Statistique de base

Programmation en R

Tabea Rebafka

Université Pierre et Marie Curie
Master 1e année
2016

Table des matières

1 Programmation sous R

Il existe un nombre très important de fonctions R prédéfinies, mais il est aussi possible d'écrire ses propres fonctions R. Cela permet d'exécuter les mêmes commandes autant que l'on veut en changeant à loisir les paramètres. Sous R une fonction est un objet comme d'autres.

1.1 Écrire ses propres fonctions R

Une fonction R est une série d'instructions comportant des paramètres variables qui retourne un objet R à la fin. En général, on n'écrit pas de fonctions directement dans la console, mais toujours dans un script.

Pour déclarer une fonction, on utilise la fonction `function`. La syntaxe générale est la suivante :

```
NomDeLaFonction <-  function(arg1,...,argk){
     suite de commandes
     sortie <- ...
     return(sortie)
}
```

Ainsi, `arg1, ..., argk` sont les noms des arguments (ou paramètres) de la fonction (on peut en mettre au tant qu'on veut, séparés par une virgule). L'instruction `return(sortie)` à la fin du corps de fonction retourne l'objet `sortie` à l'utilisateur. Notez que plusieurs objets peuvent être donnés en argument alors qu'un seul objet est renvoyé en sortie.

Pour délimiter le début et la fin des instructions, elles doivent être mises entre les caractères { et }. On peut omettre les accolades uniquement si le corps de la fonction ne contient qu'une seule instruction.

Exécutée dans la console, la fonction `function` crée on objet qui est affecté dans la variable `NomDeLaFonction`, et on peut utiliser cette fonction en appelant `NomDeLaFonction` suivi de la liste des paramètres contenue entre parenthèses.

Voici un premier exemple concret :

```
IMC <- function(poids,taille) {
    res <- poids/taille^2
    return(res)
}
```

Le nom de cette fonction est `ICM`, elle prend deux arguments, `taille` et `poids`, et elle renvoie l'indice de masse corporelle associée.

Afin de pouvoir utiliser cette fonction dans la console, il faut la rappeler ou soumettre à `R`. Pour ce faire, il y a deux possibilités :

- ou on fait un copier-coller des lignes définissant la fonction pour les exécuter dans la console,
- ou on "source" le script grâce au menu déroulant ou par la fonction `source()`. (Si le script est enregistré dans le répertoire de travail actuel, p. ex. sous le nom `MonScript.R`, on exécute la commande `source("MonScript.R")`).

Plus un programme est long, mieux vaut de "sourcer" le script afin d'éviter de surcharger la console.

Désormais on peut utiliser la fonction `IMC` dans la session en cours comme toute autre fonction de `R` :

```
IMC(90,1.85)

[1] 26.29657
```

Il est possible d'affecter des valeurs par défaut aux variables d'une fonction. C'est aussi simple que ça :

```
IMC <- function(poids=60,taille=1.75) {
    res <- poids/taille^2
    return(res)
}
```

Après avoir sourcé la fonction de nouveau, on peut l'appeler sans préciser les arguments :

```
IMC()

[1] 19.59184
```

Dans ce cas, les calculs sont fait avec les valeurs par défaut. Si on souhaite utiliser une autre valeur pour la taille, on tape :

```
IMC(taill=1.88)

[1] 16.97601
```

Ou on fait comme avant :

```
IMC(60,1.88)

[1] 16.97601
```

1.2 Objet `list`

Une liste est une collection d'objets **non nécessairement de même type**. C'est pratique pour les fonctions qui ne peuvent renvoyer un seul objet. En fait, des nombreuses fonctions `R` renvoient des listes.

Une liste peut être créée grâce à la fonction `list()`. Exemple : Créons une liste de nom `inutile` :

```
v <- 10:1
x <- pi
word <- "Wow!"
inutile <- list(vec = v, nombr = x, fn_imc = IMC, mot = word)
inutile

$vec
 [1] 10  9  8  7  6  5  4  3  2  1

$nombr
[1] 3.141593

$fn_imc
function (poids = 60, taille = 1.75)
{
    res <- poids/taille^2
    return(res)
}

$mot
[1] "Wow!"
```

Vérifions le type de l'objet `inutile` :

```
typeof(inutile)

[1] "list"
```

Les éléments d'une liste porte typiquement des noms, qui sont précisés au moment de la création de la liste dans l'appel de `list` et que l'on peut connaître (et modifier) grâce à la fonction `names()`.

```
names(inutile)

[1] "vec"    "nombr"  "fn_imc" "mot"
```

On accède aux éléments d'une liste par le symbole $:

```
inutile$mot

[1] "Wow!"
```

Pour accéder au troisième élément du vecteur `inutile$vec` :

```
inutile$vec[3]

[1] 8
```

Pour utiliser la fonction `inutile$fn` :

```
inutile$fn_imc(100,1.5)

[1] 44.44444
```

1.3 Structures de contôle

En général, tous les calculs un peu poussés nécessitent de diriger le flot d'exécution d'un programme. Très souvent, il faut faire du cas par cas, c'est-à-dire exécuter des instructions différentes selon le cas. Il est aussi fréquent de vouloir répéter un même calcul plusieurs fois. Pour cela les structures de contrôle sont indispensables. Comme d'autres langages de programmation, `R` intègre dans son langage entre autre la structure `if else`, et les boucles `for` et `while`.

1.3.1 La structure `if else`

La structure `if else` permet de faire du cas par cas. En fonction d'une condition logique `condition` qui est vérifiée ou non, les instructions `action1` sont réalisés ou les instructions `action2`. La syntaxe générale de la structure `if else` est

```
if (condition) {action1} else {action2}
```

ou dans le cas particulier où il n'a rien à faire quand la condition `condition` n'est pas vérifiée simplement :

```
if (condition) {action1}
```

Voici un exemple élémentaire :

```
age <- 17
if (age < 18) {x <- 'mineur'} else {x <- 'majeur'}
x

[1] "mineur"
```

Bien évidemment, dans la pratique, la condition `condition` est souvent le résultat d'opérations logiques faisant intervenir les opérateurs logiques. La syntaxe des conditions logiques élémentaires est la suivante :

```
age < 18   # age est strictement inférieur à 18

[1] TRUE

age <= 18  # age est inférieur ou égal à 18

[1] TRUE

age == 18  # Attention: age = 18 ne donne pas une condition

[1] FALSE

age != 18  # age est différent de 18

[1] TRUE
```

Les conditions peuvent également être des objets complexes, des combinaisons de conditions :

```
cond1 && cond2  # Est vrai si cond1 est vrai ET cond2 est vrai.
cond1 || cond2  # Est vrai si cond1 est vrai OU cond2 est vrai.
!cond1          # Est vrai quand cond1 est faux
```

Un dernier exemple pour l'utilisation de la structure `if else` :

```
IMC <- function(poids,taille) {
    val <- poids/taille^2
    if (val>=18.5 && val<=25 ){
        phrase <- "Très bien ! Tout va bien !"
    }
    else{
        phrase <- "Attention au poids !"
    }
    return(list(imc=val, comment=phrase))
}

IMC(58,1.70)
```

```
$imc
[1] 20.0692

$comment
[1] "Très bien ! Tout va bien !"

IMC(48,1.70)

$imc
[1] 16.609

$comment
[1] "Attention au poids !"
```

1.3.2 La boucle `for`

Pour répéter les mêmes instructions en variant un paramètre, on utilise la boucle `for`. Plus précisément, la boucle `for` permet d'effectuer les `instructions` pour tous les éléments `v` du `vecteur`. La syntaxe générale est

```
for (v in vecteur) {instructions}
```

Voici deux exemples élémentaires utilisant une boucle `for` :

```
for (i in 1:3){print(i)}

[1] 1
[1] 2
[1] 3

vec <- 2:5
for (k in 1:length(vec)){
  vec[k] <- vec[k]^k
  }
vec

[1]   2   9  64 625
```

Lorsque cela est possible, il est préférable d'éviter l'utilisation de boucle en `R`, car cela entraîne souvent un accroissement du temps de calcul. La plupart des opérations en `R` sont en effet vectorisées, c'est-à-dire qu'elles peuvent opérer sur des vecteurs, et ce calcul est effectué dans un langage compilé, ce qui est beaucoup plus rapide. Dans ce sens, le dernier exemple ci-dessus pour l'utilisation de la boucle `for` est très mauvais, car le même calcul est effectué plus rapidement par l'instruction

```
vec <- 2:5
vec^(1:length(vec))

[1]   2   9  64 625
```

Un autre exemple où l'utilisation de la boucle `for` est indispensable est le calcul de la suite de Fibonacci, qui est la suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1. Voici une fonction `R` pour calculer les `m` premiers termes da la suite de Fibonacci :

```
fibonacci <- function(m){
  # pour la boucle for il faut que m>=3 :
  if (m<3) {m=3}

  # créer un vecteur de longueur m dont les 2 permiers éléments ont
  # les valeurs des 2 premiers termes de la suite Fibonacci :
  suite <- 0:(m-1)
  for (k in 3:m){
    # calculer le k-ième terme de la suite :
    suite[k] <- suite[k-1]+suite[k-2]
  }
  return(suite)
}

fibonacci(25)

 [1]     0     1     1     2     3     5     8    13    21    34    55
[12]    89   144   233   377   610   987  1597  2584  4181  6765 10946
[23] 17711 28657 46368
```

1.3.3 La boucle `while`

Des méthodes mathématiques itératives nécessitent souvent l'exécution des mêmes instructions jusqu'à convergence. Dans ce cas, le nombre d'itérations est inconnu d'avance, et donc la boucle `for` est inadaptée. On utilise plutôt la boucle `while` qui repose sur une condition logique `condition` évaluée à chaque itération et on continue à effectuer les `instructions` tant que la `condition` est réalisée. La syntaxe générale est

```
while (condition) {instructions}
```

Voici un exemple élémentaire :

```
a <- 2
while (a < 100)
```

```
  {a <- a^2 }
a

[1] 256
```

Pour revenir à la suite de Fibonacci, on peut écrire une fonction pour calculer le nombre de termes nécessaires pour dépasser un valeur donné :

```
howmany_fibonacci <- function(M){
  if (M<=1) {howmany <- 'Appelez la fonction pour une valeur > 1'}
  else{
      # initialisation de 2 premiers termes de la suite :
      term_kmoins1 <- 0
      term_k <- 1
      # longueur actuelle de la suite :
      k <- 2

      # effectuer les instructions suivantes tant que le dernier terme
      # est inférieur au seuil M :
      while (term_k < M){
        # calculer le terme suivant de la suite :
        term_kplus1 <- term_k+term_kmoins1
        # mettre à jour la longueur de la suite :
        k <- k+1
        # mettre à jour les valeurs des 2 derniers termes de la suite :
        term_kmoins1 <- term_k
        term_k <- term_kplus1
      }
      howmany <- k
  }

  return(howmany)
}

howmany_fibonacci(10)

[1] 8

howmany_fibonacci(10^6)

[1] 32
```

Quand on utilise la boucle `while` il est très important de s'assurer que la condition n'est plus vérifiée à un moment, afin d'éviter que le programme tourne indéfiniment.