Capsule 3: Présentation de la syntaxe en R et R Studio

Jean Lefebvre
2022-10-04

Les commandes et concepts suivants s'appliquent tous à l'utilisation du logiciel R mais aussi lors de l'utilisation du logiciel R Studio, qui agit comme un interface facilitant la programmation en R. Le logiciel R Studio permettra entre autres de faciliter l'exécution de commandes et de deux logiciels ayant la fonction d'exécuter le code en R en dont chaque simultanément sur `ctrl+R` pour exécuter une ligne de code sur laquelle notre curseur est placé, alors que sous R Studio, on doit plutôt cliquer simultanément sur `ctrl+Enter` tout en gardant le curseur sur la ligne de code à exécuter.

Commentaires

Les commentaires sont des bouts de code qui ne sont pas exécutés avec le reste du code et qui nous permettent de commenter à l'intérieur du script pour se retrouver plus facilement.

Dans un script R qu'on exécute, par exemple, sur R Studio, on peut mettre un bout de code en commentaire en utilisant au moins un symbole dièse (#) en début de ligne de code, comme suit :

```
# ceci fonctionne comme une commentaire
# Ceci fonctionne aussi pour un commentaire

# ceci fonctionne aussi
# ceci aussi

## ici aussi
```

Notez que l'on peut aussi sélectionner et mettre en commentaire plusieurs lignes de code à la fois, en utilisant la procédure suivante sous Windows :

- Sélectionner le bloc de code dans votre script R à mettre en commentaire.
- Presser CTRL + Shift + C.
- Le bloc de texte pourra maintenant être en commentaire.

Sous MAC, on changera CTRL + Shift + C pour COMMANDE

Importons maintenant les jeux de données `Bycatchprofile.csv` et `apachefalletRfeaut.csv` que j'ai importé et respectivement appelé `bycs` et `ref.apache`:

```
bycs<-read.csv("C:/Users/Jean-Luc Lefebvre/Desktop/Bycatchprofile.csv")
ref.apache<-read.csv("C:/Users/Jean-Luc Lefebvre/Desktop/apachefalletRfeaut.csv", stringsAsFactors=TRUE)
```

```
## 'data.frame':    1000 obs. of  11 variables:
##  $ ocean_prof_id: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Year         : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ Month        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day          : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Hour         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Minute       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Second       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Latitude     : int  -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 ...
##  $ Longitude    : int  -95 -95 -95 -95 -95 -95 -95 -95 -95 -95 ...
##  $ Depth        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Temperature  : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ Salinity     : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
## 'data.frame':    1000 obs. of  11 variables:
##  $ ocean_prof_id: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Year         : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ Month        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day          : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Hour         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Minute       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Second       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Latitude     : int  -5 -5 -5 -5 -5 -5 -5 -5 -5 -5 ...
##  $ Longitude    : int  -95 -95 -95 -95 -95 -95 -95 -95 -95 -95 ...
##  $ Depth        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ Temperature  : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ Salinity     : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
```

Contrôlons les ensembles à partir de la fonction `dscrpt`, et regardons les variables qu'il y retrouvent :

```
# contrôlons les deux jeux de données sur les variables qualitatives/nominales
dscrpt(bycs,maxchar=TRUE,digits=3)
dscrpt(ref.apache,maxchar=TRUE,digits=3)
```

```
## Warning: Variable(s) ocean_prof_id, Year, Month, Day, Hour, Minute, Second, Latitude, Longitude, Depth, Temperature, Salinity converted to factor.
## Warning: Variable(s) ocean_prof_id, Year, Month, Day, Hour, Minute, Second, Latitude, Longitude, Depth, Temperature, Salinity converted to factor.
```

```
## Warning: Variable(s) ocean_prof_id, Year, Month, Day, Hour, Minute, Second, Latitude, Longitude, Depth, Temperature, Salinity converted to factor.
## Warning: Variable(s) ocean_prof_id, Year, Month, Day, Hour, Minute, Second, Latitude, Longitude, Depth, Temperature, Salinity converted to factor.
```

```
## 
## ocean_prof_id : 1000 levels, 1 to 1000
## 
## Year : 1 level, 2000
## 
## Month : 1 level, 1
## 
## Day : 1 level, 1
## 
## Hour : 1 level, 0
## 
## Minute : 1 level, 0
## 
## Second : 1 level, 0
## 
## Latitude : 1 level, -5
## 
## Longitude : 1 level, -95
## 
## Depth : 1 level, 0
## 
## Temperature : 1 level, -9999
## 
## Salinity : 1 level, -9999
## 
## ocean_prof_id : 1000 levels, 1 to 1000
## 
## Year : 1 level, 2000
## 
## Month : 1 level, 1
## 
## Day : 1 level, 1
## 
## Hour : 1 level, 0
## 
## Minute : 1 level, 0
## 
## Second : 1 level, 0
## 
## Latitude : 1 level, -5
## 
## Longitude : 1 level, -95
## 
## Depth : 1 level, 0
## 
## Temperature : 1 level, -9999
## 
## Salinity : 1 level, -9999
```

```
## 
## ocean_prof_id : 1000 levels, 1 to 1000
## 
## Year : 1 level, 2000
## 
## Month : 1 level, 1
## 
## Day : 1 level, 1
## 
## Hour : 1 level, 0
## 
## Minute : 1 level, 0
## 
## Second : 1 level, 0
## 
## Latitude : 1 level, -5
## 
## Longitude : 1 level, -95
## 
## Depth : 1 level, 0
## 
## Temperature : 1 level, -9999
## 
## Salinity : 1 level, -9999
## 
## ocean_prof_id : 1000 levels, 1 to 1000
## 
## Year : 1 level, 2000
## 
## Month : 1 level, 1
## 
## Day : 1 level, 1
## 
## Hour : 1 level, 0
## 
## Minute : 1 level, 0
## 
## Second : 1 level, 0
## 
## Latitude : 1 level, -5
## 
## Longitude : 1 level, -95
## 
## Depth : 1 level, 0
## 
## Temperature : 1 level, -9999
## 
## Salinity : 1 level, -9999
```

Variables et leur type

On définit une variable comme étant une caractéristique mesurable qui peut prendre différentes valeurs.

Comme vous le savez, certaines variables sont numériques (quantitatives) alors que d'autres sont catégoriques (valeurs qui ne sont pas quantifiables). On peut distinguer les variables numériques en deux autres catégories, soient les variables continues (Ex: la taille, le temps passé et les variables discrètes (le nombre de frères et soeurs d'un individu, l'indice de masse corporelle). Pour ce qui des variables catégoriques, on peut distinguer les variables nominales ou ordinales. Un exemple de variable catégorique nominale: la place de résidence. Un exemple de variable catégorique ordinale: Votre niveau de satisfaction sur une échelle de 1 à 5.

En R, on doit généralement spécifier le type de variable afin qu'il soit bien pris en compte dans notre analyse. Par exemple, soit la variable `Salinité` qui l'on vient de créer:

```
Salinité <- c(1,2,3,4,5,6,7,8)
```

Ce vecteur pourrait représenter une variable numérique continue, ou encore discrète, ou encore il pourrait s'agir d'une variable catégorique nominale. C'est à nous de dire à R de quoi il s'agit.

Si variable1 est une variable numérique continue, on n'a pas besoin d'agir sur la variable car la variable est numérique par défaut. Or pour le définir en utilisant la fonction `typeof`:

```
typeof(variable1)
```

```
## [1] "double"
```

On note donc que par défaut la variable est traitée comme numérique et continue (ce que R appelle double). Ainsi, même si le vecteur pourrait représenter des nombres entiers, R traite la variable comme une variable continue.

S'il s'agit plutôt d'une variable qu'on souhaite considérer comme à valeurs entières, celle peut être remaké en utilisant la fonction `as.integer`comme suit:

```
variable1<- as.integer(variable1)
is.integer(variable1)
```

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

Supposons maintenant que nous voulons plutôt traiter la variable comme une variable nominale. Par exemple, la valeur 1 représente peut-être la catégorie "Canada", la valeur 2 la catégorie "États-Unis", et ainsi de suite. Alors on peut utiliser la procédure suivante:

```
variable1<- as.factor(variable1)
is.factor(variable1)
```

```
## [1] 1 2 3 4 5 6 7 8
## Levels: 1 2 3 4 5 6 7 8
```

On peut aussi re-coder la variable, comme suit:

```
library(dplyr)
```

```
## Attaching packages: 'dplyr'
```

```
## The following objects is masked from 'package:stats':
```

```
##  filter
```

```
## The following objects is masked from 'package:base':
```

```
##  intersect, setdiff, setequal, union
```

```
variable1<-recode(variable1, "1" = "Canada", "2" = "États-Unis", "3" = "Mexique",
                  "4" = "Chine", "5" = "Japon", "6" = "Allemagne", "7" = "Brésil",
                  "8" = "Inde")
```

```
## [1] "Canada"   "États-Unis" "Mexique"  "Chine"    "Japon"    "Allemagne" "Brésil"
## [8] "Inde"
```

On peut aussi transformer une variable numérique en variable caractère (comme du texte) en utilisant au character comme suit:

```
variable1<- as.character(variable1)
class(variable1)
```

```
## [1] "character"
```

Les guillemets autour de chaque valeur nous montrent que les valeurs sont en format caractère.

Une variable caractère ne sera pas nécessairement traitée comme une variable factor, dans le sens que R ne pas pas nécessairement considérer que la variable est nominale, et quelles sont ses catégories. Dans ce cas, on peut utiliser la fonction factor comme suit:

```
variable1<- factor(variable1, levels=c("Canada", "États-Unis", "Mexique", "Chine",
                   "Japon", "Allemagne", "Brésil", "Inde"))
class(variable1)
```

```
## [1] "factor"
```

```
## [1] 1 2 3 4 5 6 7 8
## Levels: Canada États-Unis Mexique Chine Japon Allemagne Brésil Inde
```

Notez que R nous montre maintenant les niveaux de la variable. Cette variable pourrait maintenant être utilisée dans un modèle pour issues catégoriques, par exemple (comme un modèle à choix prepotionnelles).

Vérifions maintenant le format de différentes variables du jeu de données `data1`. La fonction `class` permet de savoir quel est le type de chaque variable dans le jeu de données sans avoir à quoter le jeu de données (`data1`), ce qui peut s'avérer avec la variable `intubated` directement plutôt que d'appeler `data1$intubated`:

```
class(age)
```

```
## [1] "integer"
```

```
class(gender)
```

```
## [1] "character"
```

```
class(admissiontype)
```

```
## [1] "character"
```

```
class(ethnicity)
```

```
## [1] "character"
```

```
class(intubated)
```

```
## [1] "integer"
```

```
class(vasoactive)
```

```
## [1] "integer"
```

Rappel: Le type double signifie que R entreprose la valeur comme un nombre à 2 décimales.

Pour voir le format des variables plus rapidement, ainsi que plusieurs autres détails concernant un jeu de données, on peut utiliser la fonction `str`: comme suit:

```
str(data1)
```

```
## 'data.frame':    1000 obs. of  30 variables:
##  $ patientunitstayid: int  141168 141233 141249 141296 141327 141348 141353 141400 141402 141407 ...
##  $ age              : int  54 77 54 54 54 54 54 54 54 54 ...
##  $ gender           : chr  "Female" "Male" "Male" "Male" ...
##  $ admissiontype    : chr  "Emergency" "Emergency" "Emergency" "Emergency" ...
##  $ ethnicity        : chr  "Caucasian" "Caucasian" "Caucasian" "Caucasian" ...
##  $ intubated        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ vasoactive       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ vasopressor      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ hypotension      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ hypoxemia        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ nitrogen         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ houroadmit       : int  -0.3 -0.3 -0.3 -0.3 -0.3 -0.3 -0.3 -0.3 -0.3 -0.3 ...
##  $ ph               : int  7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 ...
##  $ hematocrit       : int  31.2 31.2 31.2 31.2 31.2 31.2 31.2 31.2 31.2 31.2 ...
##  $ hemoglobin       : int  10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 ...
##  $ hospitalid       : int  167 167 167 167 167 167 167 167 167 167 ...
##  $ wardid           : int  701 701 701 701 701 701 701 701 701 701 ...
##  $ apachescore      : int  51 51 51 51 51 51 51 51 51 51 ...
##  $ apachepatientresult: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ gcsmotor         : int  6 6 6 6 6 6 6 6 6 6 ...
##  $ gcsverbal        : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ gcseyes          : int  4 4 4 4 4 4 4 4 4 4 ...
##  $ gcstotal         : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ bun              : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ creatinine       : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ sodium           : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ potassium        : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ bilirubin        : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ albumin          : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ bicarbonate      : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ lactate          : int  -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
##  $ apachepatientresult: int  1 1 1 1 1 1 1 1 1 1 ...
```

La format de chaque variable se retrouve dans la deuxième colonne de la sortie (`int`, `chr`, etc.).

On peut changer le format d'une variable en utilisant, par exemple, la commande suivante:

```
admissiontypenumeric<- as.factor(admissiontype)
table(admissiontypenumeric)
```

```
## 
##  Elective Emergency    Urgent
##       101       899       100
```

ou encore

```
admissiontypenumeric<- as.character(admissiontypenumeric)
table(admissiontypenumeric)
```

```
## 
##  Elective Emergency    Urgent
##       101       899       100
```

Dans cet exemple particulier, on n'a pas nécessairement de raison de changer le format des variables, mais dans d'autres situations, on peut vouloir forcer une variable à être en format numérique. Par exemple, si on veut faire une ANOVA, on peut vouloir transformer une variable catégorique pour utilisation dans un modèle linéaire, pour une issue catégorique ou pour une issue binaire dans un modèle de régression logistique (ou à choix proportionnelles). Par exemple, si la variable est binaire mais qu'elle est en format caractère, il se peut que vous ne soyez pas capable d'ajuster un modèle de régression logistique pour cette issue binaire. Vous pouvez alors la remettre numérique à l'aide de la `as.numeric()`, ou la mettre en format factor.

```
table(gender)
```

```
## 
## Female   Male
##    454    546
```

```
gender<- as.factor(gender)
table(gender)
```

```
## 
## Female   Male
##    454    546
```

En R, on peut aussi spécifier qu'une variable est de type date. Cela est pratique lorsqu'on veut pouvoir faire des opérations sur des dates (ex: calculer la différence de jours entre deux dates). Mais cet exemple sera une date inventée qui n'est pas dans notre jeu de données:

```
admdate<- as.Date(admdate)
admdate<- as.Date("2010-01-01", format="%Y-%m-%d")
admdate<- as.Date("01-01-2010", format="%d-%m-%Y")
class(admdate)
```

```
## [1] "Date"
```

```
## [1] "2010-01-01 GMT"
```

```
admdate<-as.POSIXct("2010-01-01 00:00:00")
```

```
## [1] "2010-01-01"
```

Exemple du calcul de nombre de jours entre deux dates distinctes:

```
admdate<- as.Date(admdate)
admdate2<- as.Date("2010-01-01", format="%Y-%m-%d")
admdate2<- as.Date(admdate2)
admdate2<- as.POSIXct("2010-01-01 00:00:00")
admdate2-admdate
```

```
## Time difference of 0 days
```

On note que R nous renvoie la phrase `Time difference of ...`, suivie du nombre de jours. Le logiciel comprend que nous sommes en train de manipuler des dates.

On peut aussi utiliser la fonction suivante:

```
difftime("2010-01-01", admdate)
```

```
## Time difference of 0 days
```

Fonctions

R contient déjà plusieurs fonctions pré-enregistrées, telles que des fonctions qui calculent la somme, la moyenne, l'écart-type, etc. En voici quelques unes:

```
sum(c(1,2,3))
```

```
## [1] 6
```

```
mean(c(1,2,3))
```

```
## [1] 2
```

```
sd(c(1,2,3))
```

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

```
var(c(1,2,3))
```

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

```
median(c(1,2,3))
```

```
## [1] 2
```

```
max(c(1,2,3))
```

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

```
min(c(1,2,3))
```

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

A partir de ces fonctions on peut donc calculer la moyenne et la variance d'une variable dans le jeu de données `data1`, par exemple:

```
mean(bycs$Depth) # Moyenne à jeu data1 pour la variable Depth
```

```
## [1] 0
```

```
var(bycs$Depth) # Variance à jeu data1 pour la variable Depth
```

```
## [1] 0
```

Vous pourriez aussi créer vos propres fonctions à partir d'une syntaxe similaire à la suivante (que vous pouvez modifier):

```
fonc1 <function(x,y)
{
  # ceci est une fonction simple
  # qui multiplie deux variables
  # ensemble
  # et ajoute 1 à la somme
  # du produit
  # et ensuite divise par 2
  # et ensuite multiplie par 3
  # et ensuite ajoute 4
  # et ensuite ajoute 5
  # et ensuite ajoute 6
  # et ensuite ajoute 7
  # et ensuite ajoute 8
  # et ensuite ajoute 9
  # et ensuite ajoute 10
  # et ensuite ajoute
```

```
# Ici, le troisième argument est la fonction à appliquer, soit function_imc dans notre cas

imcvecteur<- apply(matrix, 1 , function_imc ) # notons que j'ai mis dans matrix les deux seules variables dont on a besoin, taille et poids, qui viennent respectivement a[1] et a[2] dans la fonction function_imc
summary(imcvecteur)
```

```
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  22.18   22.18   22.48   22.73   23.14   24.63
```

Un peu plus tard, nous verrons comment calculer l'IMC à partir d'une boucle, plutôt que d'utiliser `apply` en R. Alors qu' `apply` nous permet d'appliquer une fonction aux lignes ou encore aux colonnes d'un jeu de données de façon simultanée, la boucle nous permettra de "rouler" à travers le jeu de données entier, ligne par ligne, une à la fois, et de faire toute opération désirée.

Conditions et opérateurs logiques

Les questions en R sont généralement TRUE (vraies) ou FALSE (fausses). Ainsi, on peut faire correspondre des énoncés à des valeurs TRUE ou FALSE. Par exemple:

```
2+2==4
```

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

```
TRUE==FALSE
```

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

```
if(1==1){print('Cet énoncé est vrai')}
```

```
## [1] 'Cet énoncé est vrai'
```

```
vecteur<- c((1==1), (1==0))
vecteur
```

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

Comme vous avez pu voir ci-haut, on utilise le symbole `==` pour l'égalité. Le symbole `=` est déjà utilisé en R pour entreposer des objets, similairement au symbole `<-`. voici quelques autres opérateurs logiques en action:

Le 'Non' logique (!) ou encore le n'est pas égal à (!=):

```
2 != 4 # 2 n'est pas égal à 4
```

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

Le 'Et' logique (&):

```
vecteur<- c(1,2,3)
vecteur[vecteur> 1 & vecteur <3] # Sélectionner seulement les nombres dans vecteur qui sont plus grands que 1 ET plus petits que 3
```

```
## [1] 2
```

Le 'Ou' logique (|):

```
vecteur<- c(1,2,3)
vecteur[vecteur> 1 | vecteur <3] # Sélectionner les nombres dans vecteur qui sont soit plus grands 1 ou plus petits a que 3 -- ils sont donc tous sélectionnés
```

```
## [1] 1 2 3
```

On a déjà vu plus haut que l'opérateur 'plus petit que' est donné par `<`, et l'opérateur 'plus grand que' est `>`. Que faire si on veut plus petit ou égal, ou encore plus grand ou égal? On ajoute le symbole `=` comme suit:

```
vecteur<- c(1,2,3)
vecteur[vecteur>=2]
```

```
## [1] 2 3
```

```
vecteur[vecteur<=2]
```

```
## [1] 1 2
```

Opérateur 'If' (IF):

```
a=0
if(a==1){print(a)}
```

```
## [1] 0
```

```
b=0
if(b==1){print(b)} # n'imprime rien (pas de output)
```

Attention à cette particularité: Si vous utilisez le 'Et' ou le 'Ou' logique à l'intérieur d'un 'IF', il faut les doubler comme suit:

```
a=0
b=10
if(a >2 && b==1){print('Tough! ')}
```

```
## [1] 'Tough! '
```

Autres exemples particulièrement intéressants:

```
x<- seq(1,10,2) # fait une séquence qui part de 1 et va jusqu'à 10 par des austs de 2
x
```

```
## [1] 1 3 5 7 9
```

```
x>4
```

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

```
# Notez ou peut utiliser l'opérateur : pour créer une suite comme suit
1:10
```

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

```
# Notez ou peut utiliser la fonction rep pour répéter le même nombre ou le même objet plusieurs fois:
variable2<- c( rep(4,3) ) # répète la valeur 4 trois fois et met cela dans un vecteur c()
variable2
```

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

Fonctions *any* et *all*:

```
x<- seq(1,10,2)
any(x>5) # indique si c'est vrai pour au moins l'un d'entre eux
```

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

```
all(x>5) # indique si c'est vrai pour tous les nombres dans x
```

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

Voyons comment on peut utiliser les opérateurs logiques pour manipuler les jeux de données discutés plus tôt (qu'on a combiné et appelé `data1`). Supposons qu'on veuille garder dans notre échantillon les patients qui ont un `meanbp` supérieur à 84:

```
data_sup84<- data1[data1$meanbp > 84,]

dim(data_sup84)
```

```
## [1] 1334   50
```

ou encore, dans un deuxième jeu de données, les patients qui ont une valeur de `creatinine` supérieure à la médiane:

```
data_medcreatinine<- data1[data1$creatinine> median(data1$creatinine), ]

dim(data_medcreatinine)
```

```
## [1] 1838   50
```

On voit que le premier jeu de données modifié contient 1334 lignes alors que le deuxième en contient 1838. Le jeu original en contenait plutôt 3676:

```
dim(data1)
```

```
## [1] 3676   50
```

Supposons maintenant qu'on veuille garder seulement les patients qui ont simultanément un `meanbp` supérieur à 84 ainsi qu'une valeur de `creatinine` supérieure à la médiane:

```
data_sup84_med_creat<- data1[ (data1$meanbp > 84) & (data1$creatinine> median(data1$creatinine))  , ]
dim(data_sup84_med_creat)
```

```
## [1] 670  50
```

Notez qu'on a utilisé le ET logique ici (`&`).

Finalement, utilisons l'opérateur ifelse pour créer une nouvelle variable à partir des variables dans `data1`:

```
data1$new_variable<- ifelse( data1$meanbp > 80, 'high_meanbp', 'low_meanbp')
table(data1$new_variable)
```

```
##
## high_meanbp  low_meanbp
##        1350        2326
```

Cette variable représente une version recodée de `meanbp` en deux catégories: soit au-dessus (`high_meanbp`) ou en-dessous de 80 (`low_meanbp`).

Boucles

La boucle peut être utile (par contre, notez qu'elle est souvent lourde, computationnellement parlant, c'est-à-dire qu'il ne s'agit pas de l'approche la plus efficace pour programmer).

On utilise la syntaxe `for(i in 1:1000){ }` qui signifie que la variable `i` ira de 1 à 1000 par des sauts de 1 unité et exécutera ce qu'il y a entre les accolades `{}` à chaque fois, pour chaque `i`.

Par exemple, on utilise parfois une boucle pour exécuter une étude de simulations qu'on répétera 1000 fois (par ex., on peut vouloir stocker des valeurs estimées à chaque boucle):

```
vecteur_des_moyennes<- c() # Créer un vecteur vide dans lequel entreposer des estimés plus tard

for(i in 1:1000){

  normale_var<- rnorm(n=100, mean=0, sd=1) # Je simule 100 réalisations d'une variable aléatoire de loi normale(0 ,1)

  mean0<- mean(normale_var) # Je note la moyenne

  vecteur_des_moyennes<- append(vecteur_des_moyennes, mean0) # J'ajoute (append) à la fin du vecteur vecteur_des_moyennes la moyenne nouvellement calculée. A la fin j'obtiens un vecteur de taille 1000 contenant 1000 valeurs de la moyenne mean0, une pour chaque simulation

}

length(vecteur_des_moyennes) # Taille
```

```
## [1] 1000
```

```
mean(vecteur_des_moyennes) # moyenne du vecteur
```

```
## [1] -0.001299343
```

```
sd(vecteur_des_moyennes)  # écart-type du vecteur (racine carrée de la variance)
```

```
## [1] 0.09970082
```

On utilise aussi parfois la boucle pour considérer chacune des personnes dans un jeu de données, une à une (et faire une opération sur chaque unité ou personne dans le jeu de données). Re-considérons le jeu de données `women` en R et le calcul de l'IMC:

```
data(women)
women$height
```

```
## [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
```

```
length(women$height)
```

```
## [1] 15
```

```
# Il semble dont y avoir 15 femmes dans ce jeu de données

# Notez que vous n'êtes pas obligés d'utiliser i pour la boucle,
#n'importe quelle nouvelle variable fait l'affaire (comme j ici)
imc_vecteur<-c() # initialiser le vecteur

for(j in 1:15){
  #J'choisi de l'indice de manne corporelle
  height<- (women$height)[j]*0.0254
  weight<- women$weight[j]* 0.45592 #metres en kg
  imc<- weight / height^2
  imc_vecteur<- append (imc_vecteur, imc)
}

summary(imc_vecteur) #fonction résumé (summary) nous montre le Q1, mediane, moyenne, Q3, min, max
```

```
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  22.18   22.22   22.48   22.73   23.14   24.63
```

Notez qu'ici, j'ai utilisé la variable `j=1,...,15` comme variable pour la boucle. Cela ne fait aucune différence. Par contre, il faut faire attention de ne pas utiliser d'autre variable `j` à l'intérieur de la boucle, juste celle reliée à la boucle. Par exemple, si vous avez une variable nommée `j` dans votre jeu de données, il est sûrement préférable de ne pas utiliser cette variable pour la boucle, mais plutôt, par ex., `k=1,...,15`.

Exercice

Utilisez le jeu de données `women` en R pour faire les exercices suivants. Vous pouvez télécharger le jeu de données dans R grâce à la commande suivante:

```
data(women)
```

1. Listez le type de chaque variable contenue dans le jeu de données `women` (c'est-à-dire, listez son format). De quel format s'agit-il? Qu'est-ce que cela veut dire?
2. En utilisant votre façon préférée (boucle ou fonction `apply`), créez une troisième variable que vous enregistrerez à l'intérieur du jeu de données `women` et qui représente l'indice de masse corporelle (IMC) (note, vous pouvez ajouter cette variable au jeu de données `women` grâce à la commande `women$nouvellevariable<- ...`, où vous remplacez les trois petits points par la nouvelle variable créée). Montrez ensuite un résumé de cette nouvelle variable grâce à la fonction `summary()`.
3. Créez une quatrième variable dans le jeu de données `women` qui représente une catégorisation de l'IMC en les catégories suivantes:
   - <18.5 (catégorie 1)
   - 18.5-24.9 (catégorie 2)
   - ≥ 25 (catégorie 3)