The R Langage - day 4

11 octobre 2017

Go further with functions

Advanced functions - definitions

All R functions have three parts:

- the `body()`, the code inside the function,
- the `formals()`, the list of arguments which controls how you can call the function,
- the `environment()`, the “map” of the location of the function’s variables.

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

```
#> function(x,y){x+y}
#> <environment: 0x0000000020861de0>
```

```
body(my_sum)
formals(my_sum)
environment(my_sum)
```

Advanced functions – personnalised class

R allows you to attribute a class or (several ones) to an object

```
x <- "pikachu"
class(x)
```

```
#> [1] "character"
```

```
class(x) <- "pokemon"
x
```

```
#> [1] "pikachu"
#> attr(,"class")
#> [1] "pokemon"
```

`inherits()` tests it:

```
inherits(x,"pokemon");inherits(x,"character")
```

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

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

Advanced functions – function overloading

It is possible to overload functions to adapt them to a class. `UseMethod()` does the job in the body of the function. Thus, when calling a function on a specific object, R is going to try something like :

```
function_name.object_class()
```

If no such function exists `function_name.default()` is called:

```
mydemo <- function(x,...){ UseMethod("mydemo")}
mydemo.default <- function(x,...){ message("default") }
mydemo.factor <- function(x,...){ message("factor")}

mydemo(iris)
```

```
#> default
```

```
mydemo(iris$Species)
```

```
#> factor
```

Advanced functions – function overloading

Some functions like `summary`, `plot`... are using this behavior when called. For example, `print()` on a date objects calls `print.Date()`:

```
print(Sys.Date())
print.Date(Sys.Date())
```

When implemented this way, such functions are masked from the user. `getAnywhere(print.htest)` is the way to access it's source code.

Question

Build a function that, when called on a class `pokemon` object, will elegantly display "This is a pokemon and it's name is ..."

Advanced functions – dots

It's possible to compute a function that take an infinite numbers of parameters, dots : `...`

`dots <- list(...)` will generate a list inside the body of the function

```
shopping_list<- function(...){
  dots <- list(...)
  dots
}
shopping_list(cheese = 2, lettuce = 1, tomato = 3)
formals(shopping_list)
```

Advanced functions – parameters testing

When writing a function, R doesn't check if the user is reasonnable. Defining function `my_sum()`...

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

...won't prevent the user to misuse the function and getting unexplicit error messages :

```
my_sum(1,3)
my_sum(iris,4)
my_sum(iris,ls)
```

`{assertthat}` is here to simplify the task of controling what the user is doing and inform the user

Advanced functions – parameters testing

2 main function in `{assertthat}` will do the job :

- `assert_that()` : throws an error
- `validate_that()` : returns a character string

```
assert_that(is.character(1))
assert_that(nrow(iris) == 39)
assert_that(is.dir("a_path/a_file/"))

validate_that(is.character(1))
validate_that(nrow(iris) == 39)
validate_that(is.dir("a_path/a_file/"))
```

Advanced functions – parameters testing

Other useful functions from `{assertthat}` :

```r
library(assertthat)
ls('package:assertthat')
```

```
#>  [1] "%has_args%"    "%has_attr%"    "%has_name%"    "are_equal"
#>  [5] "assert_that"   "has_args"      "has_attr"      "has_extension"
#>  [9] "has_name"      "is.count"      "is.date"       "is.dir"
#> [13] "is.error"      "is.flag"       "is.number"     "is.readable"
#> [17] "is.scalar"     "is.string"     "is.time"       "is.writeable"
#> [21] "noNA"          "not_empty"     "on_failure"    "on_failure<-"
#> [25] "see_if"        "validate_that"
```

Advanced functions – parameters testing

Build then your own customised error messages:

```
library(lubridate)

is_leap_year <- function(x){
    assert_that(is.numeric(x)|is.Date(x)|is.POSIXt(x))
    leap_year(x)}

on_failure(is_leap_year) <-  function(call, env) {
  paste0(deparse(call$x), "is not a leap year")}

assert_that(is_leap_year(2012))

assert_that(is_leap_year(2013))
```

Question

Build a function that saves a data.frame in a csv file and also outputs the full path of where the csv has been saved. Use `{assertthat}` to control the `formals()`. The function should be able to take all the arguments of `write.csv2()`

```
# should work :
iris %>% save_as_csv("output.csv") %>% browseURL()

# this instructions give explicit error
iris %>% save_as_csv("output.xlsx") %>% browseURL()
NULL %>% save_as_csv("output.csv") %>% browseURL()
iris %>% save_as_csv("C://doesnotexist/pas/out.csv") %>% browseURL()
iris %>% save_as_csv("C://Windows/System/out.csv") %>% browseURL()
```

About paths

```
my_file <- "c://Program Files/R/R-3.3.2/bin/Rscript.exe"

basename(my_file) # filename
dirname(fichier) # directory name
normalizePath(fichier) # OS compatible complete path

my_file2 <- "output.csv"
normalizePath(my_file2)

file.path("directory_1","directory_2")# builds a path
```

Answer

Your own personnal package

Reach out and touch faith

Create your own packages: tools

To create a package, we'll need:

- RStudio.
- `{devtools}`, to save us time.
- `{roxygen2}`, to handle documentation.
- Rtools.exe (only on Windows).

Rtools is available here: https://cran.r-project.org/bin/windows/Rtools/

You can check with `devtools::find_rtools()` that Rtools is well installed. This tool has everything needed to compile C++, and other useful features.

```r
install.packages(c("devtools", "roxygen2"))
```

Then go to `new project > new directory > Rpackage`. Then, choose a package name and a directory. Do not put spaces or underscore in the name of the project/package

Making your own packages - files and folders

At the heart of your package, you'll find several folders and files. This organization is standardized:

- DESCRIPTION: the general description of the package: why to use this package, developer, license …
- NAMESPACE: describes how your package connects with other packages.
- R/ folder: contains the source code of your functions. As a rule of thumb, you'll have one file per big function: function1.R, function2.R.
- man/ folder: contains the documentation of each functions: function1.Rd, function2.Rd.

The NAMESPACE file should not be edited by hand (`{roxygen2}` will do it for us).

The man folder will be populated automatically thanks to `{roxygen2}`.

The file DESCRIPTION is to be edited by hand (but again, `{devtools}` can help us).

It is up to you to put your functions in R directory.

Making your own packages - files and folders

In the folder R, you'll put functions which are considered as "clean", that is to say:

- They do not call `library()` or `require()`: the dependencies are handled elsewhere in the `NAMESPACE` file.
- Must not modify the `options()` or parameters of the user.
- Must not use `source()` to call code.
- Do not call `setwd()`.

Making your own packages - roxygen2

In `build > configure build tools`, make sure to check "generate documentation with roxygen", and all the boxes in the options window that opens. The documentation will then automatically update using `{roxygen2}`

Making your own packages

```
my_fun <- function(x, y){

  return(x+y)

}
```

-> This is an example function, to be put in the R folder, saved in a `my_fun.R` file.

You can put several functions in a `single.R`. But you'll have to find the right balance between making a `file.R` by function and putting everything in a `single.R`.

Classically the "big" functions have their `own.R` and the "small" function are grouped by theme in a `same.R`.

Making your own packages - doc

```
#' Title
#'
#' Description
#' @param x my param
#' @param y my second param
#' @return this return x+y
#' @examples
#' my_fun(4,5)
#' my_fun(5,9)
#' @export

my_fun <- function(x, y){

  return(x+y)

}
```

-> comments starting with `#'` will be read by {roxygen2} to generate the documentation. (`ctrl + alt + shift + R` generates this skeleton).

Making your own packages - doc

Help can contain different styles and can point to other functions or external links.

```
Style :
\emph{italique}
\strong{gras}
\code{r_function_call(with = "arguments")}, \code{NULL}, \code{TRUE}
\pkg{nom_du_package}

Link :
\code{\link{function}}: function from another package
\code{\link[MASS]{stats}}: function from another package
\link[ = dest]{name}: link to dest.

URL :
\url{http://google.com}.
\href{http://google.com}{google}.
\email{vincent@@thinkr.fr} (with a double @)
```

Since roxygen2 version 6, markdown is supported: https ://cran.r-project.org/web/packages/roxygen2/vignettes/markdown.html

Making your own packages - datasets

In a package, it may be useful to put sample data sets. `iris` for example.

```
mydata <- iris
devtools::use_data(mydata)
```

When it comes to documentation, you should create a `mydata_doc.R` file in the R/ folder:

```
#' My dataset
#'
#' @name mydata
#' @docType data
#' @author Vincent \email{vincent@@thinkr.fr}
#' @references \url{google.com}{a link}
#' @keywords data
NULL
```

Once the package is installed and loaded, the user can type `data(mydata)` to load the sample dataset.

Making your own packages - datasets

Good practice: create a `data-raw` folder at the root of the package, and put here the script you used to generate the sample dataset.

To create such a folder, and configure the package so that this folder ignored when the package is built, you can use:

```
devtools::use_data_raw()
```

You'll then be able to update your datasets without losing the original ones.

Making your own packages - good practices

It's a good habit to create a `devtools_history.R` files inside the `data-raw` folder. In this file you could track every `devtools::` you made for creating the package.

This is for example the `devtools_history.R` file from {fcuk}

```r
devtools::use_data_raw()
devtools::use_package("stringdist")
devtools::use_package("purrr")
devtools::use_package("magrittr")
devtools::use_package("tibble")
devtools::use_package("rstudioapi")
devtools::use_vignette("poc")
devtools::use_vignette("fcuk")
devtools::build_vignettes()
devtools::use_test("regex")
devtools::use_test("get_all_objets_from_r")
devtools::use_test("error_correction_propostion")
devtools::use_test("error_analysis")
devtools::use_test("catch_error")
```

Making your own packages - dependencies

The DESCRIPTION file contains the package dependencies (the other packages needed to run your code).

You can add them with:

```r
devtools::use_package("magrittr")
```

When typed in the console, this instruction add the `import: magrittr` in your DESCRIPTION file.

Tips: You should create a `devtools_history.R` file in the data-raw folder to track all your calls to devtools during the creation of your package. This allows you to keep track of what you've done in the console.

However, this is not enough, and the NAMESPACE file must also be modified. To do so, you'll need to edit the documentation for each function, specify the package and / or package functions used inside the function.

You'll use `@import` (loading the whole package) and `@importFrom` (specific loading of a function) as in the following example.

Making your own packages - dependencies

```r
#' Mean
#'
#' This computes the mean without na
#' @param x a numerique vector
#' @return the mean
#' @import magrittr
#' @importFrom  stats na.omit
#' @examples
#' mean_no_na(c(4,5))
#' @export

mean_no_na <- function(x){
  x <- x %>% na.omit()
  res <- sum(x)/length(x)
  return(res)

}
```

Making your own packages - startup

A file named `zzz.R` (by convention) in the R/ folder can contain some specific functions:

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

`.onAttach` and `.onLoad` are used to run statements when loading a package. Be parsimonious with those functions

Making your own packages - general documentation

Do not forget to create a help for the package. It can be called by the user by making the `?nameofthepackage`.

```
#' What it does
#'
#' Detailled description of your package
#'
#' @name nameofthepackage-package
#' @aliases nameofthepackage-package nameofthepackage
#' @docType package
#' @author vincent <vincent@@thinkr.fr>
NULL
```

Making your own packages - Vignettes

A vignette is a html / pdf file that comes with a package and which hs more information than a help page. And the style is free (you can include images, a summary, links...). (for example look at `browseVignettes("dplyr")`)

Instruction:

```
devtools::use_vignette("mypackage")
```

creates a `mypackage.Rmd` file in a vignettes/ folder at the root of the package.

This is a RMarkdown page:

http://rmarkdown.rstudio.com/authoring_pandoc_markdown.html

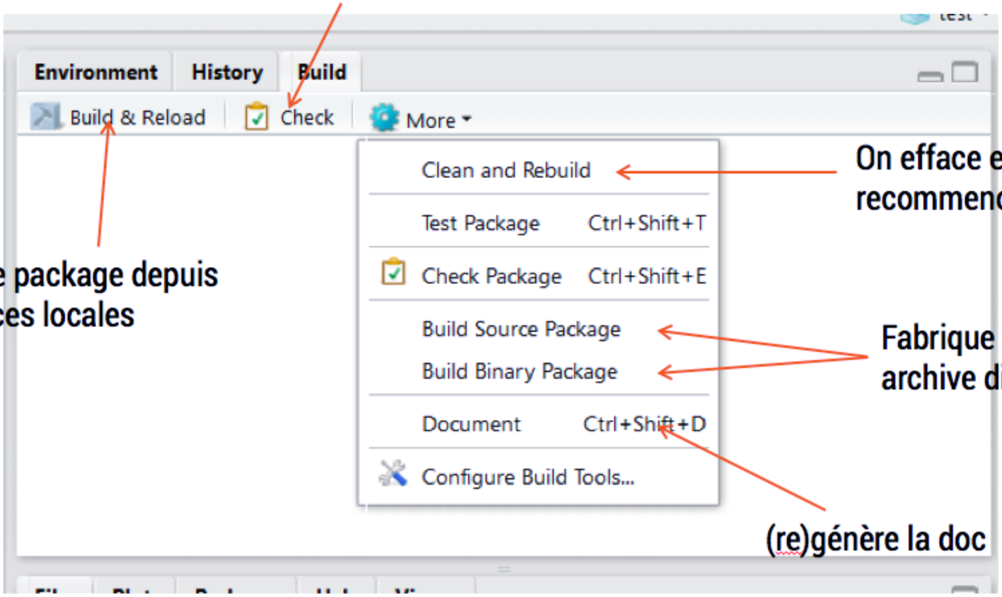
You have to edit by hand to explain your work. Once done you must type `devtools :: build_vignettes()` to create the vignette (which will be put in /inst/doc).

The end user can see the vignette with the following statement:

```
browseVignettes("monPackage")
```

Making your own packages - the build tab

+ de 50 tests réalisés
Il ne faut pas d'ERROR..



On efface et on recommence

Installe le package depuis les sources locales

Fabrique une archive distribuable

(re)génère la doc

Build your own packages – go deeper

http://r-pkgs.had.co.nz

The `.Rbuildignore` file is a file at the root of the project that must contain the folders and files you don't want the end package to contain: raw, data-raw...

Everything in the `inst` folder will be moved as is at the root of the package installation folder when your package is installed on the user's computer.

Making your own packages - memo

- Create a project (from git* or local)
- Enable roxygen2: `build> configure build tools`, and make sure to check "generate documentation with roxygen"
- If there, remove `hello.R` from the folder R/ and `hello.Rd` from the folder man/. Delete the `NAMESPACE` file.
- Fill the `DESCRIPTION` file.
- Make a first check.
- Add functions in the folder R/ (**just functions and comments, nothing else**)
- Use `devtools::use_data_raw()` and add a file `devtools_history.R` in data-raw/
- Document functions and dependencies (with `@import`, `@importFrom` and `devtools::use_package()`)
- Add the datasets that will be used as examples.
- Create a vignette

Remember to make regular commit and checks !

Web scraping

Becoming a pirate with R

Web scraping

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

```r
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:

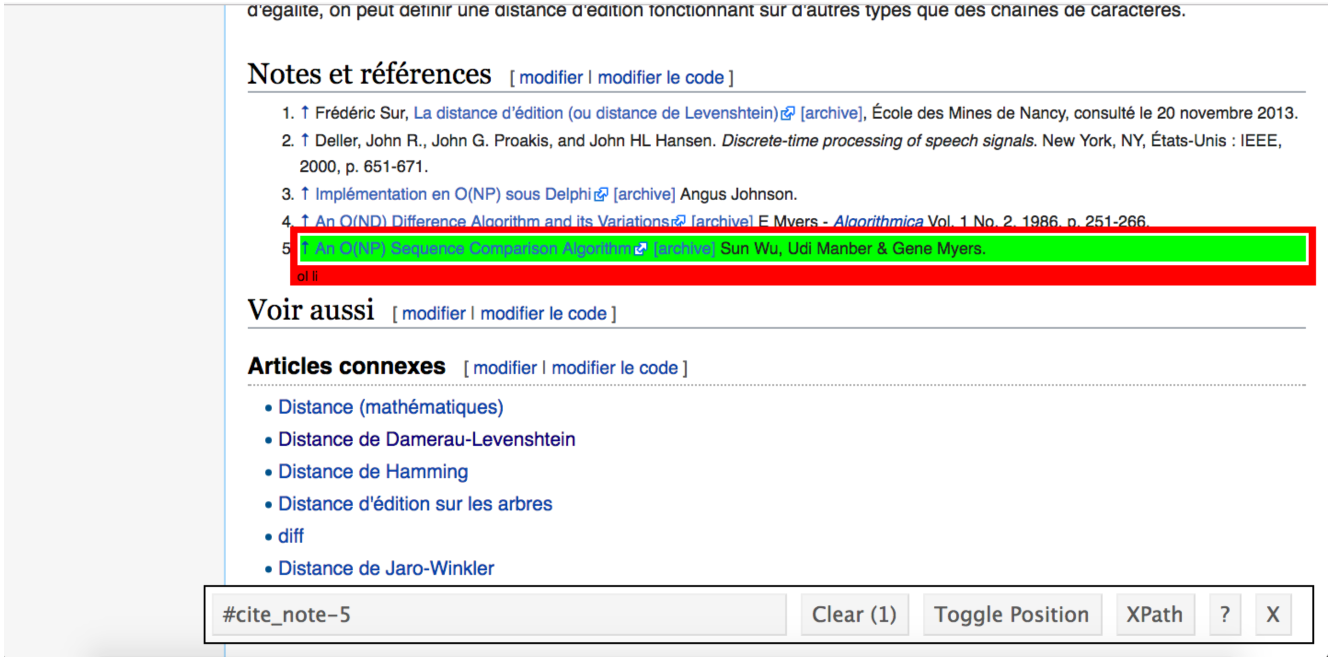
- id : the id is a unique name of an element. It is supposed to be used only once on a page. You can recognize it as it is preceded by a `#`. For example `#my-id`.
- class : a class is used to identify several elements on a page and throughout a whole website. It is precedeed by a dot. For example `.my-class`.

```
<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.

Web scraping

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



d'égalité, on peut définir une distance d'édition fonctionnant sur d'autres types que des chaînes de caractères.

Notes et références [modifier | modifier le code]

1. ↑ Frédéric Sur, La distance d'édition (ou distance de Levenshtein) [archive], École des Mines de Nancy, consulté le 20 novembre 2013.
2. ↑ Deller, John R., John G. Proakis, and John HL Hansen. *Discrete-time processing of speech signals.* New York, NY, États-Unis : IEEE, 2000, p. 651-671.
3. ↑ Implémentation en O(NP) sous Delphi [archive] Angus Johnson.
4. ↑ An O(ND) Difference Algorithm and its Variations [archive] E Myers - *Algorithmica* Vol. 1 No. 2, 1986, p. 251-266.
5. ↑ An O(NP) Sequence Comparison Algorithm [archive] Sun Wu, Udi Manber & Gene Myers.

ol li

Voir aussi [modifier | modifier le code]

Articles connexes [modifier | modifier le code]

- Distance (mathématiques)
- Distance de Damerau-Levenshtein
- Distance de Hamming
- Distance d'édition sur les arbres
- diff
- Distance de Jaro-Winkler

#cite_note-5 | Clear (1) | Toggle Position | XPath | ? | X

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 <div> elements and all <p> elements
element element	div p	Selects all <p> elements inside <div> elements
element>element	div > p	Selects all <p> elements where the parent is a <div> element
element+element	div + p	Selects all <p> elements that are placed immediately after <div> elements
element1~element2	p ~ ul	Selects every <ul> 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.

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

`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).

```
html_nodes(x = my_page, css = ".dt-blog-shortcode > .wf-cell")
articles <- html_nodes(x = my_page, css = "[data-name]")
```

`html_text()`

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

```
html_text(articles)
```

Web scraping

`html_attr()`

Used to collect the attributes.

```
html_attr(articles,"data-post-id")
```

`html_name()`

Returns the name of the elements.

```
html_name(articles)
```

`html_table()`

This function extract a table from a webpage:

```
url <- 'https://fr.wikipedia.org/wiki/Distance_de_Levenshtein'
p <- read_html(url)
html_table(p,fill = TRUE)
```

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.

```
# Image you're clicking on "Random" on the webpage you've just launched.

nav <- follow_link(nav,"Random")
```

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

Web scraping

`jump_to()`

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

```r
nav <- jump_to(nav,"https://thinkr.fr")
```

session_history()

Returns the navigation history from the current session :

```r
session_history(nav)
```

```
#>   http://www.commitstrip.com/fr/2015/05/19/data-wars/
#>   http://www.commitstrip.com/fr/2017/02/16/the-dark-side-of-coding-the-cross/
#> - https://thinkr.fr/
```

Web scraping

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

```r
library(rvest)
'http://www.commitstrip.com/fr/2015/05/19/data-wars/' %>%
  read_html() %>%
  html_nodes(".size-full") %>%
  html_attr("src") %>%
  .[1] %>%
  browseURL()
```

Question

Get the list of all the posts from `http://abcdr.guyader.pro/`

Then get the name of the authors from these posts.

ThinkR

contact@thinkr.fr

06 23 83 10 61