

# Outils de debugging/débogage

Présentation des outils de débogage avec R et RStudio

# Sommaire

- 1 Le print
- 2 Les breakpoints
- 3 La fonction browser
- 4 Les fonctions debug() / debugonce()
- 5 L'option recover
- 6 Astuce dans les chaine de traitements
- 7 Le package boomer
- 8 Les logs
- 9 Bibliographie

## 0.1 Sommaire

- Pourquoi faire du debugging/débogage ?
- Le print
- Le breakpoint / point d'arrêt
- La fonction browser
- Les fonctions debug() / debugonce()
- L'option recover
- Astuce dans les chaine de traitements
- Le package boomer
- Les logs

## 0.2 Pourquoi faire du debugging/débogage ?

Le débogage, ou debugging en anglais, est une étape cruciale dans le développement de tout programme informatique.

Cette présentation vous présente quelques outils que vous avez déjà à disposition sous **votre RStudio sur Cerise** afin de vous aider à trouver par vous-même la source de vos bugs.

## 0.3 Exemple de code

Soit la fonction suivante :

```
1 exemple1 <- function(indicateur) {  
2  
3   if (indicateur == "Sepale") {  
4     resultat <- iris |>  
5       select(Species, starts_with("Sepal")) |>  
6       head(3)  
7   } else if (indicateur == "Pétale") {  
8     resultat <- iris |>  
9       select(Species, starts_with("Petal")) |>  
10      head(3)  
11   }  
12   return(resultat)  
13  
14 }
```

## 0.4 Des erreurs/bugs

```
1 ## Tout est ok
2 exemple1("Pétale")
3
4 > exemple1("Pétale")
5   Species Petal.Length Petal.Width
6 1  setosa         1.4         0.2
7 2  setosa         1.4         0.2
8 3  setosa         1.3         0.2
```

```
1 ## Un bug
2 exemple1("petale")
3
4 > exemple1("petale")
5 Erreur dans exemple1("petale") : objet 'resultat' introuvable
```

# 1 Le print



## 1.1 Les prints (1/3)

Il existe un outil très simple (universel car non spécifique à R), parfois décrié mais qui reste efficace : ajouter la fonction `print()` dans votre code R pour examiner le contenu d'un objet à un moment donné.

Par exemple :

```
1 exemple1 <- function(indicateur) {  
2  
3   print(indicateur) # <- on ajoute un print()  
4  
5   if (indicateur == "Sepale") {  
6     resultat <- iris |>  
7       select(Species, starts_with("Sepal")) |>  
8       head(3)  
9   } else if (indicateur == "Pétale") {  
10    resultat <- iris |>  
11      select(Species, starts_with("Petal")) |>  
12      head(3)  
13  }  
14  return(resultat)  
15
```



## 1.2 Les prints (2/3)

Cela renvoie pour le cas du bug :

```
1 > exemple1("petale")  
2 [1] "petale"  
3 Erreur dans exemple1("petale") : objet 'resultat' introuvable
```

=> On récupère ainsi la valeur contenue dans l'objet indicateur “petale” ce qui nous permet de comprendre qu'on ne rentre dans aucune des 2 conditions de la structure conditionnelle de `exemple1()`.

=> L'objet `resultat` n'ayant pas été créé, R ne peut pas le trouver :).

## 1.3 Les prints (3/3)

### 👍 Avantages du print :

- Simplicité, universalité
- Pas d'interruption du code
- Compatible dans tous les environnements (RStudio, VSCode, batchs...)

### 👎 Inconvénients du print :

- Intrusif dans le code - Pas d'inspection de l'environnement à la volée
- On oublie souvent de les retirer du code une fois les bugs résolus.



## 2 Les breakpoints



## 2.1 Les breakpoints (1/6)

Avant de lancer vos traitements, il peut être utile de définir des breakpoints (“point d’arrêt”) sur une ligne de code. Lorsque R atteint cette ligne pendant l’exécution, il met le script en pause.

Pour cela, on peut cliquer dans la marge à gauche du numéro de ligne dans l’éditeur.

=> Cela permet d’avoir accès à l’environnement d’exécution, ce qui signifie que vous pouvez voir les valeurs de toutes vos variables, exécuter des commandes dans la console comme si vous étiez à cet endroit précis du code, et avancer pas à pas pour observer comment les variables changent.

```
3 exemple1 <- function(indicateur) {  
4  
5 if (indicateur == "Sepale") {  
6   resultat <- iris |>  
7     select(Species,starts_with("Sepal")) |>  
8     head(3)  
9 } else if (indicateur == "Pétale") {  
10  resultat <- iris |>  
11    select(Species,starts_with("Petal")) |>  
12    head(3)  
13 }  
14 return(resultat)  
15  
16 }
```

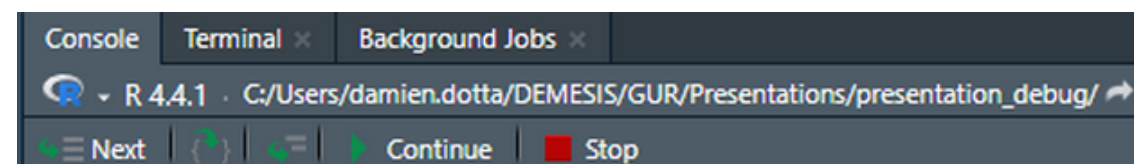
Les fonctions R qui contiennent un breakpoint ont un point rouge dans l’onglet Environnement de RStudio :



## 2.2 Les breakpoints (2/6)

En appuyant sur le bouton **source** de RStudio :

```
3 exemple1 <- function(indicateur) {  
4  
5   if (indicateur == "Sepale") {  
6     resultat <- iris |>  
7       select(Species, starts_with("Sepal")) |>  
8       head(3)  
9   } else if (indicateur == "Pétale") {  
10    resultat <- iris |>  
11      select(Species, starts_with("Petal")) |>  
12      head(3)  
13  }  
14  return(resultat)  
15  
16 }
```



La flèche verte et le texte en surbrillance permettent d'identifier le bout de code en cours d'exécution

Appuyer sur le bouton “Next” ou taper la lettre “n” dans la console de RStudio pour exécuter le script pas à pas.

## 2.3 Les breakpoints (3/6)

En appuyant sur le bouton **Next** de RStudio :

values	
indicateur	"Pétale"

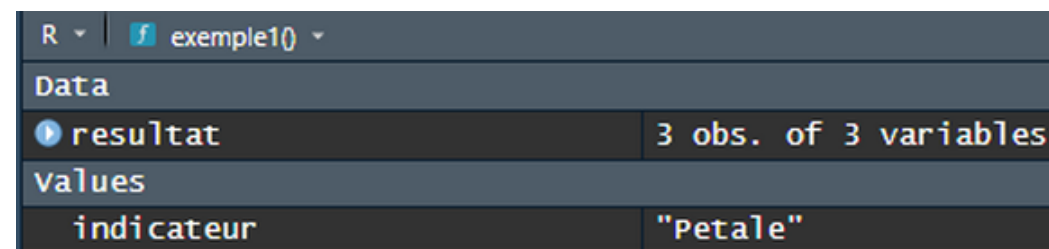
```
5  if (indicateur == "Sepale") {  
6    resultat <- iris |>  
7    select(species, starts_with("Sepal")) |>  
8    head(3)  
9  } else if (indicateur == "Pétale") {  
10   resultat <- iris |>  
11   select(species, starts_with("Petal")) |>  
12   head(3)  
13 }  
14 return(resultat)  
15  
16 }
```

Les objets se mettent à jour dans l'onglet Environnement de RStudio.

La flèche verte se positionne à l'emplacement du code qui sera exécuté à l'étape suivante.

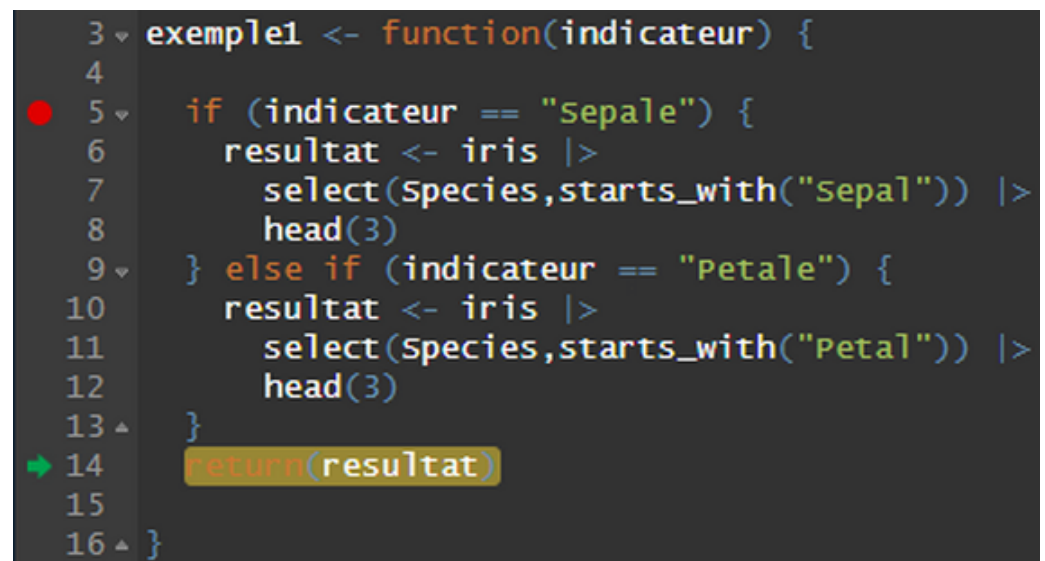
## 2.4 Les breakpoints (4/6)

En appuyant sur le bouton **Next** de RStudio :



The screenshot shows the RStudio Environment pane. At the top, it says 'R' and 'exemple1()'. Below that, under the 'Data' tab, there is an object named 'resultat' with '3 obs. of 3 variables'. Under the 'Values' tab, the variable 'indicateur' is shown with the value 'Pétale'.

Data	
▶ resultat	3 obs. of 3 variables
Values	
indicateur	"Pétale"



The screenshot shows the RStudio Source pane with a function definition. A red dot (breakpoint) is set on line 5. A green arrow points to line 14, indicating the next step in execution.

```
3 exemple1 <- function(indicateur) {  
4  
5 if (indicateur == "sepale") {  
6   resultat <- iris |>  
7     select(species, starts_with("Sepal")) |>  
8     head(3)  
9 } else if (indicateur == "Pétale") {  
10   resultat <- iris |>  
11     select(species, starts_with("Petal")) |>  
12     head(3)  
13 }  
14 return(resultat)  
15  
16 }
```

Les objets se mettent à jour dans l'onglet Environnement de RStudio.

La flèche verte se positionne à l'emplacement du code qui sera exécuté à l'étape suivante.

## 2.5 Les breakpoints (5/6)

### 👍 Avantages des breakpoints :

- Très visuels
- Pas besoin de modifier votre code R
- Permet l'exécution pas à pas

### 👎 Inconvénients des breakpoints :

- Spécifique à RStudio et aux programmes R
- Pas versionnable avec Git
- Difficile à utiliser dans certains cas dynamiques (les shiny par ex.)

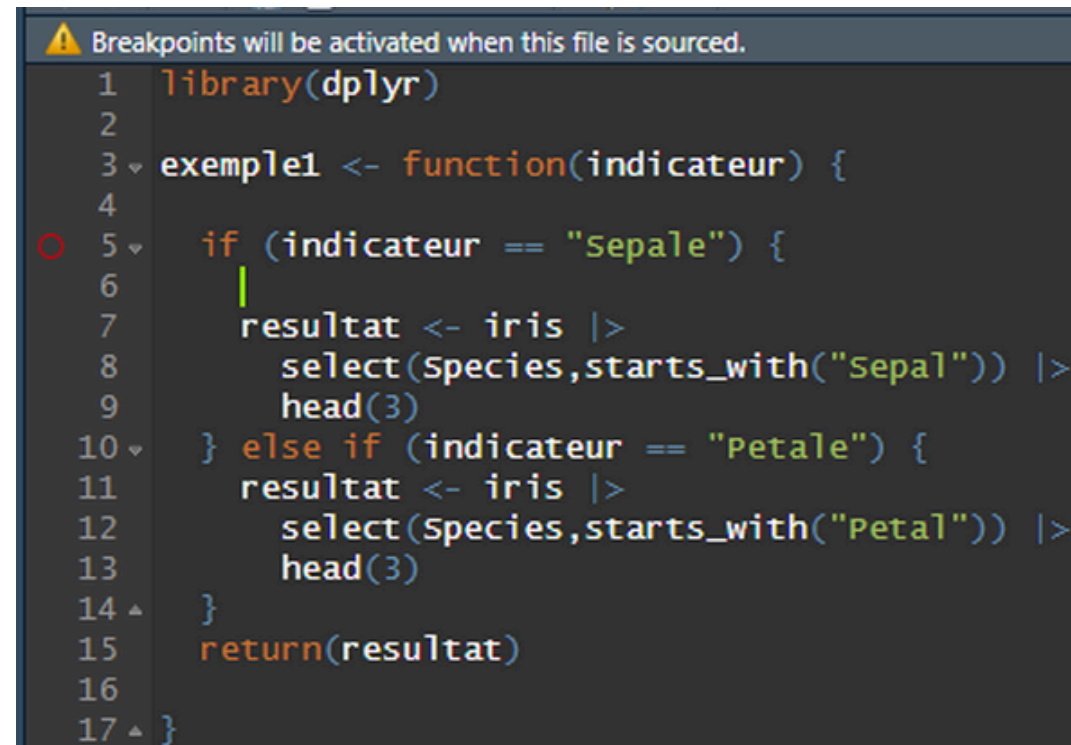




## 2.6 Les breakpoints (6/6)

### ⚠ Remarque

Si lorsque vous injectez un breakpoint dans votre programme R **celui-ci s'affiche en cercle rouge**, il faudra alors sauvegarder votre script et le sourcer pour retrouver le rond rouge.



```
1 library(dplyr)
2
3 exemple1 <- function(indicateur) {
4
5   if (indicateur == "Sepale") {
6     |
7     resultat <- iris |>
8       select(Species,starts_with("Sepal")) |>
9       head(3)
10  } else if (indicateur == "Petal") {
11    resultat <- iris |>
12      select(Species,starts_with("Petal")) |>
13      head(3)
14  }
15  return(resultat)
16
17 }
```

## 3 La fonction browser



## 3.1 La fonction browser (1/3)

La fonction browser() est un compromis entre la fonction print() et les breakpoints.

- Elle peut s'insérer à n'importe quel endroit du code comme print().
- Elle permet l'exécution pas à pas comme les breakpoints.



## 3.2 La fonction browser (2/3)

Par exemple :

```
1 exemple1 <- function(indicateur) {  
2  
3   browser() # <- on ajoute un browser()  
4  
5   if (indicateur == "Sepale") {  
6     resultat <- iris |>  
7       select(Species, starts_with("Sepal")) |>  
8       head(3)  
9   } else if (indicateur == "Pétale") {  
10    resultat <- iris |>  
11      select(Species, starts_with("Petal")) |>  
12      head(3)  
13  }  
14  return(resultat)  
15 }
```

## 3.3 La fonction browser (3/3)

### 👍 Avantages du browser :

- Facile à mettre en place
- Permet l'exécution pas à pas
- Fonctionne dans les chunks quarto

### 👎 Inconvénients du browser :

- Intrusif dans le code
- Moins visuels que les breakpoints
- On oublie souvent de les retirer du code une fois les bugs résolus.

## 4 Les fonctions debug() / debugonce()



## 4.1 Les fonctions debug() / debugonce() (1/4)

Les fonctions debug() et debugonce() permettent d'insérer la fonction browser() via l'appel à une fonction.

- debug(fonction) : active le mode pas-à-pas pour la fonction à chaque appel (nécessite la fonction undebbug() pour le désactiver).
- debugonce(f) : active le mode debug uniquement au prochain appel de la fonction.



## 4.2 Les fonctions debug() / debugonce() (2/4)

```
1 exemple1 <- function(indicateur) {  
2  
3   if (indicateur == "Sepale") {  
4     resultat <- iris |>  
5       select(Species, starts_with("Sepal")) |>  
6       head(3)  
7   } else if (indicateur == "Pétale") {  
8     resultat <- iris |>  
9       select(Species, starts_with("Petal")) |>  
10      head(3)  
11   }  
12   return(resultat)  
13  
14 }
```





## 4.3 Les fonctions debug() / debugonce() (3/4)

```
1 debug(exemple1) # pour lancer le debugueur sur la fonction
2 exemple1("petale") # maintenant le debugueur se lance qd on soumet la fonction
3 undebug(exemple1) # pour arrêter le debugueur sur la fonction
4
5 debugonce(exemple1) # Pour lancer le debugueur qu'une seule fois
6 exemple1("petale") # le debugueur ne va se lancer qu'une seule fois
```



## 4.4 Les fonctions debug() / debugonce() (4/4)

### 👍 Avantages des debug() :

- Facile à mettre en place
- Permet l'exécution pas à pas
- Ne modifie pas le code source des fonctions

### 👎 Inconvénients des debug() :

- Ne fonctionne qu'avec les fonctions nommées
- On oublie souvent de les retirer du code une fois les bugs résolus au moment des appels des fonctions.



# 5 L'option recover



## 5.1 L'option recover (1/4)

L'option `options(error = recover)` sert à déboguer une erreur quand elle se produit.

Quand une erreur se produit, cela :

- interrompt l'exécution
- affiche la pile des fonctions appelées
- permet de choisir un niveau (=frame) pour inspecter les variables à ce moment-là.

`options(error = NULL)` permet de rétablir le comportement par défaut.

## 5.2 L'option recover (2/4)

```
1 options(error = recover)
2
3 f <- function(x) {
4   y <- x + 1
5   g(y)
6 }
7
8 g <- function(z) {
9   if (z < 10) {
10     res <- z*z
11   }
12   return(res)
13 }
```



## 5.3 L'option recover (3/4)

```
1 f(2) # le mode debug ne s'active pas
2 f(100) # le mode debug s'active
```

On obtient dans la console :

```
1 > f(100) # le mode debug s'active
2 Erreur dans g(y) : objet 'res' introuvable
3
4 Enter a frame number, or 0 to exit
5
6 1: f(100)
7 2: #3: g(y)
8
9 Sélection :
```

=> Taper 0, 1 ou 2 dans ce cas pour lancer le mode debug au niveau souhaité.

## 5.4 L'option recover (4/4)

### 👍 Avantages de l'option recover :

- Facile à mettre en place
- Permet rapidement de voir toutes les fonctions imbriquées qui ont conduit à l'erreur
- Ne modifie pas le code source des fonctions

### 👎 Inconvénients de l'option recover :

- Le mode debug ne s'active qu'après une erreur
- Pas utilisable pour les batchs par exemple



## 6 Astuce dans les chaine de traitements



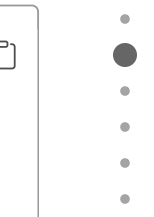


## 6.1 Une astuce (1/5)

A l'intérieur d'une chaîne de traitements {tidyverse} avec %>%, il est possible d'afficher ou de sauvegarder un résultat intermédiaire pour résoudre rapidement un bug.

**Si on veut simplement afficher le résultat intermédiaire, la syntaxe générale est :**

```
1 data |>
2 etape_une() |>
3 { print(head(.)); . } %>%
4 etape_deux()
```



## 6.2 Une astuce (2/5)

Un exemple :

```
1 iris %>%  
2   group_by(Species) %>%  
3   summarise(moyenne_long_petale = mean(Petal.Length, na.rm = TRUE)) %>%  
4   { print(head(.)); . } %>%  
5   rename(Especes = Species)
```

On obtient dans la console :

```
1 # A tibble: 3 × 2  
2   Species      moyenne_long_petale  
3   <fct>          <dbl>  
4 1 setosa          1.46  
5 2 versicolor     4.26  
6 3 virginica       5.55
```

## 6.3 Une astuce (3/5)

Si on veut stocker le résultat intermédiaire, , la syntaxe générale est :

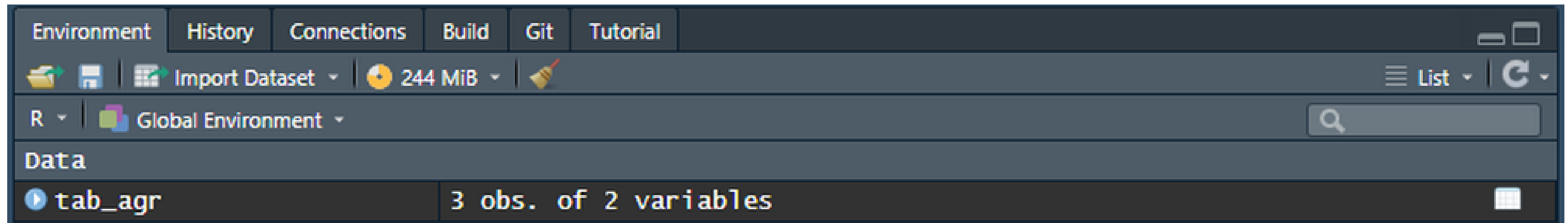
```
1 data |>  
2 etape_une() |>  
3 { result_interm <- . } %>% # On peut donner le nom que l'on veut au résultat intermédiaire  
4 etape_deux()
```



## 6.4 Une astuce (5/5)

```
1 iris %>%  
2   group_by(Species) %>%  
3   summarise(moyenne_long_petale = mean(Petal.Length, na.rm = TRUE)) %>%  
4   { tab_agr <- . } %>%  
5   rename(Especes = Species)
```

On obtient dans l'environnement :



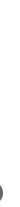
## 6.5 Une astuce (5/5)

### 👍 Avantages de l'astuce :

- Pas d'interruption du code
- Pratique à mettre en place pour un debug rapide

### 👎 Inconvénients de l'astuce :

- Ne fonctionne qu'avec `%>%` (pas avec `|>`)
- On oublie souvent de les retirer du code une fois les bugs résolus



# 7 Le package boomer



## 7.1 Le package boomer

Le package {boomer} fournit des outils de débogage qui permettent d'inspecter les résultats intermédiaires d'un code R.

L'output produit par le package se présente sous la forme de bombes  d'où le nom.

Ses 2 principales fonctions sont :

- `boom()` qui permet de diagnostiquer rapidement la performance d'un bloc de code R.
- `rig()` qui permet de tracer une fonction R et affiche automatiquement les mesures de performances à chacun de ses appels.

## 7.2 La fonction boom() (1/10)

3 écritures sont possibles :

- Ecriture en une ligne

```
1  
2 boom(iris |> head(2) |> filter(Sepal.Length > 5))
```

- Ecriture avec des crochets

```
1 boom({  
2   iris |>  
3     head(2) |>  
4     filter(Sepal.Length > 5)  
5 })
```



## 7.3 La fonction boom() (2/10)

- Ecriture à la fin de la chaîne de traitement (préférable)

```
1 iris |>  
2   head(2) |>  
3   filter(Sepal.Length > 5) |>  
4   boom()
```



## 7.4 La fonction boom() (3/10)

Le résultat dans la console :

```
1  💣 filter(head(iris, 2), Sepal.Length > 5)
2  · 💣 ✨ head(iris, 2)
3  ·   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
4  · 1           5.1           3.5           1.4           0.2   setosa
5  · 2           4.9           3.0           1.4           0.2   setosa
6  ·
7  · 💣 ✨ Sepal.Length > 5
8  · [1]  TRUE  FALSE
9  ·
10 ✨ filter(head(iris, 2), Sepal.Length > 5)
11   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
12 1           5.1           3.5           1.4           0.2   setosa
13
14   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
15 1           5.1           3.5           1.4           0.2   setosa
```

## 7.5 La fonction boom() (4/10)

En cas d'erreur, la fonction `boom()` affiche les résultats intermédiaires jusqu'à l'apparition du bug.

```
1 iris |>
2   head(2) |>
3   filter(Species == "virginica") |>
4   boom()
```

renvoie la sortie suivante :

```
1 💣 filter(head(iris, 2), Species == "virginica")
2 · 💣 💣 head(iris, 2)
3 ·   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
4 · 1           5.1           3.5           1.4           0.2 setosa
5 · 2           4.9           3.0           1.4           0.2 setosa
6 ·
7 · 💣 💣 Species == "virginica"
8 · [1] FALSE FALSE
9 ·
10 💣 filter(head(iris, 2), Species == "virginica")
11 [1] Sepal.Length Sepal.Width Petal.Length Petal.Width Species
12 <0 lignes> (ou 'row.names' de longueur nulle)
13
14 [1] Sepal.Length Sepal.Width Petal.Length Petal.Width Species
15 <0 lignes> (ou 'row.names' de longueur nulle)
```

## 7.6 La fonction boom() (5/10)

La fonction `boom()` comprend 2 arguments très utiles :

- `clock` : booléen qui permet de quantifier le temps d'exécution de chaque étape.
- `print` : qui permet de fixer la manière dont sont affichés les résultats intermédiaires.
  - Soit `str`
  - Soit `glimpse`
  - Soit `list(data.frame = str)`

## 7.7 La fonction boom() (6/10)

Un exemple d'utilisation de `boom()` avec l'argument `clock = TRUE`


```
1 iris |>  
2   head(2) |>  
3   filter(Sepal.Length > 5) |>  
4   boom(clock = TRUE)
```

Renvoie (partie 1) :

```
1 💣 filter  
2 · 💣💣 head(iris, 2)  
3 time: 0.215 ms  
4 ·   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
5 · 1           5.1           3.5           1.4           0.2   setosa  
6 · 2           4.9           3.0           1.4           0.2   setosa  
7 ·  
8 · 💣💣 Sepal.Length > 5  
9 time: 0.027 ms  
10 · [1]  TRUE  FALSE  
11 ·
```

## 7.8 La fonction boom() (7/10)

Renvoie (partie 2) :

```
1  filter(head(iris, 2), Sepal.Length > 5)
2 time: 0.001 s
3   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
4 1           5.1         3.5         1.4         0.2   setosa
5
6   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
7 1           5.1         3.5         1.4         0.2   setosa
8 >
```

## 7.9 La fonction boom() (8/10)

Un exemple d'utilisation de `boom()` avec l'argument `print = list(data.frame = str)`


```
1 iris |>
2   head(2) |>
3   filter(Sepal.Length > 5) |>
4   boom(clock = TRUE, print = list(data.frame = str))
```

Renvoie (partie 1) :

```
1  💣 filter
2  · 💣 💣 head(iris, 2)
3  time: 0.214 ms
4  · 'data.frame': 2 obs. of  5 variables:
5  ·   $ Sepal.Length: num  5.1 4.9
6  ·   $ Sepal.Width : num  3.5 3
7  ·   $ Petal.Length: num  1.4 1.4
8  ·   $ Petal.Width : num  0.2 0.2
9  ·   $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1
10 ·
11 · 💣 💣 Sepal.Length > 5
12 time: 0.021 ms
13 · [1]  TRUE FALSE
14 ·
```

## 7.10 La fonction boom() (9/10)

Renvoie (partie 2) :

```
1  filter(head(iris, 2), Sepal.Length > 5)
2 time: 0.002 s
3 'data.frame':  1 obs. of  5 variables:
4  $ Sepal.Length: num 5.1
5  $ Sepal.Width : num 3.5
6  $ Petal.Length: num 1.4
7  $ Petal.Width : num 0.2
8  $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1
9
10   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
11  1           5.1           3.5           1.4           0.2   setosa
```



## 7.11 La fonction boom() (10/10)

Quelques options utiles :

- Pour fixer l'option clock à TRUE tout le temps : `options(boomer.clock = TRUE)`
- Pour fixer l'option clock à TRUE tout le temps : `options(boomer.print = str)`
- Voir [ici](#) pour les options disponibles...

Pour aller plus loin :

- {boomer} contient les fonctions `boom_on()` et `boom_off()` qui peuvent être utilisées dans le mode debug pas à pas de RStudio (voir [cette page](#)).
- Le package contient également une fonctionnalité `boom_shinyApp()` (encore expérimentale) pour aider les développeurs à debugger les applications shiny (voir [cette page](#)).

## 7.12 La fonction `rig()` (1/3)

La fonction `rig()`

Soit la fonction suivante :

```
1 mafonction <- function(x) {  
2   y <- x^2  
3   return(y)  
4 }
```



## 7.13 La fonction `rig()` (2/3)

On appelle la fonction avec `rig()` :

```
1 rig(mafonction)(2)
```

Et on obtient :

```
1 🖱️ mafonction
2 x :
3 [1] 2
4 💣💣 x^2
5 [1] 4
6
7 🖱️ mafonction
8 [1] 4
```

Pour arrêter de surveiller une fonction, il existe la fonction `unrig()`.

## 7.14 La fonction rig() (3/3)

Avec une erreur :

```
1 rig(mafonction)("2")
```

Renvoie :

```
1 🖱️ mafonction
2 x :
3 [1] "2"
4 💣💣 x^2
5 Error: simpleError/error/condition
6 Erreur dans .Primitive("^")(x, 2) :
7   argument non numérique pour un opérateur binaire
8 🖱️ mafonction
```

## 7.15 Avantages/Inconvénients de {boomer}

### 👍 Avantages de {boomer} :

- Permet de “décortiquer” facilement les blocs de code (avec `boom()`)
- Permet de “décortiquer” facilement les fonctions (avec `rig()`)
- Peut aider à debugger les applications Shiny

### 👎 Inconvénients de {boomer} :

- Pour le debogage d’une longue chaine de traitements, le pas à pas est plus pratique.
- Surcharge le code source
- On oublie souvent de les retirer du code une fois les bugs résolus.



# 8 Les logs



## 8.1 Les logs

Pour compléter les différents outils de débogage, il reste une pratique essentielle à présenter qui est **l'écriture des logs**.

Les logs permettent de tracer l'historique des actions, de suivre le cheminement logique du code R et ainsi de repérer l'origine des bugs/erreurs.

Cette pratique est tellement répandue et utile qu'il existe de très nombreux packages R : `{logger}`, `{futile.logger}`, `{logging}`, `{lgr}`, `{loggit}`, `{log4r}` ...

Dans cette présentation, nous allons utiliser le package `{logr}`.

## 8.2 Pourquoi logr ?

Le package {logr} présente plusieurs avantages :

- Il est très simple à utiliser
- Il fournit automatique un en-tête complet très utile au débogage
- L'horodatage est automatique (utile pour des mesures de performances)





## 8.3 logr (1/15)

Avec {logr}, il y a besoin de seulement 3 étapes pour créer un fichier de log :

- Ouvrir le fichier de log (avec `log_open()`)
  - Syntaxe : `log_open(chemin_vers_fichier_log)`
- Ecrire dans le fichier de log (avec `log_print()`)
  - Syntaxe : `log_print("message à écrire dans la log")`
- Fermer le fichier de log (avec `log_close()`)
  - Syntaxe : `log_close()`

## 8.4 logr (2/15)

### Structure d'une log :

- **Un en-tête** qui contient des informations essentielles : le chemin du fichier de log, le répertoire de travail, la version de R Core, le nom de l'utilisateur, le time-stamp d'exécution, les packages utilisés...
- **Un corps/body** qui constitue le coeur de la log. Il contient tous les messages et les objets que vous voulez afficher tels qu'il s'afficherait dans la console de R.
- **Un pied de page** qui contient le timestamp de fin d'exécution ainsi que le temps total écoulé pendant l'exécution du script R.

## 8.5 logr (3/15)

Un exemple :

```
1 library(logr)
2
3 # Ouverture de la log
4 log_open("ma_log.log")
5
6 log_print("## Lecture des données")
7 donnees <- readRDS("data/iris.rds")
8
9 log_print("## Traitement des données")
10 resultat <- donnees |>
11   group_by(Species) |>
12   summarise(moy_long_petales = mean(Petal.Length, na.rm = TRUE))
13
14 # Fermeture de la log
15 log_close()
```

## 8.6 logr (4/15)

Renvoie la log suivante (partie 1) :

```
1 =====
2 Log Path: ./log/ma_log.log
3 Working Directory: /var/data/nfs/CERISE/00-Espace-Personnel/damien.dotta
4 User Name: damien.dotta
5 R Version: 4.4.1 (.2024-06-14)
6 Machine: stats-prod-rstudio-2.zsg.cdpagri.fr x86_64
7 Operating System: Linux 4.18.0-372.32.1.el8_6.x86_64 #1 SMP Fri Oct 7 12:35:10 EDT 2022
8 Base Packages: stats graphics grDevices utils datasets methods base
9 Other Packages: dplyr_1.1.4 logr_1.3.8 common_1.1.3
10 Log Start Time: 2025-06-27 14:18:06.072628
11 =====
```

# 8.7 logr (5/15)

Partie 2 :

```
1  ## Lecture des données
2
3  NOTE: Log Print Time:  2025-06-27 14:18:06.08348
4  NOTE: Elapsed Time: 0.00261092185974121 secs
5
6  ## Traitement des données
7
8  NOTE: Log Print Time:  2025-06-27 14:18:06.090004
9  NOTE: Elapsed Time: 0.00652408599853516 secs
10
11  =====
12  Log End Time: 2025-06-27 14:18:06.120677
13  Log Elapsed Time: 0 00:00:00
14  =====
```

## 8.8 logr (6/15)

La fonction `log_print()` a également un alias utile pour la présentation des logs :

- `sep()` ajoute du texte dans la log comme `log_print()` mais ajoute des séparateurs avant/après le message.  
Cet exemple :

```
1 sep("Création des graphiques")
```

Va générer dans la log :

```
1 =====  
2 Création des graphiques  
3 =====
```

- `put()` est également un alias plus court que le nom `log_print()` dont le comportement est identique.

## 8.9 logr (7/15)

La fonction `log_print()` (ou `put()`) permet également d'envoyer des résultats dans les logs.

Par exemple :

```
1 log_print("## Traitement des données")
2 resultat <- donnees |>
3   group_by(Species) |>
4   summarise(moy_long_petales = mean(Petal.Length, na.rm = TRUE)) %>%
5   log_print()
```

# 8.10 logr (8/15)

Renvoi :

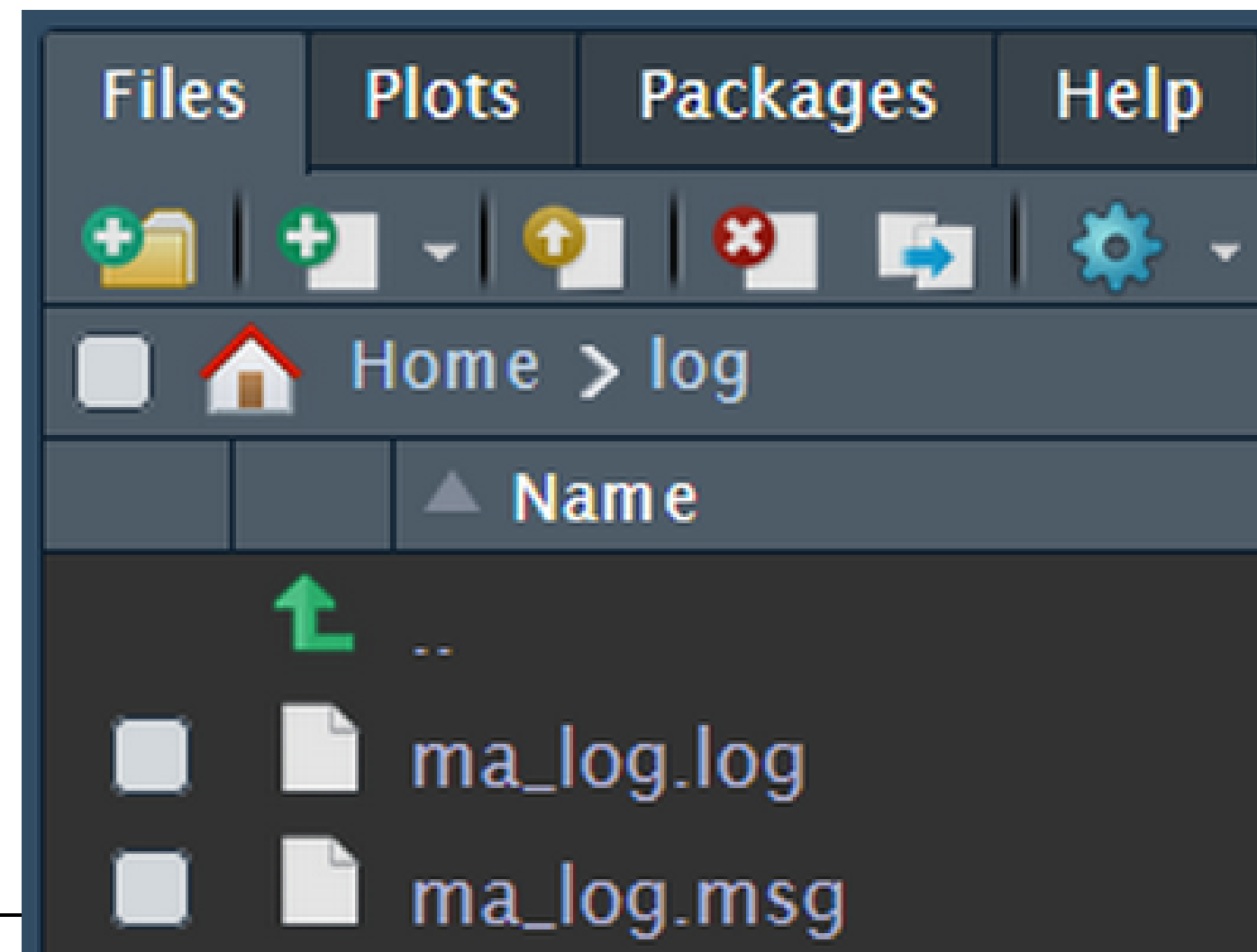
```
1 ## Traitement des données
2
3 NOTE: Log Print Time: 2025-06-27 14:35:15.474827
4 NOTE: Elapsed Time: 0.0325253009796143 secs
5
6 # A tibble: 3 × 2
7   Species      moy_long_petales
8   <fct>          <dbl>
9 1 setosa          1.46
10 2 versicolor     4.26
11 3 virginica       5.55
12
13 NOTE: Data frame has 3 rows and 2 columns.
14
15 NOTE: Log Print Time: 2025-06-27 14:35:15.572858
```



## 8.11 logr (9/15)

En cas d'erreur/bug (ou de warnings) pendant l'exécution du programme R, ceux-ci s'afficheront dans la log à l'endroit où ils ont eu lieu.

En complément, {logr} va isoler les messages d'erreurs et les warnings dans un fichier qui porte l'extension `.msg`.



## 8.12 logr (10/15)

Le traceback de R peut être très verbeux en cas d'erreurs. Si vous n'avez pas besoin de ce niveau de précision, il est possible de le désactiver ponctuellement avec `log_print("traceback" = FALSE)` ou globalement avec `options("logr.traceback" = FALSE)`.

Exemple d'erreur dans les logs :

```
1 Error in group_by(.donnees, species) : [38;5;255m [31m*[38;5;255m Column `species` is not found.[39m
2 Traceback:
3 [1] "10: stop(fallback)"
4 [2] "9: signal_abort(cnd, .file)"
5 [3] "8: abort(bullets, call = error_call)"
6 [4] "7: group_by_prepare(.data, ..., .add = .add, error_call = current_env())"
7 [5] "6: group_by.data.frame(donnees, species)"
8 [6] "5: group_by(donnees, species)"
9 [7] "4: summarise(group_by(donnees, species), moy_long_petales = mean(Petal.Length, "
10 [8] "      na.rm = TRUE))"
11 ...
```

Sans traceback :

```
1 Error in group_by(.donnees, species) : [38;5;255m [31m*[38;5;255m Column `species` is not found.[39m
```

## 8.13 logr (11/15)

Quelques conseils si vous savez que vous allez utiliser logr sur votre programme :

- Remplacer vos commentaires dans votre code R par des `log_print()`

A la place de :

```
1 # Traitement des données
```

Mettre ça :

```
1 log_print("# Traitement des données")
```

- A chaque exécution de vos programmes R, les logs seront par défaut écrasées. Si vous souhaitez les conserver, il est possible d'horodater leurs noms avec par exemple :

```
1 nom <- paste0("mylog_", format(Sys.Date(), "%Y-%m-%d"))
2
3 # Open log
4 lf <- log_open(nom)
```

## 8.14 logr (12/15)

Il est possible de logger seulement une partie d'un script R. Pour cela, il existe les options `options("logr.on" = FALSE)` et `options("logr.on" = TRUE)`.

```
3
4 # Activation de logr
5 options("logr.on" = TRUE)
6
7 log_open("ma_log.log")
8
9 log_print("## Lecture des données")
10 donnees <- readRDS("data/iris.rds")
11
12 log_close()
13
14 # Désactivation de logr
15 options("logr.on" = FALSE)
16
17 # la suite du programme
```

## 8.15 logr (13/15)

- Si vous souhaitez raccourcir la longueur de vos logs, il est possible de supprimer les lignes blanches avec l'option `options("logr.compact" = TRUE)`.
- Si vous souhaitez ne pas écrire les notes dans vos logs, il existe l'option `options("logr.notes" = FALSE)`
- Si vous souhaitez envoyer vos variables d'environnement dans vos logs, c'est possible via `put(Sys.getenv())`
- Pour plus de conseils, consultez [la FAQ du package](#) qui est très riche.

## 8.16 logr (14/15)

Si votre code est essentiellement écrit avec {dplyr} et {tidyr}, il est possible de limiter le nombre d'appels à `log_print()` en utilisant `options("logr.autolog" = TRUE)`.

Cela permet d'écrire dans les logs automatiquement (grâce à `tidylog`) à chaque appel d'un bloc de code.

Exemple sans `log_print()` :

```
1 options("logr.autolog" = TRUE)
2
3 log_open("ma_log.log")
4
5 donnees <- readRDS("data/iris.rds")
6
7 resultat <- donnees |>
8   group_by(Species) |>
9   summarise(moy_long_petales = mean(Petal.Length, na.rm = TRUE))
10
11 # Fermeture de la log
12 log_close()
```

Renvoie la log suivante :

## 9 Bibliographie





## 9.1 Liens complémentaires

Pour en savoir plus :

- [Debugging with the RStudio IDE](#)
- [Debugging - Advanced R](#)
- [Package boomer](#)
- [Package logr](#)