R : Bonnes pratiques / Développement / Production

B.Thieurmel - benoit.thieurmel@datastorm.fr

ENGIE : R Acceleration Week - 20/12/2019

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 éxécuter 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** (ligne 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

Passage des arguments

- `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

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).

---

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
```

```
# $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
```

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

```
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

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

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.

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
```

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/

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