Bonnes pratiques de programmation en R

Sophie Baillargeon, Université Laval

2019-02-11

Table des matières

Introduction

Peu importe le langage informatique, employer de **bonnes pratiques** de programmation signifie respecter certaines **normes** afin de créer du **bon code**.

Qu'est-ce que du bon code ? Un bon code est un code qui **produit les résultats escomptés**, **est facile à maintenir** et **roule suffisamment rapidement**.

Dans quels buts adoptons-nous de bonnes pratiques ? Afin que notre code soit **réellement utilisé** et que n'importe qui (en particulier soi-même dans le futur) soit capable de comprendre **ce que le code fait** et **comment s'en servir**.

À long terme, les bonnes pratiques apportent une **augmentation de notre productivité**[1] en évitant les répétitions inutiles.

Objectif #1 : Code qui produit les résultats escomptés

La priorité lors du développement de tout code informatique est certainement l'écriture d'un code qui réalise bien ce qu'il doit réaliser. Donc, un bon code doit produire les bons résultats. Pour y arriver, le code doit

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

d'abord être fonctionnel, c'est-à-dire ne pas contenir de bogues.

Pour s'assurer d'écrire du code qui fonctionne correctement, il faut simplement le **tester**. Il 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. Un courant de pensée en informatique prône même l'écriture des tests avant l'écriture du code (le *test driven development*).

En R, il est facile de tester interactivement son code dans la console. Cependant, une meilleure pratique est de formaliser ses tests afin de pouvoir facilement les faire rouler de nouveau lors de modifications futures au code. Nous verrons dans un prochain cours comment écrire des tests unitaires en R.

Objectif #2 : Code facile à maintenir

Maintenir un code informatique signifie de s'assurer qu'il continue de fonctionner correctement dans le futur, malgré les modifications qui lui sont apportées. Un code utilisé fréquemment est un code appelé à être mis à jour, soit pour y ajouter des fonctionnalités ou pour corriger des bogues non détectés par les tests, mais découverts par des utilisateurs.

Reprendre un code écrit par quelqu'un d'autre, ou écrit par nous-mêmes quelques mois auparavant, n'est pas toujours une tâche facile. Cependant, s'il s'agit d'un bon code, il ne devrait pas être trop difficile à comprendre et à modifier.

Les sous-sections suivantes présentent des conseils pour faciliter la maintenance de code R en simplifiant son **écriture**, sa **compréhension** et sa **réutilisation**.

Code facile à écrire

L'écriture de code R est facilitée par l'utilisation de **bons outils**.

Premièrement, il vaut mieux s'assurer de travailler avec la **dernière version** de R et des packages dont nous avons besoin. Ainsi, nous risquons moins de rencontrer des bogues et nous pouvons profiter des dernières fonctionnalités. De plus, utiliser un **éditeur** de code R ou même un environnement intégré de développement, tel que RStudio, facilite le travail.

Avant de se lancer dans l'écriture d'un bout de code R, il est cependant recommandé de vérifier si un code réalisant ce que nous voulons faire n'a pas déjà été écrit. R est un logiciel libre, la **réutilisation du code des autres** est permise et même encouragée (tant que nous citons nos sources). Nous perdrions notre temps à reprogrammer quelque chose qui existe déjà, à moins que l'implantation existante ne nous convienne pas.

Finalement, il est primordial de s'assurer de ne pas perdre par accident le code que nous écrivons. Un pépin informatique est si vite arrivé. Comme pour tout travail effectué à l'ordinateur, il est recommandé de faire des copies de secours (**backups**) de nos fichiers de code R. Nous pouvons simplement utiliser pour ce faire un outil de backups instantanés (avec une connexion internet) tel que Dropbox, Google Drive, OneDrive ou autre. Si nous développons du logiciel, soit des packages R, l'utilisation d'un **logiciel de gestion de versions** tel que Git ou Subversion (intégrable à RStudio) est encore plus approprié.

Code facile à comprendre

Un code facile à comprendre est **clair et se lit bien** (presque comme du texte). Il comporte souvent des instructions qui parlent d'elles-mêmes. Ces instructions sont typiquement succinctes, car une instruction trop longue effectue souvent plusieurs tâches difficilement discernables. Si la lecture d'une instruction ne permet pas à un programmeur initié dans le langage informatique employé de comprendre ce qu'elle réalise, il est alors recommandé d'insérer un commentaire dans le code pour expliquer à quoi sert l'instruction.

Documentation

En plus de commentaires pour expliquer certaines instructions, 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.

Toute **fonction** créée devrait aussi être documentée afin de comprendre dans le futur ce qu'elle fait. Une documentation de fonction devrait 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.).

Au besoin, un **guide d'utilisation** du code peut aussi être rédigé.

Guide de style

Un guide de style énonce des normes pour :

- la syntaxe :
  - espacements, indentations, opérateur d'assignation, etc. ;
- la façon de nommer les objets et fichiers ;
- l'organisation du code :
  - insertion de commentaires, ordre des éléments dans un programme, répartition du code entre plusieurs fichiers, etc. ;
- etc.

Une syntaxe uniforme rend un code beaucoup plus facile à lire.

Voici quelques exemples de guides de style R :

- Google's R style guide : https://google.github.io/styleguide/Rguide.xml
- The `tidyverse` style guide : https://style.tidyverse.org/

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.

Retours à la ligne et indentations

Une façon simple de rendre son code plus lisible est d'y insérer des retours à la ligne et des indentations appropriées. Par exemple, 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 instruction R pour réaliser cette tâche.

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

Cette instruction comporte un appel de fonction imbriqué dans un autre. Elle serait plus facile à lire comme suit :

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

ou encore comme ça :

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

Opérateur d'assignation

Voici quelques différences[2][3] entre les opérateurs d'assignation `<-` et `=`.

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

Pour **passer des valeurs d'arguments 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)
```

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)
```

[2] R Inferno, section 8.2.26 : http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
[3] http://stackoverflow.com/questions/1741820/assignment-operators-in-r-and

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
##    0.73    0.00    0.75
```

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.

La page web suivante fournit plus de renseignements concernant les différences entre les opérateurs d'assignation `<-` et `=` : https://colinfay.me/r-assignment/.

Conventions de noms

Les noms des fonctions dans les packages de base de R ne suivent pas une unique convention pour les noms[4]. 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`).

Une bonne pratique est de **choisir une convention de noms et de la respecter**.

Conseil : Il est préférable d'**éviter les accents** dans les instructions 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.

Organisation du code

Un long code R a avantage a être séparé en plusieurs fichiers, par exemple un pour les fonctions et un autre pour les instructions à soumettre interactivement dans la console, ou encore un fichier par section du code. De plus, un code R dépend souvent d'autres fichiers, par exemple des fichiers de données à importer. Comment organiser intelligemment tous ces fichiers ? Il est recommandé de créer des sous-dossiers regroupant les fichiers de même type, par exemple un sous-dossier pour les données, un autre pour les résultats, un pour les scripts R, etc.[5]

Chaque fichier devrait avoir un nom judicieusement choisi[6]. Il devrait idéalement être :

- significatif : bien décrire ce qu'il contient ;
- facile à traiter dans un programme informatique : ne pas contenir d'espaces, ni d'accents ;

[4] https://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf
[5] https://andrewbtran.github.io/NICAR/2018/workflow/docs/01-workflow_intro.html
[6] https://speakerdeck.com/jennybc/how-to-name-files

- bien adapté à l'ordonnancement par défaut dans les explorateurs de fichiers : date sous le format année-mois-jour, débuter par un nombre pour forcer un ordre (en s'assurant que les nombres ont tous la même quantité de caractères, en complétant à gauche par des zéros les plus petits nombres).

Les projets RStudio, déjà mentionnés dans les notes sur la lecture et l'écriture dans des fichiers externes à partir de R, sont parfaits pour rassembler au même endroit tous les fichiers relatifs à du code R et pour faciliter le travail sur plusieurs projets simultanément en gérant le passage d'un répertoire de travail à un autre.

Opérateur « pipe »

Le guide de style du `tidyverse` recommande l'utilisation d'un opérateur appelé « pipe », en faisant référence à la signification « tuyaux ». Cet opérateur offre une nouvelle façon d'enchaîner des instructions et de passer des arguments à des fonctions. Il est offert dans le package `magrittr`. En fait, ce package propose plusieurs opérateurs « pipes », mais 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`.

Les auteurs du `tidyverse` considèrent que cet opérateur permet d'écrire du **code R plus clair**. En lisant de gauche à droite l'instruction `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 instruction, 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 l'instruction se réalise dans l'ordre inverse de sa lecture (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 instruction 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 instruction 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 = .)`.

Reprenons un exemple précédent pour illustrer l'utilisation de l'opérateur `%>%`. Supposons que nous avons la chaîne de caractères suivante :

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

et que nous souhaitons la corriger avec l'instruction suivante :

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

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

Cette instruction 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 `%>%` :

```
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

Code facile à réutiliser

Un code facile à comprendre est aussi plus facile à réutiliser. Mais l'ultime façon de faciliter la réutilisation d'un code est d'en faire des fonctions. Et pour partager ces fonctions, rien de mieux qu'un package.

Créer des 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.

La création de fonctions est la principale façon en R d'appliquer le principe de programmation informatique DRY (*Don't Repeat Yourself*)[7]. Lorsque nous appliquons ce principe de ne pas répéter inutilement des bouts de codes, les mises à jour de programmes R sont beaucoup plus rapides à réaliser.

Créer des packages

Un package permet de regrouper des fonctions R et de les documenter dans un format uniforme. Ensuite, le package peut être facilement partagé avec nos collègues, ou même offert à tous sur le web. Nous reparlerons des avantages des packages et verrons comment en créer dans un prochain cours.

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

Objectif #3 : Code qui roule suffisamment rapidement

Après nous être assuré que notre code fonctionne correctement et qu'il est facilement maintenable, nous pouvons nous préoccuper de son temps d'exécution. Il ne s'agit pas du critère le plus important pour définir ce qu'est du bon code, mais c'est tout de même un critère à ne pas négliger, car un code trop lent risque de ne pas être utilisé.

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.

Quelques bonnes habitudes de travail spécifiques à R

- Rédiger son code dans un programme (script) R, et enregistrer ce programme fréquemment.
  - **Erreur évitée** : Perdre la trace de certaines instructions importantes parce qu'elles ont été écrites directement dans la console.
- Il est préférable de 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, je suis d'avis qu'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` lorsque notre environnement de travail n'est pas vide, 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`, ou du moins être prudent lors de son utilisation.
  - 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)
```

- Ne pas utiliser `T` et `F` au lieu de `TRUE` et `FALSE`.
  - **Erreur évitée** : Posséder un objet nommé `T` ou `F` dans notre environnement de travail (R nous empêche de nommer un objet `TRUE` ou `FALSE`, mais pas `T` et `F`), et donc référer à cet objet plutôt qu'à `TRUE` ou `FALSE` lors de l'utilisation de `T` ou `F`.

Synthèse

En résumé, pour adopter de bonnes pratiques en programmation R il faut :

1. **tester** fréquemment son code ;
2. **utiliser de bons outils** :
    - version la plus à jour de R et des packages exploités,
    - éditeur de code R ou environnement intégré de développement R,
    - code des autres (le réutiliser plutôt que de le réécrire),
    - système effectuant des copies de secours,
    - logiciel de gestion de versions pour le développement de packages ;
3. **documenter** son code ;
4. **suivre un guide de style** ;
    - utiliser une syntaxe uniforme,
    - adopter une convention de noms,
    - organiser intelligemment ses fichiers ;
5. **créer des fonctions** pour éviter de se répéter (et documenter ses fonctions) ;
6. **créer des packages** avec les fonctions à partager ;
7. **optimiser le temps d'exécution** de son code s'il est lent.