Programmation modulaire sous R
Procédures et fonctions
Découpage en modules des applications

Ricco Rakotomalala

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

Découpage des programmes

PROCÉDURES ET FONCTIONS

Pourquoi créer des fonctions ?

1. Meilleure organisation du programme (regrouper les tâches par blocs : lisibilité → maintenance)
2. Eviter la redondance (pas de copier/coller → maintenance)
3. Possibilité de partager /diffuser les fonctions (via des modules)

Qu'est-ce qu'un module sous R ?

1. Module = fichier « .r »
2. On peut regrouper dans un module les fonctions traitant des problèmes de même nature ou manipulant le même type d'objet
3. Pour charger les fonctions d'un module dans un autre module / programme principal, on utilise la commande **source**(nom du fichier module.r)
4. Les fonctions importées sont chargées en mémoire. Si collision de noms, les plus récentes écrasent les anciennes.

Expression

Expression = instruction

Ex. a <- 10 + 2

Bloc d'expressions

Suite d'instructions délimitées par { et }

```
bloc <- {
a <-  10
b <- 2
w <- a + b
}
```

- On peut nommer un bloc d'expressions
- Un bloc d'expressions renvoie toujours une valeur, celle de la dernière expression exécutée c.-à-d. 12 pour notre ex.
- On aurait mis a + b simplement, le bloc renvoie aussi la valeur 12

Appel d'un bloc d'expressions

```
d <- bloc
print(d)
```

- Affiche la valeur 12
- print(bloc) renvoie 12 également

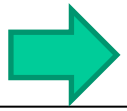
Fonction

- Fonction = Bloc d'expressions, forcément nommée
- Visible comme un objet en mémoire [avec ls()]
- Peut être supprimée avec rm(), comme tout autre objet
- Prend des paramètres en entrée (non typés)
- Renvoie une valeur en sortie (même si on n'écrit pas spécifiquement d'instruction pour cela)

Un exemple

```
petit <- function (a, b){
d <- ifelse(a < b, a, 0)
return(d)
}
```

- Le nom de la fonction est « petit »
- Les paramètres ne sont pas typés
- return() renvoie la valeur
- Si on omet return(), la fonction renvoie quand même une valeur : la dernière calculée
- return() provoque immédiatement la sortie de la fonction



Procédure = Fonction sans return(), mais renvoie quand même une valeur !!!

Passage de paramètres par position

```
print(petit(10, 12))
```

Passer les paramètres selon les positions attendues
La fonction renvoie 10

Passage par nom. Le mode de passage que je préconise, d'autant plus que les paramètres ne sont pas typés.

```
print(petit(a=10,b=12))
```

Aucune confusion possible → 10

```
print(petit(b=12, a=10))
```

Aucune confusion possible → 10

En revanche…

```
print(petit(12, 10))
```

Sans instructions spécifiques, le passage par position prévaut
La fonction renvoie → **0**

Paramètres par défaut

- Affecter des valeurs aux paramètres dès la définition de la fonction
- Si l'utilisateur ne spécifie par le paramètre lors de l'appel, cette valeur est utilisée
- Si l'utilisateur spécifie une valeur, c'est bien cette dernière qui est utilisée

Exemple

```
ecart <- function (a, b, epsilon = 0.1){
d <- abs(a-b)
res <- ifelse(d < epsilon, 0, d)
return(res)
}

ecart(a=12.2, b=11.9, epsilon = 1) #renvoie 0
ecart(a=12.2, b=11.9) #renvoie 0.3
```

La valeur utilisée est epsilon = 0.1 dans ce cas

Portée des variables, imbrications des fonctions, …

PLUS LOIN AVEC LES FONCTIONS

Variables locales et globales

1. Les variables définies localement dans les fonctions sont uniquement visibles dans ces fonctions. Elles sont absentes de la mémoire de R après l'exécution de la fonction.
2. Les variables définies (dans la mémoire globale) en dehors de la fonction sont accessibles dans la fonction.
3. Il n'est pas nécessaire que la variable globale soit disponible lors de l'implémentation de la fonction, il faut en revanche qu'elle soit disponible lors de l'exécution de la fonction.

```
#définition (implémentation) de le fonction
produit ← function(b){
d ← a * b #on peut utiliser la variable a même si elle est définie nulle part
return (d)
}
#appel
a ← 10 #a est définie dans la mémoire globale, en dehors de la fonction
z ← produit(5) #produit utilise a trouvée dans la mémoire globale
print(z) #renvoie bien la valeur 50 !!!
ls() #renvoie les objets « a », « z » et « produit » ; pas « d », ni « b » bien sûr
```

Fonctions locales et globales

Il est possible de déclarer et définir une fonction dans une autre fonction. Dans ce cas, elle est locale à la fonction, elle n'est pas visible comme objet dans la mémoire globale.

```
#définir une fonction globale
ecart <- function(a,b){
 epsilon <- 0.1

 #définir une fonction locale
 difference <- function (a,b){
  d <- abs(a-b)
  resultat <- ifelse(d < epsilon, 0, ifelse(a < b,-d,d))
  return (resultat)
  }

 #faire appel à la fonction locale
 e <- difference(a,b)
 return (e)
 }
#appel
z <- ecart(10.95,11)
print(z)
#affichage du contenu de la mémoire
ls()  # ➔ « z », « ecart » sont visibles ; pas « epsilon », « difference », et encore moins « d » et « resultat »
```

Utilisation des listes

A priori, une fonction ne sait pas renvoyer plusieurs valeurs. Nous sommes obligés d'utiliser un subterfuge en mettant à contribution une structure de liste d'objets. Les objets peuvent être de classe différente, au final l'outil est très souple.

(on verra plus en détail les listes plus loin)

```
extreme <- function(a,b){
            if (a < b) {
            return (list(mini=a,maxi=b))
            } else
            {
            return (list(mini=b,maxi = a))
            }
}
#appel de la fonction
z <- extreme(10,12)
#accès aux champs de l'objet résultat (qui est une liste)
print(z$mini) #renvoie 10 ; on peut faire aussi z[[1]]
print(z$maxi) #renvoie 12 ; idem, on la même valeur avec z[[2]]
plage <- z$maxi – z$mini #on peut l'utiliser dans les calculs
print(plage) #renvoie 2
```

```r
#met au carré une variable
CARRE <- function(x){
  return(x^2)
}

#met au cube
CUBE <- function(x){
  return(x^3)
}

#somme de 1 à n
#avec une FONCTION générique
somme <- function(FONCTION,n){
  s <- 0
  for (i in 1:n){
    s <- s + FONCTION(i)
  }
  return(s)
}

#appel avec carré
print(somme(CARRE,10)) # résultat = 385

#appel avec cube
print(somme(CUBE,10)) # résultat = 3025
```

Fonction en paramètre d'une autre fonction

Tout objet peut être paramètre d'une fonction, y compris une autre fonction. On parle alors de « **fonction callback** » (fonction de rappel en français).

R n'effectue le contrôle qu'à l'exécution. Il appelle la fonction callback pourvu que son prototype (son en-tête) est licite.

Ca peut être carré ou cube… ou n'importe quel autre fonction qui ne prend qu'un paramètre en entrée.

Passage d'une fonction en paramètres + Paramètres ouverts

```r
#met au carré une variable
CARRE <- function(x){
  return(x^2)
}

#met au cube
CUBE <- function(x){
  return(x^3)
}

#met à la puissance
PUISSANCE <- function(x,k){
  return(x^k)
}

#somme de 1 à n avec une FONCTION
somme <- function(FONCTION,n,...){
  s <- 0
  for (i in 1:n){
    s <- s + FONCTION(i,...)
  }
  return(s)
}

#appel avec carre
print(somme(CARRE,10))

#appel avec cube
print(somme(CUBE,10))

#appel avec puissance
print(somme(PUISSANCE,10,k=0.5))
```

Fonction en paramètre d'une autre fonction, sans délimiter à l'avance son prototype

La fonction callback peut prendre des paramètres supplémentaires qu'il faut prévoir.

Le … indique qu'il peut y avoir éventuellement des paramètres supplémentaires à transmettre à la fonction callback.

CARRE et CUBE qui n'en ont pas n'en tiennent pas compte.

Ce n'est pas obligatoire, mais on a intérêt à nommer le paramètre lors de l'appel pour éviter les ambigüités.

Conclusion - R est magique

De la documentation à profusion (**n'achetez jamais des livres sur R**)

Site du cours

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

Programmation R

http://www.duclert.org/

Quick-R

http://www.statmethods.net/

POLLS (Kdnuggets)

Data Mining / Analytics Tools Used

(R, 2nd ou 1er depuis 2010)

What languages you used for data mining / data analysis?

http://www.kdnuggets.com/polls/2013/languages-analytics-data-mining-data-science.html

(Août 2013, langage R en 1ère position)

Article New York Times (Janvier 2009)

"**Data Analysts Captivated by R's Power**" - http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=1