Structures de contrôle en R

Sophie Baillargeon, Université Laval

2020-02-28

Table des matières

Alternatives **2**
Structure `if ... else` . . . 2
Écriture générale d'un `if ... else` . . . 2
Écriture condensée d'un `if ... else` . . . 4
Distinction entre une structure `if ... else` et la fonction `ifelse` . . . 4
Fonction `switch` . . . 5

Boucles **6**
Boucles `for` . . . 6
Enregistrement des résultats dans une boucle . . . 9
Affichage de résultats dans une boucle . . . 10
Boucles `while` ou `repeat` . . . 10
Imbriquer des boucles . . . 11
Mots-clés `break` et `next` . . . 12
Interruption de l'exécution d'une boucle . . . 13
Éviter les boucles . . . 13
Calcul vectoriel versus boucle . . . 13
Fonction de la famille des `apply` versus boucle . . . 13

Résumé **14**

Références **16**

La matière vue jusqu'à maintenant dans le cours traitait principalement de l'utilisation du logiciel R dans le but de faire de la manipulation ou de l'analyse de données. Cette utilisation passe par la soumission de commandes dans la console R. En fait, souvent plus d'une commande est nécessaire pour produire le résultat escompté. Plutôt que de soumettre une après l'autre plusieurs commandes dans la console, nous avons appris qu'il était préférable d'écrire des programmes. Renommons « instructions » les commandes apparaissant dans un programme. Lorsqu'un programme R entier est soumis, les instructions qui le composent sont exécutées séquentiellement, c'est-à-dire l'une après l'autre, en respectant leur ordre d'apparition dans le programme.

Comme presque tout langage informatique qui adhèrent au paradigme de programmation impératif, R offre des structures de contrôles (aussi appelées séquencements). Les structures de contrôle sont des instructions particulières qui contrôlent l'ordre dans lequel d'autres instructions d'un programme informatique sont exécutées. Les appels de fonction, présents dans pratiquement toutes les instructions R étudiées dans ce cours jusqu'à maintenant, sont des structures de contrôle. Elles produisent un saut dans l'exécution des instructions d'un programme vers un sous-programme (le corps de la fonction), suivi d'un saut de retour vers le programme principal.

Ce document décrit l'utilisation en R des deux autres structures de contrôle les plus courantes en programmation impérative : les alternatives (structures conditionnelles) et les boucles (structures itératives).

Alternatives

Les alternatives ont pour but d'exécuter des instructions seulement si une certaine condition est satisfaite. Voyons ici deux outils pour créer des alternatives en R : la structure `if ... else` et la fonction switch.

Structure `if ... else`

Écriture générale d'un `if ... else`

Les mots-clés pour écrire des alternatives en R sont `if` et `else`. De façon générale, la syntaxe d'une structure de contrôle `if ... else` est la suivante.

```
if (condition) {
  instructions  # exécutées si l'évaluation de condition retourne TRUE
} else {
  instructions  # exécutées si l'évaluation de condition retourne FALSE
}
```

Il est possible d'avoir un `if` sans `else`.

```
if (condition) {
  instructions  # exécutées si l'évaluation de condition retourne TRUE
}
```

Un `if` doit être suivi d'une paire de parenthèses dans laquelle est inséré une instruction R retournant une seule valeur logique (`TRUE` ou `FALSE`). C'est la condition de l'alternative. Ensuite viennent la ou les instructions à exécuter si la condition est vraie (c'est-à-dire si l'instruction `condition` produit le résultat `TRUE`). S'il y a plus d'une instruction à exécuter, les accolades sont nécessaires pour les encadrer. Pour une seule instruction, les accolades sont optionnelles.

Voici un exemple :

```
# Simulation du lancer d'une pièce de monnaie
lancer <- sample(x = c("Pile", "Face"), size = 1)

# Structure qui affiche ou non un message, en fonction du résultat du lancer
if (lancer == "Pile")    # sans accolades
  print("Je gagne!")
# ou encore
if (lancer == "Pile") {  # avec accolades
  print("Je gagne!")
}
```

Lorsqu'il y a des instructions à exécuter si la condition est fausse, il faut ajouter un `else` à l'alternative, suivi des instructions en question. Dans ce cas, il est considéré comme une bonne pratique de toujours encadrer les blocs d'instructions d'accolades (sauf si l'écriture condensée, qui sera présentée plus loin, est utilisée), même s'ils sont composés d'une seule instruction, de façon à retrouver le mot-clé `else` précédé de `}` et suivi de `{` .

Voici un exemple :

```
if (lancer == "Pile") {
  print("Je gagne!")
} else {
  print("Je perds...")
}
```

Plusieurs structures `if ... else` peuvent être imbriquées. Pour ce faire, il suffit d'insérer une autre structure `if ... else` à la place de l'accolade suivant le dernier `else`, comme suit.

```
if (condition_1) {
  instructions  # exécutées si condition_1 retourne TRUE
} else if (condition_2) {
  instructions  # exécutées si condition_1 retourne FALSE, mais condition_2 retourne TRUE
} else {
  instructions  # exécutées si condition_1 et condition_2 retournent FALSE
}
```

Notons que la syntaxe précédente est préférée à la suivante, qui est équivalente mais plus lourde.

```
# Syntaxe non allégée de deux structures if ... else imbriquées
if (condition_1) {
  instructions  # exécutées si condition_1 retourne TRUE
} else {
  if (condition_2) {
    instructions  # exécutées si condition_1 retourne FALSE, mais condition_2 retourne TRUE
  } else {
    instructions  # exécutées si condition_1 et condition_2 retournent FALSE
  }
}
```

Voici un exemple :

```
x <- iris$Sepal.Length

# Programme qui calcule des statistiques descriptives simples, selon
# le type des éléments du vecteur sur lequel le calcul est fait
if (is.numeric(x)) {
  c(min = min(x), moy = mean(x), max = max(x))
} else if (is.character(x) || is.factor(x)) {
  table(x)
} else {
  NA
}
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

```
# Faisons rouler les instructions de nouveau, après avoir redéfini le vecteur x.

x <- iris$Species

if (is.numeric(x)) {
  c(min = min(x), moy = mean(x), max = max(x))
} else if (is.character(x) || is.factor(x)) {
  table(x)
} else {
  NA
}
```

```
## x
##     setosa versicolor  virginica
##         50         50         50
```

Il serait pratique de créer une fonction à partir de ce bout de code. Nous le ferons dans les notes sur les fonctions en R.

Écriture condensée d'un `if ... else`

Lorsque, dans chaque branche d'une alternative `if ... else`, il n'y a seulement une instruction courte servant à créer un seul objet, l'écriture condensée suivante peut être pratique :

```
nom <- if (condition) instruction else instruction
```

Cette écriture est recommandée seulement si elle rend le code plus lisible pour des alternatives très simples.

Voici un exemple :

```
message <- if (lancer == "Pile") "Je gagne!" else "Je perds..."
```

Distinction entre une structure `if ... else` et la fonction `ifelse`

Sous sa forme condensée, une structure `if ... else` fait penser à un appel à la fonction `ifelse`. Quelles sont les différences entre les deux ?

Premièrement, la condition dans une structure `if ... else` doit être une instruction qui retourne un seul `TRUE` ou un seul `FALSE`, pas un vecteur logique de longueur supérieure à 1. Si la condition est un vecteur logique de longueur supérieure à 1, seul le premier élément est utilisé et un avertissement est affiché comme dans l'exemple suivant :

```
x <- 1:10
xmodif <- if (x > 2.5 & x < 7.5) 5 else x
```

```
## Warning in if (x > 2.5 & x < 7.5) 5 else x: the condition has length > 1 and
## only the first element will be used
```

```
xmodif
```

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

Si nous avions utilisé l'opérateur logique `&&` au lieu de `&`, nous n'aurions pas obtenu cet avertissement, mais exactement le même résultat aurait été obtenu.

```
xmodif <- if (x > 2.5 && x < 7.5) 5 else x
xmodif
```

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

> **Les opérateurs logiques `&&` et `||` sont utiles pour écrire une condition d'un `if`, car ils assurent la production d'une seule valeur logique.**

Cependant, dans cet exemple, la structure de contrôle `if ... else` n'est probablement pas ce que nous voulions utiliser. Si notre but est de vérifier pour chaque élément de `x` si la valeur est comprise entre 2.5 et 7.5, si c'est le cas retourner la valeur 5, sinon retourner l'élément de `x` inchangé, alors c'est la fonction `ifelse` que nous devrions utiliser.

```
xmodif <- ifelse(test = x > 2.5 & x < 7.5, yes = 5, no = x)
xmodif
```

```
##  [1]  1  2  5  5  5  5  5  8  9 10
```

La fonction `ifelse` agit de façon vectorielle. Elle teste une condition sur tous les éléments d'un objet et retourne une valeur en fonction du résultat. La dimension de la sortie d'un `ifelse` est la même que la dimension du premier argument qu'elle reçoit.

Ainsi, la fonction `ifelse` n'est pas une structure de contrôle.

Fonction `switch`

La fonction switch est parfois utile pour remplacer plusieurs structures `if ... else` imbriquées. La syntaxe générale d'un appel à la fonction `switch` est la suivante.

```
switch(
  expression,
  "resultat_1" = {
    instructions # exécutées si expression retourne "resultat_1"
  },
  "resultat_2" = {
    instructions # exécutées si expression retourne "resultat_2"
  },
  .
  .  # autres paires (résultat, instructions à exécuter) s'il y a lieu
  .
  {
    instructions # exécutées si expression retourne tout autre résultat
  }
)
```

Dans cette syntaxe générale, la valeur fournie au premier argument, représentée par `expression`, doit être une instruction R retournant une chaîne de caractères. Les autres arguments de la fonction doivent porter les noms de ce que peut produire en sortie `expression`. Les valeurs fournies à ces arguments sont les instructions à exécuter si `expression` retourne une chaîne de caractères égale au nom de l'argument. Si un dernier argument non assigné à un nom est fourni, il sera exécuté si `expression` retourne une chaîne de caractères ne se retrouvant pas parmi les noms d'arguments présents dans l'appel à la fonction `switch`. Notons que lorsqu'un résultat doit provoquer l'exécution d'une seule instruction, celle-ci n'a pas à être encadrée d'accolades.

Voici un exemple :

```
x <- iris$Sepal.Length

# Structures if ... else imbriquées présentées précédemment, à reproduire
if (is.numeric(x)) {
  c(min = min(x), moy = mean(x), max = max(x))
} else if (is.character(x) || is.factor(x)) {
  table(x)
} else {
  NA
}
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

```
# Appel à la fonction switch équivalent
switch(
  class(x),
  "numeric"   = c(min = min(x), moy = mean(x), max = max(x)),
  "integer"   = c(min = min(x), moy = mean(x), max = max(x)),
  "character" = table(x),
  "factor"    = table(x),
  NA
)
```

```
##      min      moy      max
## 4.300000 5.843333 7.900000
```

Dans cet exemple, `expression` est l'instruction `class(x)`. Celle-ci retourne `"numeric"` pour un vecteur `x` contenant des données de type réel, mais retourne `"integer"` pour un vecteur `x` contenant des données de type entier. La condition `is.numeric(x)` retourne quant à elle `TRUE` pour tout vecteur numérique `x`, que ses données soient réelles ou entières. Afin de créer un appel à la fonction `switch` équivalent aux structures if ... else imbriquées à reproduire, il fallait donc définir les résultats `"numeric"` et `"integer"`. Les deux solutions implémentent la même alternative aussi parce que `is.character(x)` est équivalent à `class(x) == "character"` et `is.factor(x)` est équivalent à `class(x) == "factor"`.

Notons que `expression` peut aussi retourner un entier, interprété comme le numéro du bloc d'instructions à exécuter. Par exemple, si `expression` produit le résultat `2` lorsqu'exécuté, c'est le deuxième bloc d'intructions (troisième argument fourni à la fonction `switch`) qui sera exécuté. Dans ce cas, les blocs d'instructions n'ont pas besoin d'être assignés à des noms.

Boucles

Les boucles ont pour but de répéter des instructions à plusieurs reprises, c'est donc dire de les itérer. Parfois, le nombre d'itérations à effectuer est connu d'avance. D'autres fois, ce nombre d'itérations n'est pas connu d'avance, car il dépend d'une condition à rencontrer.

Boucles `for`

Lorsque le nombre d'itérations à effectuer est prédéterminé, une boucle `for` est tout indiquée.

Écriture générale d'une boucle `for` :

```
for (itérateur in ensemble) {
  instructions  # exécutées à chaque itération de la boucle
}
```

Ce type de boucle débute par le mot clé `for`, suivi des éléments suivants, dans l'ordre :

- une parenthèse ouvrante ;
- un nom quelconque, représenté par `itérateur` dans la syntaxe générale ;
- le mot-clé `in` ;
- une instruction retournant un vecteur contenant l'ensemble des valeurs sur lesquelles itérer, représenté par `ensemble` dans la syntaxe générale ;
- une parenthèse fermante.

Ensuite viennent la ou les instructions à répéter. S'il y a plus d'une instruction à répéter, les accolades sont nécessaires pour les encadrer. Dans ces instructions, l'objet nommé `itérateur` dans la syntaxe générale intervient généralement.

La boucle effectue autant de répétitions que la longueur du vecteur `ensemble`.

- Première itération : `itérateur` prend la valeur `ensemble[[1]]`.
- Deuxième itération : `itérateur` prend la valeur `ensemble[[2]]`.
- ⋮
- Dernière itération : `itérateur` prend la valeur `ensemble[[length(ensemble)]]`.

Ainsi, de façon générale, pour les itérations `i` allant de `1` à `length(ensemble)`, `valeur` contient `ensemble[[i]]`.

Voici un exemple :

```
for (lettre in LETTERS) {
  cat(lettre, " ")
}
```

```
## A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Dans cet exemple, nous avons effectué 26 itérations, car `length(LETTERS) == 26`. À l'itération `i`, nous avons affiché le i^e élément du vecteur `LETTERS`, soit la i^e lettre de l'alphabet.

Nous aurions pu effectuer exactement la même boucle en itérant sur les entiers de 1 à 26 comme suit :

```
for (i in seq_along(LETTERS)) {
  cat(LETTERS[[i]], " ")
}
```

```
## A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Lorsque nous choisissons d'itérer sur les entiers allant de 1 au nombre total d'itérations à effectuer (disons `n`), il est commun d'utiliser le nom `i` pour l'objet changeant de valeur au fil des itérations. Le vecteur `ensemble` est alors souvent créé par l'instruction `1:n`, mais il est plus prudent d'utiliser `seq_len(n)` qui retournera une erreur si `n` est négatif ou un vecteur vide si `n == 0`. Si le nombre d'itérations à effectuer `n` est égal à la longueur d'un objet, disons `a`, il est recommandé de créer le vecteur `ensemble` par l'instruction `seq_along(a)`. Pour un objet `a` de longueur non nulle, les instructions `1:length(a)` et `seq_along(a)` retournent exactement le même résultat, comme l'illustre cet exemple.

```
1:length(LETTERS)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

```
seq_along(LETTERS)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Cependant, si l'objet `a` s'adonne à être de longueur nulle, l'utilisation de `seq_along(a)` nous assure que la boucle n'effectue aucune itération. L'utilisation de `1:length(a)` entraînerait plutôt une itération sur `i = 1`, puis `i = 0`, qui risquerait de générer une erreur. Ce comportement s'explique par le fait que `1:length(a)` retourne dans ce cas particulier le vecteur (1, 0).

```
a <- list()  # supposons que a est une liste vide
1:length(a)
```

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

```
seq_along(a)
```

```
## integer(0)
```

Autre exemple :

Voici un exemple de boucle `for`, utilisant le jeu de données `attitude` (provenant du package `datasets`)

```
str(attitude)
```

```
## 'data.frame':    30 obs. of  7 variables:
##  $ rating    : num  43 63 71 61 81 43 58 71 72 67 ...
##  $ complaints: num  51 64 70 63 78 55 67 75 82 61 ...
##  $ privileges: num  30 51 68 45 56 49 42 50 72 45 ...
##  $ learning  : num  39 54 69 47 66 44 56 55 67 47 ...
##  $ raises    : num  61 63 76 54 71 54 66 70 71 62 ...
##  $ critical  : num  92 73 86 84 83 49 68 66 83 80 ...
##  $ advance   : num  45 47 48 35 47 34 35 41 31 41 ...
```

Ce jeu de données contient 7 variables numériques. Ces données ont été recueillies dans le but d'étudier les variables influençant la cote (`rating`) reçue par 30 départements d'une grande organisation financière. Supposons que nous souhaitons réaliser 6 régressions linéaires simples sur ces données. Toutes les régressions

auraient la même variable réponse, `rating`, et la variable explicative devrait être tour à tour une des autres variables du jeu de données.

L'instruction pour réaliser la régression simple avec la variable `complaints` par exemple, serait la suivante :

```
lm(rating ~ complaints, data = attitude)
```

```
# ou
lm(rating ~ ., data = attitude[, c(1, 2)])
```

```
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, 2)])
##
## Coefficients:
## (Intercept)   complaints
##     14.3763       0.7546
```

Nous souhaitons maintenant insérer cette instruction dans une boucle permettant d'effectuer les 6 régressions simples.

```
modeles <- vector(length = ncol(attitude) - 1, mode = "list")
for (i in seq_len(ncol(attitude) - 1)) {
  modeles[[i]] <- lm(rating ~ ., data = attitude[, c(1, i + 1)])
}
modeles
```

```
## [[1]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)   complaints
##     14.3763       0.7546
##
##
## [[2]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)   privileges
##     42.1087       0.4239
##
##
## [[3]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)     learning
##     28.1741       0.6468
##
##
```

```
## [[4]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)       raises
##     19.9778       0.6909
##
##
## [[5]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)     critical
##     50.2446       0.1924
##
##
## [[6]]
##
## Call:
## lm(formula = rating ~ ., data = attitude[, c(1, i + 1)])
##
## Coefficients:
## (Intercept)      advance
##     56.7558       0.1835
```

Enregistrement des résultats dans une boucle

Une affectation de valeur à un endroit précis d'un objet (ex. : `modeles[[i]] <- lm(...)`) nécessite que l'objet existe préalablement. Ainsi, une boucle est souvent précédée par l'initialisation d'un objet dédié à contenir les résultats calculés dans la boucle. Dans l'exemple précédent, nous avons initialisé la liste `modeles` avant la boucle par l'instruction :

```
modeles <- vector(length = ncol(attitude) - 1, mode = "list")
```

Remarquons que la fonction `vector` crée bien une liste ici, et non un vecteur, grâce à l'argument `mode = "list"`. Après tout, les listes sont des vecteurs récursifs.

Nous aurions pu choisir d'itérer sur les noms des variables explicatives plutôt que sur les entiers 1 à 6, comme suit :

```
modeles <- vector(length = ncol(attitude) - 1, mode = "list")
names(modeles) <- names(attitude)[-1]

for (variable in names(modeles)) {
  modeles[[variable]] <- lm(rating ~ ., data = attitude[, c("rating", variable)])
}
```

Dans ce cas, nous avons préalablement nommé les éléments de la liste initialement vide. Ainsi, dans la boucle, nous pouvons référer à des éléments spécifiques de la liste `modeles` par leur nom plutôt que par leur indice.

Affichage de résultats dans une boucle

Voici une boucle très simple.

```
for (i in seq_len(5)) {
  i
}
```

Si vous soumettez cette boucle, vous remarquerez qu'elle n'affiche rien. Pourtant, une instruction contenant uniquement le nom d'un objet affiche cet objet lorsque l'instruction est soumise dans la console.

```
i
```

```
## [1] 5
```

Ce résultat ne se produit pas dans une boucle. Il faut utiliser les fonctions `print` ou `cat` pour qu'un résultat soit affiché dans la console pendant l'exécution d'une boucle.

```
for (i in seq_len(5)) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
for (i in seq_len(5)) {
  cat(i)
}
```

```
## 12345
```

`cat` est utile pour faire afficher une trace des itérations.

```
for (i in seq_len(5)) {
  cat("itération", i, "terminée\n")
}
```

```
## itération 1 terminée
## itération 2 terminée
## itération 3 terminée
## itération 4 terminée
## itération 5 terminée
```

Rappel : Le caractère `\n` représente un retour à la ligne.

Boucles `while` ou `repeat`

Parfois, le nombre d'itérations dépend d'une condition à rencontrer, il n'est pas prédéterminé. Les boucles R `while` et `repeat` sont utiles dans cette situation.

Écriture générale d'une boucle `while` :

```
while (condition) {
  instructions  # exécutées à chaque itération de la boucle
}
```

Écriture générale d'une boucle `repeat` :

```
repeat {
  instructions  # exécutées à chaque itération de la boucle
  if (!condition) {
    break
  }
}
```

Un des intérêts d'une boucle `repeat` est de tester la condition après avoir exécuté les instructions et non avant comme dans une boucle `while`. Dans un boucle `repeat`, le mot-clé `break` doit être utilisé pour mettre fin aux itérations, sinon la boucle est infinie.

La `condition` doit encore une fois être une instruction qui retourne une seule valeur logique (`TRUE` ou `FALSE`).

Dans les écritures générales ci-dessus, remarquez qu'il y a un opérateur logique de négation devant la condition dans la boucle `repeat`. C'est pour mettre en évidence le fait qu'une boucle `while` continue d'itérer tant que `condition` demeure `TRUE`. Pour une même `condition`, il faut donc faire arrêter la boucle `repeat` lorsque `condition` devient `FALSE`.

Voici un exemple.

Nous souhaitons simuler le lancer d'un dé jusqu'à l'obtention d'un 6 et compter le nombre de lancers.

```
resultat <- 1   # initialisation à un résultat quelconque, différent de 6
n_lancers <- 0  # initialisation à 0 du nombre de lancer
while (resultat != 6) {              # tant que le résultat n'est pas égal à 6, répéter
  resultat <- sample(1:6, size = 1)  # simulation du lancer du dé
  n_lancers <- n_lancers + 1         # incrémentation du nombre de lancers
}
n_lancers       # afficher le résultat final
```

La boucle `while` peut être remplacée par une boucle `repeat` avec le mot-clé `break` comme suit.

```
n_lancers <- 0
repeat {
  resultat <- sample(1:6, size = 1)
  n_lancers <- n_lancers + 1
  if (resultat == 6) {
    break
  }
}
n_lancers
```

Ici, nous n'avons pas besoin d'initialiser `resultat`, car la condition est évaluée à la fin de la boucle, après avoir calculé `resultat` au moins une fois.

Remarque : Si les insctuctions dans une boucle `while` ou `repeat` n'ont aucun impact sur la `condition` et que celle-ci demeure toujours vraie, alors la boucle est infinie. Il est important de s'assurer que la `condition` devienne éventuellement fausse, afin que la boucle puisse s'arrêter.

Imbriquer des boucles

Dans l'exemple précédent, il serait intéressant de répéter l'expérience un grand nombre de fois et de calculer le nombre moyen de lancers requis pour obtenir un 6. Pour ce faire nous pourrions imbriquer la boucle `while` ou `repeat` dans une boucle `for` comme suit :

```r
n_rep <- 10000
n_lancers <- rep(0, n_rep)  # ou vector(length = n_rep, mode = "numeric")
for (i in 1:n_rep) {
  resultat <- 1
  while (resultat != 6) {
    resultat <- sample(1:6, size = 1)
    n_lancers[i] <- n_lancers[i] + 1
  }
}
mean(n_lancers)
```

```
## [1] 5.9225
```

Cet exemple montre une façon empirique d'estimer l'espérance d'une variable aléatoire suivant une distribution géométrique de paramètre $p = 1/6$. Plus grand est le nombre de répétitions, plus l'estimation est précise (convergence). En théorie, cette espérance vaut $1/p = 6$.

Il est simple d'imbriquer des boucles en R, peu importe leur type (`for`, `while` ou `repeat`). Cependant, nous verrons plus tard que plus l'imbrication possède de niveaux, plus le programme tend à être long à exécuter.

Mots-clés `break` et `next`

Deux mots-clés existent pour contrôler l'exécution des instructions à l'intérieur d'une boucle :

- `break` : pour terminer complètement l'exécution de la boucle (les itérations restantes ne sont pas effectuées).
- `next` : pour terminer immédiatement une itération (sans exécuter les instructions après le mot-clé `next`) et reprendre l'exécution de la boucle à la prochaine itération.

Ces deux mot-clés sont pratiquement toujours utilisés dans une structure `if`.

Le mot-clé `break` a déjà été illustré dans une boucle `repeat`. Notons cependant que nous pouvons l'utiliser dans une boucle de n'importe quel type.

Illustrons maintenant l'utilisation du mot-clé `next`. Reprenons l'exemple de l'affichage des lettres de l'alphabet. Supposons que nous souhaitons afficher seulement les consonnes.

```r
for (lettre in LETTERS) {
  if (lettre %in% c("A", "E", "I", "O", "U")) {
    next
  }
  cat(lettre, " ")
}
```

```
## B  C  D  F  G  H  J  K  L  M  N  P  Q  R  S  T  V  W  X  Y  Z
```

Dans ce programme, si la condition `lettre %in% c("A", "E", "I", "O", "U")` est rencontrée, nous passons à l'itération suivante de la boucle, sans soumettre l'instruction `cat(lettre, " ")`. Le mot-clé `next` permet donc d'omettre l'exécution de certaines instructions.

En fait, le dernier programme fait la même chose que le programme suivant.

```r
for (lettre in LETTERS) {
  if (! lettre %in% c("A", "E", "I", "O", "U")) {
    cat(lettre, " ")
  }
}
```

```
## B  C  D  F  G  H  J  K  L  M  N  P  Q  R  S  T  V  W  X  Y  Z
```

Ici, il n'y a plus de mot-clé `next`, mais l'instruction `cat(lettre, " ")` est dans le `if` plutôt qu'après le `if`.

Nous avons souvent constaté qu'il y a plusieurs façons de réaliser une même tâche en R. Cette remarque est aussi vraie pour les boucles.

Interruption de l'exécution d'une boucle

Il peut arriver que, par erreur, nous soumettions en R une boucle vraiment longue à rouler, possiblement infinie. En RStudio, l'exécution de n'importe quelle commande, incluant une boucle, peut être interrompue d'une des façons suivantes :

- la touche « Esc »,
- le bouton STOP en entête à droite de la sous-fenêtre de la console R (dans un petit logo de panneau d'arrêt rouge de forme octogonale),
- le menu « Session > Interrupt R ».

Éviter les boucles

Une des philosophies de base en programmation R est d'utiliser une boucle seulement si celle-ci est vraiment nécessaire pour réaliser la tâche à accomplir. Le fonctionnement vectoriel de plusieurs fonctions R, ainsi que les fonctions de la famille des `apply`, permettent bien souvent d'éviter l'utilisation d'une boucle.

Cette philosophie se base sur les faits suivants :

- le calcul vectoriel est plus rapide que les boucles ;
- éviter les boucles produit habituellement un code plus court, donc plus rapide à écrire et aussi potentiellement plus facile à comprendre.

Calcul vectoriel versus boucle

Voici un exemple simple d'opération vectorielle. Supposons que nous avons le vecteur numérique `x` suivant.

```
x <- 1:10
```

Nous voulons élever au carré toutes les valeurs dans ce vecteur. En R, il est recommandé de réaliser cette tâche comme suit :

```
z <- x^2
z
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Dans bien des langages informatiques, il aura fallu faire une boucle, telle que celle-ci :

```
z <- vector(length = length(x), mode = "numeric")
for (i in seq_along(x)) {
  z[i] <- x[i]^2
}
z
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Laquelle des deux solutions vous paraît la plus simple à comprendre ?

Fonction de la famille des `apply` versus boucle

Voici un exemple simple d'utilisation d'une fonction de la famille des `apply`. Supposons que nous avons la matrice numérique `mat` suivante.

```
mat <- matrix(1:12, ncol = 3, nrow = 4)
mat
```

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

Nous voulons calculer les sommes des valeurs par colonne. En R, il est recommandé de réaliser cette tâche comme suit :

```
sommesColonnes <- colSums(mat)
sommesColonnes
```

```
## [1] 10 26 42
```

La fonction `colSums` revient à un appel à la fonction `apply` optimisé pour la tâche spécifique du calcul de sommes en colonnes.

Dans bien des langages informatiques, il aurait fallu faire une boucle, telle que celle-ci :

```
sommesColonnes <- vector(length = ncol(mat), mode = "numeric")
for (i in 1:ncol(mat)){
  sommesColonnes[i] <- sum(mat[, i])
}
sommesColonnes
```

```
## [1] 10 26 42
```

En termes de temps de travail requis pour écrire le code, il est plus rapide d'appeler la fonction `colSums` que d'écrire cette boucle. Aussi, le code avec l'appel à la fonction `colSums` est plus succinct, donc potentiellement plus simple à comprendre.

Résumé

Alternatives

Structure `if ... else`

```
if (condition) {
 instructions  # exécutées si condition est TRUE
} else {
 instructions  # exécutées si condition est FALSE
}
```

- `condition` = instruction qui retourne *un seul logique* (`TRUE` ou `FALSE`).

Écriture condensée d'une alternative :

`x <- if (condition) instruction else instruction`

- Ce n'est pas la même chose que la fonction `ifelse`, qui travaille de façon vectorielle.

Fonction `switch`

```
switch(
  expression,
  "resultat_1" = {
    instructions # exécutées si expression retourne "resultat_1"
  },
  "resultat_2" = {
    instructions # exécutées si expression retourne "resultat_2"
  },
  .
  .  # autres paires (résultat, instructions à exécuter) s'il y a lieu
  .
  {
    instructions # exécutées si expression retourne tout autre résultat
  }
)
```

Boucles

```
for(valeur in ensemble) {
  instructions
}
```

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

```
repeat {
  instructions
  if (!condition) {
    break
  }
}
```

- `for` : boucle ayant un nombre prédéterminé d'itérations,
- `while` : boucle arrêtant lorsqu'une condition n'est plus rencontrée,
- `repeat` : boucle nécessitant le mot-clé `break` pour arrêter,
- `break` : mot-clé pour terminer l'exécution de la boucle,
- `next` : mot-clé pour sauter à la prochaine itération sans exécuter les instructions après le mot-clé.

À noter :

- Une boucle est souvent précédée par l'initialisation d'un objet dédié à contenir les résultats calculés dans la boucle.
- Il faut utiliser les fonctions `print` ou `cat` pour qu'un résultat soit affiché dans la console pendant l'exécution d'une boucle.
- Il est recommandé d'utiliser des boucles seulement si nécessaire, de favoriser les calculs vectoriels et l'utilisation des fonctions de la famille des `apply`.

Références

- Matloff, N. (2011). The Art of R Programming : A Tour of Statistical Software Design. No Starch Press. Chapitre 7.
- Wickham, H. et Grolemund, G. (2016). R for Data Science. O'Reilly Media, Inc. :
  - Alternatives : Section 19.4 http://r4ds.had.co.nz/functions.html#conditional-execution
  - Boucles : Chapitre 21 http://r4ds.had.co.nz/iteration.html
- Sections du tutoriel https://www.datamentor.io/r-programming :
  - Alternatives : https://www.datamentor.io/r-programming/if-else-statement
  - Boucles `for` : https://www.datamentor.io/r-programming/for-loop
  - Boucles `while` : https://www.datamentor.io/r-programming/while-loop
  - Boucles `repeat` : https://www.datamentor.io/r-programming/repeat-loop
- Fanara, C. (2018). Tutoriel web intitulé « A Tutorial on Loops in R - Usage and Alternatives ». https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r