Fonctions en R

Sophie Baillargeon, Université Laval

2019-03-10

Table des matières

Lorsqu'un bout de code R est susceptible d'être utilisé à répétition (par exemple pour faire un même calcul sur des données différentes), il est préférable d'en faire une fonction R.

Avantages des fonctions :

- sauver du temps,
- diminuer les risques de faire des erreurs,
- rédiger du code plus clair et plus court, donc plus facile à comprendre et à partager.

Bref, faire des fonctions est une bonne pratique de programmation en R.

Syntaxe générale d'une fonction R

Pour créer une fonction en R, il faut utiliser le mot-clé `function` en respectant la syntaxe suivante :

```
nomFonction <- function(arg1, arg2, arg3){
  instructions  # formant le corps de la fonction
}
```

`arg1`, `arg2` et `arg3` représentent les arguments de la fonction, soit les objets qui peuvent être fournis en entrée à la fonction, qui ne sont pas nécessairement au nombre de trois.

Voici une fonction qui reprend un exemple présenté dans les notes sur les structures de contrôle en R. Elle calcule des statistiques descriptives simples selon le type des éléments du vecteur donné en entrée.

```
statDesc <- function(x){
  if (is.numeric(x)) {
    min <- min(x)
    moy <- mean(x)
    max <- max(x)
    stats <- c(min = min, moy = moy, max = max)
  } else if (is.character(x) || is.factor(x)) {
    stats <- table(x)
  } else {
    stats <- NA
  }
  stats
}
```

Après avoir soumis le code de création de cette fonction dans la console, la fonction se retrouve dans l'environnement de travail. Il est alors possible de l'appeler.

```
statDesc(x = iris$Species)
```

```
## x
##     setosa versicolor  virginica
##         50         50         50
```

Nous pourrions ajouter un argument à cette fonction. Par exemple, nous pourrions offrir l'option d'une sortie présentée sous la forme d'une matrice plutôt que d'un vecteur.

```
statDesc <- function(x, sortieMatrice){
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x), moy = mean(x), max = max(x))
  } else if (is.character(x) || is.factor(x)) {
    stats <- table(x, dnn = NULL)
  } else {
    stats <- NA
  }
  # Production de la sortie
  if (sortieMatrice){
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  }
  stats
}
```

Le code de la fonction a aussi été un peu reformaté. Nous pouvons maintenant appeler la fonction comme

suit.

```
statDesc(x = iris$Species, sortieMatrice = TRUE)
```

```
##            frequence
## setosa            50
## versicolor        50
## virginica         50
```

Composantes d'une fonction R

Les composantes d'une fonction R sont :

- la liste de ses arguments, possiblement avec des valeurs par défaut (nous allons y revenir) ;

```
args(statDesc)
```

```
## function (x, sortieMatrice)
## NULL
```

- le corps de la fonction, soit les instructions qui la constituent.

```
body(statDesc)
```

```
## {
##     if (is.numeric(x)) {
##         stats <- c(min = min(x), moy = mean(x), max = max(x))
##     }
##     else if (is.character(x) || is.factor(x)) {
##         stats <- table(x, dnn = NULL)
##     }
##     else {
##         stats <- NA
##     }
##     if (sortieMatrice) {
##         stats <- as.matrix(stats)
##         colnames(stats) <- if (is.character(x) || is.factor(x))
##             "frequence"
##         else "stat"
##     }
##     stats
## }
```

- l'environnement englobant de la fonction (défini plus loin).

```
environment(statDesc)
```

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

Fonction sans nom

Notons qu'une fonction n'a même pas besoin de porter de nom. La grande majorité du temps, une fonction est conçue pour être appelée à plusieurs reprises et il est alors nécessaire qu'elle ait un nom. Cependant, certaines fonctions sont parfois à usage unique.

Par exemple, il est parfois utile de se créer une fonction pour personnaliser le calcul effectué par une fonction de la famille des `apply`. Si cette fonction est très courte et a peu de chance d'être réutilisée, il n'est pas nécessaire de la nommer.

Voici un exemple. Si nous voulions calculer le minimum, la moyenne et le maximum (comme le fait notre fonction `statDesc`) de toutes les variables numériques du jeu de données `iris`, mais selon le niveau de la variable `Species`, nous pourrions utiliser trois appels à la fonction `aggregate` comme suit.

```
aggregate(x = iris[, -5], by = list(iris$Species), FUN = min)
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa          4.3         2.3          1.0         0.1
## 2 versicolor          4.9         2.0          3.0         1.0
## 3  virginica          4.9         2.2          4.5         1.4
```

```
aggregate(x = iris[, -5], by = list(iris$Species), FUN = mean)
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

```
aggregate(x = iris[, -5], by = list(iris$Species), FUN = max)
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa          5.8         4.4          1.9         0.6
## 2 versicolor          7.0         3.4          5.1         1.8
## 3  virginica          7.9         3.8          6.9         2.5
```

Nous pourrions aussi créer une fonction qui calcule les trois statistiques et donner cette fonction en entrée à `aggregate` comme valeur à l'argument `FUN`.

```
aggregate(x   = iris[, -5],
          by  = list(iris$Species),
          FUN = function(x) c(min = min(x), moy = mean(x), max = max(x)))
```

```
##      Group.1 Sepal.Length.min Sepal.Length.moy Sepal.Length.max Sepal.Width.min
## 1     setosa            4.300            5.006            5.800           2.300
## 2 versicolor            4.900            5.936            7.000           2.000
## 3  virginica            4.900            6.588            7.900           2.200
##   Sepal.Width.moy Sepal.Width.max Petal.Length.min Petal.Length.moy Petal.Length.max
## 1           3.428           4.400            1.000            1.462            1.900
## 2           2.770           3.400            3.000            4.260            5.100
## 3           2.974           3.800            4.500            5.552            6.900
##   Petal.Width.min Petal.Width.moy Petal.Width.max
## 1           0.100           0.246           0.600
## 2           1.000           1.326           1.800
## 3           1.400           2.026           2.500
```

Nous n'avons jamais donné de nom à la fonction et cela n'a causé aucun problème. Nous n'avons même pas utilisé d'accolades pour encadrer le corps de la fonction. Ce n'est pas nécessaire lorsque celui-ci est composé d'une seule instruction.

Arguments en entrée

Les arguments sont définis en énumérant leurs noms entre les parenthèses après le mot-clé `function`.

```
nomFonction <- function(arg1, arg2, arg3){
  instructions  # formant le corps de la fonction
}
```

Il n'y a pas de restrictions quant au nombre d'arguments que peut posséder une fonction. Exceptionnellement,

une fonction peut même ne posséder aucun argument. C'est la cas par exemple de la fonction `getwd` et de la fonction suivante.

```
HelloWorld <- function() cat("Hello World !")
```

Comme nous le savons déjà, pour appeler une fonction sans fournir d'arguments, il faut tout de même utiliser les parenthèses.

```
HelloWorld()
```

```
## Hello World !
```

Omettre les parenthèses retourne le code source de la fonction.

```
HelloWorld
```

```
## function() cat("Hello World !")
```

Valeurs par défaut des arguments

Afin de définir une valeur par défaut pour un argument, il faut accompagner son nom dans l'énumération des arguments d'un opérateur `=` et d'une instruction R retournant la valeur par défaut. Par exemple, dans la fonction `statDesc`, il serait préférable de définir un format par défaut pour la sortie.

```
statDesc <- function (x, sortieMatrice = FALSE) {
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x), moy = mean(x), max = max(x))
  } else if (is.character(x) || is.factor(x)) {
    stats <- table(x, dnn = NULL)
  } else {
    stats <- NA
  }
  # Production de la sortie
  if (sortieMatrice){
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  }
  stats
}
```

Les arguments qui ne possèdent pas de valeur par défaut sont obligatoires. Si une fonction est appelée sans donner de valeur en entrée à un paramètre obligatoire, une erreur est produite.

```
statDesc(sortieMatrice = FALSE)
```

```
## Error in statDesc(sortieMatrice = FALSE): argument "x" is missing, with no default
```

Les arguments ayant une valeur par défaut peuvent, pour leur part, ne pas être fournis en entrée, auquel cas leur valeur par défaut est utilisée.

```
statDesc(x = iris$Sepal.Length)
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

Valeur par défaut pour un argument acceptant seulement un petit nombre de chaînes de caractères spécifiques

Attardons-nous maintenant à un cas particulier de valeur par défaut en R. Supposons qu'une fonction possède un argument qui prend en entrée une chaîne de caractères et que seulement un petit nombre de chaînes de caractères distinctes sont acceptées par cet argument. C'est le cas par exemple pour l'argument `useNA` de la fonction `table`. La fonction accepte seulement les valeurs `"no"`, `"ifany"` ou `"always"` pour cet argument. Donner une valeur autre à l'argument produira une erreur.

```
table(iris$Species, useNA = "test")
```

```
## Error in match.arg(useNA): 'arg' should be one of "no", "ifany", "always"
```

Une pratique courante en R pour un argument de ce type est de lui donner comme valeur dans l'énumération des arguments le vecteur de toutes ses valeurs possibles. C'est ce qui est fait dans la fonction table.

```
args(table)
```

```
## function (..., exclude = if (useNA == "no") c(NA, NaN), useNA = c("no",
##     "ifany", "always"), dnn = list.names(...), deparse.level = 1)
## NULL
```

La valeur par défaut de l'argument n'est pas, dans ce cas, le vecteur complet `c("no", "ifany", "always")`, mais plutôt le premier élément de ce vecteur, soit `"no"`. Il en est ainsi, car le corps de la fonction contient l'instruction suivante.

```
useNA <- match.arg(useNA)
```

La fonction `match.arg` vérifie que la valeur donnée en entrée à un argument est bien une valeur acceptée ou retourne le premier élément du vecteur de valeurs possibles si aucune valeur n'a été donnée en entrée à l'argument.

Nous devrions reproduire cette façon de faire dans nos propres fonctions qui possèdent un argument du même type que l'argument `useNA` de la fonction `table`. Par exemple, remplaçons l'argument `sortieMatrice` de notre fonction `statDesc` par l'argument `formatSortie` comme suit.

```
statDesc <- function (x, formatSortie = c("vecteur", "matrice", "liste")) {
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x), moy = mean(x), max = max(x))
  } else if (is.character(x) || is.factor(x)) {
    stats <- table(x, dnn = NULL)
  } else {
    stats <- NA
  }
  # Production de la sortie
  formatSortie <- match.arg(formatSortie)
  if (formatSortie == "matrice"){
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  } else if (formatSortie == "liste") {
    stats <- as.list(stats)
  }
  stats
}
```

La valeur par défaut de l'argument `formatSortie` est bel et bien `"vecteur"`.

```r
statDesc(x = iris$Sepal.Length)
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

```r
statDesc(x = iris$Sepal.Length, formatSortie = "vecteur")
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

La présence du vecteur des chaînes de caractères possibles dans la définition des arguments est informative, car elle indique à l'utilisateur quelles valeurs sont acceptées par l'argument.

Appel d'une fonction

Les appels à nos propres fonctions respectent les mêmes règles que les appels à n'importe quelle fonction R. En plus du fonctionnement des valeurs par défaut décrit ci-dessus, rappelons que les arguments peuvent être fournis à une fonction R par position, par nom complet ou même par nom partiel. L'association des arguments à leurs valeurs se fait en respectant les règles de préséances suivantes :

1. d'abord les arguments fournis avec un nom exact se voient attribuer une valeur,
2. puis les arguments fournis avec un nom partiel,
3. et finalement les arguments non nommés, selon leurs positions.

Voici quelques exemples.

```r
testAppel <- function(x, option, param, parametre) {
  cat("l'argument x prend la valeur", x, "\n")
  cat("l'argument option prend la valeur", option, "\n")
  cat("l'argument param prend la valeur", param, "\n")
  cat("l'argument parametre prend la valeur", parametre, "\n")
}
```

```r
testAppel(1, 2, 3, 4)
```

```
## l'argument x prend la valeur 1
## l'argument option prend la valeur 2
## l'argument param prend la valeur 3
## l'argument parametre prend la valeur 4
```

```r
testAppel(1, 2, param = 3, opt = 4)
```

```
## l'argument x prend la valeur 1
## l'argument option prend la valeur 4
## l'argument param prend la valeur 3
## l'argument parametre prend la valeur 2
```

```r
testAppel(1, par = 2, option = 3, 4)
```

```
## Error in testAppel(1, par = 2, option = 3, 4): argument 2 matches multiple formal arguments
```

Une bonne pratique de programmation en R est d'utiliser l'association par positionnement seulement pour les premiers arguments, ceux les plus souvent utiliser. Les arguments moins communs devraient être nommés afin de conserver un code facile à comprendre.

Passage d'arguments par valeur

Lors de l'appel d'une fonction R, lorsque nous assignons à un argument un objet de notre environnement de travail, une copie de cet objet est créé et les instructions du corps de la fonction affectent cette copie et non l'objet d'origine. En terminologie informatique, on dit que R utilise toujours du passage d'arguments par valeur.

Un autre type de passage d'arguments utilisé par d'autres languages informatiques est le passage par référence. Avec ce type de passage, les objets passés ne sont pas recopiés et les instructions du corps d'une fonction peuvent modifier l'objet d'origine. Cependant, cela n'arrive jamais en R.

Par exemple, supposons que notre environnement de travail comporte un objet nommé `x` contenant le nombre 5.

```
x <- 5
x
```

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

Créons une simple fonction R qui ajoute une unité à des nombres.

```
ajoute1 <- function(x) x + 1
```

Maintenant, appelons cette fonction en lui donnant en entrée l'objet `x` de notre environnement de travail.

```
ajoute1(x = x)
```

```
## [1] 6
```

La fonction retourne le résultats de `x + 1`, soit 6. Mais est-ce que l'objet `x` a pour autant changé ?

```
x
```

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

Non. Il contient toujours la valeur 5.

Remarquez qu'ici le nom `x` a été utilisé pour deux entités distinctes :

- un objet dans notre environnement de travail,
- un argument de la fonction `ajoute1`.

Dans l'instruction `ajoute1(x = x)`, nous avons cependant assigné la valeur contenu dans l'objet `x` à l'argument portant le même nom.

Comment pourrions-nous modifier l'objet `x` de notre environnement de travail à l'aide de la fonction `ajoute1` ? Il faudrait assigner le résultat de l'instruction `ajoute1(x = x)` au nom `x` comme suit.

```
x <- ajoute1(x = x)
```

En fait, cette commande écrase l'ancien objet `x` par un nouveau, contenant la valeur retournée par `ajoute1(x = x)`.

```
x
```

```
## [1] 6
```

Argument ...

Les deux utilités de l'argument `...` ont été mentionnées lors du premier cours. Nous pouvons utiliser cet argument dans nos propres fonctions, en exploitant l'une ou l'autre de ses utilités.

Utilité 1 : recevoir un nombre indéterminé d'arguments

L'argument `...` peut permettre de prendre un nombre indéterminé d'objets en entrée, comme dans cet exemple.

```
statDescMulti <- function(...){
  args <- list(...)
  lapply(X = args, FUN = statDesc)
}
```

Le corps de la fonction doit contenir une instruction telle que `list(...)` pour récupérer tous les objets.

Voici un exemple d'appel à cette fonction.

```
statDescMulti(iris$Sepal.Length, iris$Petal.Width, iris$Species)
```

```
## [[1]]
##      min      moy      max
## 4.300000 5.843333 7.900000
##
## [[2]]
##      min      moy      max
## 0.100000 1.199333 2.500000
##
## [[3]]
##     setosa versicolor  virginica
##         50         50         50
```

Il est même possible d'attribuer un nom aux arguments passés. Ces noms deviennent les noms des éléments de la liste retournée en sortie.

```
statDescMulti(Sepal.Length = iris$Sepal.Length,
              Species = iris$Species)
```

```
## $Sepal.Length
##      min      moy      max
## 4.300000 5.843333 7.900000
##
## $Species
##     setosa versicolor  virginica
##         50         50         50
```

Utilité 2 : passer des arguments à une autre fonction

L'argument `...` permet aussi de passer des arguments à une fonction appelée dans le corps de la fonction. Par exemple, l'argument `...` serait utile à notre fonction `statDesc` pour contrôler le traitement des valeurs manquantes. Dans le corps de la fonction, les appels aux fonctions auxquelles nous souhaitons permettre le passage d'arguments doivent contenir l'argument `...`, comme dans l'exemple suivant.

```
statDesc <- function (x, formatSortie = c("vecteur", "matrice", "liste"), ...) {
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x, ...), moy = mean(x, ...), max = max(x, ...))  # ... ici
  } else if (is.character(x) || is.factor(x)) {
```

```
    stats <- table(x, dnn = NULL)
  } else {
    stats <- NA
  }
  # Production de la sortie
  formatSortie <- match.arg(formatSortie)
  if (formatSortie == "matrice"){
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  } else if (formatSortie == "liste") {
    stats <- as.list(stats)
  }
  stats
}
```

Dans cet exemple, l'argument `...` permet de passer des arguments aux fonctions `min`, `mean` et `max`.

```
statDesc(x = c(iris$Sepal.Length, NA))
```

```
## min moy max
##  NA  NA  NA
```

```
statDesc(x = c(iris$Sepal.Length, NA), na.rm = TRUE)
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

Sortie d'une fonction

Une fonction retourne :

- l'objet donné en argument à la fonction `return` dans le corps de la fonction,
- ou, en l'absence d'appel à la fonction `return`, la dernière expression évaluée dans le corps de la fonction.

Par exemple, la version suivante de la fonction `statDescMulti` retourne la liste des arguments fournis en entrée plutôt que le résultat de l'appel à `lapply` à cause de la présence de la fonction `return`.

```
statDescMulti <- function(...){
  args <- list(...)
  return(args)
  lapply(X = args, FUN = statDesc)
}
```

```
statDescMulti(rating = attitude$rating, complaints = attitude$complaints)
```

```
## $rating
##  [1] 43 63 71 61 81 43 58 71 72 67 64 67 69 68 77 81 74 65 65 50 50 64 53
## [24] 40 63 66 78 48 85 82
##
## $complaints
##  [1] 51 64 70 63 78 55 67 75 82 61 53 60 62 83 77 90 85 60 70 58 40 61 66
## [24] 37 54 77 75 57 85 82
```

Une fonction ne peut retourner qu'un seul objet. Pour retourner plusieurs objets, il faut les combiner dans un seul objet (typiquement dans une liste), comme dans l'exemple suivant.

```
statDescMulti <- function(...){
  call <- match.call()
```

```
  args <- list(...)
  stats <- lapply(X = args, FUN = statDesc)
  list(stats = stats, call = call)
}
```

```
statDescMulti(rating = attitude$rating, complaints = attitude$complaints)
```

```
## $stats
## $stats$rating
##      min      moy      max
## 40.00000 64.63333 85.00000
##
## $stats$complaints
##  min  moy  max
## 37.0 66.6 90.0
##
##
## $call
## statDescMulti(rating = attitude$rating, complaints = attitude$complaints)
```

Pour faciliter la réutilisation des résultats, il est souhaitable de toujours nommer les éléments d'une liste retournée en sortie.

Fonction `match.call`

L'exemple précédent fait intervenir la fonction `match.call`. Il est commun pour des fonctions d'ajustement de modèle telles que `lm` de retourner dans la sortie une copie de l'appel de la fonction.

```
exemple <- lm(rating ~ raises, data = attitude)
exemple$call
```

```
## lm(formula = rating ~ raises, data = attitude)
```

C'est la fonction `match.call` qui permet de créer cet élément de la sortie.

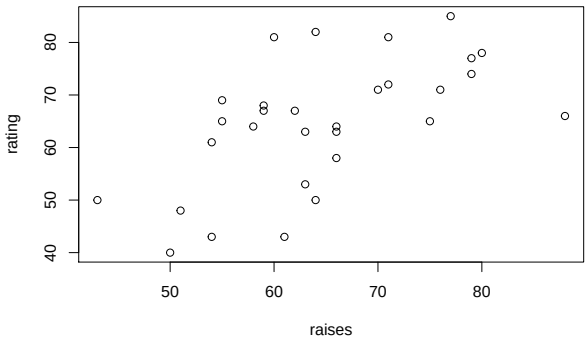
Les fonctions `match.call` et `return` sont des exemples de fonctions seulement utiles dans le corps d'une fonction. Les appeler directement dans la console retourne une erreur ou une sortie sans intérêt.

Effets de bord d'une fonction

En plus de potentiellement retourner un objet, l'exécution d'une fonction peut produire des « effets de bord » (en anglais *side effects*). Ces effets de bords peuvent être en réalité le but principal de la fonction.

L'exemple le plus courant d'effet de bord est la **production d'un graphique**. Les fonctions graphiques ont un effet puisqu'elles créent ou ajoutent des éléments à un graphique. Cependant, certaines fonctions graphiques ne retournent pas d'objet.

```
test <- plot(rating ~ raises, data = attitude)
```




```
test
```

```
## NULL
```

Un autre exemple d'effet de bord est **l'écriture dans un fichier externe**. Par exemple, la fonction `write.table` ne retourne rien dans l'environnement de travail de la session R, mais enregistre des données dans un fichier externe, sur le disque de l'ordinateur.

Finalement toute interaction avec l'environnement de travail ou la session R autre que celle de créer un objet contenant la sortie de la fonction peut être considérée comme un effet de bord. Les fonctions suivantes sont toutes des exemples de fonctions ayant des effets de bord :

- `library` : charge un package, ce qui modifie le chemin de recherche de R ;
- `setwd` : modifie le répertoire de travail ;
- `options` : modifie les options de la session R ;
- `par` : modifie les paramètres graphiques ;
- etc.

Cependant, l'effet de bord d'une fonction R ne peut jamais être de modifier un objet de notre environnement de travail puisque R utilise la passage d'arguments par valeur et non par référence.

Exécution d'une fonction et environnements associés

Lorsqu'une fonction R est appelée, un environnement est créé spécifiquement pour l'évaluation du corps de la fonction, puis **détruit lorsque l'exécution est terminée**. Rappelons que l'évaluation est simplement la façon dont R s'y prend pour comprendre ce qu'une commande R signifie. Attardons-nous à comprendre comment R fait pour trouver la valeur d'un objet lorsqu'il évalue les instructions dans le corps d'une fonction.

Au départ, l'environnement créé lors de l'appel d'une fonction contient seulement des *promesses d'évaluation*, car R utilise une évaluation d'arguments dite *paresseuse*. Il évalue un argument seulement lorsqu'une instruction du corps de la fonction le fait intervenir pour une première fois. Ainsi, au fur et à mesure que les instructions constituant le corps de la fonction sont évaluées, les arguments de la fonction deviennent des objets dans l'environnement créé spécifiquement pour l'évaluation de la fonction.

Un objet associé à un argument donné en entrée lors de l'appel de la fonction est créé en évaluant la valeur qui lui a été attribuée. Pour créer les objets associés aux arguments auxquels aucune valeur n'a été fournie dans l'appel de la fonction, R évalue l'instruction fournie comme valeur par défaut dans la définition de la fonction.

Les instructions formant le corps de la fonction créent parfois de nouveaux objets. Ceux-ci sont créés dans l'environnement d'évaluation de la fonction. En informatique, ces objets sont appelés **variables locales**.

Portée lexicale

Trouver la valeur des arguments et des variables locales en cours d'évaluation d'une fonction est simple pour R. Ces objets se trouvent directement dans l'environnement d'évaluation de la fonction. On dit en informatique qu'ils ont une **portée locale**.

Mais comment R trouve-t-il la valeur des objets appelés à l'intérieur d'une fonction, qui ne sont ni des arguments ni des variables locales ?

Chaque langage de programmation suit une certaine règle pour résoudre ce problème. Les deux règles les plus courantes sont l'utilisation d'une **portée lexicale** (en anglais *lexical scoping*) ou encore d'une **portée dynamique** (en anglais *dynamic scoping*).

Avec une portée lexicale, si un objet appelé n'est pas trouvé dans l'environnement d'évaluation d'une fonction, le programme va le chercher dans l'environnement d'où la fonction a été **créée**, nommé **environnement englobant** (en anglais *enclosing environment*). Avec une portée dynamique, le programme va plutôt le chercher dans l'environnement d'où la fonction a été **appelée**, nommé **environnement d'appel** (en anglais *calling environment*).

R utilise par défaut la portée lexicale.

Voici un petit exemple pour illustrer la portée lexicale.

```
a <- 1
b <- 2
f <- function(x) {
  a*x + b
}
```

Quelle valeur sera retournée par `f(2)` ? Est-ce 1*2 + 2 = 4 ? Oui !

```
f(2)
```

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

Les objets nommés `a` et `b` ne se retrouvaient pas dans l'environnement d'exécution de la fonction. Alors R a cherché leurs valeurs dans l'environnement englobant de la fonction `f`, qui est ici l'environnement de travail.

```
environment(f)
```

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

Il a trouvé `a` = 1 et `b` = 2. La fonction `environment` retourne l'environnement englobant d'une fonction.

Modifions maintenant l'exemple comme suit.

```
g <- function(x) {
  a <- 2
  b <- 1
  f(x)
}
```

Quelle valeur sera retournée par `g(2)` ? Est-ce 2*2 + 1 = 5 ? Non !

```
g(2)
```

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

La fonction `g` est appelée dans l'environnement de travail. Elle appelle elle-même `f`. L'environnement d'appel de `f` est donc l'environnement d'exécution de `g`. Par contre, l'environnement englobant de `f` n'a pas changé. Il est encore l'environnement de travail, car c'est dans cet environnement que la fonction a été définie.

```
environment(f)
```

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

La portée lexicale permet de s'assurer que le fonctionnement de l'évaluation d'une fonction ne dépende pas du contexte dans lequel la fonction est appelée. Il dépend seulement de l'environnement d'où la fonction a été créée.

Si la portée en R était par défaut dynamique, `g(2)` aurait retourné la valeur 5.

Et si `f` était créée à l'intérieur de la fonction `g` ?

```
g<-function(x) {
  f<-function(x) {
    a*x + b
  }
  a <- 2
  b <- 1
  f(x)
}
```

Que retourne `g(2)` maintenant ?

```
g(2)
```

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

L'environnement englobant de `f` est maintenant l'environnement d'exécution de `g`, car `f` a été défini dans le corps de la fonction `g`.

Notons que l'environnement englobant des fonctions disponibles en R autres que celles que nous avons créées en cours de session est l'espace de noms du package d'où provient la fonction. Par exemple, l'environnement englobant de la fonction `mean` est l'espace de noms du package `base`. Nous verrons plus en détail ce qu'est un espace de noms plus tard.

```
environment(mean)
```

```
## <environment: namespace:base>
```

Chemin de recherche complet

Le chemin de recherche de valeurs des objets lors de l'évaluation d'une fonction en R ne s'arrête pas à l'environnement d'exécution de la fonction suivi de l'environnement englobant de la fonction. Il remonte toujours jusqu'à l'environnement de travail. Parfois, l'environnement englobant est directement l'environnement de travail. Si l'environnement englobant est plutôt l'environnement d'exécution d'une autre fonction, alors la recherche se poursuit dans l'environnement englobant de cette fonction. En remontant ainsi le chemin des environnements englobants, R finit toujours par retomber sur l'environnement de travail. Et de là, le chemin de recherche se poursuit par les environnements de tous les packages chargés, tel que vu dans les notes sur des informations techniques concernant R. Nous pouvons donc utiliser, dans les fonctions que nous créons, des fonctions provenant d'autres packages. Il faut seulement s'assurer que ces packages soient chargés pour que nos fonctions roulent sans erreur.

Bonnes pratiques concernant les objets utilisables dans le corps d'une fonction

Il est recommandé d'utiliser dans une fonction uniquement des objets que nous sommes certains de pouvoir atteindre. L'idéal est de se limiter aux arguments de la fonction, aux objets créés dans la fonction (variables locales) ainsi qu'aux objets se trouvant dans des packages *chargés*.

Ceux qui comprennent bien le concept de portée lexical peuvent aussi s'amuser à utiliser des objets dans l'environnement englobant d'une fonction.

Cependant, il est risqué d'utiliser les objets de l'environnement de travail, même si cet environnement se retrouve toujours dans le chemin de recherche de valeurs des objets lors de l'évaluation d'une fonction. Le contenu de l'environnement de travail est constamment modifié au fil de nos sessions. Aussi, si nous partageons nos fonctions avec une autre personne, nous ne contrôlons pas le contenu de l'environnement de travail pendant la session R de cette personne.

Ces recommandations s'appliquent au code dans le corps d'une fonction, mais aussi aux instructions définissant les valeurs par défaut des arguments. Nous avons appris que ces instructions sont évaluées dans le corps de la fonction. Elles peuvent donc contenir sans problème d'autres arguments de la fonction. Cependant, nous devrions éviter d'utiliser des objets provenant de l'environnement de travail dans ces instructions.

Exemple de création d'une fonction R

Nous allons créer ensemble une fonction qui compte combien de nombres entiers impairs contient un vecteur numérique. Cet exemple est tiré de

- Matloff, N. (2011). The Art of R Programming : A Tour of Statistical Software Design. No Starch Press. Sections 1.3 et 7.4.

Étapes de développement conseillées

1. **Planifier** le travail (pas de programmation encore) :
   - définir clairement la tâche à accomplir par la fonction et la sortie qu'elle doit produire,
   - prévoir les étapes à suivre afin d'effectuer cette tâche,
   - identifier les arguments devant être fournis en entrée à la fonction.
2. **Développer le corps de la fonction**
   - 2.1 Écrire le programme par étapes, d'abord sans former la fonction, en commentant bien le code et en travaillant sur des mini-données test.
   - 2.2 Pour chaque petite étape ou sous-tâche, tester interactivement si le programme produit le résultat escompté (tester souvent en cours de travail, ainsi il y a moins de débogage à faire).
3. **Créer la fonction** à partir du programme développé.
4. **Documenter** la fonction.

D'autres étapes de développement seront abordées au prochain cours.

1. Planifier le travail :

- entrée = un vecteur de nombres (= 1 seul argument)
- sortie = le dénombrement (une seule valeur)
- utiliser l'opérateur modulo pour tester si un nombre est impair
- nous pourrions travailler de façon vectorielle ou encore utiliser une boucle sur les éléments du vecteur

2. Développer le corps de la fonction :

Création de mini-données test

```
x <- c(6, 3, 5.5, 1, 0, -5)
```

Ce vecteur contient 3 nombres entiers impairs. C'est le résultat que nous visons obtenir.

Code le plus simple qui me vient en tête :

```
sum(x %% 2 == 1)
```

```
## [1] 3
```

Nous obtenons bien 3. Ça marche pour les mini-données test.

Ce code est équivalent à la boucle suivante :

```r
k <- 0
for (n in x){
  if (n %% 2 == 1) k <- k + 1
}
k
```

```
## [1] 3
```

3. Créer la fonction à partir du programme développé :

```r
compteImpair1 <- function(x) {
  sum(x %% 2 == 1)
}

compteImpair2 <- function(x) {
  k <- 0
  for (n in x){
    if (n %% 2 == 1) k <- k + 1
  }
  k
}
```

4. Documenter la fonction.

Option 1 : Documentation en commentaire dans le corps de la fonction.

```r
compteImpair1 <- function(x) {
  # Fonction qui compte combien de nombres entiers impairs contient un vecteur numérique
  # Argument en entrée : x = vecteur numérique
  # Sortie : le nombre de nombres entiers impairs dans x
  sum(x %% 2 == 1)
}
```

Option 2 : Documentation en commentaire avant la définition de la fonction.

```r
# Fonction qui compte combien de nombres entiers impairs contient un vecteur numérique
# Argument en entrée : x = vecteur numérique
# Sortie : le nombre de nombres entiers impairs dans x
compteImpair2 <- function(x) {
  k <- 0
  for (n in x){
    if (n %% 2 == 1) k <- k + 1
  }
  k
}
```

Options supplémentaires : Nous verrons d'autres options dans le cours sur les packages.

Comparaison des 2 fonctions :

Nous avons créé 2 fonctions qui, à première vue, retournent toutes les deux le résultat escompté. Nous devrions par contre les tester sur plus de données pour en être certains. Ce sera fait dans les notes sur les tests et exceptions en R. Pour l'instant, tenons pour acquis que ces fonctions accomplissent correctement leur tâche.

Dans ce cas, laquelle des 2 fonctions devrions-nous utiliser ?

Réponse : la plus rapide.

Créons un vecteur très grand pour comparer le temps d'exécution des deux fonctions.

```
x <- round(runif(1000000, -10, 10))
```

Utilisons la fonction `system.time` pour évaluer les temps d'exécution.

```
system.time(compteImpair1(x))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

```
system.time(compteImpair2(x))
```

```
##    user  system elapsed
##    0.28    0.00    0.28
```

L'écart dans les temps d'exécution des deux fonctions se creuse encore plus si nous augmentons la longueur du vecteur `x`.

Nous devrions donc choisir d'utiliser `compteImpair1` plutôt que `compteImpair2`.

Nous allons revenir plus tard sur l'optimisation des temps d'exécution de nos fonctions.

Programmation fonctionnelle

Maintenant que vous savez écrire des fonctions en R, vous pouvez exploiter tout le potentiel de la programmation fonctionnelle, paradigme de programmation exploité par R. En fait, nous avons déjà parlé de ce paradigme dans ce cours. L'utilisation de fonctions de la famille des `apply` est une forme de programmation fonctionnelle. Contentons nous ici de parler de cet aspect de la programmation fonctionnelle : les fonctions de haut niveau qui prennent d'autres fonctions en entrées, comme les fonctions de la famille des `apply`.

Nous avons déjà donné dans les notes sur les structures de contrôle un exemple de boucle `for` remplacé par un appel à une fonction de la famille des `apply`. En fait, pratiquement n'importe quelle boucle `for` en R peut être remplacée par un appel à une fonction de la famille des `apply` une fois que nous savons comment créer de nouvelles fonctions. Par exemple, reprenons l'exemple de boucle suivant, aussi tiré des notes sur les structures de contrôle.

```
modeles <- vector(length = 6, mode = "list")
names(modeles) <- names(attitude)[-1]

for (variable in names(modeles)) {
  modeles[[variable]] <- lm(rating ~ ., data = attitude[, c("rating", variable)])
}
```

Il s'agit d'une boucle ajustant plusieurs modèles de régression linéaire simple avec les variables du jeu de données `attitude`. Modifions un peu cet exemple pour conserver des modèles uniquement les coefficients de détermination, non ajustés et ajustés.

```
R2 <- matrix(NA, nrow = 2, ncol = 6)
colnames(R2) <- names(attitude)[-1]
rownames(R2) <- c("r.squared", "adj.r.squared")

for (variable in colnames(R2)) {
  reg <- lm(rating ~ ., data = attitude[, c("rating", variable)])
  R2["r.squared", variable] <- summary(reg)$r.squared
  R2["adj.r.squared", variable] <- summary(reg)$adj.r.squared
}

R2
```

```
##               complaints privileges  learning    raises    critical     advance
## r.squared      0.6813142  0.1815756 0.3889745 0.3482640  0.02447321  0.02405175
## adj.r.squared  0.6699325  0.1523461 0.3671521 0.3249877 -0.01036703 -0.01080355
```

À l'aide de la bonne fonction, nous allons obtenir exactement le même résultat avec un code plus court, qui n'utilise pas de boucle, mais qui utilise la fonction `sapply`. Voici une fonction qui extrait des deux statistiques à conserver pour un seul modèle de régression.

```
R2_reg_rating_vs_var <- function(variable){
  reg <- lm(rating ~ ., data = attitude[, c("rating", variable)])
  c(r.squared = summary(reg)$r.squared,
    adj.r.squared = summary(reg)$adj.r.squared)
}
```

Cette fonction prend en entrée le nom d'une variable provenant de `attitude`, mais autre que la variable réponse `rating`. Nous pouvons itérer sur tous les noms de variables possibles comme suit :

```
sapply(X = names(attitude)[-1], FUN = R2_reg_rating_vs_var)
```

```
##               complaints privileges  learning    raises    critical     advance
## r.squared      0.6813142  0.1815756 0.3889745 0.3482640  0.02447321  0.02405175
## adj.r.squared  0.6699325  0.1523461 0.3671521 0.3249877 -0.01036703 -0.01080355
```

Un des avantages de l'utilisation de fonctions de la famille des `apply` en remplacement de boucles est que nous n'avons pas à initialiser préalablement un objet pour stocker les résultats. La fonction de haut-niveau gère cet aspect pour nous. Les adeptes de la programmation fonctionnelle sont également d'avis que l'utilisation de fonction de la famille des `apply` produit un code plus clair qu'une boucle.

Il existe en R d'autres fonctions de haut-niveau, qui appliquent itérativement une fonction sur les éléments d'un objet. Pour les intéressés, la fiche d'aide ouverte par la commande `help(funprog)` présente ces fonctions (`Map`, `Filter`, `Reduce`, etc.). Un package R se spécialise aussi dans ce genre de fonction, il s'agit du package `purrr`.

Synthèse

Syntaxe générale d'une fonction

```
nomFonction <- function(arg1, arg2, arg3){
  instructions  # formant le corps de la fonction
}
```

Les composantes d'une fonction R sont :

- la liste de ses arguments, possiblement avec des valeurs par défaut ;
- le corps de la fonction, soit le code qui la constitue ;
- l'environnement englobant de la fonction.

Note : Il n'est pas obligatoire pour une fonction de porter un nom, ni de posséder des arguments.

Valeurs par défaut des arguments

Les valeurs par défaut sont **définies dans la liste des arguments**, en accompagnant le nom d'un argument d'un **opérateur =** et d'une **instruction R retournant la valeur par défaut**.

Cas particulier : argument qui prend en entrée **une seule chaîne de caractères** et que seulement un **petit nombre de chaînes de caractères distinctes sont acceptées** comme valeur de cet argument :

- valeur à droite de l'opérateur `=` dans la liste des arguments : vecteur de toutes les valeurs acceptées,
- valeur par défaut : élément en position 1 dans le vecteur de toutes les valeurs acceptées,
- dans le corps de la fonction : `arg <- match.arg(arg)`.

Appel d'une fonction

L'association des arguments à leurs valeurs se fait en respectant les règles de préséances suivantes :

1. d'abord les arguments fournis avec un nom exact se voient attribuer une valeur,
2. puis les arguments fournis avec un nom partiel,
3. et finalement les arguments non nommés, selon leurs positions.

Bonne pratique :

- utiliser l'association par positionnement seulement pour les premiers arguments,
- arguments moins communs nommés (code plus facile à comprendre).

Passage des arguments par valeur

Les objets assignés à des arguments dans un appel à une fonction R sont recopiés et l'évaluation de la fonction affecte ces copies. Elle n'affecte jamais les objets d'origine.

Argument ...

Nous pouvons insérer l'argument `...` dans la liste des arguments des fonctions que nous créons.

Dans le corps de la fonction, le traitement de cet argument dépend de son utilité.

- Pour **prendre un nombre indéterminé d'objets en entrée** :
  - Le corps de la fonction doit contenir une instruction telle que `list(...)` pour récupérer tous les objets.
- Pour **permettre le passage d'arguments** à une autre fonction :
  - Dans le corps de la fonction, les appels à la ou aux fonctions auxquelles nous souhaitons permettre le passage d'arguments doivent contenir l'argument `...`.

Sortie

Une fonction retourne :

- l'objet donné en argument à la fonction `return` dans le corps de la fonction,
- ou, en l'absence d'appel à la fonction `return`, la dernière expression évaluée dans le corps de la fonction.

Une fonction **ne peut retourner qu'un seul objet**. Pour retourner plusieurs objets, il faut les combiner dans un seul objet (typiquement dans une liste).

Exécution d'une fonction

Lorsqu'une fonction R est appelée, un environnement est créé spécifiquement pour l'évaluation du corps de la fonction, puis détruit lorsque l'exécution est terminée.

$\longrightarrow$ **environnement d'exécution** (ou d'évaluation) = temporaire

Cet environnement contient :

- des objets pour les arguments fournis dans l'appel ;
- des objets pour les arguments non fournis dans l'appel, qui prennent leur valeur par défaut ;
- des objets créés dans les instructions du corps de la fonction (variables locales).

Évaluation paresseuse : Les objets associés aux arguments sont créés uniquement lorsqu'une instruction du corps de la fonction les faisant intervenir doit être évaluée. À sa création, l'environnement d'exécution contient uniquement des promesses d'évaluation.

Portée lexicale

> Comment R trouve-t-il la valeur des objets appelés dans les instructions du corps d'une fonction qui ne sont ni des arguments ni des variables locales ?

Il les cherche dans l'**environnement englobant** de la fonction = *environnement dans lequel la fonction a été créée.*

> **R utilise par défaut la portée lexicale.**

À ne pas confondre : il ne cherche pas dans l'**environnement d'appel** = *environnement dans lequel la fonction est appelée* (à moins que l'environnement englobant soit le même que l'environnement d'appel).

Chemin de recherche complet lors de l'exécution d'une fonction

- environnement d'exécution,
- chemin des environnements englobants jusqu'à :
- environnement de travail,
- environnements des packages chargés.

Bonne pratique : Utiliser dans une fonction uniquement des objets que nous sommes certains de pouvoir atteindre, soit

- les arguments de la fonction,
- les objets créés dans la fonction (variables locales),
- les objets se trouvant dans des packages chargés,
- les objets dans l'environnement englobant (si nous comprenons bien le concept de portée lexicale).

Ne pas utiliser les objets de l'environnement de travail, car le contenu de cet environnement est constamment modifié.

Références

- Matloff, N. (2011). The Art of R Programming : A Tour of Statistical Software Design. No Starch Press. Chapitre 7.
- http://adv-r.had.co.nz/Functions.html
- http://adv-r.had.co.nz/Environments.html
- https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial
- Passage d'arguments par valeur versus par référence : http://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php
- Programmation fonctionnelle :
  - http://adv-r.had.co.nz/Functional-programming.html
  - http://adv-r.had.co.nz/Functionals.html
  - https://r4ds.had.co.nz/iteration.html#for-loops-vs.functionals
  - https://purrr.tidyverse.org/