



Créer son premier **package** R

# Ajouter **une fonction** dans votre package

Juliette ENGELAERE-LEFEBVRE - Maël THEULIERE

# Objectif de cet atelier

Après cet atelier vous saurez ajouter une fonction dans un package. C'est à dire que vous aurez compris :

- ce qu'est une fonction ;
- comment ajouter une fonction dans un package ;
- comment documenter une fonction ;
- comment tester une fonction.

,

Qu'est ce qu'une fonction ?

# Qu'est ce qu'une fonction ?

Une fonction est un objet de R. C'est une opération qui prend en entrée des arguments pour produire un résultat.

Par exemple :

- `abs()` prend comme argument un vecteur de nombre et produit un vecteur de nombre contenant la valeur absolue des nombres en argument.
- `select()` de `{dplyr}` prend comme argument un dataframe et une liste de colonnes et produit en sortie un dataframe restreint à ces colonnes.
- `write.csv()` prend en argument un dataframe, un lien vers un fichier, et produit en sortie un fichier csv contenant le dataframe, à l'endroit spécifié par le lien.

,

# Définir une fonction

Une fonction classique dans R se définit de la sorte :

```
ma_fonction <- function(a = 2, b = 1){  
  resultat <- a + 2*b  
  return(resultat)  
}
```

L'instruction `function()` créer une fonction ici appelée `ma_fonction()`.

Elle prend en arguments les paramètres de notre fonction, ici `a` et `b` auxquels on peut assigner des valeurs par défaut, ici `2` et `1`.

L'intérieur de nos accolades `{}` va définir le résultat produit par notre fonction. Ce résultat doit être retourné par l'instruction `return()`.

Voilà comment se définit une fonction type qui produit en retour un objet R. Certaines fonctions ne produisent pas des objets R mais des instructions, comme par exemple `write.csv()` vu précédemment.

,

# Les bonnes pratiques

Pour créer une bonne fonction, il faut bien penser sa cohérence dans le workflow dans laquelle elle va s'inscrire :

- Pour faire telle opération, dois je créer une fonction ou deux car un résultat intermédiaire pourrait m'intéresser ailleurs ?
- Quels paramètres ?
- Quelle complémentarité avec les fonctions existantes ?
- Quelle convention de nommage ?

Ensuite cette fonction devra être correctement documentée et testée. On verra dans la suite ce qu'est un test.

,

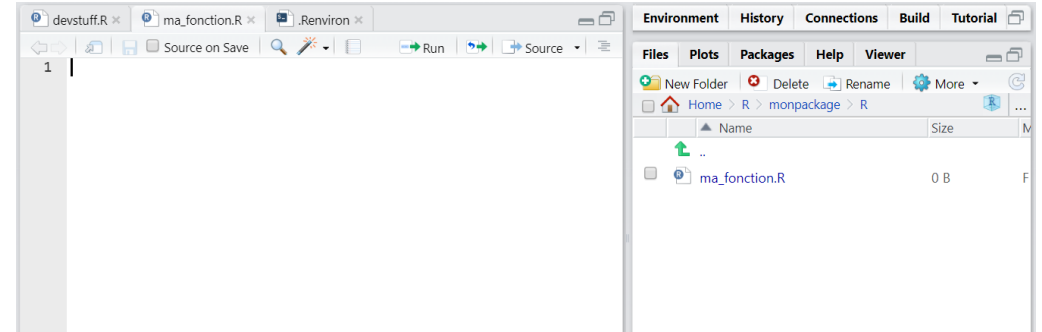
**Ajouter une fonction dans votre package**

Ajouter une fonction dans votre package

# créer le fichier .R

Pour rappel, le code d'une fonction doit être rajouté dans un script R du sous répertoire `R/`.

Pour rajouter une fonction dans votre package, `{usethis}` vous facilite le travail :  
`usethis::use_r("ma_fonction")` va créer un fichier `ma_fonction.R` dans votre répertoire `R/`.





# Ajouter votre fonction dans le fichier

Ici on crée la fonction `ma_fonction` qui prend en paramètres :

- un dataframe `data`,
- deux nombres `n_head` et `n_tail`,

et produit en sortie un dataframe contenant le début et la fin du dataframe `data`, en gardant `n_head` lignes du début et `n_tail` lignes de la fin.

⚠ Dans une fonction, il est commun d'utiliser des fonctions d'autres packages. Dans ce cas, appelez-les en utilisant la convention `packages::fonction()`.

```
ma_function <- function(data = NULL,  
                          n_head = 3,  
                          n_tail = 3){  
  res <- rbind(dplyr::slice_head(data,  
                                n = n_head),  
              dplyr::slice_tail(data,  
                                n = n_tail))  
  )  
  return(res)  
}
```

Ajouter une fonction dans votre package

# Utiliser votre fonction

`devtools::load_all()` vous permet de charger le contenu du package sur lequel vous travaillez, comme si vous l'aviez installé. Dans votre workflow habituel, vous allez utiliser souvent cette fonction pour tester les fonctions que vous ajoutez.

```
devtools::load_all()
```

Vous pouvez ensuite constater que votre fonction marche correctement 🍰

```
ma_fonction(iris, 2, 3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1         5.1         3.5         1.4         0.2    setosa
## 2         4.9         3.0         1.4         0.2    setosa
## 3         6.5         3.0         5.2         2.0 virginica
## 4         6.2         3.4         5.4         2.3 virginica
## 5         5.9         3.0         5.1         1.8 virginica
```

,

# Documenter votre fonction

## Documenter votre fonction

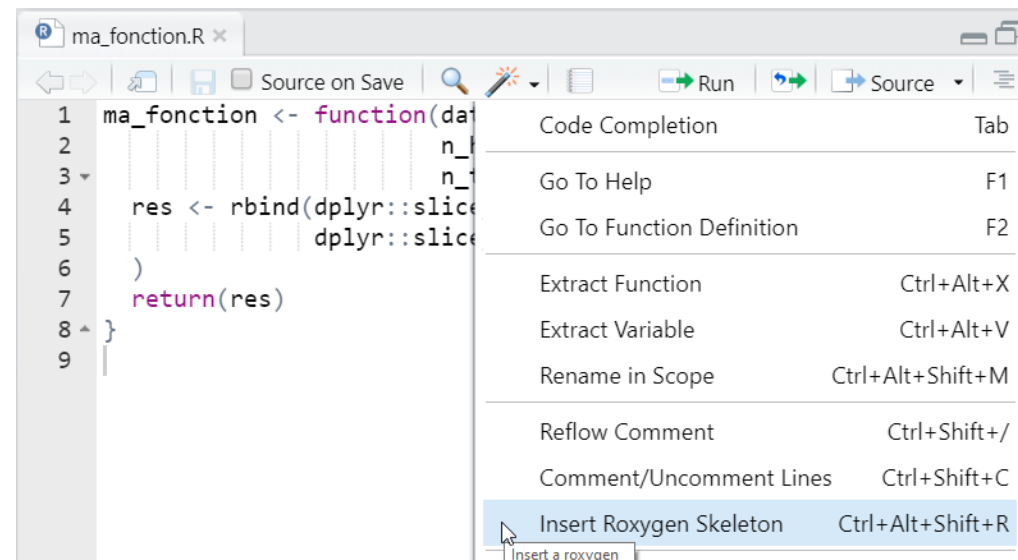
# {roxygen2}

Le package {roxygen2} va vous permettre de documenter votre fonction afin qu'une aide soit accessible pour celle-ci.

,

# {roxygen2} : créer un canevas

Pour ajouter une documentation, mettez le pointeur sur la fonction dans son script et utiliser le raccourci clavier **Ctrl + Alt + Shift + R** ou utiliser l'interface de Rstudio en cliquant sur **Code Tools**.

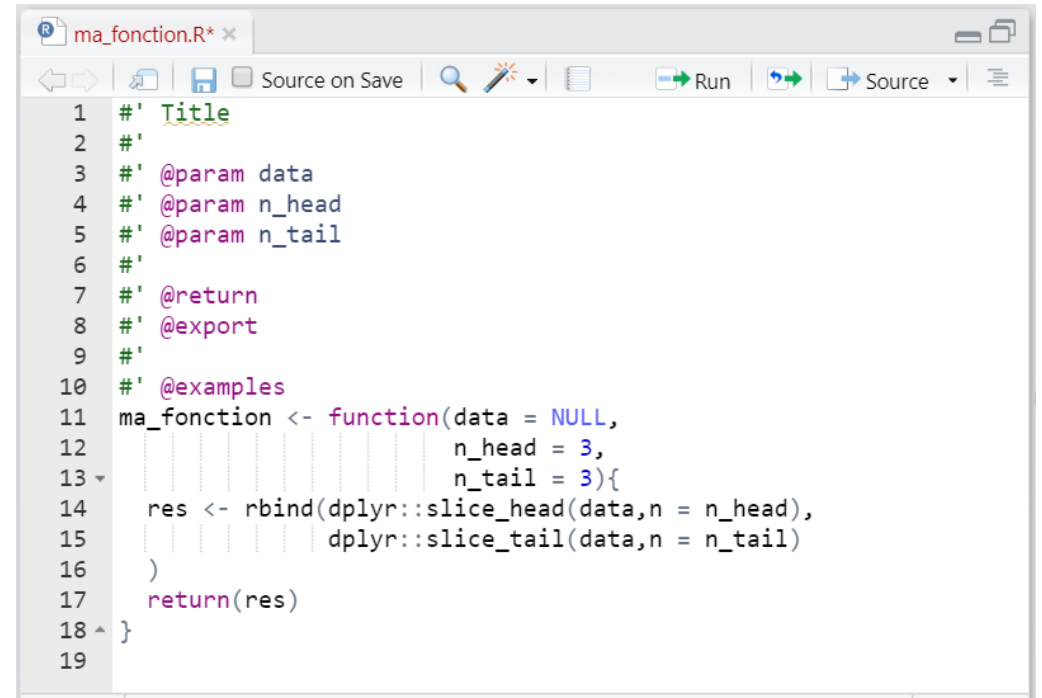


## Documenter votre fonction

# {roxygen2} : créer un canevas

Pour ajouter une documentation, mettez le pointeur sur la fonction dans son script et utilisez le raccourci clavier **Ctrl + Alt + Shift + R** ou utilisez l'interface de Rstudio en cliquant sur **Code Tools**.

Une fois activé, roxygen2 vous rajoute un canevas de documentation.



```
1 #' Title
2 #'
3 #' @param data
4 #' @param n_head
5 #' @param n_tail
6 #'
7 #' @return
8 #' @export
9 #'
10 #' @examples
11 ma_fonction <- function(data = NULL,
12                          n_head = 3,
13                          n_tail = 3){
14   res <- rbind(dplyr::slice_head(data,n = n_head),
15               dplyr::slice_tail(data,n = n_tail)
16   )
17   return(res)
18 }
19
```

# {roxygen2} : compléter votre documentation

Vous n'avez plus qu'à compléter 🍷!

```
#' Garder les lignes de début et de fin d'un datafr  
#'  
#' @param data un dataframe  
#' @param n_head Le nombre de lignes à garder du dé  
#' @param n_tail Le nombre de lignes à garder de la  
#'  
#' @return un dataframe  
#' @export  
#'  
#' @examples  
#' ma_fonction(mpg, 3, 3)  
ma_fonction <- function(data = NULL,  
                        n_head = 3,  
                        n_tail = 3){  
  res <- rbind(dplyr::slice_head(data, n = n_head),  
              dplyr::slice_tail(data, n = n_tail)  
  )  
  return(res)  
}
```

# {roxygen2} : gestion des dépendances

`{roxygen2}` permet non seulement de gérer la documentation mais aussi les dépendances et les exports de notre package. Cela se traduit par l'alimentation du fichier `NAMESPACE`.

La balise `@importFrom` permet de préciser les fonctions qu'on utilise dans le package. Cet ajout permettra de compléter le fichier `NAMESPACE` avec les dépendances de notre package.

On ajoute une balise `@importFrom` pour chaque package utilisé.

```
#' Garder les lignes de début et de fin d'un datafr  
#'  
#' @param data un dataframe  
#' @param n_head Le nombre de lignes à garder du de  
#' @param n_tail Le nombre de lignes à garder de la  
#'  
#' @return un dataframe  
#' @importFrom dplyr slice_head slice_tail  
#' @export  
#'  
#' @examples  
#' ma_fonction(mpg, 3, 3)  
ma_fonction <- function(data = NULL,  
                        n_head = 3,  
                        n_tail = 3){  
  res <- rbind(dplyr::slice_head(data, n = n_head),  
              dplyr::slice_tail(data, n = n_tail)  
  )  
  return(res)  
}
```



# {roxygen2} : gestion des exports

La balise `@export` permet aussi de compléter le fichier `NAMESPACE` en lui précisant cette fois ci que `ma_fonction()` est une fonction *exportée* de `{monpackage}`.

Si cette balise n'est pas ajoutée, dans ce cas, la fonction restera purement interne au package. Cela est une convention utile pour définir des fonctions nécessaires à d'autres fonctions du package mais pas directement utiles pour les utilisateurs.

```
#' Garder Les lignes de début et de fin d'un datafr  
#'  
#' @param data un dataframe  
#' @param n_head Le nombre de lignes à garder du de  
#' @param n_tail Le nombre de lignes à garder de la  
#'  
#' @return un dataframe  
#' @importFrom dplyr slice_head slice_tail  
#' @export  
#'  
#' @examples  
#' ma_fonction(mpg,3,3)  
ma_fonction <- function(data = NULL,  
                        n_head = 3,  
                        n_tail = 3){  
  res <- rbind(dplyr::slice_head(data, n = n_head),  
              dplyr::slice_tail(data, n = n_tail)  
              )  
  return(res)  
}
```

# Des actuces avec {prefixer}



Le package `{prefixer}` permet de finaliser votre fonction et produire la documentation plus facilement.

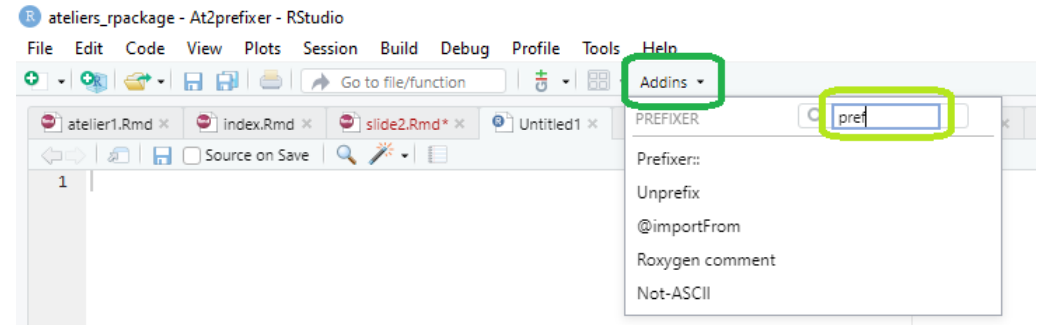
Il s'installe via :

```
remotes::install_github("dreamRs/prefixer")
```

Il s'agit d'un package qui installe un addin, c'est à dire qui ajoute des fonctionnalités à RStudio.

L'objectif des addins est généralement d'accélérer la réalisation de tâches répétitives ou fastidieuses.

On accède aux fonctionnalités nouvelles via le menu addins :

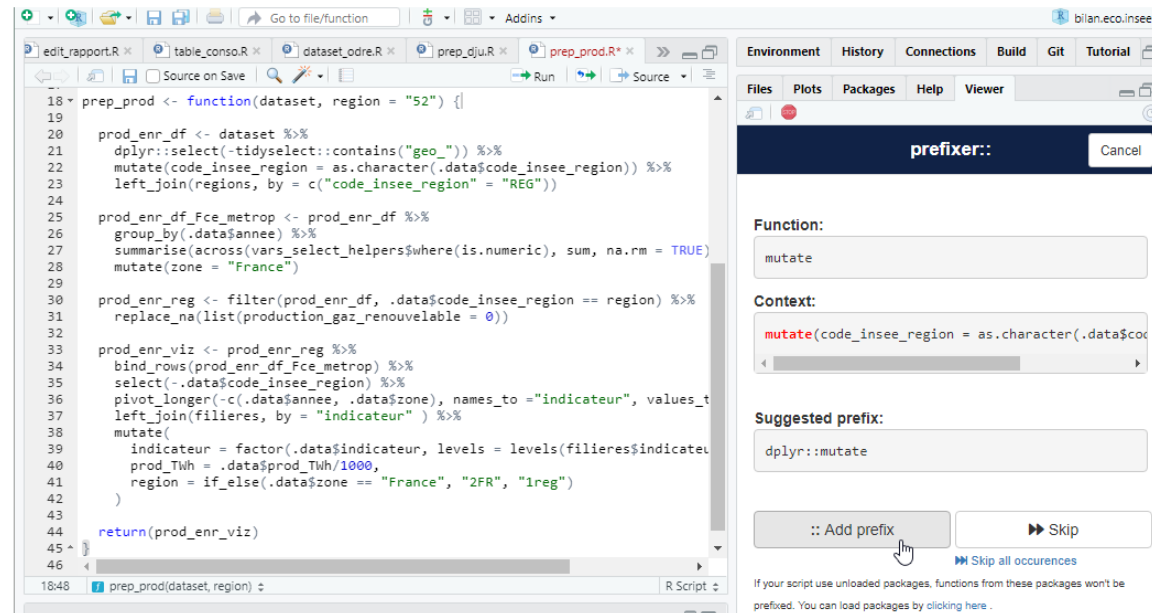


On peut ajouter des raccourcis clavier pour accélérer encore l'utilisation des fonctions des addins.

# Commande addin Prefixer::

La commande `Prefixer::` ouvre une boîte de dialogue qui vous propose d'ajouter, pour chaque fonction utilisée dans notre script de définition de fonction, le préfixe adéquat.

⚠ Seuls les packages actuellement actifs (appelés via `library`) seront proposés.



La commande `Unprefix` supprime tous les préfixes de notre script actif.

# Commande addin @importFrom

La commande @importFrom :

- parcourt votre script de définition de fonction,
- y détecte toutes les préfixes utilisés et
- ajoute au dessus de la fonction, la ou les balises @importFrom package1 fonctions1

```
#' @importFrom bilan.eco.insee regions filieres
#' @importFrom dplyr select mutate left_join group_by summarise across filter bind_rows if_else
#' @importFrom tidyr contains replace_na pivot_longer
#' @importFrom tidyselect vars select helpers
prep_prod <- function(dataset, region = "52") {
  prod_enr_df <- dataset %>%
    dplyr::select(-tidyr::contains("geo_")) %>%
    dplyr::mutate(code_insee_region = as.character(.data$code_insee_region)) %>%
    dplyr::left_join(bilan.eco.insee::regions, by = c("code_insee_region" = "REG"))

  prod_enr_df_Fce_metrop <- prod_enr_df %>%
```

⚠ Ces lignes restent à adapter :

- il faut préfixer les datasets, mais ne pas les faire figurer dans une balise importFrom qui est réservée aux fonctions,
- on n'importe pas les autres fonctions du package en cours de développement,
- le pronom `.data` n'est pas préfixé : ajouter `importFrom rlang .data` quand on y recourt.

# Commande addin Not-ASCII

Utiliser un encodage multi-plateforme est absolument nécessaire pour que notre package puisse fonctionner partout, que ce soit sur un serveur linux, un PC, un Mac...

La commande @Not-ASCII :

- scanne l'ensemble du fichier contenant le script de définition de fonction,
- y détecte tous les caractères à problème (entre "quote"),
- et les convertit avec leur code unicode.

Par exemple :

```
filter(dataset, TypeZone == "Régions")
```

devient :

```
filter(dataset, TypeZone == "R\u00e9gions")
```

Cela ne fonctionne pas dans les commentaires de documentation, de toutes façons, ils ne seraient rendus correctement.

,

# {roxygen2} : document()

Une fois votre documentation effectuée, la fonction `devtools::document()` va exploiter ces balises en

1. créant le fichier de documentation de votre fonction, `ma_fonction.Rd`, dans le répertoire `man/` et
2. en mettant à jour le fichier `NAMESPACE`.

`devtools::check()` intègre `devtools::document()` donc vous aurez au départ rarement à utiliser `devtools::document()` de façon isolée.

```
> devtools::document()  
Updating monpackage documentation  
Loading monpackage  
Writing NAMESPACE  
Writing NAMESPACE  
Writing ma_fonction.Rd
```

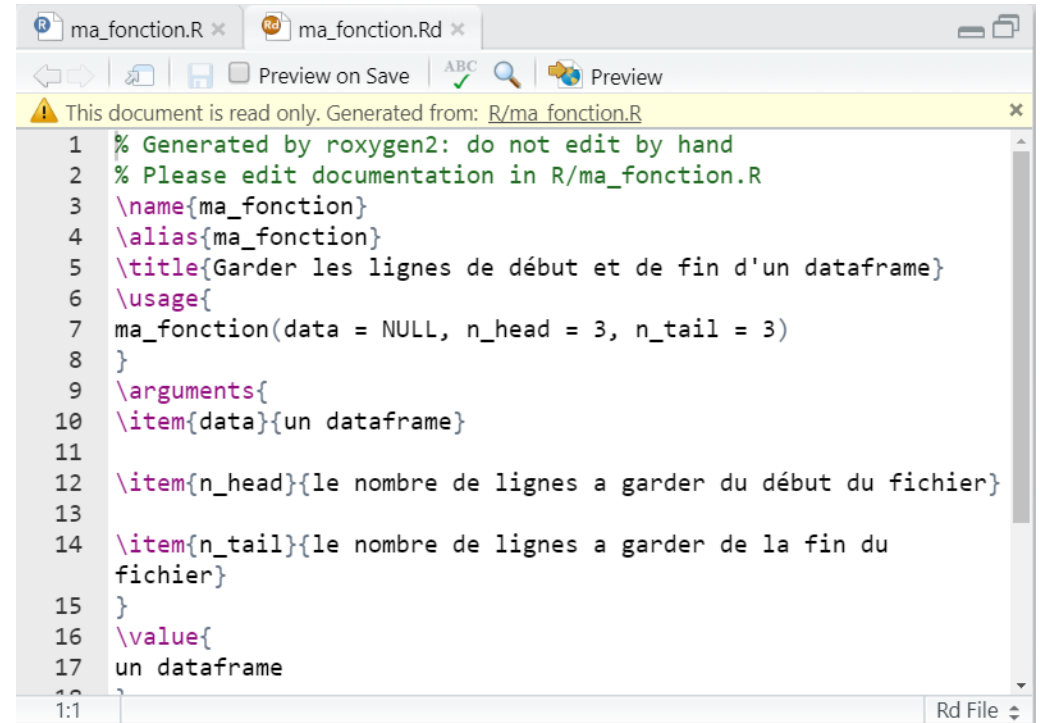
## Documenter votre fonction

# {roxygen2} : document()

Une fois votre documentation effectuée, la fonction `devtools::document()` va exploiter ces balises en

1. créant le fichier de documentation de votre fonction, `ma_fonction.Rd`, dans le répertoire `man/` et
2. en mettant à jour le fichier `NAMESPACE`.

`devtools::check()` intègre `devtools::document()` donc vous aurez au départ rarement à utiliser `devtools::document()` de façon isolée.



```
1 % Generated by roxygen2: do not edit by hand
2 % Please edit documentation in R/ma_fonction.R
3 \name{ma_fonction}
4 \alias{ma_fonction}
5 \title{Garder les lignes de début et de fin d'un dataframe}
6 \usage{
7   ma_fonction(data = NULL, n_head = 3, n_tail = 3)
8 }
9 \arguments{
10   \item{data}{un dataframe}
11
12   \item{n_head}{le nombre de lignes a garder du début du fichier}
13
14   \item{n_tail}{le nombre de lignes a garder de la fin du
15   fichier}
16 }
17 \value{
18   un dataframe
19 }
```

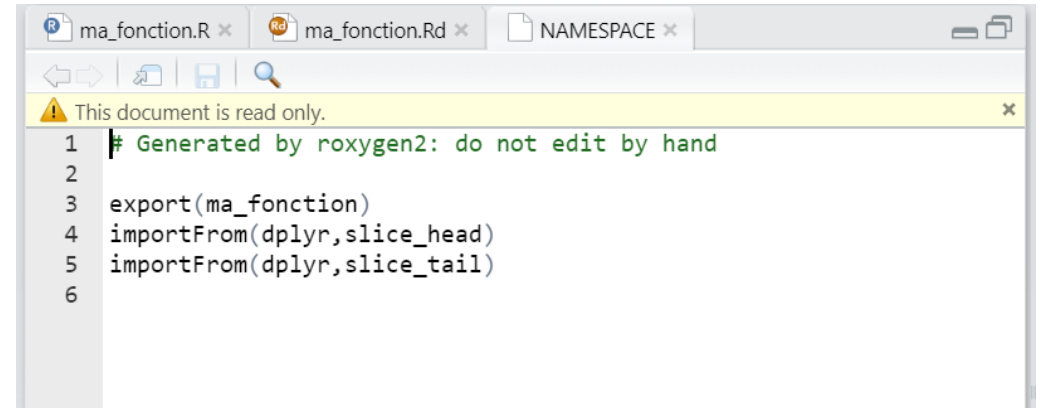
## Documenter votre fonction

# {roxygen2} : document()

Une fois votre documentation effectuée, la fonction `devtools::document()` va exploiter ces balises en

1. créant le fichier de documentation de votre fonction, `ma_fonction.Rd`, dans le répertoire `man/` et
2. en mettant à jour le fichier **NAMESPACE**.

`devtools::check()` intègre `devtools::document()` donc vous aurez au départ rarement à utiliser `devtools::document()` de façon isolée.

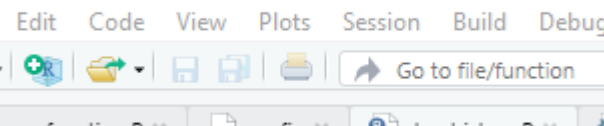


```
ma_fonction.R x ma_fonction.Rd x NAMESPACE x
This document is read only.
1 # Generated by roxygen2: do not edit by hand
2
3 export(ma_fonction)
4 importFrom(dplyr,slice_head)
5 importFrom(dplyr,slice_tail)
6
```



## Partie 'imports' de DESCRIPTION

Cela se fait notamment avec l'instruction `usethis::use_package("nomdupackage")` à consigner dans le `dev_history.R`.



monpremierpackage - master - RStudio

File Edit Code View Plots Session Build Debug Prof

+ [Save] [Go to file/function]

ma\_function.R × config × dev\_history.R × DESCRIPTION

← → [Save] [Source on Save] [Search] [Run]

```

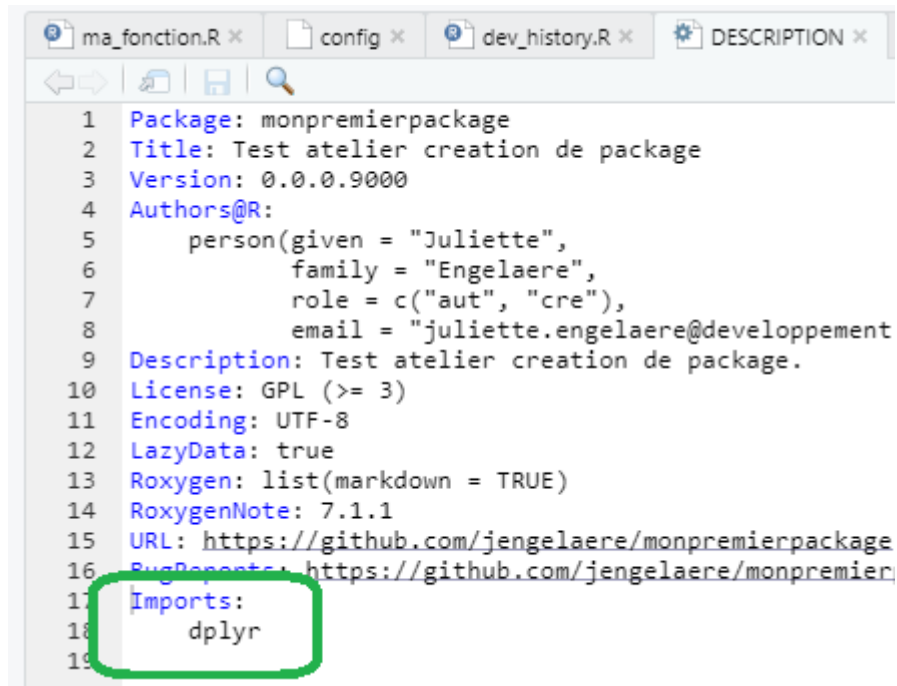
1 usethis::use_build_ignore('dev_history.R')
2 usethis::use_gpl3_license()
3 usethis::edit_file('DESCRIPTION')
4 devtools::check()
5 devtools::install()
6 usethis::use_package("dplyr")
7

```



# Partie 'imports' de DESCRIPTION

L'instruction `usethis::use_package` va compléter notre fichier DESCRIPTION au niveau de la partie 'imports' :



```
1 Package: monpremierpackage
2 Title: Test atelier creation de package
3 Version: 0.0.0.9000
4 Authors@R:
5   person(given = "Juliette",
6           family = "Engelaere",
7           role = c("aut", "cre"),
8           email = "juliette.engelaere@developpement
9 Description: Test atelier creation de package.
10 License: GPL (>= 3)
11 Encoding: UTF-8
12 LazyData: true
13 Roxygen: list(markdown = TRUE)
14 RoxygenNote: 7.1.1
15 URL: https://github.com/jengelaere/monpremierpackage
16 BugReports: https://github.com/jengelaere/monpremier
17 Imports:
18   dplyr
19
```

Le fichier DESCRIPTION peut être complété à la main.

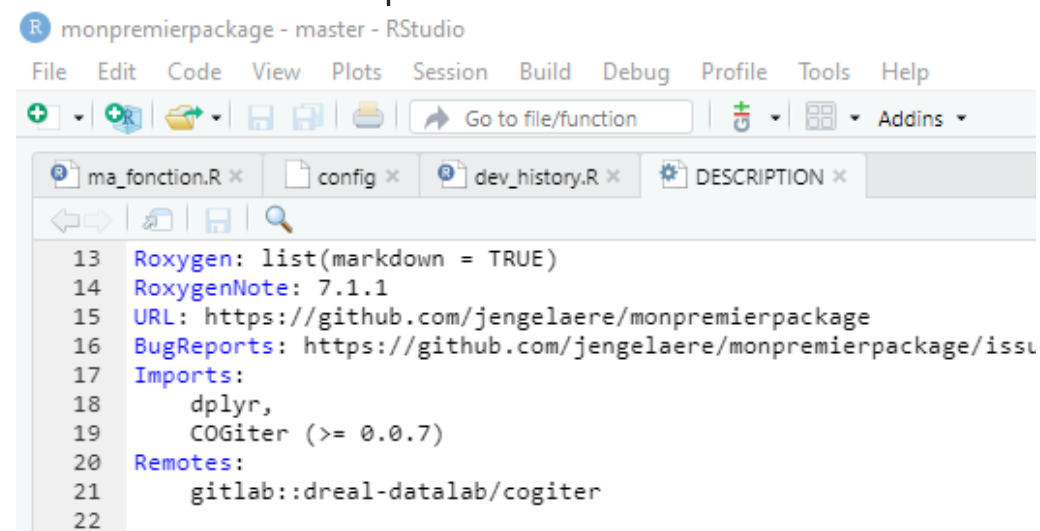
# Partie 'imports' de DESCRIPTION

Dans le cas de dépendance à des packages **qui ne sont pas hébergés par le CRAN** (par exemple COGiter), il faut le préciser. Sans ça, la gestion de cette dépendance ne sera pas traitée correctement par R lors de l'installation du package par l'utilisateur.

Cela se fait par exemple avec :

```
> usethis::use_dev_package("COGiter", type = "Import")
✓ Adding 'gitlab::dreal-datalab/cogiter' to Remotes
Refer to functions with `COGiter::fun()`
```

Cela a pour effet d'ajouter une partie '*Remotes* :' à notre fichier description :

A screenshot of the RStudio interface. The title bar shows 'monpremierpackage - master - RStudio'. The menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. The toolbar has icons for adding files, saving, and navigating. The file explorer shows 'ma\_fonction.R', 'config', 'dev\_history.R', and 'DESCRIPTION'. The editor window displays the content of the DESCRIPTION file, which includes fields for Roxygen, URL, BugReports, Imports, and Remotes. The 'Imports' field lists 'dplyr' and 'COGiter (>= 0.0.7)'. The 'Remotes' field lists 'gitlab::dreal-datalab/cogiter'.

```
13 Roxygen: list(markdown = TRUE)
14 RoxygenNote: 7.1.1
15 URL: https://github.com/jengelaere/monpremierpackage
16 BugReports: https://github.com/jengelaere/monpremierpackage/issu
17 Imports:
18   dplyr,
19   COGiter (>= 0.0.7)
20 Remotes:
21   gitlab::dreal-datalab/cogiter
22
```

**Tester votre fonction**

# Qu'est ce qu'un test ?

Une fois une fonction ajoutée à votre package, vous allez créer un ou plusieurs tests la concernant.

Un test définit un comportement attendu de votre fonction.

Par exemple, on s'attend à ce que  $2+2$  soit égal à 4. Plus généralement on s'attend à ce qu'une addition renvoie un nombre.

Sur notre exemple de fonction, le résultat de `ma_fonction()` doit être un *dataframe*.

On va pouvoir écrire un test qui cherche à vérifier cela sur un exemple particulier.

,

# Pourquoi faire un test ?

Les tests permettent de sécuriser votre développement.

Imaginez sur notre exemple que la définition de `slice_head()` et que par exemple le paramètre `n` change de nom et devient `nb`.

Le fait d'avoir défini un test pour s'assurer sur un jeu d'exemples du résultats attendu vous permettra très vite d'identifier ce changement.

,

# Comment faire un test ?

- Réfléchir au comportement attendu de la fonction. Dans notre exemple, le résultat de notre fonction est un dataframe de `n_head` + `n_tail` lignes.
- Grâce à `{usethis}` :
  - initialiser les tests dans `dev_history.R`,
  - et créer le fichier du test de la fonction `ma_fonction()`.

On recourt pour cela à la librairie `{testthat}`.

```
usethis::use_testthat()  
usethis::use_test("ma_fonction")
```

Un répertoire `tests` dédié aux tests a été créé à la racine du projet. Il contient un premier script R, `testthat.R`, qui initie les tests de notre package, et un répertoire `testthat` qui contient les fichiers de tests de chaque fonction.

,

# Comment faire un test ?

- Dans le fichier de test `test-ma_fonction.R` qui s'est ouvert, on exécute la fonction et on vérifie que le résultat a les propriétés attendues.

```
objet <- ma_fonction(data = iris, n_head = 3, n_tail = 2)

test_that("ma_fonction() renvoie un dataframe", {
  expect_is(objet, "data.frame")
})

test_that("ma_fonction() renvoie le bon nombre de lignes", {
  expect_equal(nrow(objet), 6)
})
```

Lors d'une prochaine vérification de notre package, avec `devtools::check()`, le test sera automatiquement exécuté.

,