S08: Programmation

Analyse de données quantitatives avec R

Samuel Coavoux

Les fonctions

Principe des fonctions

Lorsque l'on souhaite répéter une opération sur plusieurs objets, il est conseillé d'employer une fonction. Un certain nombre de fonctions sont déjà présentes dans R et dans ses packages ; il est possible d'en créer de nouvelles.

Cela permet d'éviter de dupliquer du code, de s'assurer que la même opération est bien réalisée sur tous les objets, de changer l'ensemble des opérations d'un seul coup (en changeant la définition de la fonction), et enfin d'automatiser une opération sur un nombre arbitraire d'objets.

Syntaxe

Une fonction se définit avec `function()`. Elle a:

- un nom
- des arguments
- un résultat (déclaré avec `return()`)

```
nom_fonction <- function(arg1, arg2 = "default"){
  res <- paste(arg1, arg2)
  return(res)
}
```

Nom de fonction

Le nom de la fonction doit obéir aux mêmes règles que les noms d'objets en général:

- pas d'espace ; préférer le _ au . pour séparer des mots
- ne pas commencer par un chiffre
- éviter les caractères accentués

Arguments

Les arguments sont déclarés dans `function(...)`. Ils prennent des noms arbitraires. On peut spécifier des valeurs par défaut de ces arguments ; dans ce cas, si l'argument n'est pas définit dans l'appel de la fonction, il prendra la fonction par détaut.

```
nom_fonction <- function(arg1, arg2 = "default"){...}
```

Traditionnellement, les arguments nécessaires à la fonction (sans défaut) arrivent en premier, suivi d'autres arguments par ordre décroissant d'importance.

Les arguments sont ensuite définis lorsque l'on appelle la fonction.

```
nom_fonction(arg1 = "Mon argument 1")
nom_fonction(arg1 = "Mon argument 1", arg2 = "Mon argument
nom_fonction("Mon argument 1", "Mon argument 2")
```

Corps de la fonction

Chaque fonction produit un nouvel *environement*. Cela signifie que *les objets créé dans la fonction ne seront pas accessibles en dehors*. Ils sont créés et manipulés uniquement à chaque fois que la fonction est employée. On peut donc créer de nombreux objets dans la fonction.

Par défaut, le dernier objet créé est retourné par la fonction ; il est préférable de préciser le résultat en utilisant `return()`.

```
ecart_type <- function(x){
  m <- mean(x, na.rm = TRUE)
  m_ecart_sq <- (x - m)^2
  variance <- sum(m_ecart_sq)
  ecart_type <- sqrt(variance)
  return(ecart_type)
}
```

Control flow

Principe

On entend ici par control flow l'ensemble des fonctions qui permettent de contrôler l'exécution d'un programme. Il s'agit de faire en sorte que le script et/ou les fonctions que l'on écrit soient capable de savoir par eux-mêmes ce qu'il faut faire lorsqu'ils sont confrontés à certaines situations.

Usages que l'on en fait:

- dans une fonction, déterminer des comportements alternatifs selon le type d'arguments employés, ou selon le résultat (que faire s'il y a des NA ? que faire s'il y a deux arguments au lieu d'un?)
- dans un programme de scraping, permet de déterminer les comportements d'une fonction selon le contenu des pages récupérés : que faire si une page n'est pas structurée comme les autres ?

Exemple If... else...

Pour l'aide, regarder `?Control`

```
var_covar <- function(x, y=NULL){
  if(is.null(y)) {
    return(var(x))
  } else {
    return(cov(x, y))
  }
}
a <- c(5, 7, 10, 12, 15)
b <- c(11, 8, 5, 6, 4)
var_covar(x = a)
```

```
## [1] 15.7
```

```
var_covar(x = a, y =b)
```

Fonctionnement : if

`if` prend comme argument *un vecteur logique de taille 1*. Pour rappel, un vecteur logique est un vecteur composé de `TRUE` et de `FALSE` (et de `NA`). L'opération suivant `if` sera executée si le vecteur est `TRUE`, pas executée sinon. Le plus souvent, on emploie if avec des fonctions ou des opérateurs de comparaison:

```
if(is.numeric(x))
if(!is.numeric(x))
if(x > 10)
if(length(x) == 10)
```

Comparaisons

Pour rappel, on peut construire des vecteurs logiques à partir de ces opérateurs de comparaison:

$<$, $>$, $==$, $<=$, $>=$

et l'on peut combiner des vecteurs logiques avec :

&, |

Fonctionnement : else

`else` est nécessairement employé après `if` ; l'opération qui suit `else` est executée seulement si celle qui suivait le `if` précédent ne l'a pas été.

```
if(is.numeric(x)){
   ...
} else {
   ...
}
```

Par convention, on place le plus souvent else sur la même ligne que l'accolade fermant le `if` précédent

Enchaînement

On peut enchaîner les `if` et `else`:

```
if(x < 0) {
   ...
} else if(x >= 0 & x < 10){
   ...
} else {
   ...
}
```

Loops

Définition

Dans un langage de programmation, les loops sont des constructions syntaxiques qui permettent d'appliquer une même opération sur une série d'objets. La plupart des langage disposent de différents types de boucle:

- boucle `for`: appliquer **pour** tous les objets répondant à une condition
- boucle `while`: appliquer **tant que** une condition est respectée

R et la vectorisation

R est un langage particulier, dans lequel la plupart des opérations qui nécessitent une boucle dans d'autres langage n'en ont pas besoin, grâce à la **vectorisation**. Cela signifie que les fonctions, par défaut, s'appliquent à des suites d'objet, les vecteurs. Ainsi:

```
x <- 1:5
y <- 31:35
x + y
```

```
## [1] 32 34 36 38 40
```

R comprend qu'il faut ajouter chaque élément de x à l'élément correspondant dans y. Dans un autre langage, il aurait fallu faire:

```
for(i in 1:5) {x[i] + y[i]}
```

R et les boucles

R **n'aime pas les boucles** pour de nombreuses raisons, mais elles sont nécessaires lors du scraping. On apprend donc à en faire. N'en faites pas si vous pouvez vous en passez.

Pour en savoir plus, lire le chapitre "Vectorization" dans *The R inferno*.

Deux types de boucles

Il existe deux manières de faire des boucles en R:

- la famille de fonction `apply`, dont les principaux représentants sont `apply`, `lapply`, et `sapply`
- les boucles classiques, `for` ou `while`

Apply

`apply` est une famille de fonctions qui permettent d'appliquer une fonction à une série d'objet. Nous utiliserons principalement `lapply` (list-apply) et `sapply` (simple-apply).

```
sapply(c(2, 3), function(x) x^2)
```

```
## [1] 4 9
```

```
lapply(c(2, 3), function(x) x^2)
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 9
```

Les boucles classiques

```
for(i in c(2, 3)) {
  print(i^2)
}
```

```
## [1] 4
## [1] 9
```

```
i = 2
while(i < 4) {
  print(i^2)
  i = i + 1
}
```

```
## [1] 4
## [1] 9
```

Ces exemples sont illustratifs: en réalité, pour trouver le carré de 2 et de 3, il est bien plus efficace d'utiliser une fonction vectorisée `c(2, 3)^2`.

Boucle while: attention

Si la condition d'une boucle `while` est toujours vraie, le programme n'en sortira jamais...

```
while(TRUE) print("je suis en train de planter l'ordinateu
```

Comment choisir quelle boucle utiliser?

Même efficacité en matière de temps de calcul.

Intérêt de `apply`:

- **ne crée pas d'objets nouveaux dans l'environnement**
- plus compact
- fonctionne bien avec les listes/data.frame

Intérêt de for/while:

- **si la boucle s'arrête, ce qui a été réalisé reste en mémoire**
- plus proche d'autres langages de programmation, plus lisible

La plupart du temps, privilégier boucles `apply` pour leur économie (pas créer d'objet).