Au coeur de R

Jean-Michel Marin

January 2024

Faculty of Sciences, University of Montpellier

Introduction

Introduction

To understand computations in R, two slogans are helpful:
Everything that exists is an object.
Everything that happens is a function call.
John Chambers

R est, dans son essence, un langage de programmation fonctionnelle.

L'utilisation de fonctions est ainsi la manière privilégiée de développer un programme en R.

C'est pourquoi il est essentiel de bien comprendre la manière dont elles fonctionnent et interagissent avec l'environnement de l'utilisateur pour développer des programmes fiables et reproductibles.

Introduction

Construire un projet autour d'un enchaînement de fonctions est une pratique qui permet de réduire les risques de bug (en identifiant mieux leur origine) et, surtout, assure un code plus flexible et reproductible.

Cela renvoie au paradigme informatique du "do not repeat yourself".

Cette approche nous intéresse particulièrement car elle rend le code plus clair, plus robuste, plus facile à répliquer et à faire évoluer.

Introduction

Néanmoins, adopter une approche de programmation fonctionnelle en R est un bon début mais ne suffit pas.

- Rappels théoriques sur les objets en R
- Construction de premières fonctions en R avec une adoption précoce de bonnes pratiques
- Compréhension de la manière dont R gère les objets dans les différents environnements de travail disponibles

Rappels sur les objets en R

Rappels sur les objets en R

Il y a deux points essentiels à retenir sur la façon dont R traite les objets:

- Comment R associe les objets et les noms;
- Comment R modifie les objets.

Distinguer noms et valeurs

La première distinction importante est celle entre un objet et son nom. A ce stade, vous devriez être habitué à l'assignation:

```
x <- c(1, 2, 3)
```

On serait tenté de lire "je crée un objet appelé 'x' contenant les valeurs 1,2 et 3". En réalité, R procède de la manière suivante:

- R crée un vecteur de valeurs c(1, 2, 3) ;
- R lie (binds) ce vecteur à un nom, ici x.

Distinguer noms et valeurs

Autrement dit, l'objet/la valeur n'a pas de nom ; c'est le nom qui a une valeur. L'opérateur assignation <- crée donc un lien entre un nom et une valeur. Un nom est une référence à une valeur: cela évite que le code ci-dessous

```
y <- x
```

crée une copie (mobilisant de la mémoire) d'un objet déjà existant (x). Cette assignation crée simplement une référence supplémentaire pour accéder au vecteur c(1,2,3), en liant ce vecteur au nom y.

Distinguer noms et valeurs

En revanche, R duplique l'objet lorsqu'on modifie un nom qui y fait référence. Le code ci-dessous lie x à y puis modifie y:

```
x <- c(1, 2, 3)
y <- x
y[[3]] <- 4
```

Distinguer noms et valeurs

On pourrait penser que cette opération a modifié x, mais ce n'est pas le cas :

```
x
## [1] 1 2 3
```

Alors que la valeur associée à y a changé, l'objet original n'a pas changé. R a simplement créé un nouvel objet le vecteur c(1,2,4), et lui a associé le nom y.

Copier des objets en R

Les objets R sont en principe non-modifiables (immutable). Parler d'immutabilité à propos des objets R peut paraître surprenant puisqu'on peut apparemment ajouter des élements à un objet R:

```
a <- list()
a[[1]] <- "je remplis ma liste"
a
## [[1]]
## [1] "je remplis ma liste"
```

Copier des objets en R

En réalité, R a dupliqué temporairement a (copy) et ensuite modifié cet objet avant le ré-assigner au nom a (modify).

Ce comportement est appelé copy-on-modify.

Cette approche rend le langage R très flexible en permettant, par exemple, de faire évoluer la classe d'un objet (à partir d'une réassignation).

Copier des objets en R

Par exemple, le code suivant transforme l'objet lié au nom a de vector en data.frame :

```
a <- c(1,2,3)
a <- data.frame("a" = a)
class(a)
## [1] "data.frame"
```

Rappels sur les fonctions

Rappels sur les fonctions

On peut décomposer une fonction en trois parties :

- Arguments (ex: formals(lm)) : la liste des arguments qui contrôlent l'appel à une fonction
- Corps (ex: body(lm)) : les commandes internes à la fonction
- Environnement (ex: environment(lm)) : le contexte qui détermine comment la fonction trouve des valeurs associées aux noms évoqués.

Rappels sur les fonctions

Dans le cas de R, ces noms peuvent renvoyer à tout type d'objet: données, variables, fonctions, packages...

Alors que les arguments et le corps de la fonction sont définis explicitement, l'environnement est lui défini implicitement, selon le contexte dans lequel est évoqué la fonction.

Rappels sur les fonctions

```r
f02 <- function(x, y) {  # Un commentaire
  x + y
}
formals(f02)
## $x
##
##
## $y
```

Rappels sur les fonctions

```
body(f02)
## {
##     x + y
## }
environment(f02)
## <environment: R_GlobalEnv>
```

Portée (scoping) d'une fonction

Comme on l'a vu, l'assignation est l'acte d'associer un nom à une valeur.

Le scoping (portée) est l'action de trouver une valeur associée à un nom.

Par exemple, que va renvoyer le code suivant: 10, 20, 25 ?

```
x <- 10
g01 <- function() {
  x <- 20
  x
}
g01()
## [1] 20
```

Noms masqués

Le principe de base de la portée est que les noms définis au sein d'une fonction masquent les noms définis en dehors.

Le code suivant montre un exemple de cette régle :

```
x <- 10 ; y <- 20
g02 <- function() {
  x <- 1 ; y <- 2
  c(x, y)
}
g02()
## [1] 1 2
```

Noms masqués

Si R ne trouve pas un nom dans une fonction, il cherchera celui-ci dans un environnement situé un niveau au-dessus.

```
x <- 2
g03 <- function() { y <- 1 ; c(x, y)
}
g03()
## [1] 2 1
# La commande ci-dessous ne change pas la valeur précédente de y
y
## [1] 20
```

Noms masqués

Cette règle s'appliquera de la même manière pour les fonctions emboîtées : R cherche d'abord dans l'environnement de la fonction, puis dans celui de la fonction au-dessus et ainsi de suite jusqu'à l'environnement global.

S'il ne trouve toujours pas le nom recherché (par exemple une fonction sum), R finira par chercher dans la liste des packages chargés.

Noms masqués

Un bon exemple de portée d'une fonction est le suivant:

```
# On reprend le vecteur a
a <- c(1,2,3)
g11 <- function() { if (!exists("a")) {
    a <- 1 } else {a <- a + 1}
  a
}
g11()
## [1] 2 3 4
g11()
## [1] 2 3 4
```

Noms masqués

Pourquoi g11() retourne-t-elle toujours la même valeur ?

C'est parce que chaque fois qu'une fonction est appelée, un nouvel environnement est créé pour héberger son exécution.

Une fonction est donc incapable de connaître les noms qui ont pu être créés par une fonction passée sans return.

Portée et évaluation paresseuse

La portée permet de lier logiquement des objets entre eux, mais ne permet pas de déterminer quand ce lien prendra place (on parle d'évaluation paresseuse ou lazy evaluation : on donne un plan à R pour lier les objets, on ne lui dit pas encore quand la commande sera exécutée).

R associe les noms aux valeurs quand la fonction est exécutée, et pas à sa création.

Portée et évaluation paresseuse

Dans le cadre de l'évaluation paresseuse, ce qui permet à R de lier les objets et environnements entre eux est la promesse (promise).

Elle comprend trois éléments:

- Une expression, comme x + y, qui amènera à de futurs calculs ;
- Un environnement, où l'expression sera évaluée, i.e. l'environnement où la fonction est évaluée ;
- Une valeur qui est calculée et gardée en mémoire au sein de l'environnement (de manière à éviter de faire deux fois le même calcul si le même plan est appelé deux fois dans le même environnement).

L'évaluation paresseuse a deux grandes conséquences sur le fonctionnement de R.

Portée et évaluation paresseuse

Premièrement, l'output d'une fonction peut différer selon les objets hors de l'environnement de la fonction.

Ces objets sont parfois appelés variables globales parce qu'ils affectent l'ensemble des fonctions, par opposition aux variables locales qui sont propres à une fonction, et n'existent que dans l'environnement d'exécution de cette fonction.

Portée et évaluation paresseuse

En voici un exemple simple:

```
g12 <- function() x + 1
x <- 15
g12()
## [1] 16
x <- 20
g12()
## [1] 21
```

La modification de x (variable globale) change donc l'output de la fonction.

Une manière de rapidement savoir quelles sont les variables globales qui peuvent avoir un effet sur les résultats est d'utiliser la fonction codetools::findGlobals().

Portée et évaluation paresseuse

Deuxièmement, les arguments d'une fonction en R ne sont évalués qu'au moment où il est nécessaire de connaître leur valeur pour continuer l'exécution du programme.

Par exemple, le code suivant ne provoquera pas d'erreur car x n'est jamais utilisé et n'est donc jamais évalué:

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
## [1] 10
```

Portée et évaluation paresseuse

C’est très pratique car cela permet de créer des fonctions qui peuvent faire des opérations très différentes selon le type d’argument. Par exemple, créez une fonction add(x,y) qui

- ajoute x et y si x et y sont numériques;
- concatène x avec y si x et y sont des chaînes de caractères;
- renvoie l’erreur “ ‘x’ et ‘y’ ne sont pas de même nature” si x et y ne sont pas de même type.

Portée et évaluation paresseuse

```
add <- function(x,y){
  if ((class(x) == "numeric") & (class(y) == "numeric")){
    x + y
  } else if ((class(x) == "character") & (class(y) == "character")){
    paste0(x,y) } else {
    stop("'x' et 'y' ne sont pas de même nature") } }
add(1,2)
## [1] 3
add("a","b")
## [1] "ab"
add(1,"b")
## Error in add(1, "b"): 'x' et 'y' ne sont pas de même nature
```

Définir des fonctions en R

Un exemple simple pour commencer

Supposons qu'on dispose d'une table de données qui utilise le code -99 pour représenter les valeurs manquantes. On désire remplacer l'ensemble des -99 par des NA.

```
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6] ; df
##   a   b   c   d  e f
## 1 7   5 -99   2  5 2
## 2 5   5   5   3  6 1
## 3 6   8   5   9  9 4
## 4 4   2   2   6  6 8
## 5 6   7   6 -99 10 6
## 6 9 -99   4   7  5 1
```

Un exemple simple pour commencer - Les défauts d'un code sans fonction

Un premier jet de code pourrait prendre la forme suivante

```
df2 <- df
df2$a[df2$a == -99] <- NA
df2$b[df2$b == -99] <- NA
df2$c[df2$c == -99] <- NA
df2$d[df2$d == -99] <- NA
df2$e[df2$e == -98] <- NA
df2$f[df2$g == -99] <- NA
```

Un exemple simple pour commencer - Les défauts d'un code sans fonction

- Le code est long et répétitif, ce qui nuit à sa lisibilité
- Le code est très dépendant de la structure des données (nom et nombre de colonnes) et doit être adapté dès que celle-ci évolue
- On a introduit une erreur humaine dans le code, difficile à détecter, dans l'instruction

```
df2$e[df2$e == -98] <- NA
```

Un exemple simple pour commencer

La tâche ici étant identique pour toutes les colonnes, on a envie de mettre en place une structure générale qui rendrait le code plus concis.

Pour cela, nous allons construire une fonction avec R. Cette fonction prendra la forme suivante

```
fix_missing <- function(x) {
  x[x == -99] <- NA ; x }
```

Un exemple simple pour commencer

La famille des fonctions apply est conçue pour appliquer une même fonction à plusieurs objets.

Ces fonctions font partie du langage R de base et sont ce qu'on appelle des fonctionnelles car elles prennent une fonction comme argument.

La plus puissante des fonctions apply est la fonction lapply qui permet de stocker le résultat sous forme de liste.

Il est préférable d'utiliser lapply plutôt que les autres fonctions apply (pour éviter des problèmes de conversion de type de données).

Un exemple simple pour commencer

Voici un exemple très simple d'utilisation de lapply:

```
ajouter1 <- function(x){ y <- x + 1 ; return(y) }
x <- list(1, 2, 3, 4)
lapply(x, ajouter1)
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 5
```

Un exemple simple pour commencer

Nous allons maintenant utiliser lapply pour améliorer notre code.

Il est possible d'appliquer lapply à un dataframe car un dataframe est en fait une liste de vecteurs (les colonnes du dataframe).

Toutefois, comme lapply renvoie une liste, on a besoin d'utiliser une petite astuce pour s'assurer que la sortie de la fonction prenne la forme d'un dataframe et non d'une liste. A la place d'assigner le résultat sous la forme df <- lapply(.....) on va faire df[] <- lapply(...).

Un exemple simple pour commencer

```
df2 <- df
fix_missing <- function(x) {
  x[x == -99] <- NA ; x }
df2[] <- lapply(df2, fix_missing)
```

Un exemple simple pour commencer

Cette approche de composition de fonctions est très puissante et explique la grande flexibilité du langage R.

Ecrire des fonctions simples et les composer dans une ou plusieurs fonctions maîtres est également très utile dans un travail collaboratif car cela permet de décomposer un traitement complexe en tâches simples et de mieux en comprendre le déroulement.

Créer des fonctions complexes

Comment faire maintenant si on désire autoriser d'autres valeurs que -99 ?

Notre fonction précédente prenait comme donnée que la valeur -99 désignait les valeurs manquantes.

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA ; x }
```

Créer des fonctions complexes

```
df2 <- df
df2[] <- lapply(df2, fix_missing, na.value = -99)
df2
##   a  b  c  d  e f
## 1 7  5 NA  2  5 2
## 2 5  5  5  3  6 1
## 3 6  8  5  9  9 4
## 4 4  2  2  6  6 8
## 5 6  7  6 NA 10 6
## 6 9 NA  4  7  5 1
```

Créer des fonctions complexes

Nous allons généraliser encore une fois le traitement.

Imaginons que toutes les variables ne soient pas codées de la même manière.

Pour simplifier, réduisons le problème aux variables a, b et c. Imaginons que les valeurs manquantes soient indiquées par les valeurs 1 à 3 pour la variable a, par les valeurs 4 à 6 pour la variable b et par les valeurs 7 à 10 pour la variable c.

Créer des fonctions complexes

```
fix_missing2 <- function(x, na.value) { x[x %in% na.value] <- NA ; return(x) }
codes_na <- list("a" = 1:3, "b" = 4:6, "c" = 7:9)
df2 <- df
df2[names(codes_na)] <- lapply(names(codes_na), function(variable)
        fix_missing2(df2[,variable], na.value = codes_na[[variable]]))
df2
##   a   b   c   d  e f
## 1 7  NA -99   2  5 2
## 2 5  NA   5   3  6 1
## 3 6   8   5   9  9 4
## 4 4   2   2   6  6 8
## 5 6   7   6 -99 10 6
## 6 9 -99   4   7  5 1
```

Méthode alternative pour créer fonctions complexes

Nous avons présenté la manière dont la fonction lapply permet de généraliser l'application d'une fonction.

- purrr::map se comporte de manière très proche de lapply ; on transforme les données avec reshape2 et applique une fonction par groupe.

```
df2_map <- df2
df2_map[] <- purrr::map(df2_map,fix_missing, na.value = -99)
```

Utiliser les outils de débuggage de Rstudio

Toutes les erreurs ne se valent pas !

Une fonction peut renvoyer une erreur pour deux raisons, avec des interprétations différentes.

- L'erreur est prévue par la fonction (bug by design) : il est possible d'introduire dans une fonction des conditions logiques qui renvoient une erreur si les conditions d'exécution de la fonction ne sont pas vérifiées (instruction stop).

La programmation défensive n'est pas une pratique obligatoire en équipe, il s'agit plutôt d'un principe de prudence.

Toutes les erreurs ne se valent pas !

- L'erreur est involontaire : un comportement non anticipé d'une fonction, une erreur de type de variable...

Il s'agit de cas fréquents. Dans ce cas, le principe général est d'essayer d'identifier l'erreur à partir de cas types (les tests d'un package visent à automatiser cette recherche) pour être en mesure d'améliorer progressivement une fonction.

Toutes les erreurs ne se valent pas !

On peut utiliser trois outils pour débugger:

- Inspecteur l'erreur Rstudio et utiliser Show Traceback() lister la séquence des appels de fonctions qui a amené une erreur ;
- Utiliser Rerun with Debug et la commande options(error = browser) pour exécuter pas à pas le code qui génère l'erreur et analyser l'environnement d'exécution du code ;
- Introduire dans une fonction un point de débuggage (breakpoint) ou la commande browser() qui ouvre une session sur un point déterminé par le créateur de la fonction.

Show Traceback

Cet outil est parfois appelé call stack. Il permet de voir l'empilement d'appels de fonctions ayant généré l'erreur.

Par exemple, voici une séquence ayant générant une erreur:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

Show Traceback

En cliquant sur le bouton Show Traceback, on voit la séquence de fonctions exécutées par la commande f(10).

Cette séquence se lit de bas en haut : on part de l'appel de fonction le plus élevé (f(10)) et on remonte la suite de fonctions appelées.

Cela permet d'identifier que l'origine de l'erreur provient de la fonction i(.).

S'il s'agit d'un code qui a été lu à partir de la commande source(), le traceback affichera également la localisation de la fonction dans le fichier.

Rerun with debug

traceback() montre à quel moment et dans quelle fonction l'erreur s'est produite, mais n'explique pas pourquoi l'erreur a eu lieu.

Cliquer sur le bouton *Rerun with debug* ouvre une fenêtre interactive qui permet de mettre en pause l'exécution d'une fonction et explorer interactivement l'état de celle-ci

Le debuggeur permet d'analyser les différents environnements, de s'assurer que l'environnement d'exécution de la fonction accède bien aux objets désirés (packages, paramètres, dataframe, functions...), et que les environnements emboîtés accèdent bien aux objets attendus.

Rerun with debug

On dispose également d'une barre dans la console

- Next: exécuter la prochaine étape de la fonction ;
- Step into: fonctionne comme next mais si la prochaine ligne est une fonction, cela vous enverra dans la fonction afin de progresser ligne par ligne ;
- Finish: finit l'exécution de la boucle ou fonction actuelle ;
- Continue: retour à l'environnement supérieur. C'est utile si le problème sur la fonction a été réglé et qu'on veut tester s'il est bien corrigé ;
- Stop: sortir du debuggeur.

Breakpoint

Il est possible d'introduire un point d'arrêt (breakpoint) dans une fonction.

Quand R arrivera au point d'arrêt, l'exécution de la fonction sera stoppée et le même debuggeur que celui de Rerun with debug s'ouvrira à cet endroit.

L'intérêt d'un breakpoint est qu'il permet au développeur de faire le point sur l'exécution d'une fonction à un moment précis.

On peut introduire un point d'arrêt de deux façons:

- cliquer sur la gauche du numéro de ligne d'un script un point rouge apparaît ;
- introduire la commande browser() dans le code de la fonction à l'endroit où on désire l'arrêter.

Comprendre les environnements

Notions sur les environnements

L'environnement est la structure qui assure la portée.

Un environnement est en fait une liste nommée d'objets avec des règles un peu plus restrictive qu'une liste classique:

- Chaque nom doit être unique ;
- Les noms dans un environnement n'ont pas d'ordre ;
- Un environnement a un parent ;
- La copie d'environnements ne suit pas le principe de copy-on-modify.

Notions sur les environnements

Un environnement sert à lier un ensemble de noms à un ensemble de valeurs.

Quelques fonctions du package rlang sont dédiées aux environnements. Par exemple, créons un nouvel environnement e1 contenant 4 objets

```
e1 <- rlang::env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3
)
e1
## <environment: 0x10f17b200>
```

Notions sur les environnements

Si on désire connaître les caractéristiques de celui-ci, on peut utiliser la fonction rlang::env_print.

```
rlang::env_print(e1)
## <environment: 0x10f17b200>
## Parent: <environment: global>
## Bindings:
## • a: <lgl>
## • b: <chr>
## • c: <dbl>
## • d: <int>
## names(e1)
## [1] "a" "b" "c" "d"
```

Notions sur les environnements

On a plusieurs environnements en R qui communiquent entre eux avec des relations de hiérarchie.

Le current environment ou environnement d'exécution, qu'on peut afficher avec rlang::current_env() est l'environnement dans lequel les commandes sont actuellement exécutées.

Quand vous expérimentez avec R, l'environnement d'exécution est l'environnement global (.GlobalEnv).

L'environnement global est l'espace de travail (workspace) car c'est là où tous les objets sont stockés.

Notions sur les environnements

Comme toute fonction est appelée avec un nouvel environnement, l'environnement d'exécution se distingue de l'environnement global dès qu'on appelle une fonction.

Dans l'exécution du code suivant, on voit que les deux appels de la fonction a() génèrent deux environnements différents.

```
print(rlang::current_env())
## <environment: R_GlobalEnv>
a <- function() print(rlang::current_env())
a()
## <environment: 0x10a3f4c00>
a()
## <environment: 0x10ae35108>
```

Notions sur les environnements

Tout environnement a un parent, qui est lui-même un environnement.

Le parent sert à définir la portée : si un nom n'est pas trouvé dans le current environment, R cherchera dans son parent puis dans le parent de celui-ci et ainsi de suite.

On peut afficher le parent d'un environnement avec la fonction rlang::env_parent ou la fonction de base parent.env

```
rlang::env_parent(e1)
## <environment: R_GlobalEnv>
parent.env(e1)
## <environment: R_GlobalEnv>
```

Notions sur les environnements

On appelle ancêtres l'ensemble des environnements parents d'un environnement.

Le principe du masquage de nom qu'on a évoqué fait que le nom d'un objet pris en compte par le current_env est toujours le premier trouvé dans les environnements ancêtres.

L'assignation traditionnelle, <-, crée toujours une variable dans l'environnement actuel.

Notions sur les environnements

La super assignation, «-, ne crée jamais la variable dans l'environnement courant mais, à la place, modifie la variable trouvée dans l'ancêtre le plus récent.

```
x <- 0
f <- function() { x <<- 1 ; return(2) }
f()
## [1] 2
x
## [1] 1
```

Si «- ne trouve pas de variable existant sous ce nom, elle sera créée dans l'environnement global.

L’environnement des packages et le chemin de recherche (search path)

Tous les packages attachés par la commande library() ou require() deviennent un environnement parent de l’environnement global.

Le parent immédiat de l’environnement global est le dernier environnement attaché, le parent suivant est celui attaché avant, etc.

Le principe de l’héritage des fonctions s’applique de la même manière qu’expliqué précédemment pour les environnements.

Si un nom de fonction n’est pas trouvé dans l’environnement global, R va donc chercher ce nom jusqu’au premier parent où ce nom fera référence à un objet.

C’est une source commune d’erreur lors d’un partage de code.

L’environnement des packages et le chemin de recherche (search path)

En effet, en cas de conflits entre noms de fonctions, la priorité sera Environnement Global > Dernier package chargé > Avant dernier package chargé > ...

Donc, si deux personnes ont chargé les mêmes packages mais dans un ordre différent, elles peuvent ne pas obtenir le même résultat (une erreur vs un output).

Un cas typique est la fonction select.

Un exemple de problème d'environnement avec les packages

Les packages MASS et dplyr proposent une fonction ayant le même nom mais dont l'objet est différent.

```
data(iris) ; library(MASS) ; library(dplyr)
## Attaching package: 'dplyr'
## The following object is masked from 'package:MASS': select
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect, setdiff, setequal, union
iris %>% select(Sepal.Length) %>% head()
##   Sepal.Length
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5.0
## 6          5.4
```

L’environnement des packages et le chemin de recherche (search path)

```
detach("package:dplyr")
detach("package:MASS")
library(dplyr)
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect, setdiff, setequal, union
library(MASS)
## Attaching package: 'MASS'
## The following object is masked from 'package:dplyr': select
iris %>% select(Sepal.Length) %>% head()
## Error in select(., Sepal.Length): unused argument (Sepal.Length)
```

Comment bien utiliser les packages ?

Si on pense régulièrement à nettoyer notre environnement, voire à repartir d'un environnement propre avec la commande lm(list = ls()), on ne pense pas nécessairement à nettoyer les packages chargés.

C'est un vrai enjeu de réplicabilité et de robustesse du code.

La bonne pratique à adopter est de systématiquement utiliser les fonctions issues de packages sous la forme pkg::function, par exemple dplyr::select.

Cela enlève toute ambiguïté à la fonction : R sait qu'il est nécessaire d'aller chercher la fonction select dans l'espace de noms (namespace) dplyr.

Quels que soient les packages chargés ou les fonctions dans l'environnement, la commande dplyr::select répondra toujours à l'intention du développeur.

Comment bien utiliser les packages ?

```
library(dplyr)
library(MASS)
iris %>% dplyr::select(Sepal.Length) %>% head()
##   Sepal.Length
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5.0
## 6          5.4
detach("package:MASS")
detach("package:dplyr")
```

Comment bien utiliser les packages ?

Notez la différence entre un package attaché et chargé:

Un package est chargé automatiquement dès qu'on accède à une de ses fonctions avec par exemple dplyr::select.

Le chargement est une dépendance souple : elle n'amène pas le package à devenir un parent mais permet quand même de trouver l'aide d'une fonction facilement.

Un package est attaché quand on fait library(pkg) ou require(pkg).

Comment bien utiliser les packages ?

En suivant à rebours l'ensemble des environnements parents (le search path), on voit l'ordre dans lequel les packages ont été attachés.

```
search()
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
library(rlang)
library(dplyr)
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats': filter, lag
## The following objects are masked from 'package:base': intersect, setdiff, setequal, union
search()
##  [1] ".GlobalEnv"        "package:dplyr"     "package:rlang"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"
```

Les Namespaces

L'objectif des espaces de noms (namespaces) est d'assurer que chaque package fonctionne de la même manière quels que soient les autres packages attachés par l'utilisateur.

Prenons un exemple à partir de la fonction sd() pour calculer des écarts-types. sd() est calculée à partir de la fonction var() (variance). sd est-elle affectée si une fonction s'appelle var dans l'environnement global ou dans les packages attachés ?

R évite ce problème en prenant avantage de la différence entre l'environnement d'une fonction et les environnements liés à cette fonction.

Les Namespaces

Chaque fonction intégrée dans un package est associée à deux environnements :

- environnement du package (package environment): interface externe d'un package. C'est la manière dont R trouve une fonction dans un package attaché ou avec par exemple dplyr::select.
- environnement de l'espace de nom (namespace environment): interface interne du package. Il contrôle la manière dont une fonction trouve les objets internes (fonctions, variables...).

Les Namespaces

Chaque environnement de noms a le même ensemble d'ancêtres.

Chaque package contient un fichier NAMESPACE qui contient le lien entre le package et ses dépendances.

Parce qu'importer toutes les fonctions de base R serait pénible, l'espace de noms des fonctions de base est un parent de tout espace de noms.

Les Namespaces

Le dernier parent, i.e. le premier dans la lignée des ancêtres, est l'environnement global.

Donc, si un package nécessite une variable myvar et ne la trouve dans aucun parent, il considérera qu'il s'agit d'une variable globale et donc ira chercher dans l'environnement global.

Les Namespaces

Reprenons l'exemple de sd. On peut vérifier l'environnement dans lequel on trouve sd

```
environment(sd)
## <environment: namespace:stats>
# pryr::where("sd")
body(sd)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
```

Les Namespaces

La définition de sd() utilise donc var(). Qu'arrive-t-il lorsqu'on crée notre propre version de var ?

```
x <- seq_len(10)
sd(x)
## [1] 3.02765
var <- function(x) "I do something really different"
sd(x)
## [1] 3.02765
var(x)
## [1] "I do something really different"
```

Cela n'affecte pas sd.