L'essentiel sur les fonctions

Hadrien Commenges, Timothée Giraud

06/10/2014

Généralités sur les fonctions

Il y plusieurs raisons qui peuvent motiver la conception de fonctions propres à l'utilisateur, voici les principales:

1. Je dois répéter plusieurs fois la même procédure et je ne souhaite pas ré-écrire à chaque fois le code correspondant.
2. Je veut rendre la procédure plus générique et pas seulement appliquée à des données particulières.
3. J'ai écris une procédure générique et je veux la mettre à disposition d'autres utilisateurs.

Les fonctions sont des "boîtes" auxquelles on donne des valeurs d'entrée (arguments de la fonction) et desquelles on retire des valeurs de sortie (résultats de la fonction). Avec R, il n'y a pas de contraintes fortes sur les types d'objets qui peuvent être en argument ou en sortie d'une fonction.

La syntaxe générale pour créer une fonction est la suivante :

```
MaFonction <- function(arguments) {
  traitements
  return(sorties)
}
```

Voici la syntaxe pour écrire une fonction qui fait une somme. Pour pouvoir l'utiliser, il faut d'abord "sourcer la fonction", c'est-à-dire exécuter le bloc de code dans son enesmble. Ceci crée un objet nommé `MySum` dans l'environnement de travail. Cet objet peut être utilisé comme n'importe quelle fonction prédéfinie dans R, en fournissant les arguments demandés (ici `a` et `b`) on peut afficher ou récupérer une sortie (ici la somme de `a` et `b`) :

```
MySum <- function(a, b) {
  sumAB <- a + b
  return(sumAB)
}

MySum(2, 3)
```

```
## [1] 5
```

L'utilisateur qui veut entrer dans le détail de la programmation avec R trouvera de nombreuses ressources en anglais (par exemple http://adv-r.had.co.nz. En français, ces ressources sont plus rares, on citera en particulier les manuels en ligne de Christophe Genolini http://christophe.genolini.free.fr/webTutorial/index.php.

Exemple d'application : exploration de variables quantitatives

Lorsque j'explore les variables quanitatives je fais toujours les mêmes traitements : j'utilise la fonction `summary()`, j'y ajoute la fonction `sd()` parce que `summary()` ne calcule pas l'écart-type et je fais un histogramme avec la fonction `hist()`. Voici un exemple avec des données de population communale entre 1936 et 2008 pour les 143 communes de Paris et de petite couronne (département 75, 92, 93, 94).

```
popHist <- read.csv(file = "Donnees/PopCom3608.csv", stringsAsFactors = FALSE)
popHist$CODGEO <- as.character(popHist$CODGEO)
```

Voici l'exploration des variables de population avec les trois fonctions.

```
summary(popHist$POP1936)
sd(popHist$POP1936)
hist(popHist$POP1936)

summary(popHist$POP1954)
sd(popHist$POP1954)
hist(popHist$POP1954)

# ... etc. etc.
```

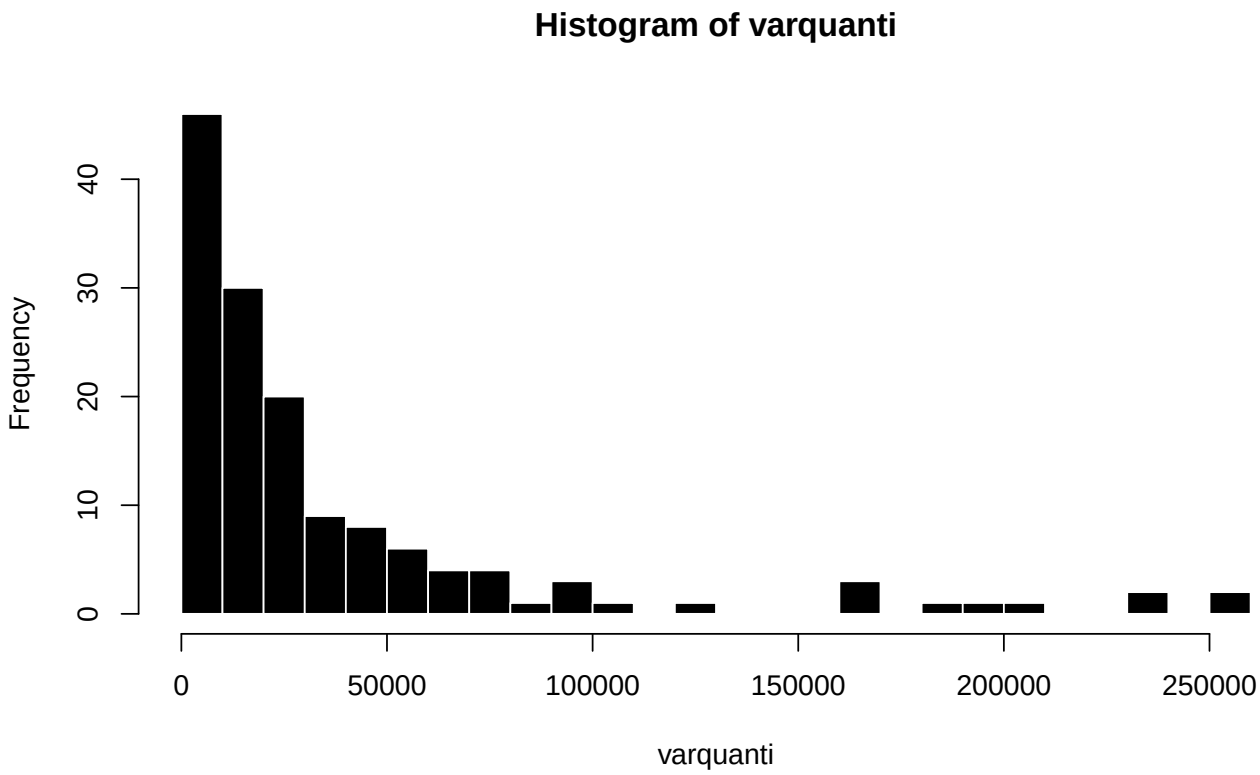
Je dois donc répéter ces commandes pour chacune des variables, ce qui comporte plusieurs inconvénients :

1. Cette répétition prend du temps à écrire.
2. Elle prend de la place dans le script.
3. Elle est source d'erreurs parce que je ferais certainement des copier-coller en changeant les noms de variables.
4. L'utilisation de la fonction `hist()` sans arguments graphiques n'est pas très satisfaisante, je voudrais personnaliser ce graphique mais ce sont encore des lignes de code qui s'ajoutent et que je devrai répéter à chaque variable.

```
ExploreQuanti <- function(varquanti){
  # calcul des quartiles
  varQuartiles <- quantile(varquanti, probs = seq(0, 1, 1/4), na.rm = TRUE)

  # combinaison des mesures de centralité et de dispersion
  summaryVar <- c(MIN = min(varquanti, na.rm = TRUE),
                  QUART1 = varQuartiles[2],
                  MEDIAN = varQuartiles[3],
                  QUART3 = varQuartiles[4],
                  MAX = max(varquanti, na.rm = TRUE),
                  MEAN = mean(varquanti, na.rm = TRUE),
                  SD = sd(varquanti, na.rm = TRUE))

  # affiche les résultats
  hist(varquanti, breaks = 20, col = "black", border = "white")
  print(summaryVar)

  # désigne le(s) objet(s) à renvoyer en sortie
  return(summaryVar)
}
```

Cette fonction peut être utilisée sur une des variables du tableau :

```
mySummary36 <- ExploreQuanti(varquanti = popHist$POP1936)
```




```
##         MIN QUART1.25% MEDIAN.50% QUART3.75%         MAX        MEAN
##         276        6952       18172       43384      258599       37140
##          SD
##       52966
```

Elle peut aussi être appliquée à un ensemble de variables avec la fonction `apply()`. Dans cet exemple, elle est appliquée colonne à colonne (`MARGIN = 2`) sur l'ensemble des variables de population.

```
mySummary3608 <- apply(popHist[3:12], MARGIN = 2, FUN = ExploreQuanti)
```

Cette fonction convient à l'usage fréquent que j'en fait mais je voudrais garder le contrôle sur l'argument `breaks` de la fonction `hist()` pour le modifier selon la variable que j'analyse. Je remplace donc le nombre fixe de seuils (20) par un nom et je fais remonter ce nom dans les arguments de ma fonction, ici nbbreaks.

```
ExploreQuanti <- function(varquanti, nbbreaks){
  # calcul des quartiles
  varQuartiles <- quantile(varquanti, probs = seq(0, 1, 1/4), na.rm = TRUE)

  # combinaison des mesures de centralité et de dispersion
  summaryVar <- c(MIN = min(varquanti, na.rm = TRUE),
                  QUART1 = varQuartiles[2],
                  MEDIAN = varQuartiles[3],
                  QUART3 = varQuartiles[4],
```

```
                  MAX = max(varquanti, na.rm = TRUE),
                  MEAN = mean(varquanti, na.rm = TRUE),
                  SD = sd(varquanti, na.rm = TRUE))

  # affiche les résultats
  hist(varquanti, breaks = nbbreaks, col = "black", border = "white")
  print(summaryVar)

  # désigne le(s) objet(s) à renvoyer en sortie
  return(summaryVar)
}
```

Pour utiliser cette fonction je devrai maintenant aussi spécifier l'argument `nbbreaks`.

```
mySummary36 <- ExploreQuanti(varquanti = popHist$POP1936, nbbreaks = 10)
```

Exemple d'application : exploration d'une relation entre deux variables quantitatives

Cet exemple est très similaire au précédent mais cette fois-ci la fonction doit prendre comme argument deux variables quantitatives à mettre en relation. Il s'agit de créer une fonction :

1. qui calcule les coefficients associés à cette relation (covariance, corrélation, détermination),
2. qui affiche un nuage de points entre les deux variables,
3. qui trace la droite de régression sur ce graphique.

L'utilité de ma fonction reste assez réduite, elle permet simplement de faire tout-en-un un traitement qui m'aurait demandé plusieurs fonctions : `plot()` pour le nuage de points, `lm()` et `abline()` pour la droite de régression, et enfin `cov()` et `cor()` pour les coefficients.

Cette fonction prend comme argument le tableau contenant les variables (`tab`) ainsi que le nom des deux variables (`varquanti` et `varquali`). À la différence de l'exemple précédent, `varquanti` et `varquali` ne désigneront pas les variables elles-mêmes mais le nom de ces variables dans le tableau. On appellera la variable entre crochets, par exemple `popHist[ , "POP1936"]`, syntaxe équivalente à `popHist$POP1936`.

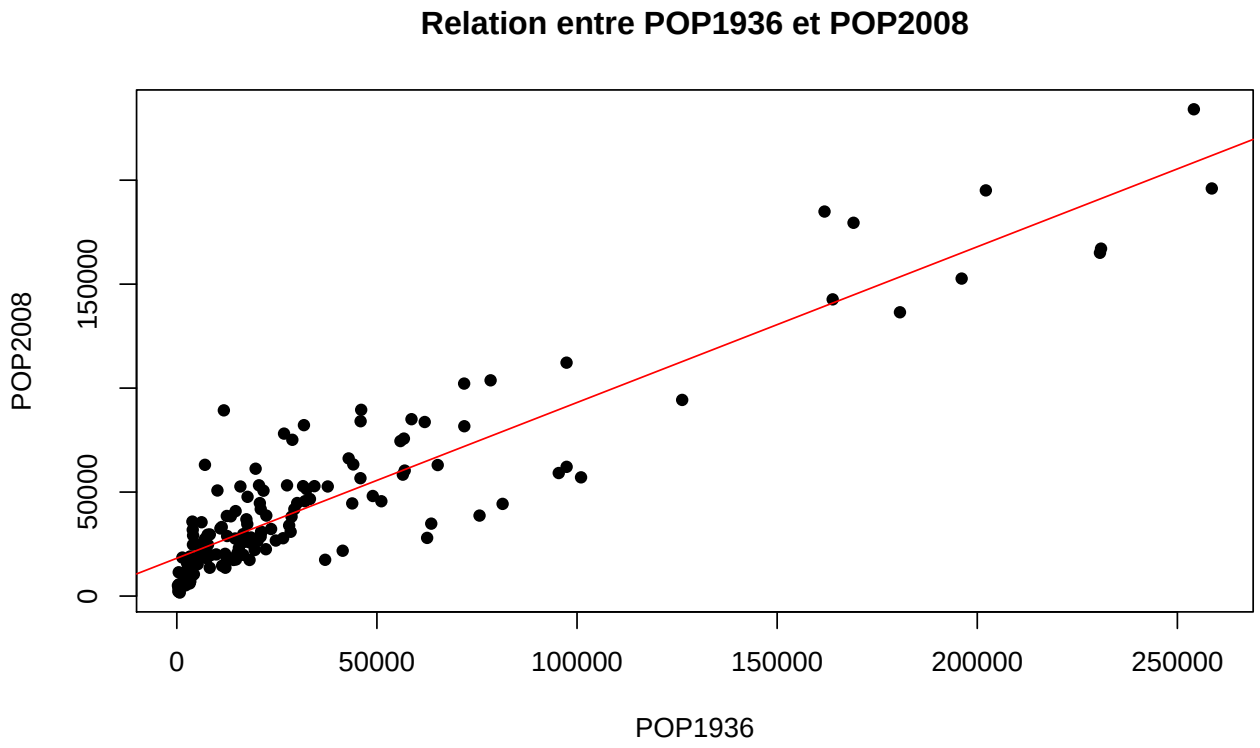
```
ExploreQuantiQuanti <- function(tab, var1, var2){
  # covariance, coefficient de corrélation et coefficient de détermination
  covCoef <- round(cov(tab[, var1], tab[, var2]), digits = 2)
  corCoef <- round(cor(tab[, var1], tab[, var2]), digits = 2)
  detCoef <- round(corCoef ^ 2, digits = 2)

  # affiche les coefficients en précisant le nom de chacun d'eux
  print(paste("Covariance :", covCoef, sep = " "))
  print(paste("Corrélation :", corCoef, sep = " "))
  print(paste("Détermination :", detCoef, sep = " "))

  # calcule le modèle de régression linéaire
  linMod <- lm(formula = tab[ ,var2] ~ tab[ ,var1])
```

```r
  # affiche le graphique et la droite de régression
  plot(x = tab[, var1], y = tab[, var2],
       main = paste("Relation entre", var1, "et", var2, sep = " "),
       xlab = var1,
       ylab = var2,
       pch = 16)

  abline(linMod, col = "red")

  # désigne le(s) objet(s) à renvoyer en sortie
  return(linMod)
}
```

Exécutons le bloce de code définissant la fonction (sourcer la fonction) et testons la sur les variables de population du tableau, qui devraient montrer une relation linéaire très nette :

```r
linearReg <- ExploreQuantiQuanti(tab = popHist, var1 = "POP1936", var2 = "POP2008")
```

```
## [1] "Covariance : 2100890678.11"
## [1] "Corrélation : 0.92"
## [1] "Détermination : 0.85"
```




```r
# vérification que la fonction a bien renvoyé le modèle (objet de type "lm")
class(linearReg)
```

```
## [1] "lm"
```

Exemple d'application : transformation et exploration des matrices de flux

Je travaille sur des données qui comportent souvent des couples de lieux assortis d'une mesure : des distances, des flux migratoires entre les régions françaises, des flux commerciaux entre les pays du monde, des navettes domicile-travail entre les communes d'une aire urbaine. En générale, je dispose dans ce cas d'une tableau long, c'est-à-dire un tableau à 3 champs : ORIGINE, DESTINATION, FLUX.

Une manipulation fréquente que je dois faire est de :

1. Transformer cette matrice "longue" en matrice "large" : une matrice carrée de n par n, les lignes sont les n origines, les colonnes sont les n destinations, la matrice est remplie de valeurs correspondantes à des flux ou à des distances.
2. Explorer cette matrice de flux en calculant des résumés numériques sur les lignes ou les colonnes : quelle est la somme des flux sortants (donc à l'origine, somme ligne à ligne) ? Quelle est la somme des flux entrants (donc à destination, somme colonne à colonne) ? Quelle est la distance moyenne de chaque lieu vers tous les autres (moyenne sur les lignes ou les colonnes, indifférent si la matrice est symmétrique)

Je cherche donc à créer une fonction qui effectue ces deux traitements de façon automatisée. L'exemple d'application porte ici sur les navettes domicile-travail entre les communes de Paris et de petite couronne (départements 75, 92, 93, 94, 143 communes).

Chargement et préparation des données :

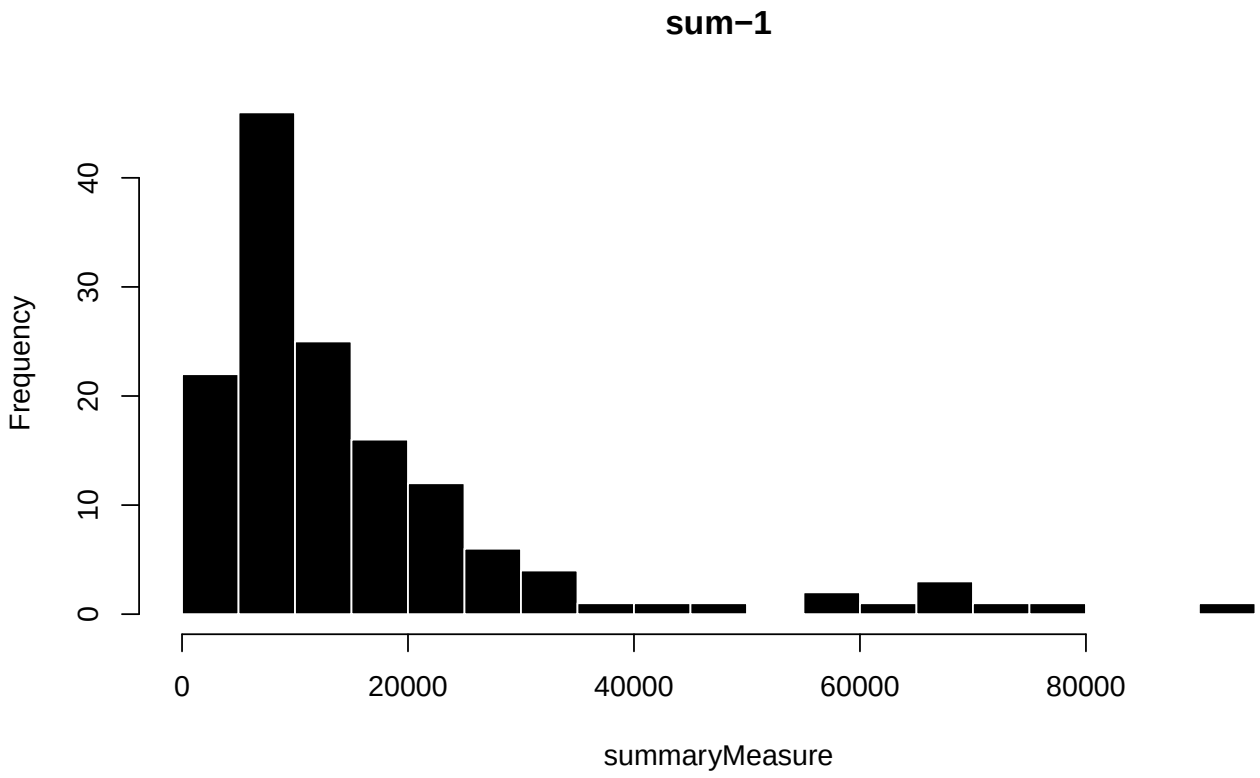
```
# charge les tableaux
navMat <- read.csv(file = "Donnees/NavLongPPC.csv", stringsAsFactors = FALSE)
```

Création de la fonction :

```
library(reshape2)

ReshapeAndSummary <- function(longmat, orivar, desvar, flowvar,
                              measure = "sum", mydim = 1, showplot = TRUE){
  # transforme le tableau du format long au format large
  wideMat <- dcast(longmat, formula = longmat[, orivar] ~ longmat[, desvar],
                   value.var = flowvar, fill = 0, drop = FALSE)

  wideMat <- as.matrix(wideMat[ , -1])
  row.names(wideMat) <- colnames(wideMat)

  # résumé numérique sur l'une des deux dimensions du tableau (1 = row, 2 = col)
  summaryMeasure <- apply(wideMat, mydim, measure)

  # affiche l'histogramme correspondant
  if(showplot == TRUE){
      hist(summaryMeasure, main = paste(measure, mydim, sep = "-"),
           breaks = 20, col = "black", border = "white")
  } else {}

  return(list(MAT = wideMat, INDIC = summaryMeasure))
}
```

Utilisation de la fonction :

```
wideMatrix <- ReshapeAndSummary(longmat = navMat, orivar = "ORI", desvar = "DES", flowvar = "FLOW")
```




```
str(wideMatrix$MAT)
```

```
##  num [1:143, 1:143] 4028 185 245 164 259 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:143] "75101" "75102" "75103" "75104" ...
##   ..$ : chr [1:143] "75101" "75102" "75103" "75104" ...
```

```
str(wideMatrix$INDIC)
```

```
##  Named num [1:143] 7394 10404 15024 10487 22007 ...
##  - attr(*, "names")= chr [1:143] "75101" "75102" "75103" "75104" ...
```

Approfondissement sur les fonctions

Cette section est fortement inspirée des matériaux pédagogiques produits par Hadley Wickham (http://had.co.nz et http://adv-r.had.co.nz).

Composition d'une fonction

Les fonctions se composent de trois types d'éléments : le corps de la fonction (son code), ses arguments, son environnement de référence. Les trois fonctions suivantes - `body()`, `formals()`, `environment()` - renvoient l'information correspondante à ces trois éléments:

```
body(ExploreQuanti)
```

```
## {
##     varQuartiles <- quantile(varquanti, probs = seq(0, 1, 1/4),
##         na.rm = TRUE)
##     summaryVar <- c(MIN = min(varquanti, na.rm = TRUE), QUART1 = varQuartiles[2],
##         MEDIAN = varQuartiles[3], QUART3 = varQuartiles[4], MAX = max(varquanti,
##             na.rm = TRUE), MEAN = mean(varquanti, na.rm = TRUE),
##         SD = sd(varquanti, na.rm = TRUE))
##     hist(varquanti, breaks = nbbreaks, col = "black", border = "white")
##     print(summaryVar)
##     return(summaryVar)
## }
```

```
formals(ExploreQuanti)
```

```
## $varquanti
##
##
## $nbbreaks
```

```
environment(ExploreQuanti)
```

```
## <environment: R_GlobalEnv>
```

Certaines fonctions de base ne présentent pas ces trois éléments, ce sont les fonctions dites "primitives" qui ne sont codées dans R mais dans un langage compilé. La function R fait appel alors au code compilé (voir la Section 3.3 du manuel *R et espace*, http://framabook.org/16-r-et-espace). Essayez d'utiliser `body()`, `formals()` et `environment()` avec la function `sum()`.

Portée lexicale

La portée lexicale (*lexical scoping*) deśgne la portée du lien qui est fait entre un nom et un objet.

Lorsqu'un objet est créé au sein d'une fonction, le lien entre le nom et l'objet se fait dans l'environnement de la fonction et l'objet n'existe qu'à l'intérieur de fonction. On parle dans ce cas de *variable locale*. Par exemple, lors de l'exécution de la fonction `ExploreQuanti()` un objet `varQuartiles` a été créé mais il n'existe pas dans l'environnement global (il ne s'affiche pas dans l'onglet `Environment` de RStudio).

Lorsqu'un objet est créé directement dans l'environnement global, il existe globalement et peut être utilisé n'importe où dans le code, au sein des fonctions mais également hors des fonctions. On parle dans ce cas de *variable globale*.

```
x <- 1              # variable créée dans l'environnement global
f0 <- function(){
  y <- 2            # variable créée dans l'environnement local de la fonction f0()
  exists("x")
  exists("y")
}

f0()
```

```
## [1] TRUE
```

Dans l'exemple ci-dessus, la variable globale `x` comme la variable locale `y` existent au sein de la fonction `f1()`. Dans l'exemple ci-dessous, on teste l'existence de `x` et `y` dans l'environnement global, seule la variable `x` existe et non la variable `y`.

```
exists("x")
```

```
## [1] TRUE
```

```
exists("y")
```

```
## [1] FALSE
```

Voici un exemple pour mieux saisir les enjeux de la portée lexicale : un objet `x` est créé dans l'environnement global ainsi que trois fonctions `f1()`, `f2()` et `f3()`.

```
# création d'une variable globale nommée "x"
x <- 1

# fonction 1
f1 <- function(){
  print(x ^ 2)
}

# fonction 2
f2 <- function(x){
  print(x ^ 2)
}

# fonction 3
f3 <- function(x){
  x <- 100
  print(x ^ 2)
}
```

Ces fonctions renvoient trois résultats différents :

```
f1()
```

```
## [1] 1
```

```
f2(x = 10)
```

```
## [1] 100
```

```
f3(x = 10)
```

```
## [1] 10000
```

Que s'est-il passé ? Ces trois fonctions cherchent à passer un objet nommé `x` au carré :

1. Dans la fonction `f1()`, R ne trouve pas d'objet `x` au sein du corps de la fonction, puis il ne trouve pas non plus d'objet `x` dans les arguments de la fonction, finalement il trouve un objet `x` (qui vaut 1) dans l'environnement global et c'est celui-là qu'il utilise.
2. Dans la fonction `f2()`, R ne trouve pas d'objet `x` au sein du corps de la fonction mais il le trouve dans les arguments de la fonction (il vaut 10), et c'est donc celui-là qu'il utilise.
3. Dans la fonction `f3()`, R trouve un objet `x` au sein du corps de la fonction : c'est celui-là qui prime sur l'objet `x` défini dans les arguments et sur l'objet `x`définit dans l'environnement global.

Ainsi, la portée lexicale peut être vue comme un emboîtement d'environnements du plus local (corps de la fonction) au plus global (environnement global) dans lesquels R va chercher les objets nécessaires. Tout ce qui est créé dans le corps d'une fonction n'existe que dans son environnement local.

Redondance des noms de variables et des noms de fonction

Il arrive que l'utilisateur nomme des objets avec des noms de fonctions, par exemple il crée un objet `var` pour stocker une variable. Or, il existe déjà un objet nommé `var`dans R, il s'agit d'une fonction qui calcule la variance.

```
# création d'une variable nommée "var"
var <- c(2, 4, 6)

# utilisation de la variable "var"
var ^ 2
```

```
## [1]  4 16 36
```

```
# utilisation de la fonction "var"
var(popHist$POP1936)
```

```
## [1] 2805362710
```

```
# confusion totale
var(var)
```

```
## [1] 4
```

À la création d'un objet `var`, R fait la différence entre l'objet variable et l'objet fonction. En revanche, si l'utilisateur crée et source une fonction qui porte le même nom qu'une fonction pré-existante, cette dernière est masquée :

```
sum(c(2, 4, 6))
```

```
## [1] 12
```

```
sum <- function(vec){
  mean(vec)
}

sum(c(2, 4, 6))
```

```
## [1] 4
```

C'est aussi ce qui arrive lors du chargement de *packages* contenant des fonctions du même nom, par exemple avec les packages `plyr` et `dplyr` qui est une ré-écriture du premier :

```
library(plyr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
##
## The following object is masked from 'package:stats':
##
##     filter
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Dans tous les cas, utiliser des noms de fonctions pré-existantes pour désigner des variables ou de nouvelles fonctions est une très mauvaise pratique qu'il faut éviter à tout prix.

Désignation des arguments de la fonction

Lorsqu'une fonction attend plusieurs arguments, ceux-ci peuvent être désignés sans être spécifiés uniquement si l'ordre des arguments est respecté. Par exemple, je souhaite calculer la moyenne des populations communales selon le département en utilisant la fonction `tapply()`. Cette fonction attend trois arguments, dans cet ordre : une variable quantitative, une variable qualitative (agrégation), une fonction (résumé numérique).

```
tapply(X = popHist$POP1936, INDEX = popHist$DEP, FUN = mean)
```

```
##        75        92        93        94
## 141487.75  28322.97  19409.45  14578.81
```

Si l'utilisateur spécifie les arguments, leur ordre n'importe pas. En revanche, si l'ordre n'est pas respecté et que les arguments ne sont pas spécifiés, la fonction ne fait pas ce qui est attendu :

```
# Arguments spécifiés, ordre non respecté
tapply(FUN = mean, X = popHist$POP1936, INDEX = popHist$DEP)

# Arguments non spécifiés, ordre non respecté
tapply(mean, popHist$POP1936, popHist$DEP)
```
