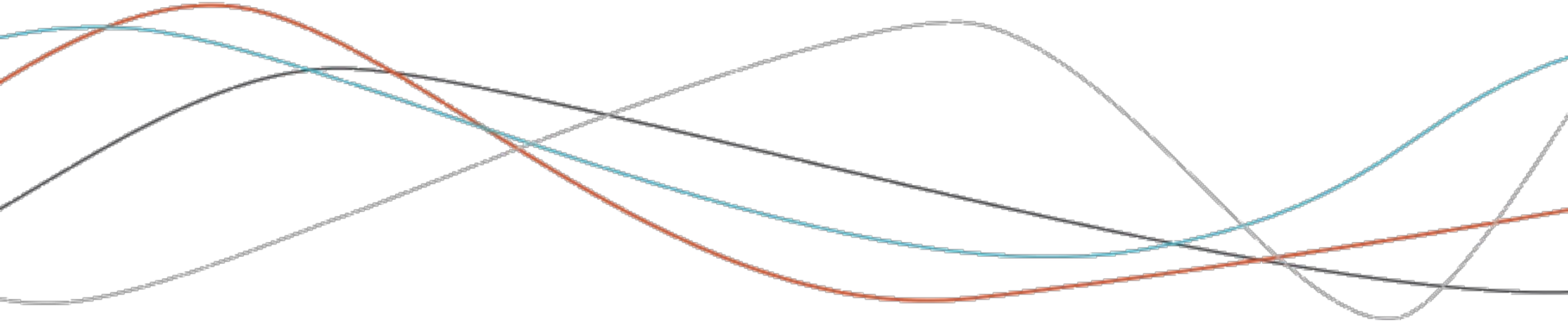
Handling adverb results

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

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

possible_log <- possibly(log, otherwise = NULL)

l %>% map(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

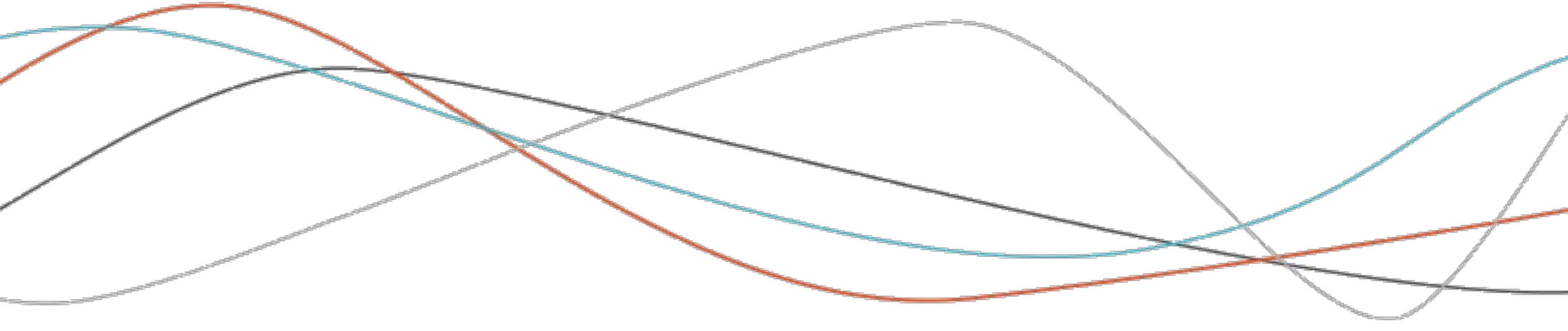
`partial()` & `compose()`

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>
```

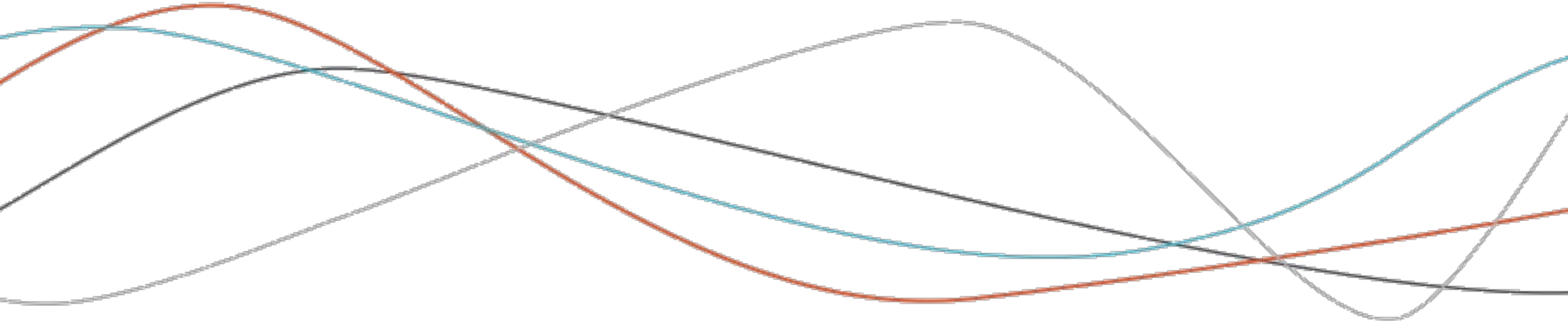
Example


```
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
```

Web scraping

Becoming a pirate with R



Web scraping

With web scraping, you're collecting data straight from the source code of a web page.

Examples :

- Scrap Wikipedia to get the content of html tables.
- Scrap the yellow pages to create a client base.
- Download all the cat pictures from a website (if you're into that).
- Watch eBay.
- Get a list of tweets.

The R package used to do web scraping is called `{rvest}` (to be pronounced as "harvest"). Created by Hadley Wickham, this package gives you a full toolbox to scrap the web.

```
install.packages("rvest")
```

Web scraping

It's quite easy to scrap a full website with R. It becomes more complex when you need to tidy the collected data, and to get precise information from the source code.

In an html page, elements can be indentified by:

- id : the id is a unique name of an element. It is supposed to be used only once on a page. You can recognize it as it is preceded by a `#`. For example `#my-id`.
- class : a class is used to identify several elements on a page and throughout a whole website. It is precedeed by a dot. For example `.my-class`.

```
<h1 id = "principal" class = "titre"> mon titre </h1>
```

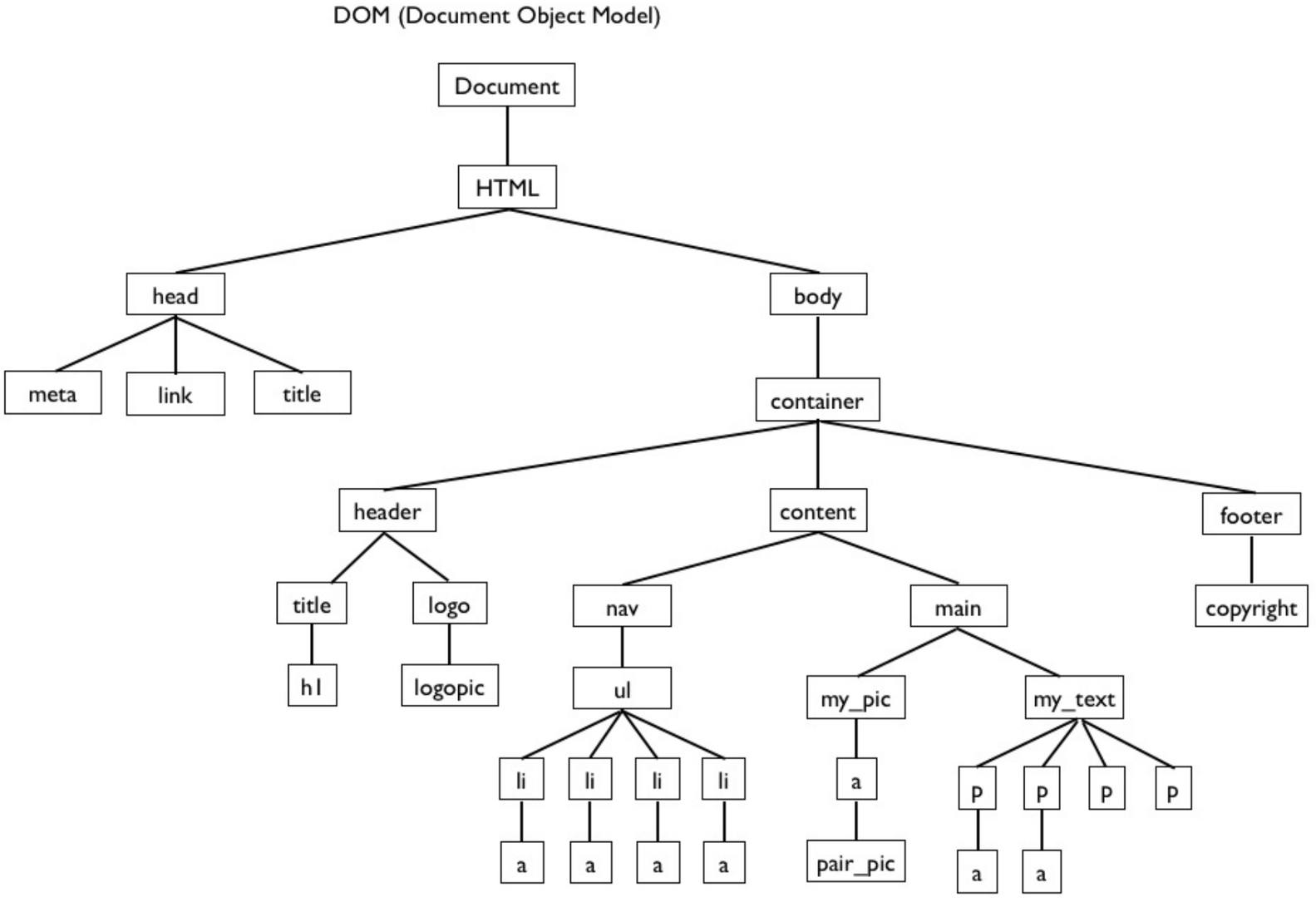
You can also refer to a specific element with its position inside the DOM (Document Object Model), which is the "tree structure" of the webpage. Using this path is more complex, but from time to time there is no other way but to do so.

Understanding DOM


```
readr::read_lines("https://cran.r-project.org/")
```

```
#>  [1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Frameset//EN\"
\"http://www.w3.org/TR/html4/frameset.dtd\">"
#>  [2] "<html>"
#>  [3] "<head>"
#>  [4] "<title>The Comprehensive R Archive Network</title>"
#>  [5] "<META HTTP-EQUIV=\"content-type\" CONTENT=\"text/html; charset=utf-8\">"
#>  [6] "<link rel=\"icon\" href=\"favicon.ico\" type=\"image/x-icon\">"
#>  [7] "<link rel=\"shortcut icon\" href=\"favicon.ico\" type=\"image/x-icon\">"
#>  [8] "<link rel=\"stylesheet\" type=\"text/css\" href=\"R.css\">"
#>  [9] "</head>"
#> [10] ""
#> [11] "<FRAMESET cols=\"1*, 4*\" style=\"border: none;\">"
#> [12] "<FRAMESET rows=\"120, 1*\">"
#> [13] "<FRAME src=\"logo.html\" name=\"logo\" frameborder=0>"
#> [14] "<FRAME src=\"navbar.html\" name=\"contents\" frameborder=0>"
#> [15] "</FRAMESET>"
#> [16] "<FRAME src=\"banner.shtml\" name=\"banner\" frameborder=0>"
#> [17] "<noframes>"
#> [18] "<h1>The Comprehensive R Archive Network</h1>"
#> [19] ""
```

Web scraping



Web scraping

You can get elements from the html source code, from right-clicking on an element and click 'Inspect', or with the Chrome extension SelectorGadget.

Web scraping – css selectors

Selector	Example	Example description
.class	.intro	Selects all elements with class="intro"
#id	#firstname	Selects the element with id="firstname"
*	*	Selects all elements
element	p	Selects all <p> elements
element,element	div, p	Selects all elements and all <p> elements
element element	div p	Selects all <p> elements inside elements
element>element	div > p	Selects all <p> elements where the parent is a element
element+element	div + p	Selects all <p> elements that are placed immediately after elements
element1~element2	p ~ ul	Selects every element that are preceded by a <p> element

Full list : http://www.w3schools.com/cssref/css_selectors.asp

Web scraping

In {rvest}, there is two families of functions:

extract

- `read_html()`
- `html_nodes()`
- `html_text()`, `html_attrs()`, `html_name()`
- `html_table()`

Simulate an html browser

- `html_session()`
- `jump_to()`, `follow_link()`
- `session_history()`

Web scraping

`read_html()`

The first function you need to use is `read_html()`, as it imports the content of a webpage in your R session.

```
library(rvest)
```

```
#> Loading required package: xml2
```

```
my_page <- read_html("http://thinkr.fr")
my_page
```

```
#> {xml_document}
#> <html class="no-js" lang="fr-FR">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
#> [2] <body class="home page-template-default page page-id-35625 do-etfw f ...
```

Web scraping

`html_nodes()`

This function is used to extract a specific element from the imported page. It takes as arguments the page, and a css selector (or a Xpath).

```
my_page %>% html_nodes(css = ".dt-blog-shortcode > .wf-cell")
```

```
#> {xml_nodeset (9)}
#> [1] <div class="wf-cell" data-date="2018-10-26T10:57:50+00:00" data-name ...
#> [2] <div class="wf-cell" data-date="2018-09-27T21:02:11+00:00" data-name ...
#> [3] <div class="wf-cell" data-date="2018-08-31T14:00:04+00:00" data-name ...
#> [4] <div class="wf-cell" data-date="2018-08-26T14:20:26+00:00" data-name ...
#> [5] <div class="wf-cell" data-date="2018-07-18T11:32:03+00:00" data-name ...
#> [6] <div class="wf-cell" data-date="2018-07-16T08:56:11+00:00" data-name ...
#> [7] <div class="wf-cell" data-date="2018-07-02T07:29:50+00:00" data-name ...
#> [8] <div class="wf-cell" data-date="2018-06-19T12:01:17+00:00" data-name ...
#> [9] <div class="wf-cell" data-date="2018-05-08T11:11:17+00:00" data-name ...
```

Web scraping

`html_text()`

When you only need to keep the text from the source code.

```
"https://rtask.thinkr.fr/" %>%
  read_html() %>%
  html_nodes("h4 a") %>%
  html_text()
```

```
#>  [1] "Shiny applications (design and deployment)"
#>  [2] "R server and infrastructure"
#>  [3] "Code and R scripts optimization"
#>  [4] "R Software Engineering"
#>  [5] "From SAS to R"
#>  [6] "Open-source contributions"
#>  [7] "RConsortium – Building an R Certification"
#>  [8] "A Tale of Two Shiny Apps: Google Auth & ShinyProxy"
#>  [9] "Shiny application in production with ShinyProxy, Docker and Debian"
#> [10] "Data visualisation pitfalls: how to avoid barbarplots ?"
#> [11] "Playing around with RStudio Package Manager"
#> [12] "Remaking ‘Luminance-gradient-dependent lightness illusion’ with R"
```

Web scraping

`html_attr()`

Used to collect the attributes.

```
"https://rtask.thinkr.fr/" %>%
  read_html() %>%
  html_nodes("h4 a") %>%
  html_attr("href")
```

```
#>  [1] "https://rtask.thinkr.fr/design-and-deployment-of-shiny-applications/"
#>  [2] "https://rtask.thinkr.fr/r-server-and-infrastructure-installation/"
#>  [3] "https://rtask.thinkr.fr/code-and-r-scripts-optimization/"
#>  [4] "https://rtask.thinkr.fr/developments-in-the-r-ecosystem/"
#>  [5] "https://rtask.thinkr.fr/migration-de-code-sas-vers-r/"
#>  [6] "https://rtask.thinkr.fr/open-source-contributions/"
#>  [7] "https://rtask.thinkr.fr/blog/rconsortium-building-an-r-certification/"
#>  [8] "https://rtask.thinkr.fr/blog/a-tale-of-two-shiny-apps-google-auth-
shinyproxy/"
#>  [9] "https://rtask.thinkr.fr/blog/shiny-application-in-production-with-
shinyproxy-docker-and-debian/"
#> [10] "https://rtask.thinkr.fr/blog/data-visualisation-pitfalls-how-to-avoid-
barbarplots/"
#> [11] "https://rtask.thinkr.fr/blog/rstudio-package-manager/"
```

Web scraping

`html_name()`

Returns the name of the elements.

```
"https://rtask.thinkr.fr/" %>%
  read_html() %>%
  html_nodes("h4") %>%
  html_name()
```

```
#>  [1] "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4" "h4"
```

Web scraping

`html_table()`

This function extract a table from a webpage:

```
'https://fr.wikipedia.org/wiki/Distance_de_Levenshtein' %>%
  read_html() %>%
  html_table(fill = TRUE)
```

```
#> [[1]]
#>       C H I E N S
#> 1   0 1 2 3 4 5 6
#> 2 N 1 0 0 0 0 0 0
#> 3 I 2 0 0 0 0 0 0
#> 4 C 3 0 0 0 0 0 0
#> 5 H 4 0 0 0 0 0 0
#> 6 E 5 0 0 0 0 0 0
#>
#> [[2]]
#>     C H I E N S
#> 1 N 1 1 1 1 0 1
#> 2 I 1 1 0 1 1 1
#> 3 C 0 1 1 1 1 1
#> 4 H 1 0 1 1 1 1
```

Web scraping

`html_session()`

Simulate the launch of an html session, as in a web browser.

```
library(rvest)
nav <- html_session('http://www.commitstrip.com/fr/2015/05/19/data-wars/')
```

`follow_link()`

Follows a link in your html session.

```
# Mimic a click on "Random" on the webpage you've just launched.
nav <- nav %>% follow_link("Random")
```

```
#> Navigating to /?random=1
```

Web scraping

`jump_to()`

Here, you simulate moving from your current page to another url (absolute path or relative).

```
nav <- nav %>% jump_to("https://thinkr.fr")
```

session_history()

Returns the navigation history from the current session :

```
session_history(nav)
```

```
#>    http://www.commitstrip.com/fr/2015/05/19/data-wars/
#>    http://www.commitstrip.com/fr/2014/12/11/lets-talk-about-code-shall-we/
#> - https://thinkr.fr/
```

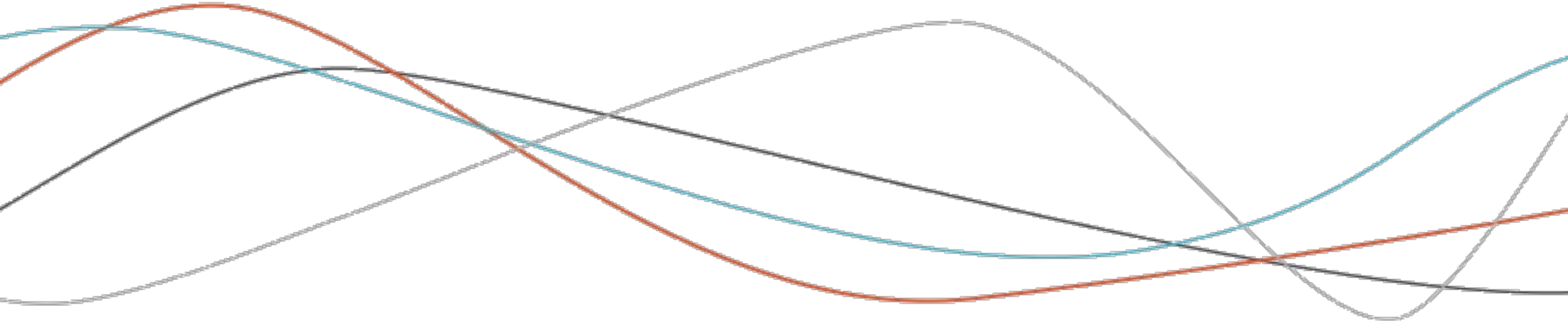
Web scraping

As every package in the tidyverse, {rvest} is best used with the pipe: `%>%`

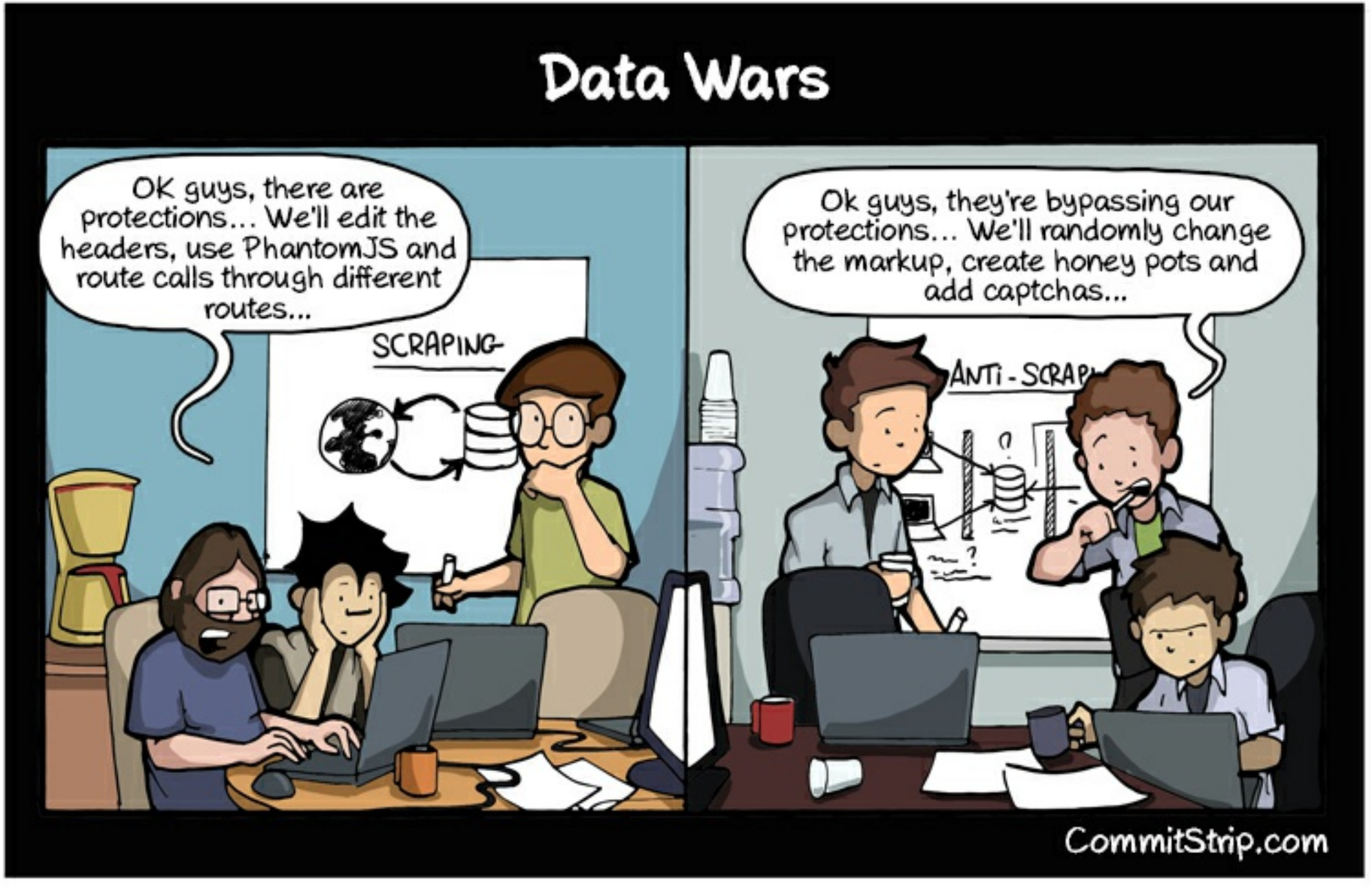
```
library(tidyverse)
library(rvest)
'http://www.commitstrip.com/fr/2015/05/19/data-wars/' %>%
  read_html() %>%
  html_nodes(".size-full") %>%
  html_attr("src") %>%
  browseURL()
```

Ethical Web scraping

From Pirate to Corsair



Web scraping



Ethical Web scraping

It's not because you can, that you should, or are allowed to.

=> Are you allowed to scrape a website?

=> Don't overflow the website with hundreds of connexions.

About robots.txt

`robots.txt` are webfiles designed to tell what a robot is allowed to do on a website :

```
readr::read_lines("https://thinkr.fr/robots.txt")
```

```
#>  [1] "# This virtual robots.txt file was created by the Virtual Robots.txt
WordPress plugin: https://www.wordpress.org/plugins/pc-robotstxt/"
#>  [2] "User-agent: *"
#>  [3] "Disallow: /wp-login.php"
#>  [4] "Disallow: /*?"
#>  [5] ""
#>  [6] "Disallow: /*/comments"
#>  [7] "Disallow: */trackback"
#>  [8] "Disallow: /*/feed"
#>  [9] ""
#> [10] "Disallow: /cgi-bin"
#> [11] "Disallow: /*.php$"
#> [12] "Disallow: /*.inc$"
#> [13] "Disallow: /*.gz"
#> [14] "Disallow: /*.cgi"
#> [15] ""
#> [16] "Allow: /*css?*"
#> [17] "Allow: /*js?*"
```

Understand robots.txt

- `User-agent: *`: which robots are concerned (here, all)
- `Disallow: /page-a/`: disallow robots from going to `/page-a`
- `Disallow: /page*`: disallow robots from going to every url starting with `page`

{robotstxt}

```
# install.packages("robotstxt")
library(robotstxt)
robot <- robotstxt("https://thinkr.fr")
robot$text
```

```
#> # This virtual robots.txt file was created by the Virtual Robots.txt WordPress
plugin: https://www.wordpress.org/plugins/pc-robotstxt/
#> User-agent: *
#> Disallow: /wp-login.php
#> Disallow: /*?
#>
#> Disallow: /*/comments
#> Disallow: */trackback
#> Disallow: /*/feed
#>
#> Disallow: /cgi-bin
#> Disallow: /*.php$
#> Disallow: /*.inc$
#> Disallow: /*.gz
#> Disallow: /*.cgi
#>
#> Allow: /*.css?*
```

{robotstxt}

Use `robot$check()`

```
robot$check(path = "/contact")
```

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

```
robot$check(path = "/wp-login.php")
```

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

{robotstxt}

Use `paths_allowed()`

```
paths_allowed("http://google.com/")
```

```
#>
 google.com                          No encoding supplied: defaulting to UTF-
8.

#> [1] TRUE
```

```
paths_allowed("http://google.com/search")
```

```
#>
 google.com                          No encoding supplied: defaulting to UTF-
8.

#> [1] FALSE
```

{robotstxt}

```
scrape_this <- function(url){
  if ( paths_allowed(url) ){
    httr::GET(url)
  } else {
    message("Can't scrape this website.")
  }
}
scrape_this("http://google.com/search")
```

```
#>
 google.com                          No encoding supplied: defaulting to UTF-
8.
#>
#>
#> Can't scrape this website.
```

{memoise}

- Situation: you want to scrape something multiple times.
- Issue: you don't want to flood the server / website.
- Solution: memoise your calls!

```
library(memoise)
```

{memoise}

`{memoise}` is caching the output of a function when it's called in with the same parameters.

```
library(httr)
```

```
#>
#> Attaching package: 'httr'

#> The following object is masked from 'package:memoise':
#>
#>     timeout
```

```
m_GET <- memoise(GET)
a <- m_GET("http://google.com")
a$date
```

```
#> [1] "2018-11-25 21:36:18 GMT"
```

```
Sys.sleep(5)
b <- m_GET("http://google.com")
b$date
```

{memoise}

Memoising your `read_html()` & `GET()`, so that you don't call again and again the same URL.

```
library(rvest)
```

```
#> Loading required package: xml2
```

```
read_html_safe <- memoise(read_html)
read_html_safe("http://thinkr.fr")
```

```
#> {xml_document}
#> <html class="no-js" lang="fr-FR">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
#> [2] <body class="home page-template-default page page-id-35625 do-etfw f ...
```

{polite}

> Be polite on the web.

```
# remotes::install_github("dmi3kno/polite")
library(polite)
library(rvest)
thinkr <- bow("http://thinkr.fr", force = TRUE)
thinkr
```

```
#> <polite session> http://thinkr.fr
#>      User-agent: polite R package - https://github.com/dmi3kno/polite
#>      robots.txt: 14 rules are defined for 3 bots
#>     Crawl delay: 5 sec
#>   The path is scrapable for this user-agent
```

{polite}

```
scrape(thinkr)
```

```
#> {xml_document}
#> <html class="no-js" lang="fr-FR">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
#> [2] <body class="home page-template-default page page-id-35625 do-etfw f ...
```

{polite}

```
google_search <- bow("http://google.com/search", force = TRUE)
google_search
```

```
#> <polite session> http://google.com/search
#>      User-agent: polite R package - https://github.com/dmi3kno/polite
#>      robots.txt: 269 rules are defined for 3 bots
#>     Crawl delay: 5 sec
#>   The path is not scrapable for this user-agent
```

{polite}

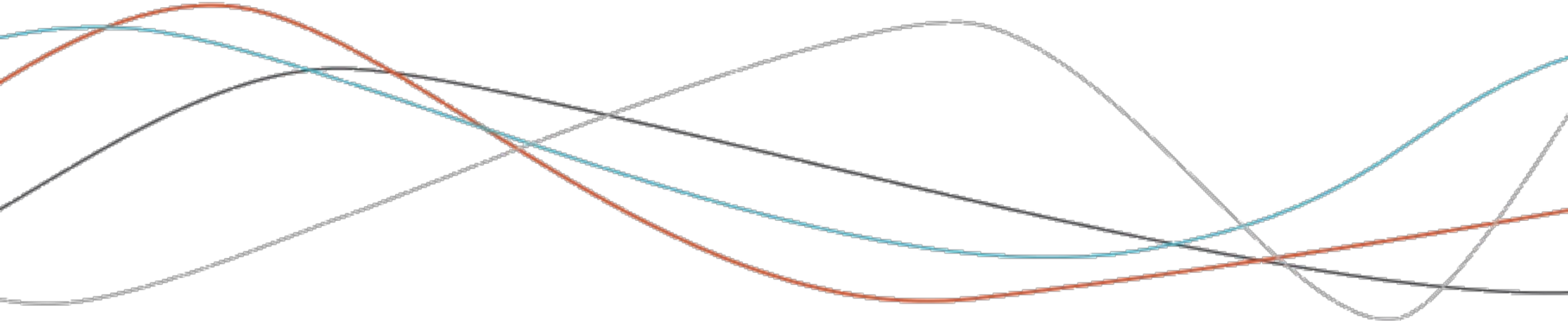
```
scrape(google_search)
```

```
#> No scraping allowed here!
```

```
#> NULL
```

Calling a web API from R

Hello, is it me you're looking for?



What is an API

Short for of Application Programming Interface, an API is a series of protocols, routines, and tools that allow two pieces of software to communicate:

- a software A (the server)
- a software B (the client)

All communications between two softwares are to be considered as using an API. Today, we'll see how we can retrieve data from the web inside R, using an architecture called REST (REpresentational State Transfer).

A REST API is a web-based interface between you (aka your computer as a browser) and a database. The API comes with a language, and a set of tools you need to be familiar with if you want to be granted access to the database.

Anatomy of a call

Most web API servers are based on http, and typically return JSON or XML. You can access them with custom calls. These are composed of :

- URI, an Uniform Resource Identifier (which is an URL on the web).
- a method
- headers
- a body

In return, you'll get a response containing :

- a status code: 200 is ok, 404 is not. (See the full list of status code)
- a header with metadata.
- a body, which is the content of the response.

Methods

There are several methods you can use to communicate with an API. You can `PUT` elements, `DELETE` contents, `POST` data... but mainly you'll need to use the `GET` method, which is the basic method to retrieve data from the database to your R session.

Some API are designed to process information: with these one, you'll need to use the `POST` method. For example, if you want to do image recognition on images, you'll `POST` elements on the server.

{httr}

The R package designed to communicate through http is called {httr}. It was developped by Hadley Wickham.

The simplest call you can make is `GET`, which gets the content located on the provided url (remember that a url is a special kind of uri).

```
library(httr)
GET(url = "http://www.google.com")
```

R packages for calling API

A lot of packages has been developped to make API calls from R. For example:

- {Rfacebook}
- {rtweet}
- {gmailr}
- {RGoogleAnalytics}
- {rpinterest}
- {searchConsoleR}
- {rtimes}
- {trelloR}

And many more...

Building your own calls

API calls need an URL. On a simple API, you simply need to `GET` this URL, and the body from the response is the data you need. 90% of the time, the content is returned in JSON, which is a notation based on javascript. You'll need the {jsonlite} package to convert this format to an R list.

```
library(jsonlite)
```

You can either call the url inside `fromJSON`, or pass the result from your `httr` call. The advantage of using the second method is that you can check which status call is returned by your call.

```
fromJSON("http://xkcd.com/1128/info.0.json")
# or with error handling
res <- GET(url = "http://xkcd.com/1234/info.0.json")
if(res$status_code != 200){
  print("API error")
} else {
  content(res)
}
```

Building your own API calls

If you need to scale your API calls, you'll have to write a function that takes as an argument the part of the url which is to be changed. For example, in `"http://xkcd.com/1128/info.0.json"`, the changing part is the number of the comic.

Then, the function will paste the generic url from the API (which you'll find in every API documentation) and the parameters provided by the user. The most useful package to do this pasting is {glue}, which parses a string and replace everything between brackets with the objects from the environment.

```
library(glue)
string <- Sys.Date()
glue("Today is the {string}")
```

The query string

API allow you to build a query with parameters. That is to say special filters to specifiy what type data you want to retrieve: from date, the number of results, an access token, words...

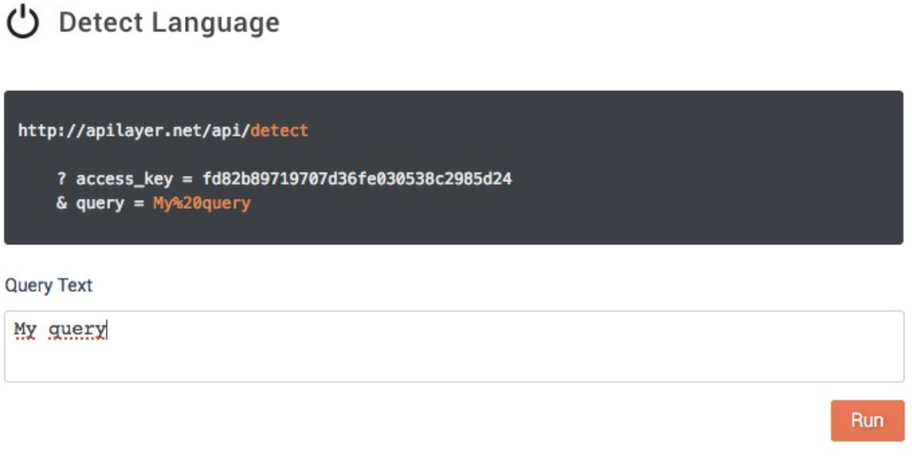
These parameters are to be put at the end of the url you're sending to the server. The parameters belong to the end of the url, after a question mark. Multiple parameters are possible, and need to be separated with an ampersand (`&`).

Example with language layer API:

```
"http://api.url.com&query=
my%20query&since=20170801"
```

Don't forget that some special characters need to be converted in ASCII to be read in a url. E.g.," " (space) has to be converted as `%20`.

You can get these parameters if you dig in the documentation :



Headers

Some more advanced API need specific information in the header. These parameters can be passed in the `httr` call with the `add_headers` function, inside your `GET`.

```
POST(url,
     add_headers(.headers =
                   c("Content-Type" = "application/json",
                     "Ocp-Apim-Subscription-Key" = api_key)))
```

You can easily spot these parameters if you look in the documentation of the API, specifically for the curl (client URL request library) calls.

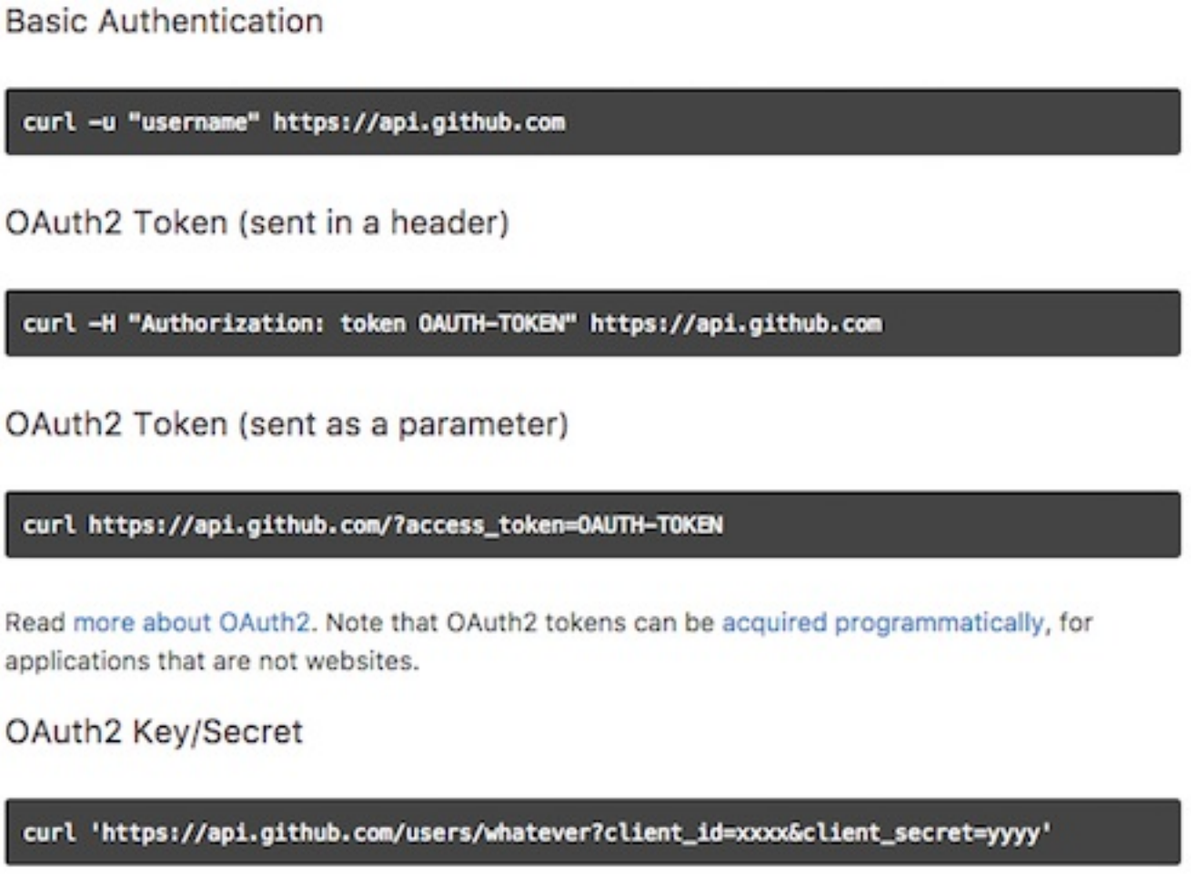
```
curl -v -X POST "https://westcentralus.api.cognitive.microsoft.com/vision/v1.0/analyze?visual
-H "Content-Type: application/json"
-H "Ocp-Apim-Subscription-Key: {subscription key}"
```

As you can see, every header element is an element of the `.headers` list.

`{httr}` is a wrapper around curl, and at the end of the day the {httr} call you'll make is quite similar to the format of the curl call (as you can see above). In general, API documentation provide curl examples you can use, so feel free to use them as examples to build your `{httr}` call.

Mix models

Many API accept either the header based authentification or the query string authentification:



Full {httr} calls

Instead of pasting parameters to create an url, you can also use {httr} as a query builder. For example, we can parse this curl call (taken from the documentation of the Geo API):

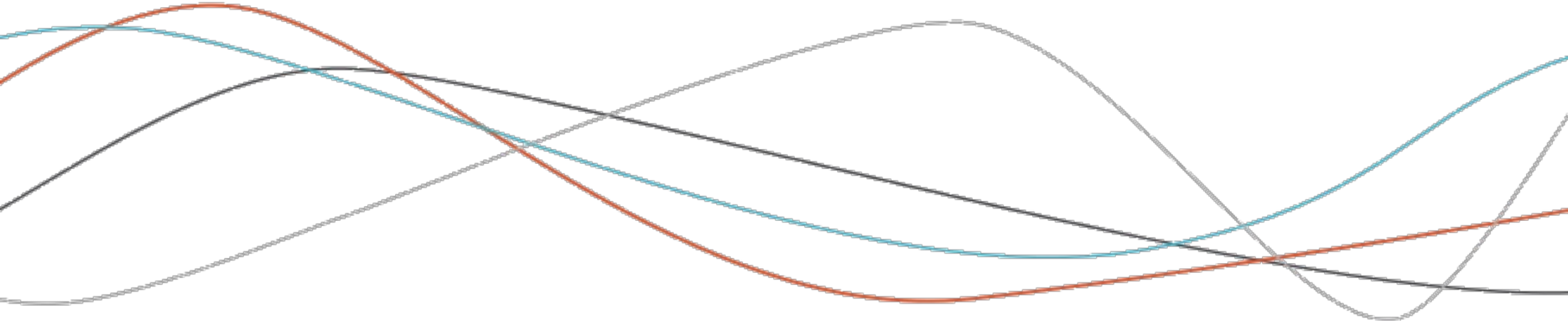
```
curl -X GET --header 'Accept: application/json'
'https://geo.api.gouv.fr/communes?
codePostal=35000&fields=codesPostaux&format=json&geometry=centre'
```

As:

```
GET(url = "https://geo.api.gouv.fr/communes",
    add_headers('Accept' = "application/json"),
    query = list(codePostal= 35000,
                 field = "codesPostaux",
                 format = "json",
                 geometry = "centre")) %>% content()
```

Text-Mining with R

The Lost Word



Text-Mining

Text mining is more than just string manipulation: you're trying to automatically extract information from a text.

There are many ways to do this with R (see : https://cran.r-project.org/web/views/NaturalLanguageProcessing.html).

We will see here how to do text-mining with{tidytext}, a package from the tidyverse galaxy.

```
library(tidyverse)
```

```
#> ── Attaching packages ─────────────── tidyverse 1.2.1 ──

#>    ggplot2 3.1.0       purrr   0.2.5
#>    tibble  1.4.2       dplyr   0.7.8
#>    tidyr   0.8.1       stringr 1.3.1
#>    readr   1.1.1       forcats 0.3.0

#> ── Conflicts ──────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
```

one-token-per-row

As with (almost) all the tools of the tidyverse, our dataset must be in rectangular format (in table form).

The first step is to move from a "classic" table format, with each textual observation in a column to a "one-token-per-row" format: a token is a unit of language, which can be one or more words, or even a sentence, a paragraph...

```
download.file("https://github.com/ThinkR-
open/datasets/raw/master/%23RStudioConf.RDS", "data.RDS")
tweets <- readRDS("data.RDS")
unlink("data.RDS")
```

one-token-per-row

```
tweets %>% select(screen_name, text)
```

```
#> # A tibble: 5,055 x 2
#>    screen_name     text
#>    <chr>           <chr>
#>  1 grod_rf         "RT @dataandme:    Check it, @ajmcoqui's \"Debugging in…
#>  2 kevinmontrose   RT @kara_woo: Aftereffects of #rstudioconf: I spent fi…
#>  3 JoshIzzard6     @juliasilge Hi Julia really enjoyed having lunch with …
#>  4 ellisvalentiner "RT @d4tagirl:    Great analysis of #RStudioConf tweets…
#>  5 LuisDVerde      "RT @cpsievert: Nice collection of links to #rstudioco…
#>  6 umairdurrani87  Can we hope for a #rstudioconf in Canada in near futur…
#>  7 rstatsbot1234   "RT @LuisDVerde: .@wmlandau @krlmlr\n#rstats #rstudioc…
#>  8 onianiaa        "RT @drob: Among the points in my #rstudioconf talk is…
#>  9 onianiaa        "RT @drob: Slides for my #rstudioconf talk, \"Teach th…
#> 10 neuromusic      RT @kara_woo: Aftereffects of #rstudioconf: I spent fi…
#> # ... with 5,045 more rows
```

one-token-per-row

We'll need to use the `unnest_tokens(()` function.

You need to write it as `unnest_tokens(table, output_col, input_col)` :

```
corpus <- tweets %>%
  unnest_tokens(word, text)
corpus %>% select(screen_name, word)
```

```
#> # A tibble: 104,879 x 2
#>    screen_name word
#>    <chr>       <chr>
#>  1 grod_rf     rt
#>  2 grod_rf     dataandme
#>  3 grod_rf     check
#>  4 grod_rf     it
#>  5 grod_rf     ajmcoqui's
#>  6 grod_rf     debugging
#>  7 grod_rf     in
#>  8 grod_rf     rstudio
#>  9 grod_rf     slides
#> 10 grod_rf     and
#> #... with 104,869 more rows
```

one-token-per-row

You've now got a column you can manipulate with `{dplyr}` and friends:

```
library(dplyr)
corpus %>%
  count(word) %>%
  top_n(10) %>%
  arrange(desc(n))
```

```
#> Selecting by n

#> # A tibble: 10 x 2
#>    word            n
#>    <chr>       <int>
#>  1 rstudioconf  5078
#>  2 rt           3306
#>  3 https        3114
#>  4 to           3100
#>  5 the          3050
#>  6 t.co         2968
#>  7 of           1638
#>  8 for          1613
#>  9 a            1526
```

Remove stop words

You can use join functions to removes "stop words": words which are not meaningful (I, you, the, a...).

`{tidytext}` has three built in datasets to to that.

```
tidytext::stop_words
```

```
#> # A tibble: 1,149 x 2
#>    word        lexicon
#>    <chr>       <chr>
#>  1 a           SMART
#>  2 a's         SMART
#>  3 able        SMART
#>  4 about       SMART
#>  5 above       SMART
#>  6 according   SMART
#>  7 accordingly SMART
#>  8 across      SMART
#>  9 actually    SMART
#> 10 after       SMART
#> # ... with 1,139 more rows
```

Remove stop words

```
stop_words_filtered <- stop_words %>% filter(lexicon == "SMART")
corpus %>%
  count(word) %>%
  anti_join(stop_words_filtered) %>%
  top_n(5) %>%
  arrange(desc(n))
```

```
#> Joining, by = "word"

#> Selecting by n

#> # A tibble: 5 x 2
#>   word            n
#>   <chr>       <int>
#> 1 rstudioconf  5078
#> 2 rt           3306
#> 3 https        3114
#> 4 t.co         2968
#> 5 rstats       1183
```

Remove stop words

You can add your own stop-words:

```
stop_words_custom <- tibble(word = c("rt", "https","http", "t.co", "amp"))
corpus %>%
  count(word) %>%
  anti_join(stop_words_filtered) %>%
  anti_join(stop_words_custom) %>%
  top_n(5) %>%
  arrange(desc(n))
```

```
#> Joining, by = "word"
#> Joining, by = "word"

#> Selecting by n

#> # A tibble: 5 x 2
#>   word            n
#>   <chr>       <int>
#> 1 rstudioconf  5078
#> 2 rstats       1183
#> 3 rstudio       906
#> 4 talk          755
```

Bi-grams

```
stop_words_custom <- bind_rows(stop_words_custom, stop_words_filtered)
tweets %>%
  select(screen_name, text) %>%
  unnest_tokens(bigram, text, token = "ngrams", n = 2) %>%
  count(bigram) %>%
  top_n(5) %>%
  arrange(desc(n))
```

Bi-grams with cleaning

```
stop_words_custom <- bind_rows(stop_words_custom, stop_words_filtered)
tweets %>%
  select(screen_name, text) %>%
  unnest_tokens(bigram, text, token = "ngrams", n = 2) %>%
  separate(bigram, c("word1", "word2"), sep = " ") %>%
  filter(!word1 %in% stop_words_custom$word) %>%
  filter(!word2 %in% stop_words_custom$word) %>%
  unite(bigram, word1, word2, sep = " ") %>%
  count(bigram, sort = TRUE) %>%
  top_n(10)
```

Bi-grams with cleaning

```
#> Selecting by n
```

```
#> # A tibble: 10 x 2
#>    bigram                    n
#>    <chr>                 <int>
#>  1 rstudioconf talk        417
#>  2 rstudioconf rstats      265
#>  3 rstudio workshop        264
#>  4 rstudioconf intro       213
#>  5 comparison cheatsheet   211
#>  6 syntax comparison       211
#>  7 good time               182
#>  8 important books         178
#>  9 online fo               177
#> 10 data science            161
```

Sentiment analysis

`{tidytext}` has tables for sentiment analysis.

```
get_sentiments("bing")
```

```
#> # A tibble: 6,788 x 2
#>    word        sentiment
#>    <chr>       <chr>
#>  1 2-faced     negative
#>  2 2-faces     negative
#>  3 a+          positive
#>  4 abnormal    negative
#>  5 abolish     negative
#>  6 abominable  negative
#>  7 abominably  negative
#>  8 abominate   negative
#>  9 abomination negative
#> 10 abort       negative
#> # ... with 6,778 more rows
```

```
tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words_custom) %>%
```

Sentiment analysis

`{tidytext}` has tables for sentiment analysis.

```
get_sentiments("nrc")
```

```
#> # A tibble: 13,901 x 2
#>    word        sentiment
#>    <chr>       <chr>
#>  1 abacus      trust
#>  2 abandon     fear
#>  3 abandon     negative
#>  4 abandon     sadness
#>  5 abandoned   anger
#>  6 abandoned   fear
#>  7 abandoned   negative
#>  8 abandoned   sadness
#>  9 abandonment anger
#> 10 abandonment fear
#> # ... with 13,891 more rows
```

```
tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words_custom) %>%
```

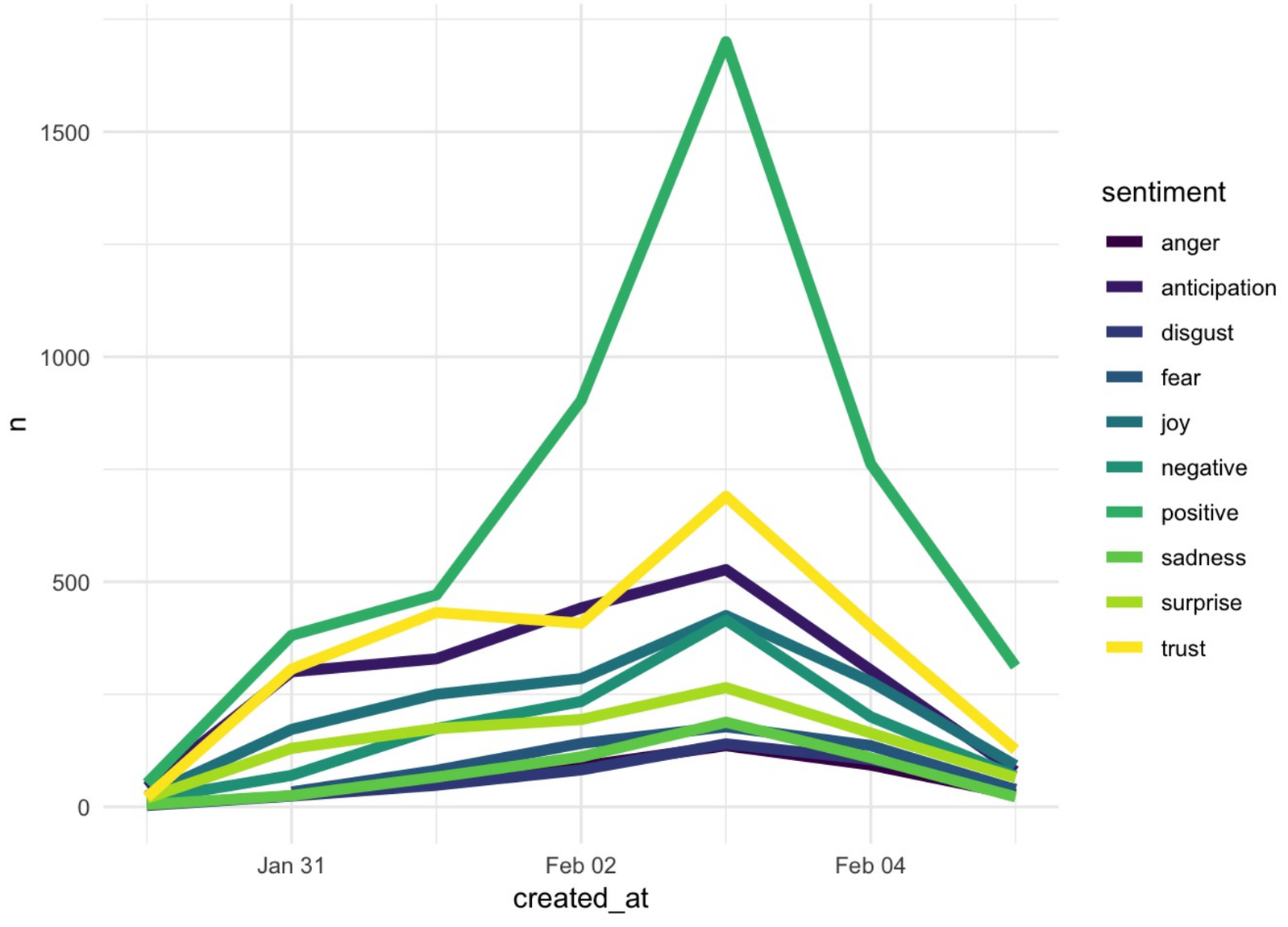
Analyse de sentiments

```
tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words_custom) %>%
  inner_join(get_sentiments("nrc")) %>%
  mutate(created_at = lubridate::floor_date(created_at, unit = "hour"))
%>%
  group_by(created_at) %>%
  count(sentiment)
```

Analyse de sentiments

```
library(ggplot2)
tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words_custom) %>%
  inner_join(get_sentiments("nrc")) %>%
  mutate(created_at = as.Date(created_at)) %>%
  group_by(created_at) %>%
  count(sentiment) %>%
  filter( between(created_at, as.Date("2018-01-30"), as.Date("2018-02-
05")) ) %>%
  ggplot() +
  aes(created_at, n, group = sentiment, color = sentiment) +
  geom_line(size = 2) +
  scale_color_viridis_d() +
  theme_minimal()
```

Analyse de sentiments



Colin Fay

colin@thinkr.fr

+33 (0)6.24.21.32.18