Bonnes pratiques de programmation en R

Sophie Baillargeon, Université Laval

2018-02-09

Table des matières

Employer de **bonnes pratiques** de programmation, c'est respecter certaines **normes** afin de créer du code :

- utile :
  - roulant sans erreurs,
  - produisant les résultats escomptés,
  - roulant suffisamment vite ;
- facile à écrire :
  - développé en utilisant de bons outils,
  - en évitant les répétitions inutiles ;
- facile à comprendre :
  - qui se lit bien (presque comme du texte),
  - bien documenté ;
- facile à réutiliser, partager et mettre à jour.

Dans quels buts adopte-t-on de bonnes pratiques ? [1] Afin que

- le code soit **réellement utilisé** et que
- n'importe qui (en particulier soi-même dans le futur) soit capable de prendre un de nos projets et de comprendre
  - **ce que le code fait** et
  - **comment l'utiliser**.

À long terme, les bonnes pratiques apportent une **augmentation de notre productivité**.

Code qui produit des résultats valides

Les priorités lors du développement de tout code informatique sont certainement de produire du

- **code fonctionnel** (sans bogues),

[1] https://nicercode.github.io/blog/2013-04-05-why-nice-code/

- qui **produit les résultats escomptés**.

Pour arriver à ça, il est faut **tester** son code !

Vaut mieux tester fréquemment, à chaque petit ajout, plutôt que de produire beaucoup de code avant de le tester. Ainsi, il y a beaucoup **moins de débogage** à faire.

Code qui roule suffisamment vite

Après s'être assuré que les résultats sont valides, nous pouvons nous préoccuper du temps d'exécution.

Un code trop lent ne sera pas utilisé en pratique.

Pour produire du code qui roule vite, il faut :

- mettre en pratique quelques trucs simples,
- **comparer le temps d'exécution** de différentes façons de programmer une tâche,
- parfois faire du calcul en parallèle,
- parfois programmer des bouts de code dans un autre langage.

Un des derniers cours de la session sera consacré à l'optimisation du temps d'exécution de programmes R.

Code facile à écrire

- S'assurer de travailler avec la **dernière version** de R et des packages dont nous avons besoin :
  - nous risquons moins de rencontrer des bogues,
  - nous pouvons profiter des dernières fonctionnalités.
- Réutiliser le **code des autres** (R est un logiciel libre !) :
  - Nous perdrions notre temps à reprogrammer quelque chose qui existe déjà, à moins que l'implantation existante ne nous convienne pas.
- Utiliser un **éditeur** de code R, par exemple RStudio.
- Faire des copies de secours de nos fichiers de code R (**backups**) : en cas de pépin informatique, nous ne voulons pas perdre notre travail.
  - Bons outils pour des backups instantanés (avec une connexion internet) : Dropbox, Google Drive, OneDrive ou autre.
- Si nous développe du logiciel, soit des packages R, utiliser un **système de gestion de versions** (p. ex. Git ou Subversion intégré à RStudio).

Code facile à comprendre

Documenter

- Tout fichier de code R devrait comporter un **entête** pour informer de ce qu'il contient, qui l'a rédigé et quand il a été créé.
- Il devrait aussi y avoir des **commentaires dans le code** afin d'expliquer ce que les bouts de code font.
- Au besoin, un **guide d'utilisation** du code devrait être rédigé.

Suivre un guide de style

Un guide de style énonce des normes pour

- la syntaxe :
  - espacements, indentations, etc. ;

- l'organisation du code :
  - insertion de commentaires, ordre des éléments dans un programme, etc. ;
- la façon de nommer les objets et fichiers ;
- etc.

Quelques exemples de guides de style R :

- Google's R Style Guide : https://google.github.io/styleguide/Rguide.xml
- Hadley Wickham's R Style Guide : http://adv-r.had.co.nz/Style.html

Note : Les espacements proposés dans ces guides de style concordent avec la façon dont R reformate automatiquement le code des fonctions dans des packages avant de les afficher dans la console.

Conventions de noms

Les noms des fonctions dans les packages de base de R ne suivent pas une unique convention pour les noms[2]. Voici, par exemple, quelques conventions retrouvées en R :

- tout en minuscules (ex. `typeof`, `colnames`),
- mots séparés par un point (ex. `data.frame`, `read.table`),
- mots séparés par un trait de soulignement (ex. `seq_along`),
- premières lettres des mots en majuscule, sauf pour le premier mot (ex. `colMeans`, `rowSums`).

Bonne pratique = **choisir une convention de noms et la respecter**.

Conseil : **Éviter les accents** dans les commandes R, donc dans les noms d'objets (ou de sous-objets tels que les colonnes d'un data frame) afin d'éviter des problèmes lors du partage de code ou d'objets R entre des utilisateurs qui ne travaillent pas sous le même système d'exploitation.

Opérateur d'assignation

Voici quelques différences entre les opérateurs d'assignation `<-` et `=`. [3] [4]

Passage de valeurs à des arguments dans des appels de fonction :

Pour **passer des valeurs d'argument dans un appel de fonction**, l'opérateur d'assignation `=` est nettement préférable, car il ne crée pas d'objets dans l'environnement de travail, contrairement à `<-`. Par exemple, supposons que l'on travaille à partir d'un environnement de travail vide.

```
ls()
```

```
## character(0)
```

Soumettons la commande suivante.

```
mean(x = 1:5)
```

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

La commande a retourné le résultat escompté, sans modifier l'environnement de travail.

```
ls()
```

```
## character(0)
```

[2] https://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf
[3] R Inferno, section 8.2.26 : http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
[4] http://stackoverflow.com/questions/1741820/assignment-operators-in-r-and

Tentons d'utiliser l'opérateur `<-` pour passer la valeur de l'argument `x`.

```
mean(x <- 1:5)
```

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

Que s'est-il passé ?

Le bon résultat a été retourné par la commande. Cependant, un objet nommé `x` a été créé dans l'environnement de travail.

```
ls()
```

```
## [1] "x"
```

```
x
```

```
## [1] 1 2 3 4 5
```

Conclusion : Utiliser `<-` pour passer des valeurs aux arguments dans des appels de fonctions n'est pas recommandé. Cette pratique a pour conséquence de polluer l'environnement de travail.

Assignation d'une valeur à un nom (création ou modification d'objets) :

Maintenant, pour **assigner des valeurs à des noms** et ainsi créer ou modifier des objets R, est-ce qu'il y a une différence entre l'utilisation de `<-` et de `=` ?

Dans la majorité des situations, les deux opérateurs produiront exactement le même résultat. Cependant, l'inclusion d'une assignation avec l'opérateur `=` dans une expression passée en argument à une fonction produira presque toujours une erreur.

Par exemple, la fonction `system.time` est très utile pour évaluer le temps pris pour évaluer une expression R. Considérons par exemple la commande suivante.

```
res <- apply(X = matrix(1, nrow = 10000, ncol = 1000), MARGIN = 1, FUN = median)
```

Nous pouvons facilement évaluer son temps d'exécution en encadrant l'expression d'un appel à la fonction `system.time` comme suit.

```
system.time(res <- apply(X = matrix(1, nrow = 10000, ncol = 1000), MARGIN = 1, FUN = median))
```

```
##    user  system elapsed
##    0.61    0.02    0.64
```

Cependant, si dans cette expression l'opérateur d'assignation `=` avait été utilisé, nous aurions obtenu une erreur.

```
system.time(res = apply(X = matrix(1, nrow = 10000, ncol = 1000), MARGIN = 1, FUN = median))
```

```
## Error: in system.time(res = apply(X = matrix(1, nrow = 10000, ncol = 1000),  :
##   unused argument (res = apply(X = matrix(1, nrow = 10000, ncol = 1000), MARGIN = 1, FUN = median))
```

Cette erreur est générée parce que R croit que nous tentons de passer un argument nommé `res` à la fonction `system.time` alors qu'elle n'accepte pas d'argument portant ce nom.

Résumé : Toute erreur est évitée en utilisant

- `<-` (ou `->`) pour l'assignation d'une valeur à un nom (création ou modification d'objets),
- `=` pour passer des valeurs d'argument dans un appel de fonction.

Opérateur « pipe »

Le package `magrittr` propose de nouveaux opérateurs R appelés « pipes », en faisant référence à la signification « tuyaux ». Ces opérateurs offrent de nouvelles façons d'enchaîner des commandes et de passer des arguments à des fonctions. Je présente ici seulement l'opérateur principal, le « forward-pipe operator » `%>%`.

Pour résumer le fonctionnement de cet opérateur, voici comment il transforme quelques appels de fonction :

- `f(x)` devient `x %>% f`,
- `f(x, y)` devient `x %>% f(y)`,
- `h(g(f(x)))` devient `x %>% f %>% g %>% h`.

Cet opérateur est présenté dans ces notes sur les bonnes pratiques de programmation en R, car certains considèrent qu'il permet d'écrire du **code R plus clair**. En lisant de gauche à droite la commande `h(g(f(x)))`, nous voyons d'abord l'appel à la fonction `h`, puis l'appel à la fonction `g` et finalement l'appel à la fonction `f`. Pourtant, pour évaluer cette commande, R va d'abord :

- évaluer `f(x)`,
- puis il passera le résultat obtenu à la fonction `g` et évaluera le résultat,
- qui sera passé à la fonction `h` et le résultat final sera retourné.

Ainsi, l'évaluation de la commande se réalise dans l'ordre inverse de la lecture de la commande (à condition de lire de gauche à droite).

Si nous voulions écrire un code qui reflète bien l'ordre des évaluations, nous pourrions écrire :

```
res1 <- f(x)
res2 <- g(res1)
h(res2)
```

Mais ce code a le défaut de créer des objets que nous ne souhaitons pas nécessairement conserver. L'opérateur `%>%` n'a pas ce défaut ! **Une commande écrite en utilisant l'opérateur `%>%` permet de suivre l'ordre des évaluations**, sans créer inutilement d'objets dans l'environnement de travail.

Pour encore plus de clarté, certains étendent sur plusieurs lignes une commande contenant plus d'un opérateur `%>%` comme suit :

```
x %>%
  f %>%
  g %>%
  h
```

Si l'argument que nous souhaitons passer avec l'opérateur `%>%` n'est pas celui en première position, il faut utiliser un `.` comme suit :

- `f(y, x)` devient `x %>% f(y, .)`,
- `f(y, z = x)` devient `x %>% f(y, z = .)`.

Voici un exemple d'utilisation de l'opérateur `%>%`. Supposons que nous avons la chaîne de caractères suivante :

```
text <- "Ceci est un example"
```

et que nous souhaitons corriger deux fautes dans cette phrase : le mot example écrit en anglais plutôt qu'en français et le point manquant à la fin de la phrase. Voici une commande R pour réaliser cette tâche.

```
paste0(gsub(pattern = "example", replacement = "exemple", x = text), ".")
```

```
## [1] "Ceci est un exemple."
```

Cette commande est un peu difficile à lire en raison de l'appel à la fonction `gsub` imbriqué dans un appel à la fonction `paste0`. Nous pourrions la réécrire comme suit avec l'opérateur `%>%` :

```r
library(magrittr)
text %>%
  gsub(pattern = "example", replacement = "exemple", x = .) %>%
  paste0(., ".")
```

```
## [1] "Ceci est un exemple."
```

Références pour plus d'information sur l'opérateur `%>%` et les autres opérateurs de du package `magrittr` :

- https://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html
- https://github.com/tidyverse/magrittr
- https://www.datacamp.com/community/tutorials/pipe-r-tutorial

Retours à la ligne et indentations

Une autre façon de rendre son code plus lisible est d'y insérer des retours à la ligne et des indentations appropriées. Reprenons le code précédent.

```r
paste0(gsub(pattern = "example", replacement = "exemple", x = text), ".")
```

Il est plus facile à lire comme suit,

```r
paste0(gsub(pattern = "example",
            replacement = "exemple",
            x = text),
       ".")
```

ou encore comme ça.

```r
paste0(
  gsub(
    pattern = "example",
    replacement = "exemple",
    x = text
  ),
  "."
)
```

Code facile à réutiliser, partager et mettre à jour

Un code facile à comprendre est aussi plus facile à réutiliser. Il y a aussi un autre truc important pour rendre un code facile à réutiliser, que voici.

Création de fonctions

Pour réutiliser un bout de code, nous pouvons le copier/coller, puis le modifier un peu pour l'adapter à la nouvelle tâche à réaliser (par exemple utiliser un nouveau jeu de données, ajuster un modèle un peu différent, etc.). Cette façon de procéder fait souvent très bien l'affaire.

Cependant, si nous nous retrouvons à copier/coller souvent le même bout de code, c'est le signe que nous aurions avantage à créer une fonction à partir de ce code. Créer la fonction prend un certain temps, mais cet investissement en temps finit par faire sauver du temps si nous utilisons souvent la fonction.

Ainsi **créer des fonctions** pour les tâches que nous effectuons souvent est une bonne pratique de programmation. Cette pratique rend le code plus facilement réutilisable.

De plus, le risque d'erreur est moins grand lors de l'appel d'une fonction que lors de la modification manuellement d'un bout de code copié/collé.

Il faut bien sûr s'assurer de documenter toute fonction créée, afin de comprendre dans le futur ce qu'elle fait. Une documentation de fonction devrait toujours comprendre minimalement une description de :

- ce que fait la fonction,
- les arguments acceptés en entrée,
- les résultats produits (sortie, graphique, écriture dans un fichier, etc.).

La création de fonctions est la principale façon en R d'appliquer le principe de programmation informatique DRY (*Don't Repeat Yourself*)[5]. Lorsque nous appliquons ce principe de ne pas répéter inutilement des bouts de codes, les mises à jour de programmes R sont beaucoup plus rapide à réaliser.

Bonnes habitudes de travail en R

- Rédiger son code dans un programme (script) R, et enregistrer ce programme fréquemment.
  - **Erreur évitée** : Perdre la trace de certaines commandes importantes parce qu'elles ont été écrites directement dans la console.
- Débuter toute session de travail en R avec un environnement de travail vide. Pour ce faire, il faut empêcher la restauration automatique d'une image de session. En fait, pour empêcher ça, il vaut mieux ne jamais enregistrer d'images de session.
  - **Erreur évitée** : Ne pas être conscient de la présence de certains objets dans notre environnement de travail et les faire intervenir dans notre code sans s'en rendre compte, ce qui pourrait produire des résultats inattendus et difficiles à comprendre.
- Ne pas utiliser la fonction `load`, ou du moins être prudent lors de son utilisation.
  - **Erreur évitée** : Modifier un objet de l'environnement de travail en l'écrasant (sans s'en rendre compte) par un objet portant le même nom.
- Ne pas utiliser la fonction `attach`.
  - Nous verrons pourquoi dans un prochain cours.
- Avant de modifier des options ou des paramètres graphiques, garder une copie de leurs valeurs par défaut, et réactiver ces valeurs par défaut au moment opportun.
  - **Erreur évitée** : Oublier qu'une option ou un paramètre graphique n'a plus sa valeur par défaut, qu'il a été modifié.

```
# Avant toute modification
options.default <- options()
par.defaut <- par(no.readonly = TRUE)

# Après le bout de code concerné
options(options.default)
par(par.defaut)
```

[5] http://fr.wikipedia.org/wiki/Ne_vous_r%C3%A9p%C3%A9tez_pas

Résumé des bonnes pratiques en programmation R

1. **tester** fréquemment son code ;
2. **optimiser le temps d'exécution** de son code s'il est lent ;
3. **utiliser de bons outils** :
   - version la plus à jour de R et des packages exploités,
   - code des autres (le réutiliser plutôt que de le réécrire),
   - éditeur de code R,
   - système effectuant des copies de secours,
   - système de gestion de versions pour le développement de packages ;
4. **documenter** son code ;
5. **suivre un guide de style** ;
6. **créer des fonctions** pour éviter de se répéter (et documenter ces fonctions).