Utilisation de packages R

Sophie Baillargeon, Université Laval

2018-03-23

Table des matières

Lors de l'ouverture d'une session R, nous avons accès à un bon nombre de fonctions et de jeux de données. Ces objets R sont accessibles, car ils proviennent de packages R chargés automatiquement à l'ouverture de session.

Un package R est simplement un **regroupement de fonctions et de données documentées**.

Installation et chargement d'un package R

Il est simple de charger en R des packages supplémentaires à ceux chargés par défaut. Il suffit d'utiliser la commande `library` comme dans cet exemple :

```
library(ggplot2)
```

Cette commande fonctionne uniquement si le package a été préalablement installé sur notre ordinateur. **L'installation d'un package et le chargement d'un package sont donc deux étapes distinctes.** L'installation n'a pas à être faite à chaque fois que le package est chargé.

Certains packages R sont installés automatiquement lors de l'installation de R. Ils ne sont pas pour autant tous chargés lors d'une ouverture de session.

La fonction `installed.packages` retourne des informations à propos des packages R installés sur l'ordinateur local.

```
i <- installed.packages()
head(i[, "Package"])
# Sortie non affichée, car trop longue
```

En **RStudio**, nous avons aussi accès à la liste des packages installés à partir de la sous-fenêtre nommée « Packages ». Parmi ces packages, ceux qui étaient inclus dans les fichiers d'installation de R sont les suivants :

```
i[ i[,"Priority"] %in% c("base","recommended"), c("Package", "Priority")]
```

```
##            Package      Priority
## base       "base"       "base"
## boot       "boot"       "recommended"
## class      "class"      "recommended"
## cluster    "cluster"    "recommended"
## codetools  "codetools"  "recommended"
## compiler   "compiler"   "base"
## datasets   "datasets"   "base"
## foreign    "foreign"    "recommended"
## graphics   "graphics"   "base"
## grDevices  "grDevices"  "base"
## grid       "grid"       "base"
## KernSmooth "KernSmooth" "recommended"
## lattice    "lattice"    "recommended"
## MASS       "MASS"       "recommended"
## Matrix     "Matrix"     "recommended"
## methods    "methods"    "base"
## mgcv       "mgcv"       "recommended"
## nlme       "nlme"       "recommended"
## nnet       "nnet"       "recommended"
## parallel   "parallel"   "base"
## rpart      "rpart"      "recommended"
## spatial    "spatial"    "recommended"
## splines    "splines"    "base"
## stats      "stats"      "base"
## stats4     "stats4"     "base"
## survival   "survival"   "recommended"
## tcltk      "tcltk"      "base"
## tools      "tools"      "base"
## utils      "utils"      "base"
```

Lorsque nous souhaitons utiliser pour une première fois un package autre qu'un de la liste précédente, il faut d'abord l'installer.

De plus, tout comme le logiciel R lui-même, les packages sont périodiquement mis à jour par leurs auteurs. Il est bon de tenir à jour les packages que nous avons installés.

Différentes façons d'installer un package

L'installation de package a déjà été abordée dans le Guide d'installation ou de mise à jour de R et RStudio. Voici de l'information plus détaillée à ce sujet.

À partir du CRAN :

Le CRAN est le dépôt informatique de packages géré par le *R core team*. C'est là que la majorité des packages R sont rendus disponibles publiquement. Pour installer un package à partir d'un des miroirs du CRAN, il suffit d'utiliser la fonction `install.packages` comme dans cet exemple :

```
install.packages("ggplot2")
```

Cette commande lance le téléchargement du fichier compressé d'installation du package, puis lance la décompression du fichier dans le dossier approprié. Bien sûr, il faut être connecté à internet pour que la commande fonctionne. La version du package correspondant à notre système d'exploitation est automatiquement sélectionnée :

- .tar.gz (*package source*) pour Linux / Unix,
- .zip pour Windows,
- .tgz pour Mac OS X / OS X / macOS.

Aussi, `install.packages` télécharge et installe par défaut les packages dont dépend le package installé. Cette option est très pratique, car certains packages dépendent d'un grand nombre de packages (`ggplot2` en est un bon exemple).

Le répertoire dans lequel les packages R sont installés par défaut est identifié dans le premier élément du vecteur retourné par la commande R suivante :

```
.libPaths()
```

```
## [1] "C:/Users/Sophie/Documents/R/win-library/3.4"
## [2] "C:/Program Files/R/R-3.4.3/library"
```

Sur mon ordinateur, les packages provenant de l'installation de R sont installés dans le répertoire `"C:/Program Files/R/R-3.4.3/library"` et les packages supplémentaires que j'ai installés sont dans le répertoire utilisateur `"C:/Users/Sophie/Documents/R/win-library/3.4"`.

RStudio offre une fonctionnalité pour rendre encore plus conviviale l'installation de packages. Dans la sous-fenêtre « Packages », le bouton « Install » (en haut à gauche) permet d'installer des packages. En fait, cette fonctionnalité ne fait qu'écrire et soumettre la commande `install.packages` en fonction des options choisies par l'utilisateur.

À partir d'un fichier compressé d'installation :

Afin d'installer un package non publicisé, qui n'est pas téléchargeable sur le web, il faut d'abord avoir sur son ordinateur le fichier compressé d'installation du package correspondant au système d'exploitation. Le package peut être installé avec la commande `install.packages`, comme dans cet exemple :

```
install.packages("C:/coursR/ggplot2_2.1.0.zip", repos = NULL)
```

Si nous possédons plutôt le *package source*, il est aussi possible d'installer le package à partir d'une plateforme Windows, mais à la condition d'avoir une installation des Rtools (voir le Guide d'installation ou de mise à jour de R et RStudio). Dans l'appel à la fonction `install.packages`, l'argument `type = "source"` doit être ajouté comme suit

```
install.packages("C:/coursR/ggplot2_2.1.0.tar.gz", type = "source", repos = NULL)
```

Cette technique d'installation à partir d'un fichier compressé est aussi utile lorsque nous devons télécharger manuellement le fichier compressé d'installation du package. Ça arrive par exemple lorsqu'un package est seulement disponible à partir du site web personnel de l'auteur du package. Nous devons aussi procéder à un téléchargement manuel lorsque nous avons besoin d'une ancienne version d'un package. Si nous laissons `install.packages` télécharger du CRAN un package, il téléchargera la dernière version. Il est possible que des mises à jour d'un package rendent certains de nos programmes non fonctionnels et qu'en conséquence nous souhaitions continuer d'utiliser une version antérieure du package. Pour installer une version d'un package qui n'est pas sa dernière version, il faut aller sur sa page web du CRAN (par exemple http://CRAN.R-project.org/package=ggplot2), et aller télécharger manuellement la version dont nous avons besoin dans « Old Sources ».

Ce type d'installation peut se réaliser par le menu « Install » de la sous-fenêtre « Packages » de **RStudio**, en sélectionnant : « Install from : > Package Archive File (. . .) ».

À partir d'un dépôt informatique autre que le CRAN :

En plus du CRAN, on retrouve des packages R à d'autres endroits sur le web, notamment sur :

- le dépôt informatique de packages en bio-informatique Bioconductor :
  des instructions d'installation sont disponibles sur la page web : http://www.bioconductor.org/install/,
- un service web d'hébergement et de gestion de développement de logiciels tel que Github ou Bitbucket :
  le package `devtools` offre des fonctions pour télécharger et installer directement à partir de ces sites (ex. fonctions `install_github` et `install_bitbucket`).

Accéder au contenu d'un package R chargé

Une fois un package chargé en R avec la commande `library`, son contenu est accessible dans la session R. Nous avons vu dans des notes précédentes comment fonctionne l'évaluation d'expressions en R. Nous savons donc que le chargement d'un nouveau package ajoute un environnement dans le chemin de recherche de R, juste en dessous de l'environnement de travail.

Le chargement de certains packages provoque aussi le chargement de packages dont ils dépendent. Ainsi, parfois plus d'un environnement est ajouté au chemin de recherche de R lors du chargement d'un package.

L'environnement d'un package contient les fonctions publiques et les données du package.

Fonctions publiques versus privées

Dans la phrase précédente, le mot « publiques » n'est pas anodin. Les fonctions publiques d'un package sont celles destinées à être appelées par des utilisateurs. Un package peut aussi contenir des fonctions privées (ou internes, ou cachées) qui sont uniquement destinées à être appelées par d'autres fonctions du package.

Les fonctions privées sont très utiles pour partitionner du code en bloc de calculs indépendants et éviter de répéter des bouts de code. Si un calcul doit être exécuté fréquemment par les fonctions d'un package, le développeur peut choisir de répéter un même bout de code pour ce calcul aux endroits appropriés dans le code source. Cependant, la présence de bouts de code identiques dans un programme est une mauvaise pratique de programmation. Si nous avons besoin de modifier le calcul fait dans ces bouts de code, il faut penser à aller faire les modifications à tous les endroits où le code est répété. Il est risqué d'oublier de modifier un des bouts. Si au contraire, nous avons créé une fonction pour faire le calcul, la modification doit être faite à un seul endroit. Nous gagnons du temps à long terme, le code est plus clair et les risques d'erreur sont minimisés. Il s'agit du principe *Don't Repeat Yourself*, mentionné dans les notes sur les bonnes pratiques de programmation.

Environnement d'un package dans le chemin de recherche :

La fonction `ls` permet de lister le contenu de l'environnement ajouté dans le chemin de recherche de R lors du chargement d'un package comme dans l'exemple suivant :

```
ls("package:stats")
```

```
## [1] "acf"                  "acf2AR"               "add.scope"
## [4] "add1"                 "addmargins"           "aggregate"
## [7] "aggregate.data.frame" "aggregate.ts"         "AIC"
```

Seulement les 9 premiers éléments de la liste sont affichés ici, car cette liste compte en réalité 447 éléments.

Espace de noms (*namespace*) d'un package :

Il est aussi possible de lister toutes les fonctions d'un package, publiques ou privées, grâce à la fonction `getNamespace`, comme dans l'exemple suivant

```
ls(getNamespace("stats"))
```

```
##  [1] "[.acf"          "[.formula"      "[.terms"        "[.ts"           "[.tskernel"
##  [6] "[[.dendrogram" "[<-.ts"         "acf"            "acf2AR"         "add.scope"
```

Seulement les 10 premiers éléments de la liste sont affichés ici, car cette liste compte en réalité 1101 éléments.

Cet environnement, où tout le contenu d'un package est présent, à l'exception des jeux de données, se nomme **espace de noms** (en anglais *namespace*). L'espace de nom n'est pas dans le chemin de recherche de R, mais son contenu est tout de même accessible grâce à l'opérateur `:::` et à des fonctions telles que `getAnywhere` et `getS3method`.

Par exemple, le package `stats` contient la fonction privée `Tr` qui permet de calculer la trace d'une matrice. Il n'est pas possible d'accéder directement à cette fonction, c'est-à-dire à partir du chemin de recherche.

```
Tr
```

```
## Error in eval(expr, envir, enclos): object 'Tr' not found
```

Nous arrivons cependant à y accéder par les commandes suivantes :

```
stats:::Tr
```

```
## function (matrix)
## sum(diag(matrix))
## <bytecode: 0x0000000016906548>
## <environment: namespace:stats>
```

```
getAnywhere(Tr)
```

```
## A single object matching 'Tr' was found
## It was found in the following places
##   namespace:stats
## with value
##
## function (matrix)
## sum(diag(matrix))
## <bytecode: 0x0000000016906548>
## <environment: namespace:stats>
```

Aussi, la méthode S3 de la fonction générique `plot` pour la classe `lm` fait partie du package `stats`, mais elle n'est pas publique.

```
plot.lm
```

```
## Error in eval(expr, envir, enclos): object 'plot.lm' not found
```

Nous pouvons tout de même y accéder avec la fonction `getS3method`

```
str(getS3method("plot", "lm"))
```

```
## function (x, which = c(1L:3L, 5L), caption = list("Residuals vs Fitted",
##     "Normal Q-Q", "Scale-Location", "Cook's distance", "Residuals vs Leverage",
##     expression("Cook's dist vs Leverage  " * h[ii]/(1 - h[ii]))),
##     panel = if (add.smooth) panel.smooth else points, sub.caption = NULL,
##     main = "", ask = prod(par("mfcol")) < length(which) && dev.interactive(),
##     ..., id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
```

```
##     qqline = TRUE, cook.levels = c(0.5, 1), add.smooth = getOption("add.smooth"),
##     label.pos = c(4, 2), cex.caption = 1, cex.oma.main = 1.25)
```

Nous aurions aussi pu y accéder avec `:::` ou `getAnywhere`.

Environnement englobant des fonctions d'un package :

Dans les notes sur les fonctions en R, le sujet de l'exécution d'une fonction est abordé. L'environnement englobant d'une fonction y est défini. Il permet de déterminer le chemin de recherche utilisé lors de l'exécution d'une fonction. Quel est l'environnement englobant des fonctions d'un package ? Il s'agit de l'espace de noms du package, comme en fait foi l'exemple suivant.

```
environment(var)
```

```
## <environment: namespace:stats>
```

C'est pour cette raison que les fonctions d'un package peuvent accéder directement aux fonctions internes du package. Donc, dans le code source d'un package, nul besoin d'utiliser `:::` ou `getAnywhere` pour accéder aux fonctions internes.

Jeux de données

Souvent, les jeux de données inclus dans un package se retrouvent directement dans l'environnement d'un package dans le chemin de recherche. C'est le cas, par exemple, des jeux de données du package `datasets`.

```
ls("package:datasets")
```

```
##   [1] "ability.cov"   "airmiles"      "AirPassengers" "airquality"    "anscombe"
##   [6] "attenu"        "attitude"      "austres"       "beaver1"       "beaver2"
```

Seulement les 10 premiers éléments de la liste sont affichés ici, car cette liste compte en réalité 104 éléments.

Cependant, les jeux de données sont parfois cachés. Ils sont traités différemment des fonctions privées et ne se retrouvent même pas dans l'espace de noms du package. C'est le cas par exemple dans le package `copula`.

```
library(copula)
```

La fonction `data` est très utile dans ce cas. Cette fonction a en fait plusieurs utilités. Premièrement, elle permet d'énumérer tous les jeux de données contenus dans un package.

```
data(package = "copula")
```

Le résultat est affiché dans une fenêtre indépendante, pas dans la console. Nous y apprenons que le package `copula` contient un jeu de données nommé `uranium`. Cependant, ce jeu de données n'est pas accessible directement.

```
str(uranium)
```

```
## Error in str(uranium): object 'uranium' not found
```

La fonction `data` permet alors de charger le jeu de données dans l'environnement de travail.

```
data(uranium)
str(uranium)
```

```
## 'data.frame':    655 obs. of  7 variables:
##  $ U : num  0.544 0.591 0.531 0.633 0.568 ...
##  $ Li: num  1.57 1.34 1.41 1.34 1.2 ...
##  $ Co: num  1.03 1.17 1.01 1.06 1.01 ...
##  $ K : num  4.21 4.34 4.22 4.16 4.22 ...
```

```
##  $ Cs: num  1.66 1.92 1.85 1.85 1.73 ...
##  $ Sc: num  0.839 0.934 0.903 0.908 0.763 ...
##  $ Ti: num  3.57 3.38 3.7 3.66 3.44 ...
```

Cette pratique de cacher les jeux de données d'un package tend à disparaître.

Documentation

Tout package doit obligatoirement contenir de la documentation sous forme de fiches d'aide. Ces fiches d'aide s'ouvrent avec la fonction `help` ou l'opérateur `?`, en fournissant comme argument le nom de la fonction. Les éléments suivants d'un package sont documentés dans une fiche d'aide :

- les fonctions publiques,
- les jeux de données,
- les classes et méthodes S4 publiques,
- les méthodes S3 pour des fonctions génériques (optionnel),
- le package lui-même (recommandé, mais pas obligatoire).

Une fiche d'aide peut servir à documenter plus d'un élément, par exemple un groupe de fonctions similaires ou une fonction retournant un objet d'une nouvelle classe et des méthodes pour des objets de cette classe.

La documentation complète d'un package s'ouvre en passant par :

> `help.start()` > Packages > nom du package.

Nous avons ainsi accès aux fiches d'aide, mais aussi à un fichier descriptif du package, parfois à des guides d'utilisation (souvent appelées vignettes), parfois à un fichier *NEWS* documentant les changements apportés au package lors de mises à jour, etc.

Pour les packages sur le CRAN, toutes les fiches d'aide et le fichier descriptifs sont regroupés dans un fichier PDF. Ce PDF se retrouve directement sur le CRAN (dans *Reference manual*), mais pas dans la fenêtre d'aide de R (par exemple http://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf).

La documentation sert à décrire ce que permettent de faire les fonctions et expliquer comment les utiliser. En plus d'exemples dans les fiches d'aide, certains packages possèdent des démonstrations, qui se démarrent avec la commande `demo`, comme dans cet exemple :

```
demo(lm.glm, package = "stats", ask = TRUE)
```

La commande `demo()` ouvre une fenêtre contenant la liste de toutes les démonstrations disponibles pour les packages chargés dans notre session R.

Il arrive (assez fréquemment malheureusement) que les fiches d'aide soient peu bavardes à propos des calculs effectués par une fonction. La fiche d'aide contient parfois des références qui peuvent nous éclairer. Si ce n'est pas le cas, nous sommes parfois contraints à aller voir directement dans le code source pour comprendre ce que fait une fonction.

Code source

Étant donné que R est un logiciel libre, le code source de tout package R est accessible. Accéder au code source est parfois simple. Il suffit de soumettre le nom d'une fonction dans la console (sans parenthèses) pour que le code source de la fonction soit affiché. Le code peut aussi être affiché dans une fenêtre indépendante avec la fonction `edit`. Pour les fonctions cachées, il faut utiliser `:::` ou `getAnywhere` comme mentionné ci-dessus.

Pour certaines fonctions, le code source est par contre écrit dans un autre langage que R. Par exemple, le code R de la fonction `var` du package `stats` contient uniquement de la validation d'arguments.

```
var
```

```
## function (x, y = NULL, na.rm = FALSE, use)
## {
##     if (missing(use))
##         use <- if (na.rm)
##             "na.or.complete"
##         else "everything"
##     na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs",
##         "everything", "na.or.complete"))
##     if (is.na(na.method))
##         stop("invalid 'use' argument")
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     else stopifnot(is.atomic(x))
##     if (is.data.frame(y))
##         y <- as.matrix(y)
##     else stopifnot(is.atomic(y))
##     .Call(C_cov, x, y, na.method, FALSE)
## }
## <bytecode: 0x0000000019185da0>
## <environment: namespace:stats>
```

Le coeur du calcul est fait dans une fonction C, appelée à la dernière ligne du corps de la fonction à l'aide de la fonction `.Call`. Il est aussi possible de voir ce code C. Il se trouve dans les fichiers source de R (voir Ligges (2006) pour des explications). Ces fichiers sont téléchargeables sur le site web suivant : https://cran.r-project.org/sources.html

---

Références

- Wickham, H. (2015). *R packages*. O'Reilly Media, Inc. URL http://r-pkgs.had.co.nz/

Accéder au code source :

- Ligges, U. (2006). R Help Desk : Accessing the Sources. *R News*, vol. 6, no. 4, p. 43-45. URL http://cran.r-project.org/doc/Rnews/Rnews_2006-4.pdf