Programmation en langage R

Résumé

Un aperçu de la syntaxe du langage S mis en œuvre dans R : fonctions, instructions de contrôle et d'itérations, fonction `apply`.
Organisation des tutoriels R.

- *Démarrer rapidement avec R*
- *Initiation à R*
- *Fonctions graphiques de R*
- *Programmation en R*
- *MapReduce pour le statisticien*

Les aspect statistiques sont développés dans les différents scénarios de Wikistat.

1 introduction

R est la version *GNU* du langage S conçu initialement aux Bell labs par John Chambers à partir de 1975 dans une syntaxe très proche du langage C. En septembre 2013, l'index TIOBE le classe en 18ème position loin derrière le C (1er) ou Java (2ème) mais devant MATLAB (19) ou SAS (21).

2 Structure de contrôle

Il est important d'intégrer que R, comme Matlab, est un langage interprété donc lent, voire très lent, losqu'il s'agit d'exécuter des boucles. Celles-ci doivent être éviter dès qu'une syntaxe, impliquant des calculs matriciels ou les commandes de type `apply`, peut se substituer.

2.1 Structures conditionnelles

`if(condition){instructions}` est la syntaxe permettant de calculer les instructions uniquement si la condition est vraie.

`if(condition){ A }else{ B }` calcule les instructions A si la condition est vraie et les instructions B sinon. Dans l'exemple suivant, les deux commandes sont équivalentes :

```
if (x>0) y=x*log(x) else y=0
y=ifelse(x>0,x*log(x),0)
```

2.2 Structures itératives

Ces commandes définissent des boucles pour exécuter plusieurs fois une instruction ou un bloc d'instructions. Les trois types de boucle sont :

```
for var in seq) {commandes}
```

```
while (condition) {commandes}
```

```
repeat {commandes; if (condition) break }
```

Dans une boucle `for`, le nombre d'itérations est fixe alors qu'il peut être infini pour les boucles `while` et `repeat` ! La condition est évaluée avant toute exécution dans `while` alors que `repeat` exécute au moins une fois les commandes.

```
for (i in 1:10) print(i)
y=z=0;
for (i in 1:10) {
   x=runif(1)
   if (x>0.5) y=y+1
   else z=z+1 }
y;z
for (i in c(2,4,5,8)) print(i)
x = rnorm(100)
y = ifelse(x>0, 1, -1) # condition
y;i=0
while (i<10){
   print(i)
   i=i+1}
```

Questions

1. Que pensez-vous de :
   `for (i in 1:length(b)) a[i]=cos(b[i])`
2. Obtenir l'équivalent de y et z dans la deuxième boucle `for` sans boucle.

3. Dans l'enchaînement de commandes ci-dessous, supprimer d'abord la boucle `for` sur `j` puis les 2 boucles.

```
M=matrix(1:20,nr=5,nc=4)
res=rep(0,5)
for (i in 1:5){
   tmp=0
   for (j in 1:4) {tmp = tmp + M[i,j]}
res[i]=tmp}
```

Réponses

1. Cette boucle est inutile. Il suffit de saisir `a=cos(b)`. L'élément de base de R est la matrice dont le vecteur est un cas particulier.
2. Une solution consiste à sommer les éléments `TRUE` d'un vecteur logique `x=runif(10);y=sum(x>0.5);z=10-y`
3. Suppression de boucles
   - Boucle `for` sur `j` : `for (i in 1:5) res[i]=sum(M[i,])`
   - Les 2 boucles : `res=apply(M,1,sum)`

3 Fonctions

3.1 Principes

Il est possible sous R de construire ses propres fonctions. Il est conseillé d'écrire sa fonction dans un fichier `nomfonction.R`.

`source("nomfonction.R")` a pour effet de charger la fonction dans l'environnempent de travail. Il est aussi possible de définir directement la fonction par la syntaxe suivante :

```
nomfonction=function(arg1[=exp1],arg2[=exp2],...)
    {
    bloc d'instructions
    sortie = ...
    return(sortie)
    }
```

Les accolades signalent le début et la fin du code source de la fonction, les crochets indiquent le caractère facultatif des valeurs par défaut des arguments. L'objet sortie contient le ou les résultats retournés par la fonction, on peut en particulier utiliser une liste pour retourner plusieurs résultats.

3.2 Exemples

Création d'une fonction élémentaire.

```
MaFonction=function(x){x+2}
ls()
MaFonction
MaFonction(3)
x = MaFonction(4);x
```

Gestion des paramètres avec une valeur par défaut.

```
Fonction2=function(a,b=7){a+b}
Fonction2(2,b=3)
Fonction2(5)
```

Résultats multiples dans un objet de type liste.

```
Calcule=function(r){
    p=2*pi*r;s=pi*r*r;
    list(rayon=r,perimetre=p,
    surface=s)}
resultat=Calcule(3)
resultat$ray
2*pi*resultat$r==resultat$perim
resultat$rsurface
```

Questions

1. le recours à un objet de type `list` est-il indispensable pour la fonction `Calcule()` ?

2. Écrire une fonction qui calcule le périmètre et la surface d'un rectangle à partir des longueurs l1 et l2 des deux côtés. La fonction renvoie également la longueur et la largeur du rectangle.
3. Écrire une fonction qui calcule les n premiers termes de la suite de Fibonacci ($u_1 = 0, u_2 = 1, \forall n > 2, u_n = u_{n-1} + u_{n-2}$)
4. Utiliser cette fonction pour calculer le rapport entre 2 termes consécutifs. Représenter ce rapport en fonction du nombre de termes pour $n = 20$. Que constatez-vous ? Avez-vous lu *Da Vinci Code* ?
5. Écrire une fonction qui supprime les lignes d'un data.frame ou d'une matrice présentant au moins une valeur manquante.

Réponses

1. Les 3 éléments à renvoyer étant de type numérique, un vecteur peut suffire.
2. Fonction `rectangle()` (la fonction `rect()` existe déjà) :

```
rectangle=function(l1,l2){
    p=(l1+l2)*2
    s=l1*l2
    list(largeur=min(l1,l2),longueur=max(l1,l2),
    perimetre=p,surface=s)}
```

3. Utilisation de la fonction : rectangle(4,6) ;res=rectangle(8,7) pour calculer les n premiers termes de la suite de Fibonacci :

```
fibo=function(n){
    res=rep(0,n);res[1]=0;res[2]=1
    for (i in 3:n) res[i]=res[i-1]+res[i-2]
    res}
# Calcul du rapport de 2 termes consécutifs
        res=fibo(20)
        ratio=res[2:20]/res[1:19]
        plot(1:19,ratio,type="b")
```

Le rapport tend vers le nombre d'or $\frac{1+\sqrt{5}}{2} \approx 1.618034$.

4. Une façon, parmi beaucoup d'autres, de répondre à la question consiste à créer une fonction `ligne.NA` qui repère s'il y a au moins une valeur manquante dans un vecteur. Cette fonction filtre les lignes en question.

```
ligne.NA=function(vec){any(is.na(vec))}
filtre.NA=function(mat){
    tmp = apply(mat,1,ligne.NA)
    mat[!tmp,]}
# Application sur une matrice de test
matrice.test = matrix(1:40,nc=5)
matrice.test[2,5]=NA;matrice.test[4,2]=NA
matrice.test[7,1]=NA;matrice.test[7,5]=NA
filtre.NA(matrice.test)
```

4 Commandes de type `apply`

Comme déjà expliqué, il est vivement recommandé d'éviter les boucles très chronophages. La fonction `apply` et ses variantes sur des vecteurs, matrices ou listes permettent d'appliquer une même fonction `FUN` sur toutes les lignes (MARGIN=1) ou les colonnes (MARGIN=2) d'une matrice MAT :

```
apply(MAT , MARGIN, FUN)
```

Les fonctions `lapply` et `sapply` calculent la même fonction sur tous les éléments d'un vecteur ou d'une liste.

`lapply(X,FUN, ARG.COMMUN)` permet d'appliquer la fonction `FUN` à tous les éléments du vecteur ou de la liste X. Les valeurs de X sont affectées au premier argument de la fonction FUN. Si la fonction FUN a plusieurs paramètres d'entrée, ils sont spécifiés dans `ARG.COMMUN`. Cette fonction retourne le résultat sous la forme de listes. La fonction `sapply` est similaire à `lapply` mais le résultat est retourné si possible sous forme de vecteurs.

`tapply(X,GRP,FUN,...)` applique une fonction FUN sur les sous-groupes d'un vecteur X définis par une variable de type factor GRP.

Exemples :

```
data(iris)
apply(iris[,1:4],2,sum)
lapply(iris[,1:4],sum)
```

```
sapply(iris[,1:4],sum)
tapply(iris[,1],iris[,5],sum)
```