Bonnes pratiques / Développement / Packages / Mise en production

B.Thieurmel - benoit.thieurmel@datastorm.fr

ENEDIS : R avancé & packages - 21/01/2021

BONNES PRATIQUES

Bonne pratique de codage

Librement inspiré du Style Guide, by Hadley Wickham

- C'est important d'adopter des bonnes pratiques de codages :
  - permettre une lecture et une compréhension simple et rapide du code
  - tant pour le(s) développeur(s), que pour les utilisateurs, et favoriser le travail collaboratif
- Il n'y a pas un style parfait, le principal est d'en adopter un et de s'y tenir

```
# dur à lire
aze=data.frame(cole=rnorm(1000),refdzf=LETTERS[1:2]);ff=lapply(split(aze$cole,aze$refdzf),
                                                               function(x){mean(x)});ff
```

```
# c'est mieux quand même... ?
data <- data.frame(value = rnorm(1000), group = LETTERS[1:2])
mean.group <- lapply(
  split(data$value, data$group),
  function(x){
    mean(x)
  })
mean.group
```

Fichiers

Les noms doivent être **explicites** et se terminer par `.R`. Si les scripts sont ordonnés, les pré-fixer par un numéro.

```
# Good            # Bad         0-download.R
modelisation.R    toto.r        1-parse.R
```

Variables et fonctions

- Noms **courts** et **explicites**, de préférence en minuscule, en évitant d'utiliser des noms de fonctions connues...
- Utilisation d'un underscore (_) pour séparer les noms. Eviter le point (.), il peut amener de mauvaises intéractions avec d'autres langages (java, javascript, ...)
- Variable == noms, fonctions == verbes, autant que possible....
- **Pas d'accents !**

```
# Good                    # Bad
day_one                   first_day_of_the_month
day_1                     DayOne
                          mean <- function(x) sum(x)
```

Espacer son code

- Mettre des espaces **autour** de tous les opérateurs (`=`, `+`, `-`, `<-`, etc.), **surtout** à l'intérieur de l'appel d'une fonction.
- Mettre un espace **après** une virgule, **pas avant**
- Essayer de mettre un espace avant l'ouverture d'une parenthèse, **sauf dans l'appel d'une fonction**

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

- Exception pour `:`, `::` and `:::`

```
# Good                    # Bad
x <- 1:10                 x <- 1 : 10
base::get                 base :: get

if (debug) do(x)          if(debug)do(x)
plot(x, y)                plot (x, y)
```

Namespace et appel d'une fonction

- `::` accès aux fonctions exportées d'un package
- `:::` accès aux fonctions *cachées* d'un package

Bonne pratique

- essayer de toujours préfixer l'appel à une fonction par `::`
  - obligatoire pour une soumission d'un package sur le CRAN
  - meilleure lisibilité des appels / dépendances
  - évite des conflits potentiels : deux fonctions du même nom dans deux packages différents...
- éviter l'utilisation des fonctions *cachées* `:::`
  - interdit pour une soumission d'un package sur le CRAN

```
require(FactoMineR)

# Good
FactoMineR::PCA(X, scale.unit = TRUE)
```

Accolades et indentation

- L'ouverture d'une accolade doit **toujours** être suivi d'un passage à la ligne.
- La fermeture d'un accolade doit être suivi d'un passage à la ligne, sauf dans le cas d'un **else**
- Le code à l'interieur des acccolades doit être indenté

```
# Good                          # Bad
if (y == 0) {                   if (y == 0) {
  log(x)                          log(x)
} else {                        }
  y ^ x                         else{ y ^ x}
}
```

- Indenter son code, de préférence en utilisant deux espaces. **Raccourci RStudio : Ctrl+A, Ctrl+I**

Assignement

- Utiliser `<-`, et **banir** `=`, lors de l'assignement

```
# Good
x <- 5
# Bad
x = 5
```

Commentaires

- Commenter son code, toujours dans un soucis de lecture et de collaboration. **Raccourci RStudio : Ctrl+Shift+C**
- un commentaire comportant au-moins `----` créé une section pouvant être réduite

```
# Load data ---------------------------

# Plot data ---------------------------
```

QUELQUES MOTS SUR LES FONCTIONS

Les fonctions

On définit une nouvelle fonction avec la syntaxe suivante :

```
fun <- function(arguments) expression
```

- **fun** le nom de la fonction
- **arguments** la liste des arguments, séparés par des virgules. *formals(fun)*
- **expression** le corps de la fonction. une seul expression, ou plusieurs entre des accolades. *body(fun)*

```
test <- function(x) x^2
test               # function(x) x^2

formals(test)      # $x
body(test)         # x^2
environment(test) # <environment: R_GlobalEnv>
```

- Une fonction appartient à un environnement. Le plus souvent un package, ou alors l'environnement global **GlobalEnv**. *environment(fun)*

Les arguments

- **Valeur par défaut**
  - via une affetaction, avec '=', dans la définition de la fonction
  - optionnel lors de l'appel

```
test <- function(x, y = 2){
  x + y
}
test(x = 2)           # 4
test(x = 2, y = 10)   # 12
```

- **Quelques fonctions utiles de contrôle :**

- *missing(arg)* : retourne TRUE si l'argument est manquant lors de l'appel
- *match.arg()* : en cas d'input tronqué...
- *typeof(arg)*, *class(arg)*, *is.vector()*, *is.data.frame()*, ...

```
match.arg("mea", c("mean", "sum", "median"))  # "mean"
class(10)                                      # "numeric"
```

Dépendances entre arguments

On peut définir un argument en fonction d'autres arguments

```
# avec une expression simple
test <- function(x, y = x + 10){
  x + y
}
test(5)  # 20

# un peu plus compliqué
test <- function(x,
                 fun = if(class(x) %in% c("numeric", "integer")){
                   "sum"
                 }else{
                    "length"
                 }){

  do.call(fun, list(x = x))

}

test(1:10)            #55
test(LETTERS[1:10])   #10
```

Evaluation des arguments

Point Important : les arguments ne sont évalués que lorsqu'ils sont appelés, sinon ils n'existent pas dans la fonction.... Pour forcer l'évaluation, on peut utiliser la fonction *force()*. Démonstration :

```
f <- function(x) {
  10
}
f(stop("This is an error!"))

# la fonction retourne 10 alors que l'argument est un stop...
# 10

# utilisation de force
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))
```

```
# Error: This is an error!
```

Comprendre les '...' (1/2)

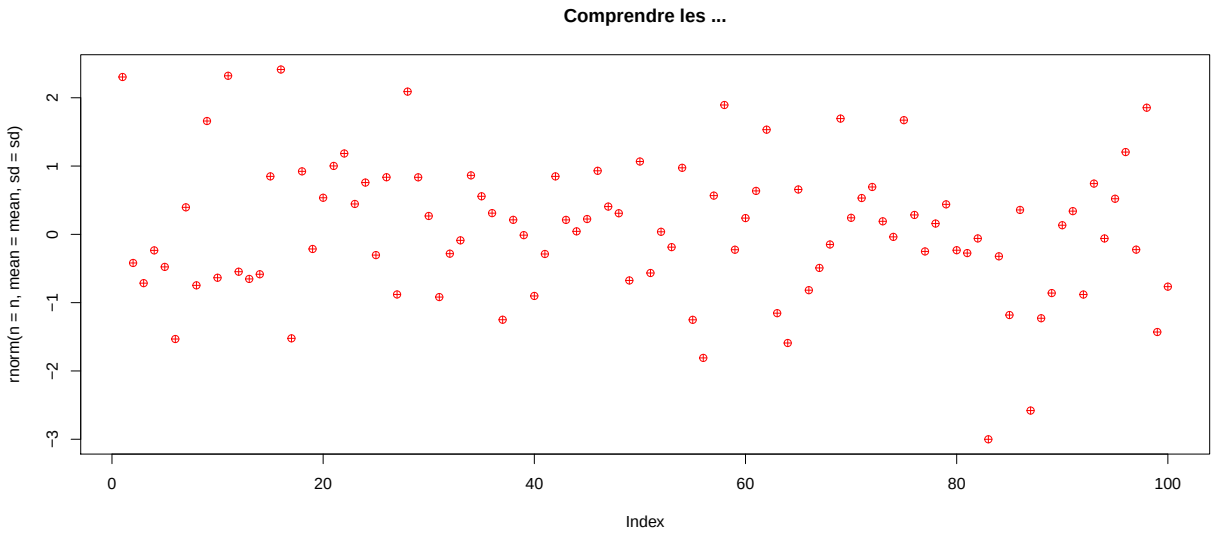
- Signifie que la fonction accepte d'autres arguments que ceux définis explicitement
- Sert généralement à passer ces arguments à une autre fonction
- Se récupère facilement avec : *list(...)*

```
viewdot <- function(arg, ...){
  list(...)
}
viewdot(arg = 1, x = 2, name = "name")

#$x
#[1] 2
#
#$name
#[1] "name"

rnormPlot <- function(n, mean = 0, sd = 1, ...){
  plot(rnorm(n = n, mean = mean, sd = sd), ...)
}
rnormPlot(n = 100, main = "Comprendre les ...", col = "red", pch = 10)
```

Comprendre les '...' (2/2)



Retourner un résultat

Une fonction retourne par défaut le résultat de la dernière expression

```r
test <- function(x, y = 2){
  x + y
}
test(2)
```

```
## [1] 4
```

```r
somme <- test(x = 2, y = 2)
somme
```

```
## [1] 4
```

- Renvoi d'un résultat avant la fin de la fonction : fonction *return()*
- Utilisation de *return()* pour la dernière expression ? **Inutile**.
- Retour de plusieurs résultats : liste nommée.
- Aucun résultat ? Possible avec par example la fonction *invisible()*

Fonction return()

```r
test <- function(x, y = 2){
  if(y == 0){
    return(x)
  }
  x + y
}
test(2)
```

```
## [1] 4
```

- Plusieurs résultats

```r
test <- function(x, y = 2){
  list(x = x, y = y)
}
test(2)
```

```
## $x
## [1] 2
##
## $y
## [1] 2
```

Fonction invisible()

"This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned"

```r
test <- function(x, y = 2){
  x + y
  invisible()
}
test(2)  # no print on console
res <- test(2)
res     # and NULL result
```

```
## NULL
```

```
test <- function(x, y = 2){
  invisible(x + y)
}
test(2)  # no print on console
res <- test(2)
res      # but a result !
```

```
## [1] 4
```

Variables locales et globales

- Une variable définie dans une fonction est **locale** :
  - elle ne sera pas présente ensuite dans l'espace de travail
  - elle n'écrasera pas une variable du même nom existante

```
x <- 100
test <- function(x, y){
  x <- x + y
  x
}

# la fonction retourne bien 10
test(5, 5)
```

```
## [1] 10
```

```
# et x vaut bien toujours 100
x
```

```
## [1] 100
```

Affectation globale

- Via l'opérateur d'affectation `<<-`, on peut affecter au modifier une variable **globale**
- Autant que possible **non-recommandé**...!

```
x <- 100
test <- function(x, y){
  x <<- x + y
  y <<- y
  x
}

# la fonction retourne ...  5 ?
test(5, 5)
```

```
## [1] 5
```

```
# et x vaut maintenant 10, et y 5
x ; y
```

```
## [1] 10
```

```
## [1] 5
```

Appel d'une variable non-définie ?

```
test <- function(x){
  x + z
}

# Erreur, z n'existe pas
test(5)

#> Error in test(5) : object 'z' not found

# Si, à tout hasard, une variable 'z' existe dans un autre environnement
# au moment de l'appel, la fonction l'utilise...
z <- 5
test(5)

#> 10
```

- **R** va chercher une variable d'une même nom dans les environnements *parents*. Pratique également à éviter.

Il faut passer tous les arguments en paramètres, et retourner l'ensemble des résultats souhaités en sortie

---

Fonctions anonymes

Comme son nom l'indique, une fonction qui n'a pas de nom...

- fonction courte, utilisée dans une autre fonction
- qui n'a pas pour but d'être ré-utilisée par la suite

```
f <- function(x){
  x + 1
}

res1 <- sapply(1:10, f)

res2 <- sapply(1:10, function(x) x + 1)

res1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
res2
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

Communication

Quand on développe, il est important d'anticiper les problèmes potentiels du code :

- mauvais type d'argument
- fichier non-existant

- données manquantes, valeurs infinies, ...

Et communiquer avec l'utilisateur. Trois niveaux sont disponibles :

- fonction *stop()* : erreur "fatale", l'éxécution se termine. A utiliser quand la suite du code ne peut pas être éxécutée
- fonction *warning()* : problème "potentielle", l'éxécution continue, mais il y aura peut-être un soucis...
- fonction *message()* : message "informatif", l'éxécution continue.

Communication : exemple

```
test <- function(x){
  # pour une erreur plus compréhensible
  if(missing(x)){
    stop("x is missing. Please enter a valid argument")
  }
  if(!class(x) %in% c("numeric", "integer")){
    x <- as.numeric(as.character(x))
    warning("x is coerced to numeric")
  }
  message("compute x*2")
  x*2
}

try(test())
```

```
## Error in test() : x is missing. Please enter a valid argument
#> Error: x is missing. Please enter a valid argument

test("5")
```

```
## Warning in test("5"): x is coerced to numeric
```

```
## compute x*2
```

```
## [1] 10
```

Et la documentation dans tout ça ?

La documentation est très importante :

- pour que l'utilisateur sache comment utiliser la fonction
- pour vous et d'autres développeurs, lors d'améliorations

Adopter la convention *doxygen*

- simple d'utilisation
- utiliser dans de nombreux langages de programmation
- via le package roxygen2, vous simplifiera ensuite la vie si vous créez des packages !

Utilisation dans R

Le plus simple : placer le curseur au-niveau de la fonction et faire *Code -> Insert roxygen Skeleton* ou bien utiliser le raccourci clavier associé

- en commençant la ligne par `#'`

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

Les balises indispensables

- @param : pour les arguments
- @return : pour le résultat
- @examples : pour les exemples
- @import : packages dépendants utilisés
- @importFrom : packages dépendants utilisés (mais importation uniquement de quelques fonctions)

Penser à préfixer le nom des fonctions utilisées par le `package::fonction`, et cela même pour les packages de base :

```
# Bad                           # Good
res_pca <- PCA(decathlon)       res_pca <- FactoMineR::PCA(decathlon)
```

---

exemple de documentation

```
#' le titre de ma fonction
#'
#' Une description succinte de ma fonction
#' sur plusieurs lignes si on veut
#'
#' @param nom : Character. Nom de la personne
#' @param prenom : Character. Prénom de la personne
#'
#' @return : Character. Identification de la personne
#'
#' @importFrom base paste0
#'
#' @examples
#' # les examples sont éxécutables dans RStudio avec Ctrl+Entrée
#' identify("Thieurmel", "Benoit")
identify <- function(nom, prenom){
  base::paste0("Nom :", nom, ", prénom : ", prenom)
}
```

LES PACKAGES

Pourquoi ?

Diffusion

- utilisation du même code à plusieurs endroits = **une fonction**
- plusieurs fonctions complémentaires = **un package**

Cadre de développement

- Bonnes pratiques / standardisation du code
- Documentation (fonctions principales, vignettes, ...)
- Gestion des dépendances / des versions

- Implémentation de tests
- Versionning

Un package = un sujet

- Ne pas hésiter à découper votre code en plusieurs packages

Les outils / la documentation

Le document de référence

R packages d'Hadley Wickham

devtools et usethis

Ensemble d'outils pour faciliter le développement sous R (devtools, usethis)

testthat et covr

Tests unitaires (Site) ; Couverture du code (Site)

roxygen2

Documentation (Site)

Fil rouge : installer les packages indispensables

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

Initialisation d'un package

Le plus simple, après l'installation de `devtools` :

File -> New Project -> New Directory -> R Package using devtools

- renseigner le nom du dossier, qui sera également le nom du package

sans espace et sans ponctuation, commençant par une lettre et de préférence en minuscule

- et l'endroit de destination

RStudio va initier un nouveau projet avec la structure minimale pour un package R

Fil rouge : initier le package demopck

L'onglet Build : un nouvel ami

Quand **RStudio** détecte la structure d'un package **R**, un nouvel onglet `Build` apparaît automatique avec les outils et raccourcis indispensables lors de la phase de dévélopppement :

Structure minimale (1/2)

Structure minimale (2/2)

obligatoire

- `DESCRIPTION` : metadata sur le package, informations sur les packages dépendants et les versions minimales nécessaires
- `NAMESPACE` : informations détaillées sur les dépendances et sur les fonctions disponibles dans le package. Il sera édité automatiquement en se basant sur la documentation des fonctions avec **roxygen2**
- `R` : ensemble des fonctions **R** du package

facultatif mais recommandé

- `.gitignore` : relatif à l'utilisation de *git*, pour ignorer certains fichiers
- `.Rbuildignore` : ignorer certains fichiers lors de la compilation du package
- `demopck.Rproj` : projet RStudio

et d'autres dossiers / fichiers que nous verrons plus loin. . . !

DESCRIPTION (1/2)

- `Title`, `Version`, `Authors@R` et `Description` à éditer manuellement
- Les autres champs seront majoritairement modifiés automatiquement par l'appel à des fonctions **R**

```
Package: demopck
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

DESCRIPTION (2/2)

Fil rouge : éditer le fichier DESCRIPTION

- le champs `Title` : sur une seule ligne, avec des majuscules en début de mot sauf pour les déterminants
- le champs `Description`
- l'auteur
- et ajouter une licence

```
usethis::use_gpl3_license("Enedis")
```

Et pour finir, faire un `check` du package pour voir si à ce stade, tout va bien !

```
devtools::check()
```

ou depuis le bouton Check dans le menu Build, ou avec Ctrl + Shift + E

Ma première fonction !

- Obligatoirement dans un script *.R* présent dans le dossier **R**
- pas de `require` ou `library` pour les appels à d'autres packages, mais l'utilisation dans un premier temps de la syntaxe `package::function()`. On reviendra sur la gestion des dépendances plus tard

Fil rouge : ajouter la fonction ci-dessous

```
is_premier <- function(x){
  numbers::isPrime(x)
}
```

N.B : installer préalablement le package numbers

Documenter (1/2)

`roxygen2` : documentation au-même niveau que la fonction dans le script **R**, génération automatique de la documentation *LaTeX* présente dans le dossier **man** et édition du fichier **NAMESPACE** !

Le plus simple : placer le curseur au-niveau de la fonction et faire *Code -> Insert roxygen Skeleton* ou bien utiliser le raccourci clavier associé `Ctrl+Alt+Shift+R`.

Les balises obligatoires

- @param : pour les arguments
- @return : pour le résultat
- @examples : pour les exemples
- @export : fonction principale du package, et donc exportée / visible pour l'utilisateur

Et aussi...

- `\code{character}` pour une syntaxe de code, `\link[pkg]{function}` pour des liens
- pour les exemples : `\dontrun{}`, `\dontshow{}`, `\donttest{}`
- `@rdname` pour une aide partagée entre plusieurs fonctions

Documenter (2/2)

Fil rouge : documenter votre fonction

- le titre de la fonction (première ligne)
- la description de ce que fait la fonction (deuxième paragraphe)
- troisième paragraphe ? la section "Details" de la page d'aide
- la signification des paramètres (nom, type, fonctionnalité, valeur par défaut)
- le résultat retourné
- rajouter des exemples

Générer la documentation

- *Build -> More -> Document*, `Ctrl+Shift+D` ou `devtools::document()`

Création du dossier `man` contenant les aides au format *.Rd* et édition du **NAMESPACE**

Et pour finir, faire un `check` du package !

Tester les développements en cours

Pour tester le package, vous devez le charger préalablement dans R

Build -> More -> Load All, `Ctrl+Shift+L` ou `devtools::load_all()`

- Utilisation de la fonction
- Affichage de l'aide avec `?is_premier`

A utiliser donc sans limite lors du développement du package !

Fil rouge : à vous de jouer !

Il est aussi possible de l'installer

Build -> More -> Install & Restart ou `Ctrl+Shift+L`. Cela installera le package et redémarrera R. Attention donc si vous avez beaucoup d'objets en mémoire...! (Car RStudio fera préalablement une sauvegarde de votre environnement courant afin de le recharger en mémoire une fois la session redémarrée)

Gestion des dépendances

Appeler une fonction externe proprement avec la syntaxe `package::fonction` ne suffit pas pour bien définir et gérer nos dépendances, comme le *check* vous le rappelle gentillement, avec dans notre exemple :

```
-- R CMD check results -------------------- demopck 0.0.0.9000 ----
Duration: 25.4s

> checking dependencies in R code ... WARNING
  '::' or ':::' import not declared from: 'numbers'
```

En effet, les dépendances doivent être définies à 3 endroits :

- dans le documentation **roxygen2** de la fonction, avec les balises `@import` ou `@importFrom`
- dans le fichier **NAMESPACE**. Ce dernier s'écrit automatiquement quand on re-génère la documentation
- et dans le fichier **DESCRIPTION**

Dépendances : dans la documentation

La première étape est donc d'ajouter dans la documentation des fonctions les informations sur les packages dépendants

- `@import` : importation de tout le package dépendant au chargement de notre package. A utiliser si on utilise beaucoup de fonctionnalités d'un autre package

```
#'@import package1 package2 package3
```

- `@importFrom` : importation d'un sous-ensemble de fonctions d'un package dépendant au chargement de notre package. A privilégier.

```
#'@importFrom package fonction1 fonction2 fonction3
```

Le fichier **NAMESPACE** sera alors automatiquement complété quand on mettra à jour la documentation :

```
import(shiny)
import(yaml)
importFrom(DT,DTOutput)
importFrom(DT,datatable)
```

Dépendances : dans DESCRIPTION

Mettre des `import` et des `importFrom` ne suffit pas pour satisfaire le *check*... Il faut également rajouter ces dépendances dans le fichier **DESCRIPTION**. C'est possible de l'éditer manullement, cependant il est assez *psychorigide* sur le format attendu. Il est conseillé d'utiliser **usethis** pour cela avec `use_package()` qui s'en chargera donc pour nous :

```
usethis::use_package("numbers")
```

DESCRIPTION

```
Imports:
    numbers
```

La gestion d'une version minimale se fera en rajoutant la syntaxe `(>= version)` à la suite dans le **DESCRIPTION**. Cela peut se faire avec l'argument `min_version` de `use_package()`, soit avec `TRUE` pour mettre la version installée, soit en mettant directement le numéro de la version minimale souhaitée.

```
Imports:
    numbers (>= 0.7.5)
```

Fil rouge : configurer proprement la dépendance au package numbers

---

Tests automatiques

Il est très important de mettre en place des tests unitaires. Ces tests seront *à minima* lancés à chaque *check* du package, et on pourra également configurer des lancements automatiques avec un gestionnnaire de code (à chaque *commit* par exemple...).

Le package **testthat** est là pour ça !

Initialisation :

```
usethis::use_testthat()
```

- création d'un dossier `tests` avec
  - un fichier *testthat.R* (à ne pas modifier en général)
  - et un dossier *testthat* dans lequel on va insérer nos tests sous la forme de scripts **R**

Bonnes pratiques : Couvrir l'ensemble de la fonction avec des tests **ET** rajouter un nouveau test à chaque bug rencontré et corrigé

Exécution : Build -> More -> Test package, `Ctrl+Shift+T` ou `devtools::test()`

Testthat : syntaxe

- écriture de scripts **R**, à sauvegarder dans *tests/testthat*. Convention de nommage : `test-*.R`

Principales fonctions :

- `context("infos")` : Information sur les tests qui suivent
- `test_that("info", {tests})` : Définition d'un bloc de test
- `expect_equal()` : égalité avec une tolérance de précision, `expect_identical()` : égalité stricte
- `expect_false()` | `expect_true()` : retourne effectivement `TRUE` ou `FALSE`
- `expect_message()` | `expect_warning()` | `expect_error()` : affichage de message, warning ou erreur
- et pleins d'autres...!

```
context("Nombres premiers")

test_that("Bons résultats", {
  expect_false(is_premier(1))
  expect_true(is_premier(3))
})
```

Couverture du code avec covr

Finalement, on peut avoir une vue de la couverture de code, c-à-d des lignes de code effectivement testées avec le package `covr`.

De façon intéractive depuis R :

```
# devtools se charge d'appeler le package covr
devtools::test_coverage()
```

Il se peut qu'il faille préalablement décharger notre package, ou redémarrer notre session R

De façon automatique :

Avec un gestionnaire de code, il sera aussi possible de configurer le lancement automatique de la couverture du code, avec l'affichage d'un *badge* associé.

Fil rouge : tester la fonction is_premier

Vignette(s) (1/2)

En complément de la documentation **R** des fonctions principales, nous pouvons rédiger une ou plusieurs *vignettes* thématiques sur notre package. Les vignettes sont écrites en *rmarkdown*.

Vignettes disponibles dans les packages installés :

```
utils::vignette() # pour tous les packages
utils::vignette(package = "devtools") # pour le package devtools
```

Afichage d'une vignette :

```
vignette("dependencies", package = "devtools")
```

Vignette(s) (2/2)

Initialisation :

```
usethis::use_vignette("nom-de-ma-vignette")
```

- création d'un dossier `vignettes` avec *nom-de-ma-vignette.Rmd*
- gestion des dépendances dans le **DESCRIPTION**
- update du *.gitignore*

Génération de la vignette :

- la vignette sera générée automatiquement lors du *check* ou du *build*, et déposée dans le dossier *doc*
- C'est également possible avec `devtools::build_vignettes()`

Fil rouge : initier une vignette pour le package

Ajout de données (1/2)

Avec `usethis::use_data()` et `usethis::use_data_raw()`

Données visibles après pour tous les utilisateurs (exemples de fonctions)

```
data_ex <- 1
usethis::use_data(data_ex, internal = FALSE, overwrite = FALSE)
```

- `internal = FALSE`, récupérables par l'utilisateur avec `data(data_ex)`
- sauvegardées en *data_ex.rda* dans le répertoire **data**

Données internes au package

```
param_pck <- 1
usethis::use_data(param_pck, internal = TRUE, overwrite = FALSE)
```

- `internal = TRUE`, récupérables uniquement à l'intérieur des fonctions du package en appelant la variable `param_pck`
- sauvegardées en *R/Sysdata.rda*

Ajout de données (2/2)

Bonnes pratiques :

Utilisation de `usethis::use_data_raw("nom_data")` qui créera un script dans **data-raw** se terminant par `usethis::use_data()`. Cela permet de garder un trace de la génération des données.

```
usethis::use_data_raw("demo_premier")
```

data-raw/demo_premier.R

```
## code to prepare `demo_premier` dataset goes here
demo_premier <- 1:10
usethis::use_data(demo_premier, overwrite = TRUE)
```

Il faudra également documenter les données...!

https://r-pkgs.org/data.html#documenting-data

Fil rouge : rajouter ce jeu de données de démo, et compléter les exemples de la fonction

Partager / Compiler le package (1/2)

Build -> Build Source package

- Création d'un fichier compressé en *tar.gz* (dit *bundle* dans le schéma ci-dessous) qui contient le code source du package
- Partageable et installable
  - sans outils de développement si uniquement du code **R**
  - avec outils de développement (RTools en windows par exemple) si également du code `C/C++ ou autre`
- Format pour la soumission sur le CRAN (https://cran.r-project.org/submit.html)

Build -> Build Binary package

- Version *compilée* du package pour macOS (*.tgz*) ou windows (*.zip*)
- Partageable et installable sans outils de développement

Partager / Compiler le package (2/2)

Aller plus loin : documentation partagée (1/2)

```r
#' @export
#' @rdname sum23
sum2 <- function(x, y) x + y

#' Title
#'
#' @param x :
#' @param y :
#' @param z :
#'
#' @return ...
#' @export
#'
#' @examples
#' sum2(2, 3)
#' sum3(2, 3, 4)
#'
#' @rdname sum23
sum3 <- function(x, y, z) x + y + z
```

Aller plus loin : documentation partagée (2/2)

- L'aide partagée doit être rédigée sur une des fonctions partagées
- Elle doit comportée l'ensemble des arguments de toutes les fonctions partagées
- Ainsi que les exemples souhaitées
- Utilisation de la balise `#' @rdname identifiant` pour faire ensuite le lien

Aller plus loin : méthodes S3 usuelles

Pour appliquer des fonctions type `plot`, `summary`, `predict`, ... sur un objet ayant une classe spécifique

- Simplement en utilisant la syntaxe `function.class`, par exemple `plot.myclass()`

```r
#' Plot of object of class "custom"
#'
#' @param x an object of class \code{custom}
#' @param ... graphical parameters passed to \code{plot()} function.
#'
#' @return Nothing is returned, only a plot is given.
#' @export
#'
#' @examples
#' custom_x <- list(x = 1:10, y = 1:10)
#' class(custom_x) <- "custom"
#' plot(custom_x)
```

```r
plot.custom <- function(x, ...) {
  plot(x$x, x$y, type = "l", ...)
}
```

Aller plus loin : nouvelles méthodes S3

Comme précédemment, avec en plus l'ajout de la définition de la fonction générique :

```r
myplot <- function (x, ...) {
  UseMethod("myplot", x)
}

#' Plot of object of class "custom"
#'
#' @param x an object of class \code{custom}
#' @param ... graphical parameters passed to \code{plot()} function.
#'
#' @return Nothing is returned, only a plot is given.
#' @export
#'
#' @examples
#' custom_x <- list(x = 1:10, y = 1:10)
#' class(custom_x) <- "custom"
#' myplot(custom_x)
myplot.custom <- function(x, ...) {
  plot(x$x, x$y, type = "l", ...)
}
```

Aller plus loin : le dossier inst/

Le dossier **inst** permet de rajouter d'autres ressources à notre package, comme par exemple :

- des fichiers plats utilisées dans des exemples ou dans les fonctions, de configuration
- des scripts de tests / démo
- une application **shiny**

Lors de l'installation du package, le dossier **inst/** s'efface et l'ensemble de son contenu est alors présent à la racine du package. On utilise alors la fonction `system.file` pour y accéder.

exemple :

- nous avons une application **shiny** dans **inst/app/**

```r
#' Launch app
#' @import shiny # plus les autres dépendances de l'appli
run_demo_app <- function(...) {
  shiny::runApp(appDir = system.file("app", package = "demopck"), ...)
}
```

Aller plus loin : devenir un expert ?

Avec un peu de lecture et beaucoup de pratique....

R packages d'Hadley Wickham

devtools

usethis

testthat

covr

roxygen2

OPTIMISATION DU CODE

Accélérer son code ?

R non efficace pour interpréter et exécuter des boucles for et donc A EVITER!

- Vectorisation
- Fonctions de type `Apply`
- Utilisation du package `compiler` :

http://homepage.divms.uiowa.edu/~luke/R/compiler/compiler.pdf

- Implémenter les points chauds de calcul avec des langages compilés et utiliser le package **Rcpp**

http://www.rcpp.org/

Gestion de la mémoire

Initialiser l'espace pour un résultat. Sinon **R** prend du temps pour agrandir itérativement la mémoire allouée à un objet :

Dans tous les cas éviter la concaténation de résultats quand cela est possible

```
x <- rnorm(100000) ; y <- rnorm(100000)
res <- integer(100000)  # initialisation

# calcul de la somme via une boucle avec initialisation
system.time(for(i in 1:length(x)){
  res[i] <- x[i] + y[i]
})
```

```
##    user  system elapsed
##    0.02    0.00    0.02
```

```
res <- c()
# avec concaténation
system.time(for(i in 1:length(x)){
  res <- c(res, x[i] + y[i])
})
```

```
##    user  system elapsed
##   12.28    6.86   19.40
```

Retour sur la vectorisation

'*La vectorisation est le processus de conversion d'un programme informatique à partir d'une implémentation scalaire, qui traite une seule paire d'opérandes à la fois, à une implémentation vectorielle qui traite une opération sur plusieurs paires d'opérandes à la fois. Le terme vient de la convention de mettre les opérandes dans des vecteurs ou des matrices.*' (Wikipédia)

- R est un langage **interprété**
- Beaucoup de calculs pouvant être réalisés par une boucle peuvent se faire en utilisant la vectorisation, avec une performance accrue :
  - opérations sur des vecteurs
  - opérations sur des matrices (= un ensemble de vecteurs)
  - opérations sur des data.frame
- Une performance accrue, pourquoi ?
  - **R**, et ses fonctions "de base" sont codés en **C**, **Fortran**, ...
  - avec l'utilisation efficace et optimisée dans "routines" d'algèbre linéaire (*BLAS*, *LAPACK*, ...)

Exemple : la somme de deux vecteurs

```
x <- rnorm(1000000)
y <- rnorm(1000000)

res <- integer(1000000)
# calcul de la somme via une boucle
system.time(for(i in 1:length(x)){
  res[i] <-  x[i] + y[i]
})
```

```
##    user  system elapsed
##    0.13    0.00    0.14
```

```
# avec la vectorisation
system.time(res2 <- x + y)
```

```
##    user  system elapsed
##       0       0       0
```

```
identical(res, res2)
```

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

Remember :

- opérations entre vecteurs / matrices

```
x <- matrix(ncol = 2, nrow = 2, 1)
y <- matrix(ncol = 2, nrow = 2, 2)

z <- x + y
z
```

```
##      [,1] [,2]
## [1,]    3    3
## [2,]    3    3
```

- Création / modification de colonne

```
data <- data.frame(x = 1:10, y = 100:109)
data$z <- data$x + data$y
head(data, n = 2)
```

```
##   x   y   z
## 1 1 100 101
## 2 2 101 103
```

GESTION DES ERREURS ET DES MESSAGES

Fonctions utiles

Quand **R** rencontre une erreur, il s'arrête net. Dans certains cas, on voudrait pouvoir continuer notre calcul. Trois fonctions sont diponibles dans R :

- *try()* : la plus simple pour contrôler l'apparition d'erreurs
- *tryCatch()* : la plus complète, avec la définition d'action en cas d'erreurs / warnings / messages
- *withCallingHandlers()* : une variante de tryCatch()

```
test <- sapply(list(1:5,"a", 6:10), log)
#>Error in FUN(X[[2L]], ...) :
#  non-numeric argument to mathematical function
```

---

try

```
try(expr, silent = FALSE)
```

- `silent` : affichage ou non d'erreur
- retourne un objet de **class** `try-error` incluant le message d'erreur

```
test <- sapply(list(1:2,"a"), function(x) try(log(x), silent = TRUE));test
```

```
## [[1]]
## [1] 0.0000000 0.6931472
##
## [[2]]
## [1] "Error in log(x) : argument non numérique pour une fonction mathématique\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in log(x): argument non numérique pour une fonction mathématique>
```

```
# on récupére un object de class "try-error", avec le message d'erreur
class(test[[2]])
```

```
## [1] "try-error"
```

```
test[[2]][1]
```

```
## [1] "Error in log(x) : argument non numérique pour une fonction mathématique\n"
```

---

tryCatch

`tryCatch(expr, ..., finally)`

- `error = function(e)` : fonction à exécuter en cas d'erreur, `e` étant le message.
- idem avec `warning = function(e)` et `message = function(e)`

Si ces fonctions sont définies, elles seront donc évaluées le cas échéant **ET le calcul sera arrêté**

```
test <- tryCatch(log("a"), error = function(e){
  print(e)
  return(0)
})
```

```
## <simpleError in log("a"): argument non numérique pour une fonction mathématique>
```

```
test
```

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

---

withCallingHandlers

`withCallingHandlers(expr, ..., finally)`

- `error = function(e)` : fonction à exécuter en cas d'erreur, `e` étant le message
- idem avec `warning = function(e)` et `message = function(e)`

Si ces fonctions sont définies, elles seront donc évaluées le cas échéant **MAIS le calcul continuera**

```
f <- function(){message("message") ; 0}
test <- withCallingHandlers(f(), message = function(e){e})
```

```
## message
```

```
test
```

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

```
# tryCatch
test <- tryCatch(f(), message = function(e){e})
test
```

```
## <simpleMessage in message("message"): message
## >
```

MONITORING, PROFILING & DEBUG

---

microbenchmark : temps de calculs

Pour monitorer le temps de calculs, la fonction `system.time()` peut-être utilisée, mais le package `microbenchmark` permet de monitorer avec plus de précision en répétant les appels.

```
suppressWarnings(require(microbenchmark, quietly = TRUE))
x <- runif(1000)
microbenchmark(sqrt(x),x^{1/2},times=1000)
```

```
## Unit: microseconds
##           expr  min   lq    mean median   uq    max neval
```

```
##        sqrt(x)  4.5  4.8  8.8364    8.0 11.2   43.5  1000
##  x^{     1/2 } 31.5 32.3 48.5865   36.2 50.9 3338.1  1000
```

Profilage du code via Rprof (1/2)

Utiliser la fonction `Rprof` qui procède par échantillonnage : elle stoppe l'éxécution du code par intervalles (`interval`) et différencie le temps de calcul réalisé par chaque fonction (`self.time`) et le temps global (`total.time`).

```
is.prime <- function(n){
  n == 2L || all(n %% 2L:ceiling(sqrt(n)) != 0)
}

all.prime <- function(n){
  v <- integer(0)
  for(i in 2:n){
    if(is.prime(i)){
      v <- c(v,i)
    }
  }
  v
}
```

```
Rprof("Rprof.out", interval = 0.001)
prime.number <- all.prime(100000)
Rprof(NULL)
```

Profilage du code via Rprof (2/2)

```
summaryRprof("Rprof.out")
```

```
##             self.time self.pct total.time total.pct
## "is.prime"      0.071    47.97      0.109     73.65
## "%%"            0.033    22.30      0.033     22.30
## "c"             0.029    19.59      0.029     19.59
## "all.prime"     0.010     6.76      0.148    100.00
## "all"           0.005     3.38      0.005      3.38
```

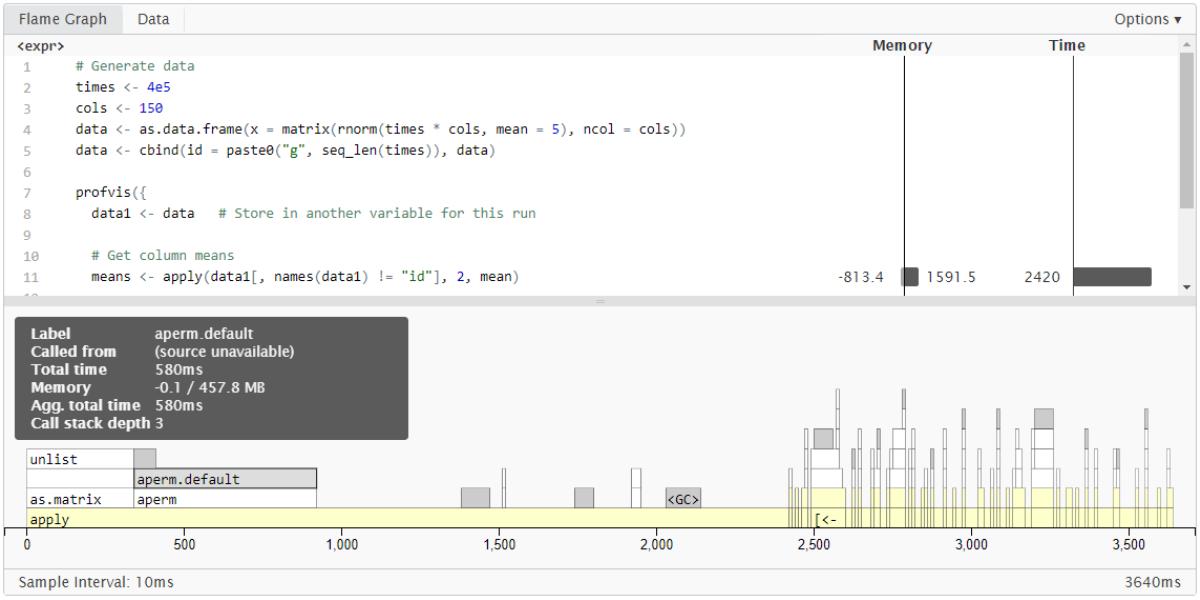
```
##             total.time total.pct self.time self.pct
## "all.prime"      0.148    100.00     0.010     6.76
## "is.prime"       0.109     73.65     0.071    47.97
## "%%"             0.033     22.30     0.033    22.30
## "c"              0.029     19.59     0.029    19.59
## "all"            0.005      3.38     0.005     3.38
```

Profilage avec profvis ou proftools

D'autres outils existent, avec notamment les packages **proftools** ou **profvis**

https://rstudio.github.io/profvis/

https://cran.r-project.org/web/packages/proftools/vignettes/proftools.pdf



Impact mémoire

- Dans **R** de base, avec la fonction `object.size()`. **Problème** : ne prend pas en compte toute la complexité potentielle des objects (environnements rattachés)
- Avec le package **pryr**
  - `object_size()`
  - `mem_used()` : mémoire utilisée, `mem_change(code)` : impact du code sur la mémoire

```
# différence integer / numeric
v_int <- rep(1L, 1e8) ; v_num <- rep(1, 1e8)
object_size(v_int); object_size(v_num)
```

```
## 400 MB
```

```
## 800 MB
```

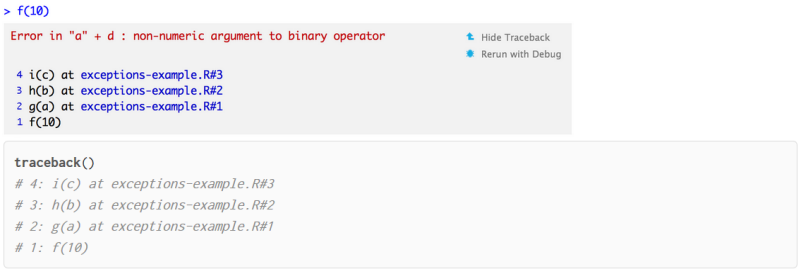
```
mem_change(x <- 1:1e6) ; mem_change(rm(x))
```

```
## -2.68 kB
```

```
## 592 B
```

Un petit mot sur le déboggage

- Pour voir simplement les informations : utilisation de *print()* dans la fonction
- Quand une erreur se produit, information du **traceback**
  - Disponible par défaut dans la console RStudio
  - via la fonction *traceback()* dans R
- Utilisation de la fonction *browser()* n'importe où dans le code : elle stoppe l'éxécution et lance un environnement dans lequel on peut accèder aux variables actuelles et continuer l'éxécution
- Insertion de points d'arrêt dans le code
- **RStudio** : menu *Debug*



Plus d'informations ici : https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio

---

R EN PRODUCTION ?

Lancement de R en ligne de commande

Il existe deux commandes pour lancer **R** : `Rscript` ou `R CMD BATCH`

Documentation

- Depuis **R** avec les commandes `?Rscript` et `?BATCH`
- Depuis un terminal via `R --help` et `Rscript -help`

Syntaxe

```
R CMD BATCH [options] infile [outfile]
```

- **infile** : Le nom/chemin du script R à exécuter.
- **options** Une liste d'options, pour la plupart partagées avec `Rscript`
- **outfile** Le nom du fichier de sortie.

```
Rscript [options] [-e expr [-e expr2 ...] | file] [args]
```

- **options** : Une liste d'options, pour la plupart partagées avec `R CMD BATCH`
- **expr, expr2** : R expression(s) (pour exécuter du code directement)
- **file** : Le nom/chemin du script R à exécuter.
- **args** : Arguments à passer au script.

Différences : écriture d'un fichier de sortie

- `R CMD BATCH` permet, avec l'argument **outfile**, l'écriture dans un fichier de toute la console **R** (lignes de codes appelées et affichages console), complétée du temps d'exécution global.

```
R CMD BATCH script.R console.txt
```

- `Rscript` ne propose rien par défaut. Une redirection du terminal vers un fichier est possible, mais restreint aux affichages console.

```
Rscript script.R > console.txt 2>&1
```

Différences : arguments et packages

- `R CMD BATCH` : utilisation de l'options `--args` :

```
R CMD BATCH '--args 2 c(1:3)' script.R
```

- `Rscript` : directement dans la commande à la suite du script. **Entourés obligatoirement de quote si complexes**

```
Rscript script.R 2 'c(1:3)'
```

Packages chargés

`Rscript` ne charge pas le package **methods** (gain de 60% au lancement). On peut cependant décider de le charger.

Récupération des arguments

Utilisation de la fonction `commandArgs()` dans le script **R**.

- Récupérés sous la forme d'un **vecteur de caractères**
- Ne pas oublier de les typer le cas échéant avant de les utiliser
- Ou de les évaluer pour les arguments plus complexes (vecteurs, matrices, ...)

```r
# récupération des arguments
args <- commandArgs(trailingOnly = T)

input_char <- args[1]
input_numeric <- as.numeric(args[2])
input_vector <- eval(parse(text = args[3]))
```

Valeur de retour

Par défaut, **R** renverra :

- **0** en cas de succès
- **1** dans la majorité des erreurs
- **2** pour le suicide, cas de force majeure...

On peut contrôler et modifier cette valeur de retour en utilisant directement la fonction **quit()** et l'argument **status** dans notre script.

```
quit(save = "default", status = 10, runLast = TRUE)
```

Principales options

`Rscript` et `R CMD BATCH`

- `--save` sauvegarde des données à la fin de la session **R**. (`.RData`)
- `--no-save` pas de sauvegarde des données
- `--no-environ` ne pas lire les fichiers utilisateur pour affecter des variables d'environnement
- `--restore` restaure le fichier `.RData` si présent dans le répertoire de lancement
- `--vanilla` combinaison de `--no-save`, `--no-environ`, `--no-site-file`, `--no-init-file` et `--no-restore`
- `--quiet`, `--silent`, et `-q` suppressions des messages initaux dans la console (version, copyright, ...)
- `--slave` makes R run as quietly as possible.

`Rscript`

- `--default-packages` : packages à charger au lancement (**methods** par exemple)

Exemple - script

Soit le script *exemple_r_cmd.R* **R** suivant :

```
# recuperation des argumnets
args <- commandArgs(trailingOnly = TRUE)

# controle et retour custom
if(length(args) < 3){
  stop("Veuillez renseigner 3 arguments")
  # .Last <- function(){
  #   cat("Veuillez renseigner 3 arguments\n")
  # }
  # quit(save = "no", status = 5, runLast = TRUE)
}
input_char <- args[1]
input_numeric <- as.numeric(args[2])
input_vector <- eval(parse(text = args[3]))

stopifnot(!is.na(input_numeric))
stopifnot(is.vector(input_vector))
# affichage
print(input_char) ; print(input_numeric) ; print(input_vector)
```

Exemple - Appel avec R CMD BATCH

```
R CMD BATCH --vanilla "--args Benoit 45 c(1:3)" exemple_r_cmd.R output.txt
```

```
R version 3.2.2 (2015-08-14) -- "Fire Safety"...
> # recuperation des argumnets
> args <- commandArgs(trailingOnly = TRUE)
>
> # controle et retour custom
> if(length(args) < 3){
+   stop("Veuillez renseigner 3 arguments")
+   # .Last <- function(){
+   #   cat("Veuillez renseigner 3 arguments\n")
+   # }
+   # quit(status = 5, runLast = TRUE)
+ }
> input_char <- args[1]
> input_numeric <- as.numeric(args[2])
> input_vector <- eval(parse(text = args[3]))
>
> stopifnot(!is.na(input_numeric))
> stopifnot(is.vector(input_vector))
>
> print(input_char) ; print(input_numeric) ; print(input_vector)
[1] "Benoit"
[1] 45
[1] 1 2 3
```

```
>
> proc.time()
user  system elapsed
0.408   0.576   0.343
```

Exemple - Appel avec Rscript (et redirection)

```
Rscript --vanilla exemple_r_cmd.R  Benoit 45 'c(1:3)' > output.txt 2>&1
```

Uniquement les affichages dans la sortie :

```
[1] "Benoit"
[1] 45
[1] 1 2 3
```

Mauvais appel : récupération de l'erreur

```
Rscript --vanilla exemple_r_cmd.R  Benoit 45 > output.txt 2>&1
```

```
Error: Veuillez renseigner 3 arguments
Execution halted
```

```
Rscript --vanilla exemple_r_cmd.R  Benoit 45 'matrix(0)'> output.txt 2>&1
```

```
Error: is.vector(input_vector) is not TRUE
Execution halted
```

Fichier de configuration

.Rprofile

Ce fichier, présent dans un répertoire à partir duquel **R** sera lancé, ou dans le *home*, s'exécutera automatiquement au lancement.

Cependant, il n'est pas forcément bien adapté dans le cadre d'un passage d'une configuration «utilisateur» ou d'arguments. En effet :

- Impossibilité de passer un **.Rprofile** dédié lors d'un lancement en ligne de commande (on peut seulement désactiver ceux existants)
- Lié à l'utilisateur ou à l'emplacement

Langage YAML

Une alternative est d'utiliser le langage YAML (http://yaml.org/) et le package **yaml** associé.

- Passage simple et lisible de valeurs (listes, tableaux, scalaires).

Langage YAML (1/2)

exemple.yml :

```
# Déclaration d'un chemin
path: /home/bthieurmel/file.log

# Déclaration du paramètre test, en boolean
test: false
```

```
# Liste de configuration pour une base de données
db:
  host : 10.244.36.68
  port : 5432
  uid : user
  pwd : pwd
  dbname : database
```

Dans **R**, on charge ensuite le fichier avec la fonction `yaml.load_file`

```
require(yaml)
conf <- yaml.load_file("C:/Users/Benoit/Desktop/exemple.yml")
conf
```

Langage YAML (2/2)

```
# $path
# [1] "/home/bthieurmel/file.log"
#
# $test
# [1] FALSE
#
# $db
# $db$host
# [1] "10.244.36.68"
#
# $db$port
# [1] 5432
#
# $db$uid
# [1] "user"
#
# $db$pwd
# [1] "pwd"
#
# $db$dbname
# [1] "database"
```

Ecriture de rapports

Utilisation de la fonction `sink()` afin d'effectuer une redirection temporaire des affichages console dans un fichier, couplée aux fonctions `cat()` (affichage de chaînes de caractères) et `print()` (affichage d'objets R).

Exemple de script R :

```
data <- data.table(lettre = sample(LETTERS[1:10], 50, replace = T),
                   n = sample(1:5, 50, replace = T))

# Initialisation du journal de chargement et redirection
sink("C:/Desktop/rapport.txt")

# stats
uni_stat <- data[, list(eff = .N), keyby = lettre]
```

```r
uni_stat[, pct := round(eff/sum(eff)*100,3)]

# ecriture
cat("Exemple de rapport \n")
cat(format(Sys.time(), "%a, %d %b %Y %H:%M:%S"), "\n\n")
print(uni_stat)
cat("\n\n")

# fermeture de la redirection
sink()
```

Ecriture de rapports

```
Exemple de rapport
jeu., 30 nov. 2017 17:58:30

      lettre eff pct
1:      A   5  10
2:      B   5  10
3:      C   3   6
4:      D   8  16
5:      E   6  12
6:      F   5  10
7:      G   5  10
8:      H   6  12
9:      I   3   6
10:     J   4   8
```

Autres formats ? (plus sexy...)

- **officer** https://davidgohel.github.io/officer/
- ...

Logs

Les sorties console ne sont pas des *logs* à proprement parlé. Ils existent différents packages pour pallier à cela :

- **futile.logger** : https://cran.rstudio.com/web/packages/futile.logger/index.html
- **log4r** : https://cran.r-project.org/web/packages/log4r/index.html
- **logging** : https://cran.r-project.org/web/packages/logging/index.html

Nous présenterons ici le package **futile.logger**. Il est relativement récent et simple dans son utilisation.

---

futile.logger - format

Initialisation du fichier :

`flog.appender()` et `appender.file()`

Format des logs :

`flog.layout()` et `layout.format()`

- ~**l** : niveau du log

- **~t** : date et heure
- **~n** : namespace
- **~f** : fonction appelée
- **~m** : le message

---

futile.logger - niveaux

Niveau de logs à afficher

`flog.threshold()` ("INFO", "WARN", "ERROR", "DEBUG")

Génération des logs

`flog.trace()`, `flog.info()`, `flog.warn()`, `flog.error()`, `flog.fatal()`

Utilisation de plusieurs fichiers

La référence au «logger» (fichier) souhaité se fait ensuite par l'argument `name` dans les différentes fonctions.

---

futile.logger - exemple

```
require(futile.logger)

# initialisation du fichier
path_file <- "C:/Users/Benoit/Desktop/file.log"
flog.appender(appender.file(path_file), name = "log.io")

# configuration du format
layout <- layout.format("[~t] [~l] ~m")
flog.layout(layout, name = "log.io")

# niveau des logs
flog.threshold("WARN", name = "log.io")

# logs
flog.info("Log d'information, pas affiché", name = "log.io")
flog.warn("Log de warnings, affiché", name = "log.io")
flog.error("Log d'%s, affiché", "erreur", name = "log.io")
```

```
[2017-11-30 16:22:10] [WARN] Log de warnings, affiché    [2017-11-30 16:22:10] [ERROR] Log
d'erreur, affiché
```

---

futile.logger : redirection des messages R

- Plus généralement, on peut rédiriger les *messages*, *warnings* et *erreurs* de **R** dans un fichier de logs.
- Cela évite de devoir adapter des codes **R** et d'utiliser les fonctions `flog.info`, `flog.warn` ... en complément de `message`, `warning`...
- Possible en utilisant la fonction `withCallingHandlers`

```
withCallingHandlers({
  # initialisation du fichier de logs
  flog.appender(appender.file("file.log"), name = "log.io")
```

```
  ...
  # calculs R
  ...
  # redirection
}, simpleError  = function(e){
  futile.logger::flog.fatal(gsub("^(Error in withCallingHandlers[[:punct:]]{3}[[:space:]]*)|(\n)*$", ""
}, warning = function(w){
  futile.logger::flog.warn(gsub("(\n)*$", "", w$message), name = "log.io")
}, message = function(m){
  futile.logger::flog.info(gsub("(\n)*$", "", m$message), name = "log.io")
})
```

Version des packages

Afin d'éviter des mauvaises surprises dues à un changement de version de packages **R**, il est préférable et conseillé de *figer* les versions par projet.

Le package **packrat** permet cela en reliant un projet à un dossier contenant les libraires nécessaires :

- **Isolation** : Installer / Mettre à jour un package n'a aucun impact sur les autres projets
- **Portable** : Multi-plateforme, passage simple d'un ordinateur à un autre
- **Reproducible** : Assurance d'éxéctuer le code avec les versions enregistrées

Nous allons présenter ici les opérations de base.

Plus d'informations : https://rstudio.github.io/packrat/

Packrat

Gestion du projet :

- `packrat::init()` : Initialisation du répertoire comme un projet **packrat**
- `packrat::snapshot()` : Sauvegarde de l'état / des versions actuelles des packages. (rappatriement du code source et des dépendances)
- `packrat::clean()` : Suppression des packages inutiles
- `packrat::status()` : Informations / statut

Partage du projet :

- `packrat::bundle()` : Création d'un "bundle", prêt à être partagé (code + librairies)
- `packrat::unbundle()` : Installation d'un "bundle"

Activation :

- `packrat::on()` : Activation de l'utilisation de packrat et des packages correspondants
- `packrat::off()` : Désactivation de l'utilisation de packrat

Aller plus loin...!

- The R Manuals : https://cran.r-project.org/manuals.html
- R Contributed Documentation : https://cran.r-project.org/other-docs.html
- Advanced R by Hadley Wickham : http://adv-r.had.co.nz/
- R packages by Hadley Wickham : http://r-pkgs.had.co.nz/
- Tests using testthat : http://r-pkgs.had.co.nz/tests.html

- Code coverage with covr : https://github.com/r-lib/covr
- How-to go parallel in R - basics + tips : http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/
- State of the Art in Parallel Computing withR : http://www.jstatsoft.org/v31/i01/paper
- R tutorial on the Apply family of functions : http://www.r-bloggers.com/r-tutorial-on-the-apply-family-of-functions/
- A Tutorial on Loops in R - Usage and Alternatives : http://blog.datacamp.com/tutorial-on-loops-in-r/