Cours 1 : Les fonctions

Encore plus avancé

Concepts plus avancés censurés de la présentation pour ne pas faire trop peur !

# Closures

Le résultat d'une fonction est un objet. Une fonction est un objet comme un autre.

Une fonction peut donc retourner une autre fonction : ce sont les **closures**.

f <- function(x){  
 g <- function(y){  
 x^y  
 }  
}

carre <- f(2)  
carre(2)

## [1] 4

cube <- f(3)  
cube(2)

## [1] 9

Un exemple plus appliqué : un rééchantilloneur.

create\_resampler <- function(x) {  
 function(n) sample(x, size = n, replace = TRUE)  
}  
rs1 <- create\_resampler(1:10)  
rs1(2)

## [1] 7 2

rs1(9)

## [1] 4 7 4 6 1 6 9 4 2

rs2 <- create\_resampler(c("M", "F"))  
rs2(5)

## [1] "F" "M" "F" "M" "F"

Ce rééchantilloneur a un problème potentiel causé par la *lazy evaluation* : x n'est pas évalué à l'execution de create\_resampler() car il n'est pas nécessaire au calcul. Il n'est évalué que à l'execution de rs1(). Et s'il a changé entre temps, c'est un problème...

ma\_distribution <- 1:10  
rs <- create\_resampler(ma\_distribution)  
  
ma\_distribution <- c("a", "b", "c")  
rs(5)

## [1] "b" "c" "a" "c" "c"

Il faut donc *forcer* l'évalution de x dès l'appel de create\_resampler(). En pratique il suffit d'ajouter une ligne arbitraire dans create\_resampler() qui nécessite l'évaluation de x. Mais pour plus de lisibilité on peut aussi utiliser la fonction force().

create\_resampler <- function(x) {  
 x # pour évaluer x, si on veut être plus lisible: force(x)  
 function(n) sample(x, size = n, replace = TRUE)  
}  
  
ma\_distribution <- 1:10  
rs <- create\_resampler(ma\_distribution)  
  
ma\_distribution <- c("a", "b", "c")  
rs(5)

## [1] 7 5 4 9 8

# Pour combiner les concepts

Closure + *enclosing environment* + scoping. On fait un compteur.

Pour qu'une fonction puisse stocker une valeur il faut qu'elle la garde dans un environnement :

* Permanent, non détruit entre deux appels de la fonction (donc pas dans l'*execution environment*) ;
* Et accessible facilement par la fonction.

L'*enclosing environment* correspond bien à ces deux critères. Le seul problème est que si lire une variable dans l'*enclosing environment* est facile (il suffit de l'appeler alors qu'elle n'est pas définie dans l'*execution environment*), écrire est par contre plus difficile.

C'est là qu'on introduit <<-, la *super-assignation*. Contrairement à <-, l'assignation classique, qui agit ainsi :

* x <- 2 ;
* Si x existe dans l'environnement, remplacer sa valeur par 2 ;
* Sinon créer x dans l'environnement.

<<- agit ainsi :

* x <<- 2 ;
* Si x existe dans l'environnement, remplacer sa valeur par 2 ;
* Sinon remonter le *searchpath* jusqu'à trouver une variable nommée x, et remplacer sa valeur.

On voit donc que comme le premier élément sur le *searchpath* après l'*execution environment* est l'*enclosing environment*, il suffit d'y créer à priori une variable x, puis de la modifier avec <<-.

create\_counter <- function() {  
 n <- 0  
 function() {  
 n <<- n + 1  
 return(n)  
 }  
}  
c1 <- create\_counter()  
c1()

## [1] 1

c1()

## [1] 2

Le compte n'est pas partagé entre les compteurs car si on appelle à nouveau create\_counter(), un second *execution environment* de create\_counter() indépendant est créé, avec un nouvel n, initialisé à 0.

c2 <- create\_counter()  
c2()

## [1] 1

c1()

## [1] 3

Il faut stocker n dans un environement partagé par tous les compteurs créés.

* Les compteurs ont comme *enclosing environment* l'*execution environment* de l'appel spécifique de create\_counter() qui les a créé ;
* Par contre tous les appels de create\_counter() ont le même *enclosing environment* ;
* C'est donc là qu'il faut stocker n : dans l'*enclosing environment* de create\_counter().

On voit que là pour trouver n, notre compteur doit remonter de deux positions dans le *searchpath* : d'abord dans l'*enclosing environment* du compteur, qui est l'*execution environment* de create\_counter(), puis dans l'*enclosing environment* de create\_counter().

Solution paresseuse et dangereuse : stocker n dans l'environnement global. C'est embêtant car n est trop facilement accessible et modifiable par n'importe quelle autre fonction.

n <- 0  
create\_counter <- function() {  
 function() {  
 n <<- n + 1  
 return(n)  
 }  
}  
  
c1 <- create\_counter()  
c1()

## [1] 1

c2 <- create\_counter()  
c2()

## [1] 2

c1()

## [1] 3

Meilleure solution : créer un nouvel environnement juste pour ça, définir create\_counter() dedans pour que ça devienne son *enclosing environment*, et y stocker n. Seuls les compteurs auront accès à n.

# local() permet de créer un nouvel environment  
create\_counter <- local({  
 n <- 0  
 function() {  
 function() {  
 n <<- n + 1  
 return(n)  
 }  
 }  
})  
c1 <- create\_counter()  
c1()

## [1] 1

c2 <- create\_counter()  
c2()

## [1] 2

c1()

## [1] 3