

S'exprimer dans R

Fousseynou Bah

25-Jun-2020

- 1 Introduction
- 2 Les déclarations
- 3 Les boucles
- 4 Les fonctions

Section 1

Introduction

Objectif de ce cours

Dans le cours précédent, il a été question d'objets dans R. Certains types ont été présentés. Il a surtout été fait état de la différence qui les sépare, de ce en quoi ils se démarquent les uns des autres. Ici, nous allons continuer en explorant l'expression dans R.

Les objets permettent de stocker des données. Celles-ci ne deviennent vivantes et parlantes qu'à travers le dialogue que le *data scientist* entretient avec elles. Et en quels termes ce dialogue se pose-t-il? Là est le début de notre démarche ici.

Nous allons:

- revenir sur les questions logiques;
- introduire déclarations conditionnelles;
- introduire la notion de boucle et de fonction.

Que nous faut-il?

- R (évidemment)
- RStudio (de préférence)
- Les données utilisées dans le cadre du cours. Nous utiliserons ici des données de recensement au Mali

Section 2

Les déclarations

Données (1)

Regardons l'objet que nous avons dans notre environnement: la liste "pop_groupage_list"
`str(pop_groupage_list)`

```
## List of 4
## $ 1976:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1976 1976 1976 1976 1976 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 589394 482851 321959 333508 265842 ...
## ..$ homme : num [1:18] 587015 492272 342807 308607 218391 ...
## ..$ total : num [1:18] 1176409 975123 664766 642115 484233 ...
## $ 1987:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1987 1987 1987 1987 1987 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 713507 611562 414302 379522 315753 ...
## ..$ homme : num [1:18] 719804 633206 452166 348200 260215 ...
## ..$ total : num [1:18] 1433311 1244768 866468 727722 575968 ...
## $ 1998:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1998 1998 1998 1998 1998 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 824505 797057 589603 529270 409584 ...
## ..$ homme : num [1:18] 839795 830211 637495 492480 364333 ...
## ..$ total : num [1:18] 1664300 1627268 1227098 1021750 773917 ...
## $ 2009:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 2009 2009 2009 2009 2009 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 1321275 1178850 882725 799081 624565 ...
## ..$ homme : num [1:18] 1353418 1225145 935796 745757 538927 ...
## ..$ total : num [1:18] 2674693 2403995 1818521 1544838 1163492 ...
```

Il s'agit de la population par groupe d'âge au Mali en 1976, 1987, 1998 et 2009 (RGPH).

Données (2)

```
# Sélectionnons seulement les données relatives à l'année 2009.
pop_groupepage_2009 <- pop_groupepage_list[["2009"]]
# (vous vous rappelez cette technique?...sinon, voir cours précédent.)
# Maintenant, regardons la structure de ce data frame.
str(pop_groupepage_2009)
```

```
## 'data.frame':   18 obs. of  5 variables:
## $ annee : num  2009 2009 2009 2009 2009 2009 ...
## $ groupepage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## $ femme : num  1321275 1178850 882725 799081 624565 ...
## $ homme : num  1353418 1225145 935796 745757 538927 ...
## $ total : num  2674693 2403995 1818521 1544838 1163492 ...
```

```
# Regardons la tête, les 3 premières observations par exemple.
head(x = pop_groupepage_2009, n = 3)
```

```
##   annee groupepage  femme  homme  total
## 55  2009         0-4 1321275 1353418 2674693
## 56  2009         5-9 1178850 1225145 2403995
## 57  2009        10-14 882725  935796 1818521
```

```
# Regardons la queue, les 3 dernières observations par exemple.
```

```
##   annee groupepage  femme  homme  total
## 70  2009        75-79 36949 41667 78616
## 71  2009        80+ 44504 42779 87283
## 72  2009         ND     0     0     0
```

```
knitr::asis_output("\\normalsize")
```


Données (3)

```
# Supposons que l'on veuille déterminer les groupes d'âge pour lesquels il y a plus de femmes que d'hommes
# On pose la condition suivante: "femme > homme"
# Tirons deux vecteurs de notre data frame
pop_femme_2009 <- pop_groupage_2009$femme
pop_homme_2009 <- pop_groupage_2009$homme
pop_femme_2009 > pop_homme_2009
```

```
## [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE FALSE
## [13] FALSE FALSE FALSE FALSE TRUE FALSE
```

```
# Pour une meilleur lisibilité, insérons ce résultat dans le data frame
pop_groupage_2009$femme_sup_homme <- pop_groupage_2009$femme > pop_groupage_2009$homme
# Ici assignant à une variable du data frame le résultat de l'opération de comparaison,
# R crée lui-même une variable booléenne (TRUE/FALSE).
# Regardons les groupes d'âge qui répondent au critère posé.
pop_groupage_2009[pop_groupage_2009$femme_sup_homme,]
```

```
##   annee groupage  femme  homme  total femme_sup_homme
## 58  2009    15-19 799081 745757 1544838          TRUE
## 59  2009    20-24 624565 538927 1163492          TRUE
## 60  2009    25-29 557627 457139 1014766          TRUE
## 61  2009    30-34 436501 391919  828420          TRUE
## 62  2009    35-39 333542 330907  664449          TRUE
## 63  2009    40-44 281004 276149  557153          TRUE
## 65  2009    50-54 196356 192875  389231          TRUE
## 71  2009      80+  44504  42779   87283          TRUE
```

Déclarations: formulations simples

```
# Sur la base de ces résultats, on voit clairement que R sait comparer des valeurs numériques.  
# Juste pour confirmer, reprenons sur le groupe d'âge 0-4 ans:  
1353418 < 1321275
```

```
## [1] FALSE
```

```
# Qu'en est-il des réels?  
1.000002 > 1 # (Notez que l'assignation se fait avec "=", mais le test d'égalité se fait avec "==")
```

```
## [1] TRUE
```

```
# De toute évidence, ça marche avec les nombres. Qu'en est-il des caractères? Testons!  
"MALI" == "Mali"
```

```
## [1] FALSE
```

```
# Cette égalité est rejetée par que R est sensible à la taille des lettres (majuscule/minuscule).  
# Maintenant regardons la logique  
TRUE == 1
```

```
## [1] TRUE
```

```
# Tentez de m'expliquer ce résultat! (Google est votre ami!)
```

Déclarations: critères additifs (et = &)

```
# Il est souvent possible que l'on souhaite combiner plusieurs résultats.
# Supposons que l'on veuille connaître les groupes d'âge pour lesquels:
# - les femmes sont plus nombreuses que les hommes; et
# - la population totale (hommes + femmes) est en dessous de 1 millions de personnes
# Définissons nos critères.
pop_groupe_2009$femme_sup_homme <- pop_groupe_2009$femme > pop_groupe_2009$homme
pop_groupe_2009$moins_de_1_million <- pop_groupe_2009$total < 1000000
# Combinons!
pop_groupe_2009[pop_groupe_2009$femme_sup_homme & pop_groupe_2009$moins_de_1_million,]
```

##	annee	groupe	femme	homme	total	femme_sup_homme	moins_de_1_million
## 61	2009	30-34	436501	391919	828420	TRUE	TRUE
## 62	2009	35-39	333542	330907	664449	TRUE	TRUE
## 63	2009	40-44	281004	276149	557153	TRUE	TRUE
## 65	2009	50-54	196356	192875	389231	TRUE	TRUE
## 71	2009	80+	44504	42779	87283	TRUE	TRUE

```
# L'addition de critères se fait avec l'opérateur "&".
# Le résultat donne les observations qui répondent à toutes les conditions posées.
```

Déclarations: critères alternatifs (ou = |) (1)

```
# La combinaison de critères ne se pose pas toujours sous la forme additive.
# Il arrive qu'on veuille procéder sur la base de: soit...soit...
# Dans ce cas, il faut une autre expression.
# Cherchons à connaître les groupe pour lesquels:
# - soit les femmes sont plus nombreuses que les hommes;
# - soit la population totale (hommes + femmes) est en dessous de 1 millions de personnes
# Combinons à nouveau et regardons juste les 6 premières observations.
head(
  pop_groupage_2009[pop_groupage_2009$femme_sup_homme | pop_groupage_2009$moins_de_1_million,]
)
```

##	annee	groupage	femme	homme	total	femme_sup_homme	moins_de_1_million
## 58	2009	15-19	799081	745757	1544838	TRUE	FALSE
## 59	2009	20-24	624565	538927	1163492	TRUE	FALSE
## 60	2009	25-29	557627	457139	1014766	TRUE	FALSE
## 61	2009	30-34	436501	391919	828420	TRUE	TRUE
## 62	2009	35-39	333542	330907	664449	TRUE	TRUE
## 63	2009	40-44	281004	276149	557153	TRUE	TRUE

```
# La validation de l'une des conditions suffit ici.
# On voit des groupes au dessus de 1 million de personne (violation du critère n°2).
# Toutefois, les femmes y sont plus nombreuses (validation du critère n°1)
# A l'inverse, certains groupes ont moins de femmes (violation du critère n°1),
# mais comptent moins d'1 millions de personnes (validation du critère n°2).
```

Déclarations: critères alternatifs (ou = |) (2)

*# Souvent, il arrive qu'on veuille accumuler des critères à l'intérieur d'une seule variable.
 # Supposons que l'on souhaite voir les informations concernant juste les moins de 15 ans.
 # On sait que, dans ce cas, on aura à sélectionner trois groupes d'âge: 0-4, 5-9, et 10-14.
 # Reprenons la logique des critères alternatifs (soit...soit...).*

```
pop_groupeage_2009[pop_groupeage_2009$groupeage == "0-4" | # condition 1
  pop_groupeage_2009$groupeage == "5-9" | # condition 2
  pop_groupeage_2009$groupeage == "10-14" # condition 3
]
```

```
##      annee groupeage  femme  homme  total
## 55  2009      0-4 1321275 1353418 2674693
## 56  2009      5-9 1178850 1225145 2403995
## 57  2009     10-14 882725 935796 1818521
```

Maintenant, ajoutons au critère de "moins de 15 ans" celui d'un "total < 2000000"

```
pop_groupeage_2009[(pop_groupeage_2009$groupeage == "0-4" | # condition 1
  pop_groupeage_2009$groupeage == "5-9" | # condition 2
  pop_groupeage_2009$groupeage == "10-14") # condition 3
  &
  pop_groupeage_2009$total < 2000000
]
```

```
##      annee groupeage  femme  homme  total
## 57  2009     10-14 882725 935796 1818521
```

*# Nous avons isolé les critères alternatifs entre parenthèses "()"
 # avant d'y ajouter le critère additif.*

Déclarations: critères opposés (opposé = !)

```
# Souvent, il arrive que l'on souhaite sélectionner sur la base de l'opposition à un critère.
# Explorons à travers un exemple
# Plus haut, nous avons défini les groupes où "femme > homme".
# Ceci revient à définir les groupes où la condition "homme >= femme" est violée.
# Voyons voir comment on part de la négation pour parvenir à ce même résultat
# Rappelons
pop_groupepage_2009$femme_sup_homme <- pop_groupepage_2009$femme > pop_groupepage_2009$homme
# Récréons la même variable, mais autrement
pop_groupepage_2009$homme_pas_sup_femme <- !(pop_groupepage_2009$homme >= pop_groupepage_2009$femme)
# Regardons les nouvelles variables
head(pop_groupepage_2009, n = 2)
```

```
##   annee groupepage  femme  homme  total femme_sup_homme homme_pas_sup_femme
## 55  2009      0-4 1321275 1353418 2674693          FALSE          FALSE
## 56  2009      5-9 1178850 1225145 2403995          FALSE          FALSE
```

```
# Comme dans bien de cas, on ne peut pas tout inspecter à l'oeil nu.
# Passons par une déclaration pour vérifier que les deux variables sont pareilles.
pop_groupepage_2009$femme_sup_homme == pop_groupepage_2009$homme_pas_sup_femme
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [16] TRUE TRUE TRUE
```

```
# Les deux nouvelles variables s'équivalent.
```

Déclarations conditionnelles

```
# Jusque là, nous avons parlé de déclarations dans une formulation simple.
# Nous les avons pas inscrites dans le cadre d'un arbre de décision.
# Il s'agit du schéma suivant: "si condition remplie, alors action 1, sinon action 2".
# Ceci est pratiquement le début de l'intelligence artificielle.
# On délègue à la machine l'exécution de tâches sur la base de critères définis
# (et plus tard la prise de décision plus complexes...mais ça commence au niveau simple).

# Nous savons que d'un groupe à un autre la supériorité numérique alterne entre hommes et femmes.
# Si nous voulons par exemple créer une variable qui indiquera le groupe le plus nombreux,
# nous procédons comme suit:
pop_groupepage_2009$test_max <- ifelse(test = pop_groupepage_2009$femme > pop_groupepage_2009$homme, # condition
                                       yes = "femme", # action si condition satisfaite
                                       no = "homme" # action si condition non satisfaite
                                       )# Regardons les données.

# Nous pouvons même assigner la valeur la plus élevée à une nouvelle variable.
pop_groupepage_2009$valeur_max <- ifelse(test = pop_groupepage_2009$femme > pop_groupepage_2009$homme, # condition
                                          yes = pop_groupepage_2009$femme, # action si condition satisfaite
                                          no = pop_groupepage_2009$homme # action si condition non satisfaite
                                          )

head(pop_groupepage_2009)
```

##	annee	groupepage	femme	homme	total	test_max	valeur_max
## 55	2009	0-4	1321275	1353418	2674693	homme	1353418
## 56	2009	5-9	1178850	1225145	2403995	homme	1225145
## 57	2009	10-14	882725	935796	1818521	homme	935796
## 58	2009	15-19	799081	745757	1544838	femme	799081
## 59	2009	20-24	624565	538927	1163492	femme	624565
## 60	2009	25-29	557627	457139	1014766	femme	557627

Section 3

Les boucles

Les boucles, la solution aux tâches répétitives

Un grand avantage de la programmation est la capacité de déléguer à la machine l'exécution de tâches répétitives. R dispose de diverses fonctions qui permettent d'effectuer celles-ci en boucle. Ceci est très commode surtout quand le nombre de répétitions est élevé et le risque d'erreur considérable quand la répétition est conduite manuellement. Toutefois, la nécessité des boucles varie d'un objet à un autre. Si pour certains, des solutions alternatives et plus simples existent, pour d'autres, elles sont la meilleure option.

Dans ce cours, nous allons nous limiter à la fonction *for*. Vous pouvez regarder la fonction *while*. Entrez dans la console: *help("while")*.

La fonction *for*

```
# La fonction "for" est très pratique pour l'exécution des boucles dans `R`.
# Elle est structurée de la façon suivante:
## for(var in seq){
##   expr
## }
# "var" désigne une variable dans la séquence "seq".
# On peut dire "i in c(1:10)" pour dire que pour tout "i" élément de la séquence allant de 1 à 10".
# "expr" désigne la fonction que l'on peut appliquer à ces éléments.
# Supposons que nous voulons multiplier par deux tous les éléments contenant dans la séquence 1:10.
# Voici cette séquence:
c(1:10)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

```
# Voici la boucle:
for(i in c(1:10)){
  print(i*2)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

```
knitr::asis_output("\\normalsize")
```

Application sur vecteurs (1)

```
# Prenons un vecteur de chiffres.
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# Utilisons une boucle avec la fonction "for" pour élever les éléments à leur carré
# et stockons dans un vecteur nommé "y":
y <- c() # Création d'une coquille vide de vecteur.
for(i in x){ # Pour chaque élément dans le vecteur x,
  y[[i]] <- i^2 # créer un élément dans le vecteur y qui en serait le carré.
}
# Regardons y
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
##
## [[6]]
## [1] 36
##
## [[7]]
## [1] 49
```

Application sur vecteurs (2)

```
# Pour nous assurer que cette règle n'est pas limité qu'aux chiffres, testons avec les lettres.
# Prenons un vecteur de caractères.
x <- c("Mamadou", "Amadou", "Ahmed", "Ahmad", "Abdoul", "Zan", "Tchiè", "Mady")
# Cherchons à détecter les prénoms qui contiennent la lettre "a".
# R a des fonctions natives qui peuvent exécuter cette tâche dont "grepl".
y <- c() # Création d'une coquille vide de vecteur.
for(i in x){ # Pour chaque élément dans le vecteur x,
  y[i] <- grepl(pattern = "(a)", x = i) # identifier les éléments contenant la lettre "a".
}
# Regardons y
y
```

```
## Mamadou Amadou Ahmed Ahmad Abdoul Zan Tchiè Mady
## TRUE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

```
# Encore une fois, on peut remarque que R est sensible à la taille de la lettre (minuscule/majuscule).
# Regardez les résultats pour "Ahmad" et "Ahmed".
# Reprenons en appliquant directement la formule
y <- grepl(pattern = "(a)", x)
y
```

```
## [1] TRUE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

```
# Même résultat. Moins de codage. Donc solution optimale!
# La fonction native (grepl) s'exécute déjà en boucle sur tous les éléments du vecteur.
# Leçon: chaque fois, qu'une fonction native existe et peut exécuter une tâche,
# il est préférable de se passer de la boucle.
```

Application sur matrices (1)

```
# Maintenant, essayons sur une matrice.
x <- matrix(data = c(1:12), nrow = 3, byrow = TRUE)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
# Comme avant, élevons les éléments à leur carré et stockons dans une matrice nommée "y":
y <- c() # Création d'une coquille vide de matrice
for(i in x){ # Pour chaque élément dans la matrice x,
  y[[i]] <- i^2 # créer un élément dans la matrice y qui en serait le carré.
}
# Regardons y
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
```

Application sur matrices (2)

```
# Ajustement le format de la matrice qui recevra les résultats.
# Créons une coquille vide de matrice.
y <- matrix(data = rep(NA, times = 12), nrow = 3, byrow = TRUE)
y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  NA  NA  NA  NA
## [2,]  NA  NA  NA  NA
## [3,]  NA  NA  NA  NA
```

```
# Rerenons la boucle
for(i in 1:nrow(x)){ # Pour chaque ligne (i) de la matrice x, et
  for(j in 1:ncol(x)){ # pour chaque colonne (j) de la matrice x,
    y[i, j] <- x[i, j]^2 # créer un élément dans la matrice y qui en serait le carré.
  }
}
# Regardons y
y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    9   16
## [2,]   25   36   49   64
## [3,]   81  100  121  144
```

```
# Nous avons le bon résultat et le bon format.
# Mais que de lignes de codes!!!!
# Il doit y avoir une voie plus simple!
```

Application sur matrices (3)

```
# Maintenant, regardons une autre solution:
# l'implémentation directe de la formule ( $x^2$ ).
y <- x2
y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    9   16
## [2,]   25   36   49   64
## [3,]   81  100  121  144
```

```
# A l'instar du vecteur, l'on peut appliquer des formules directement aux matrices.
# L'objet qui en résulte hérite de la structure et du format de la matrice de départ.
# Ce qui marche pour les chiffres, marche-t-il pour les lettres aussi?
```

Application sur matrices (4)

```
# Comme pour les vecteurs, testons avec une matrice de caractère.
# Cherchons dans la matrice suivante les éléments qui contiennent la lettre "z" (minuscule!)
x <- matrix(data = c("Zégoua", "Hamdallaye", "Zanbougou",
                    "Farimaké", "Cinzani", "Tinzawatene",
                    "Nara", "Hawa Dembaya", "Bozobougou"),
            nrow = 3, byrow = TRUE)

x
```

```
##      [,1]      [,2]      [,3]
## [1,] "Zégoua"  "Hamdallaye" "Zanbougou"
## [2,] "Farimaké" "Cinzani"   "Tinzawatene"
## [3,] "Nara"    "Hawa Dembaya" "Bozobougou"
```

```
# Appliquons directement la formule à la matrice "x"
y <- grepl(pattern = "(z)", x)
y
```

```
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE TRUE
```

```
# Nous avons le bon résultat, mais pas le bon format.
# R a généré le résultat sous format de vecteur.
# Ce qui en érode fortement la lisibilité.
# Ajustons!
```


Application sur matrices (5)

x

```
##      [,1]      [,2]      [,3]
## [1,] "Zégoua"  "Hamdallaye" "Zanbougou"
## [2,] "Farimaké" "Cinzani"    "Tinzawatene"
## [3,] "Nara"     "Hawa Dembaya" "Bozobougou"
```

Nous allons déclarer le résultat sous le format matrice.

```
y <- grepl(pattern = "(z)", x) # étape 1
```

```
y <- matrix(data = y, nrow = 3, byrow = TRUE) # étape 2
```

ou tout simplement

```
y <- matrix(data = grepl(pattern = "(z)", x), nrow = 3, byrow = TRUE) # combinaison des 2 étapes
```

y

```
##      [,1] [,2] [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE  TRUE  TRUE
```

Malgré cet ajustement, l'application directe de la formule est préférable à la boucle

car une fonction native existe déjà pour l'exécution de la tâche souhaitée.

Sachant que les matrices sont fortement sollicitées en algèbre, il n'est pas surprenant de trouver

que le format est respecté quand les opérations pour sur des chiffres, mais défait quand il s'agit

de lettres ou caractères.

Application sur *data frame*

```
# Partant de ce qu'on a vu avec les vecteurs et les matrices, on peut se douter que les boucles
# ne sont pas toujours le meilleur choix pour les data frame non plus
# Supposons que l'on veuille calculer pour chaque groupe d'âge de notre data frame
# l'écart entre les femmes et les hommes: "femme - homme".
# Une formulation en boucle serait:
ecart_femme_homme <- c()
for(i in 1:nrow(pop_groupage_2009)){
  ecart_femme_homme[i] <- pop_groupage_2009[i, "femme"] - pop_groupage_2009[i, "homme"]
}
ecart_femme_homme
```

```
## [1] -32143 -46295 -53071 53324 85638 100488 44582 2635 4855 -11070
## [11] 3481 -14467 -3894 -11252 -1136 -4718 1725 0
```

```
# Un détournement fort peu utile quand on peut faire ça:
pop_groupage_2009$ecart_femme_homme <- pop_groupage_2009$femme - pop_groupage_2009$homme
head(x = pop_groupage_2009, n = 3)
```

```
##   annee groupe  femme  homme  total test_max valeur_max ecart_femme_homme
## 55 2009    0-4 1321275 1353418 2674693   homme 1353418      -32143
## 56 2009    5-9 1178850 1225145 2403995   homme 1225145      -46295
## 57 2009   10-14 882725 935796 1818521   homme 935796      -53071
```

Application sur listes (1)

```
# C'est avec les listes que les boucles prennent tout leur sens.  
# Les vecteurs, matrices et data frame constitue tous des objets  
# unitaires eux-mêmes. Ils ont leur propriétés propres à eux-mêmes  
# (structure et comportements). Ceci veut dire qu'ils prêtent à  
# l'assimilation par les fonctions. Celles-ci vont systématiquement  
# s'appliquer sur toutes les éléments désignés au sein de l'objet.  
# Qu'il s'agisse d'une opération mathématiques (élévation au carré)  
# ou de la détection d'un caractère, l'objet peut servir d'intrant  
# direct à la fonction. Avec la liste, les choses sont différentes.  
# La liste est un objet "hôte". Bien qu'elle ait ses propriétés,  
# elle sert de contenant à d'autres objets. De ce fait, elle peut  
# abriter plusieurs objets sur lesquels l'on peut souhaiter exécuter  
# la même opération en boucle.  
# Et c'est là, qu'on est content que les boucles existent!  
  
# Illustrons
```

Application sur listes (2)

```
# Nous avons vu plus haut qu'avec les déclarations conditionnelles,
# l'on peut exécuter des tâches sur la base d'un arbre de décision.
# Maintenant, imaginez que vous avez à répéter une même tâche sur plusieurs objets.
# Nous avons vu que la liste contient 4 data frames, tirés de 4 recensements (1976, 1987, 1998 et 2009).
# Imaginez que vous souhaitez déterminer qui des hommes et des femmes
# sont les plus nombreux et ce pour tous les années de recensement.
# Là, vous allez devoir définir une tâche et l'exécuter en boucle.
# Pensez-vous comme un agent de vaccination qui passe dans toutes les concessions (data frame)
# d'une rue (liste) pour vacciner des enfants (le test "femme > homme").
# Rappelons les objets contenus dans la liste "pop_groupage_list"
names(pop_groupage_list)
```

```
## [1] "1976" "1987" "1998" "2009"
```

```
# Pour montrer comment ça marche, voyons juste la première observation de chaque data frame
for(i in pop_groupage_list) { # Pour chaque élément de la liste
  obs1 <- head(x = i, n = 1) # Assigner l'affichage de la 1ère observation à une variable
  print(obs1) # Affiche toutes les 1ères variables extraites
}
```

```
##   annee groupage femme homme total
## 1  1976      0-4 589394 587015 1176409
##   annee groupage femme homme total
## 19 1987      0-4 713507 719804 1433311
##   annee groupage femme homme total
## 37 1998      0-4 824505 839795 1664300
##   annee groupage femme homme total
## 55 2009      0-4 1321275 1353418 2674693
```

Application sur listes (3)

```
# Maintenant qu'on a une idée du résultat de la boucle sur des data frame, continuons avec notre exemple.
# Générons dans chacun des data frame une variable "femme_sup_homme" qui est vrai (TRUE)
# quand "femme > homme" et faux (FALSE) dans le cas contraire.
for(i in pop_groupeage_list){# Pour élément "i" de la liste "pop_groupeage_2009"
  i[, "femme_sup_homme"] <- i[, "femme"] > i[, "homme"] # Exécuter l'opération "femme > homme"
  obs3 <- head(x = i, n = 3) # Assigner l'affichage des 3 premières observations à une variable
  print(obs3) # Afficher toutes les 3 premières extraites.
}
```

```
##   annee groupeage  femme  homme  total femme_sup_homme
## 1  1976      0-4 589394 587015 1176409             TRUE
## 2  1976      5-9 482851 492272  975123             FALSE
## 3  1976     10-14 321959 342807  664766             FALSE
##   annee groupeage  femme  homme  total femme_sup_homme
## 19  1987      0-4 713507 719804 1433311             FALSE
## 20  1987      5-9 611562 633206 1244768             FALSE
## 21  1987     10-14 414302 452166  866468             FALSE
##   annee groupeage  femme  homme  total femme_sup_homme
## 37  1998      0-4 824505 839795 1664300             FALSE
## 38  1998      5-9 797057 830211 1627268             FALSE
## 39  1998     10-14 589603 637495 1227098             FALSE
##   annee groupeage  femme  homme  total femme_sup_homme
## 55  2009      0-4 1321275 1353418 2674693             FALSE
## 56  2009      5-9 1178850 1225145 2403995             FALSE
## 57  2009     10-14 882725  935796 1818521             FALSE
```

Application sur listes (4)

```
# Allons plus loin en enrichissant les conditions. Voici la démarche:
# - sélectionnons seulement les moins de 15 ans: groupage est 0-4 ou 5-9 ou 10-14;
# - créons ensuite une colonne "test_max" qui indique qui a l'é supériorité numérique;
# - créons ensuite une colonne "valeur_max" qui donne la valeur de la population.
for(i in pop_groupage_list){# Pour élément "i" de la liste "pop_groupage_2009"
  # sélection des groupes d'âge dans 0-15 ans.
  i <- i[i["groupage"]=="0-4" | i["groupage"]=="5-9" | i["groupage"]=="10-14",]
  # déclarations conditionnelles pour les variables "test_max" et "valeur_max".
  i[, "test_max"] <- ifelse(test = i[, "femme"] > i[, "homme"], # condition
                           yes = "femme", # action si condition satisfaite
                           no = "homme") # action si condition non satisfaite
  i[, "valeur_max"] <- ifelse(test = i[, "femme"] > i[, "homme"], # condition
                             yes = i[, "femme"], # action si condition satisfaite
                             no = i[, "homme"]) # action si condition non satisfaite
  print(head(x = i, n = 2)) # Afficher toutes les 2 premières extraites.
}
```

```
##   annee groupage femme homme   total test_max valeur_max
## 1  1976      0-4 589394 587015 1176409   femme    589394
## 2  1976      5-9 482851 492272  975123   homme    492272
##   annee groupage femme homme   total test_max valeur_max
## 19 1987      0-4 713507 719804 1433311   homme    719804
## 20 1987      5-9 611562 633206 1244768   homme    633206
##   annee groupage femme homme   total test_max valeur_max
## 37 1998      0-4 824505 839795 1664300   homme    839795
## 38 1998      5-9 797057 830211 1627268   homme    830211
##   annee groupage femme homme   total test_max valeur_max
## 55 2009      0-4 1321275 1353418 2674693   homme    1353418
## 56 2009      5-9 1178850 1225145 2403995   homme    1225145
```

Arrêtons-nous un instant!

```
# Qu'avons-nous vu jusque là?  
# Nous avons vu comment:  
# - poser des critères et les insérer dans des déclarations;  
# - poser un raisonnement en arbre de décision avec les déclarations conditionnelles;  
# - les boucles marchent avec divers objets (vecteurs, matrices, data frame et listes).  
  
# Nous avons vu que c'est avec les listes que les boucles révèlent leur plus grande utilité.  
# Il se trouve que R contient aussi des fonctions taillées spécialement pour  
# tourner des fonctions en boucle sur les éléments d'une liste.  
# Dans R-base seulement, il y a une grande famille de fonction dont "lapply", "sapply", "vapply",  
# "tapply", "mapply", "rapply", "eapply"...  
# Toutes ces fonctions sont des outils du paradigme "split-apply-combine" qui consiste à  
# - diviser des données en morceaux;  
# - à appliquer sur chaque morceau une fonction donnée;  
# - à rassembler les résultats en un nouveau morceau.  
  
# Nous allons nous limiter à "lapply" ici.  
# Explorons "lapply" avec quelques exemples.
```

Paradigme *split-apply-combine*: illustration avec *lapply* (1)

Considérons la liste suivante avec deux vecteurs et deux matrices:

```
malist <- list(monvect2 = seq(from = 0, to = 20, by = 0.5),
              monvect1 = rnorm(20, mean = 9.88, sd = 1.23),
              mamat1 = matrix(data = c(1:20), nrow = 4, byrow = TRUE),
              mamat2 = matrix(data = rnorm(20, mean = 7.43, sd = 1.80), nrow = 4, byrow = TRUE)
            )
```

Pour chaque objet de la liste, procédons à une agrégation avec la fonction "sum".

```
for(i in malist){
  print(sum(i))
}
```

```
## [1] 410
## [1] 200.8776
## [1] 210
## [1] 160.1925
```

Faisons la même chose avec "lapply"

```
lapply(X = malist, FUN = sum)
```

```
## $monvect2
## [1] 410
##
## $monvect1
## [1] 200.8776
##
## $mamat1
## [1] 210
##
## $mamat2
## [1] 160.1925
```


Paradigme *split-apply-combine*: illustration avec *lapply* (2)

```
# Un autre exemple: au lieu des sommes, générons les moyennes. D'abord, avec la boucle "for"...
```

```
for(i in malist){  
  print(mean(i))  
}
```

```
## [1] 10  
## [1] 10.04388  
## [1] 10.5  
## [1] 8.009627
```

```
# ...ensuite, avec "lapply"  
lapply(X = malist, FUN = mean)
```

```
## $monvect2  
## [1] 10  
##  
## $monvect1  
## [1] 10.04388  
##  
## $mamat1  
## [1] 10.5  
##  
## $mamat2  
## [1] 8.009627
```

```
# Vous voyiez la logique?
```

Paradigme *split-apply-combine*: illustration avec *lapply* (3)

```
# Nous avons vu que, pour les matrices et les vecteurs, l'on trouve souvent des fonctions  
# qui sont déjà capables d'exécuter les tâches souhaitées  
# (ne paniquez pas, c'est avec la pratique que votre répertoire de fonction s'agrandira!)  
# Et quand cela est possible, une boucle n'est pas nécessaire.  
  
# Le même principe va pour les listes.  
# Quand il y a des fonctions natives qui peuvent:  
# - exécuter la tâche souhaitée (générer une somme ou une moyenne par exemple);  
# - insérer cette tâche dans une boucle (exécuter sur tous les objets d'une liste);  
# alors, il est préférable d'embrasser cette voie.  
# Les exemples précédents ont clairement illustré cela.  
  
# Maintenant, retournons à notre liste pour illustrer davantage.
```

Paradigme *split-apply-combine*: illustration avec *lapply* (4)

```
# Revenons à notre liste sur les données de population.
# Chercons à connaître les dimensions des objets de la liste (nombre de lignes, nombres de colonnes)
lapply(X = pop_groupage_list, FUN = dim)
```

```
## $`1976`
## [1] 18 5
##
## $`1987`
## [1] 18 5
##
## $`1998`
## [1] 18 5
##
## $`2009`
## [1] 18 5
```

```
lapply(X = pop_groupage_list, FUN = colnames)
```

```
## $`1976`
## [1] "annee"      "groupage" "femme"    "homme"    "total"
##
## $`1987`
## [1] "annee"      "groupage" "femme"    "homme"    "total"
##
## $`1998`
## [1] "annee"      "groupage" "femme"    "homme"    "total"
##
## $`2009`
## [1] "annee"      "groupage" "femme"    "homme"    "total"
```

Paradigme *split-apply-combine*: illustration avec *lapply* (5)

```
# Maintenant, supposons qu'on veut déterminer la population totale pour chaque année
# en faisant la somme de la colonne "total". Comment faire?
# Avec une boucle, c'est facile.
for(i in pop_groupe_list){
  print(sum(i["total"]))
}
```

```
## [1] 6392918
## [1] 7696349
## [1] 9810912
## [1] 14528662
```

```
# Avec "lapply".
lapply(X = pop_groupe_list, FUN = sum)
```

```
## Error in FUN(X[[i]], ...): only defined on a data frame with all numeric variables
```

```
# "lapply" n'arrive pas à s'exécuter car nous n'avons pas spécifié qu'à l'intérieur de chaque
# data frame, il fallait prendre la variable "totale"!
# Certes, "lapply" est destinée à exécuter les tâches en boucle, mais encore
# faudrait-il que celles-ci soient bien définies.
# Et c'est ça que fait une fonction. Elle exécute des tâches!
# Et ça, c'est le début d'un autre pan de la Data Science: la programmation fonctionnelle.
# Dans ce terme, on va englober, l'art de faire des fonctions.
# Tout y passe: de la conception à la rapidité.
# Dans le prochain chapitre, nous allons reprendre les idées déjà présentées,
knitr::asis_output("\\normalsize")
```

Section 4

Les fonctions

Les fonctions, l'épine dorsale de R

```
# Pour un data scientist, les fonctions sont d'une importance capitale car  
# son flux de travail consiste à faire passer les données d'une fonction à une autre  
# pour recadrer ses questions ou trouver des réponses à celles-ci.  
# Depuis le début, nous parlons de fonctions.  
# Qu'est-ce que c'est?  
# Dans R, la fonction agit comme dans les mathématiques.  
# C'est un règle ou une procédure qui détermine la transformation d'un intrant en extrant.  
# Prenons l'exemple suivant:
```

$$y = f(x) = x^2$$

```
# Dans cet exemple, la fonction élève les intrants au carré pour donner les extrants.  
# Dans R, c'est la même chose!  
# Nous avons déjà mentionnées certaines fonctions et avons montré ce qu'elles font.  
# Revenons sur quelques unes.
```

Retour sur quelques fonctions (1)

```
# Prenons le vecteur suivant:  
monvect <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
# Si nous voulons compter le nombre d'éléments composants cet éléments, une fonction...  
length(monevect)
```

```
## [1] 10
```

```
# ...faire la somme de ces éléments, une fonction...  
sum(monevect)
```

```
## [1] 55
```

```
#...faire la moyenne de ces éléments, une fonction...  
mean(monevect)
```

```
## [1] 5.5
```

Retour sur quelques fonctions (2)

Revenons a notre liste. Pour voir sa structure, une fonction...

```
str(pop_groupage_list)
```

```
## List of 4
## $ 1976:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1976 1976 1976 1976 1976 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 589394 482851 321959 333508 265842 ...
## ..$ homme : num [1:18] 587015 492272 342807 308607 218391 ...
## ..$ total : num [1:18] 1176409 975123 664766 642115 484233 ...
## $ 1987:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1987 1987 1987 1987 1987 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 713507 611562 414302 379522 315753 ...
## ..$ homme : num [1:18] 719804 633206 452166 348200 260215 ...
## ..$ total : num [1:18] 1433311 1244768 866468 727722 575968 ...
## $ 1998:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 1998 1998 1998 1998 1998 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 824505 797057 589603 529270 409584 ...
## ..$ homme : num [1:18] 839795 830211 637495 492480 364333 ...
## ..$ total : num [1:18] 1664300 1627268 1227098 1021750 773917 ...
## $ 2009:'data.frame':  18 obs. of  5 variables:
## ..$ annee : num [1:18] 2009 2009 2009 2009 2009 ...
## ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<...: 1 2 3 4 5 6 7 8 9 10 ...
## ..$ femme : num [1:18] 1321275 1178850 882725 799081 624565 ...
## ..$ homme : num [1:18] 1353418 1225145 935796 745757 538927 ...
## ..$ total : num [1:18] 2674693 2403995 1818521 1544838 1163492 ...
```


Retour sur quelques fonctions (3)

```
# Pour voir le nom des éléments qu'elle contient, une fonction  
names(pop_groupage_list)
```

```
## [1] "1976" "1987" "1998" "2009"
```

```
# Vous voyez l'idée?  
knitr::asis_output("\\normalsize")
```

Pourquoi faire une fonction?

```
# Vu la richesse de R on peut bien être amener à se demander  
# pourquoi se donner la peine de faire une fonction.  
# N'en exist-il pas déjà dans R ? La plupart du temps, oui!  
# Mais pas tout le temps.  
# Autant, les questions que la data scientist soulève sont nombreux,  
# autant les voies qui s'offrent à lui pour parvenir à des solutions  
# sont nombreuses. Les particularités de la question peuvent  
# faire qu'il est souhaitable voire indispensable de "personnaliser"  
# la réponse. Ceci peut signifier soumettre ses données  
# à un flux de transformation dans le  
# cadre duquel diverses autres fonctions sont utilisées.
```

Les basiques de la fonction (1)

```
# Bien que tous les sept milliards d'humains peuplant la terre partagent cette appellation commune,
# il demeure que l'on s'attache à donner à chacun d'entre eux une appellation particulière: le prénom!
# Ou bien?
# De même, une fonction a besoin d'un nom!
# A ce niveau, il est utile d'indiquer qu'il y a des mots réservés qui ne peuvent pas être utilisés. Voir:
## help("reserved")
# Après le nom, il y a les "arguments". Ceci est l'appellation donnée aux intrants.
# Ensuite on a le "corps" qui est la procédure à laquelle sont soumis ces intrants.
# Schématisons tout ça!
mafonction <- function(x){
  x^2
}
# Nous avons défini ici une fonction où x est l'argument et la procédure à laquelle
# il est soumis est l'élévation au carré. Testons le résultat.
mafonction(x = 25)
```

```
## [1] 625
```

```
# Juste pour ironiser un peu, rappelons que c'est par une fonction "function"
# que nous venons de créer une fonction. Trop méta ça!!!!!!
```

Les basiques de la fonction (2)

```
# Avançons un peu ici en créant une fonction avec deux arguments: x et y.
mafonction <- function(x, y){
  x + y
}
# Testons
mafonction(x = 1, y = 2)
```

```
## [1] 3
```

```
# Souvent, il est possible d'assigner à un argument ou à tous une valeur par défaut.
mafonction <- function(x, y = 10){
  x + y
}
# Regardons ce qu'on obtient qu'on ne spécifie pas la valeur passé à l'argument y.
mafonction(x = 3)
```

```
## [1] 13
```

```
# Certaines fonctions contiennent plusieurs arguments.
# Par commodité, on a assigné à beaucoup des valeurs par défaut qui sont validées sauf si
# l'utilisateur en décide autrement.
```

Les basiques de la fonction (3)

```
# Souvent, la fonction comprends des étapes intermédiaires.
# A vrai dire, c'est dans ça que réside la nécessité des fonctions,
# le séquençage de procédures multiples en une seule commande.
# Revenant à notre exemple, nous pouvons assigner le résultat à une variable intermédiaire z.
mafonction <- function(x, y = 10){
  z <- x + y
}
mafonction(x = 3)
# L'exécution de la fonction sur "3" car nous n'avons pas demandé à la fonction de sortir le résultat.
# Pour que le résultat sorte, il faut expliciter.
mafonction <- function(x, y = 10){
  z <- x + y
  return(z)
}
mafonction(x = 3)
```

```
## [1] 13
```

```
# "return()" est très commode quand on doit passer par plusieurs étapes à l'intérieur de la fonction.
```

Fonctions et boucles (1)

```

# Les fonctions (écriture, évaluation, etc.) constitue un domaine vaste de la data science.
# Nous ne pourrions pas tout voir d'un seul coup.
# La maîtrise des règles (les do's et les don't's) viennent avec la pratique.
# Ayant couvert les basiques, nous allons retourner à nos données pour illustrer.
# Vous vous rappelez la boucle suivante...
## for(i in pop_groupe_list){
##   print(sum(i["total"]))
## }
# qu'on avait pas réussi à insérer dans la fonction "lapply"?
# La raison est simple.
# "lapply" exécute des fonctions sur les objets contenus dans une liste.
# Elle ne peut pas systématiquement atteindre les éléments contenus dans ces objets.
# Nous avons pu faire des moyennes et des médianes sur des vecteurs et matrices à partir de "lapply".
# La raison était simple: ces objets sont des ensembles homogènes. Ils contenaient tous des chiffres,
# qui sont des éléments assimilables par "mean" et "median".
# Or, le data frame est un objet hétérogène, contenant des chiffres et des lettres.
# Les fonctions natives qu'on a utilisées ne peuvent faire la différence d'elles-mêmes.
# Elles doivent être guidées.
# De ce fait, nous devons inscrire la procédure souhaitée dans une fonction avant de passer celle-ci
# à "lapply" qui va l'exécuter en boucle sur tous les objets de la liste.
# Voici la fonction:
pop_somme <- function(df){
  sum(df["total"])
}
# Nous venons de définir une fonction où "df" est l'argument principal.
# On s'attend à un data frame comme intrant
# On s'attend à ce que celui-ci ait une colonne nommée ("total") dont les éléments seront agrégés par
# la fonction "sum".
# Vous voyez? C'est une solution très personnalisée!
# Regardons les résultats!

```

Fonctions et boucles (2)

Voici le résultat pour la boucle:

```
for(i in pop_groupage_list){
  print(sum(i["total"]))
}
```

```
## [1] 6392918
## [1] 7696349
## [1] 9810912
## [1] 14528662
```

Voici le résultat pour "lapply"

```
lapply(X = pop_groupage_list, FUN = pop_somme)
```

```
## $`1976`
## [1] 6392918
##
## $`1987`
## [1] 7696349
##
## $`1998`
## [1] 9810912
##
## $`2009`
## [1] 14528662
```

Qu'est-ce qui est préférable?

Comme avant, les fonctions existantes sont toujours meilleures.

"lapply" est intégrée à R. Elle constitue une meilleure boucle.

Aussi, elle peut générer en extrayant une liste, qui peut être assignée à un objet donné.

Fonctions et boucles (3)

Voici le résultat pour la boucle:

```
for(i in pop_groupage_list){
  print(sum(i["total"]))
}
```

```
## [1] 6392918
## [1] 7696349
## [1] 9810912
## [1] 14528662
```

Voici le résultat pour "lapply"

```
lapply(X = pop_groupage_list, FUN = pop_somme)
```

```
## $`1976`
## [1] 6392918
##
## $`1987`
## [1] 7696349
##
## $`1998`
## [1] 9810912
##
## $`2009`
## [1] 14528662
```

Qu'est-ce qui est préférable?

Comme avant, les fonctions existantes sont toujours meilleures.

"lapply" est intégrée à R. Elle constitue une meilleure boucle.

Aussi, elle peut génère en extrant une liste, qui peut être assignée à un objet donné.