TD n°4 - Appréhender la logique algorithmique avec R

Joris FALIP & Morgan COUSIN

Année Universitaire 2018-2019

Contents

Objectifs de séance

A l'issue de ce cours, vous devez être capables de :

- Utiliser les principaux opérateurs arithmétiques et logiques
- Comprendre le fonctionnement d'une fonction
- Utiliser une fonction et ses arguments
- Rechercher le détail d'une fonction dans l'aide
- Utiliser les principales fonctions de base
- Manipuler quelques fonctions avancées
- Construire et Utiliser votre propre fonction
- Construire et Utiliser les blocs de code conditionnels (**if**, **elseif** et **else**)
- Utiliser la fonction **ifelse()**, notamment pour recoder des variables
- Construire et Utiliser une boucle **for**

Rappel des opérateurs arithmétiques et logiques

Table 1: Liste des opérateurs arithmétiques et logiques à connaître

Opérateur	Signification
+	Addition
-	Soustraction
étoile	Multiplication
/	Division
double étoile ou chapeau : ^	Puissance
%%	Modulo
%/%	Division entière
:	Générer une suite
<	Plus petit que
<=	Plus petit ou égale
>	Plus grand que
>=	Plus grand ou égale
==	Egale à
!=	Différent de
&	ET
|	OU

Le tableau ci-dessus rappelle les opérateurs mathématiques et logiques les plus fréquemment employés. Il est impératif de les connaître.

Les fonctions

Construire un algorithme est un bien grand mot. En effet, dès lors qu'une commande est exécutée, des opérations sont effectuées. Il s'agit là, déjà, d'algorithme. Aussi, les opérateurs arithmétiques et logiques rappelés ci-avant, sont en réalité des *"raccourcis"* permettant d'appeler des commandes algorithmiques, élémentaires certes, mais algorithmiques tout de même. Pour toutes les autres opérations, avec un niveau de complexité supérieur, il est nécessaire d'appeler une **fonction**. Dans les séances précédentes, nous avons utilisé certaines d'entre elles, à la fois simples et déjà implémentées dans *R*. Nous allons voir dans cette section comment se structure une fonction de manière générale et nous verrons aussi comment construire une fonction *ad hoc*, c'est-à-dire construite par vos soins en fonction de votre besoin.

La structure élémentaire d'une fonction

Une fonction s'écrit de manière générale d'une seule et unique façon : **action(argument)**

Appeler une *fonction* signifie appeler une série de commandes spécifiques à exécuter selon certains paramètres dits *arguments*.

Le ou les argument(s) sont toujours localisés à l'intérieur des parenthèses. On fera remarquer qu'il n'y a pas de limite au nombre d'arguments que peut avoir une fonction. Les arguments sont séparés par une virgule.

Attention ! : Les arguments d'une fonction ont un ordre spécifique ! Il s'agit de l'ordre dans lequel ils apparaissent dans la définition de la fonction. C'est pourquoi, il est plus prudent de spécifier le nom de l'argument lors de l'appel de la fonction. Pour cela, on place entre parenthèse : **nom_argument = valeur**. Ce conseil est d'autant plus pertinent si la fonction possède plusieurs arguments et que vous ne les utilisez pas tous ou que vous ne les utilisez pas forcément dans l'ordre.

Exemple :

Souvenez-vous du cours sur les matrices, nous avons vu la fonction **matrix()**. Si vous reprenez votre cours, ou que vous saisissez matrix dans l'onglet **Help**, voici ce que nous voyons concernant la fonction **matrix()**:

- **data =** : les données à enregistrer dans la matrice. Par défaut : *data = NA* (valeur manquante).
- **nrow =** : le nombre de lignes à créer. Par défaut : *nrow = 1.*
- **ncol =** : le nombre de colonnes à créer. Par défaut : *ncol = 1.*
- **byrow =** : (Saisir **TRUE** ou **FALSE**) *R* fonctionne en mode 'colonne'. Cela signifie qu'il va d'abord remplir la première colonne, puis la deuxième, etc... Dans notre conception naturelle, notre cerveau complète ligne par ligne. C'est pourquoi **byrow = T** permet de dire à *R* de compléter ligne après ligne. Par défaut : *byrow = FALSE.*
- **dimnames =** : Nommer les dimensions de la matrice (il y en a 2). Que désigne les lignes? les colonnes? Par défaut : *dimnames = NULL.*

```
# Appeler la fonction matrix juste avec les arguments par défaut
matrix()
```

```
##      [,1]
## [1,]   NA
```

Comme convenu dans les paramètres par défaut, nous avons créé une matrice de dimension 1 x 1, contenant un **NA** (valeur manquante) et les colonnes et lignes n'ont pas de labels.

Voici un exemple d'utilisation de la fonction **matrix()** avec des arguments spécifiés et dans un ordre différent par rapport à celui de la définition.

```
# Création d'une matrice test avec des arguments spécifiés
# dans un ordre différents de la définition

mat_test <- matrix(byrow = T,
                   data = c(2:9),
                   ncol = 4,
                   nrow = 2)
```

Il s'agit là d'un exemple tout simple, sans signification particulière. Cependant, nous avons la preuve que peu importe l'ordre des arguments, si les noms sont spécifiés, nous pouvons utiliser la fonction sans souci. Les arguments non utilisés (ici *dimnames =*) prennent la valeur par défaut (ici *dimnames = NULL*).

Aide pratique : pour connaitre rapidement la liste des arguments disponibles, vous pouvez placer votre curseur après la parenthèse ou une virgule et appuyer sur **TAB**.

Bonne pratique : Quand les arguments commencent à être nombreux, pour faciliter la lecture nous allons à la ligne pour définir l'argument suivant. Nous avons de suite une vision très simple des arguments utilisés et des valeurs données.

Help : Quand R a réponse à tout...

Si vous vous souvenez des premiers conseils donnés lors du premiers cours, il a été dit qu'un codeur a le sens de la débrouille. Aussi, il y a un réflex à avoir quand vous utilisez une fonction pour la première fois : Trouver son utilité et ses arguments. Pour cela, plusieurs solutions :

- La fonction **help()** dans la console. Il existe en effet une fonction dont l'argument est un nom de fonction et qui retourne le détail de la fonction demandée. *Ex:* **help(matrix)**
- Le **?**. Dans la console, le raccourci pour la fonction help() est le **?**. *Ex:* **?matrix**
- L'onglet **Help** de la zone **OUTILS** de l'environnement. Il est possible d'y saisir directement le nom de la fonction à expliquer

Dans l'aide, vous avez plusieurs parties, dont voici les plus importantes et les plus utiles :

- **Description** : L'utilité de la fonction
- **Usage** : Comment appeler la fonction
- **Arguments** : La liste des arguments et des informations sur ces derniers
- **Examples** : Comme son nom l'indique, des exemples d'utilisation de la fonction

Les fonctions vues jusqu'à présent

Si vous reprenez vos cours des premières séances, vous verrez que nous avons utilisé quelques fonctions. Celles-ci sont assez simples car pour la plupart d'entre elles, elles n'ont qu'un argument.

Définir une structure/objet de base :

- **c()** : Définir un vecteur
- **matrix()** : Définir une matrice
- **data.frame()** : Définir un data frame
- **list()** : Définir une liste

Connaitre le mode d'un objet :

- **class()** : renvoie le mode d'un objet
- **is.null()** : renvoie **TRUE** si l'objet n'existe pas. **FALSE** sinon.
- **is.numeric()** : renvoie **TRUE** si l'objet est de mode *numeric*. **FALSE** sinon.
- **is.character()** : renvoie **TRUE** si l'objet est de mode *character*. **FALSE** sinon.
- **is.logical()** : renvoie **TRUE** si l'objet est de mode *logical*. **FALSE** sinon.

Connaître les noms (labels) donnés aux éléments d'un objet :

- **names()** : renvoie les noms (labels) donnés à un vecteur. Si l'argument est un data frame, cela renvoie les noms de variables (colonnes).
- **dimnames()** : renvoie les noms de lignes puis de colonnes d'une matrice ou d'un data frame
- **colnames** et **rownames** : renvoie les noms de colonnes / noms de lignes d'un objet à 2 dimensions (matrice, data frame)

Connaître les dimensions d'un objet :

- **length()** : renvoie la longueur d'un objet à 1 dimension (vecteur ou liste). Pour un data frame, length() renvoie le nombre de colonnes. Pour une matrice, length() renvoie le nombre d'éléments dans la matrice (en d'autres termes le résultat du produit : lignes x colonnes)
- **dim()** : renvoie les dimensions d'une matrice ou d'un data frame
- **t()** : permet de transposer une matrice ou un data frame. Les lignes deviennent les colonnes et les colonnes des lignes

Les fonctions arithmétiques de base

Il y a quelques fonctions de base à connaître pour la partie arithmétique. Les voici :

- **sum()** : permet de calculer la *somme* d'une série
- **min()** : permet d'obtenir le *minimun* d'une série
- **max()** : permet d'obtenir le *maximum* d'une série
- **mean()** : permet de calculer la *moyenne* d'une série
- **sd()** : permet de calculer l'*écart-type* d'une série (min 2)
- **var()** : permet de calculer la *variance* (carré de l'écart-type)
- **median()** : permet de calculer la *médiane*
- **quantile()** permet de sortir différents *quantiles* d'une série de données
- **summary()** : donne un descriptif statistique d'une série quantitative (on retrouve plusieurs résultats données par les fonctions ci-dessus)

Attention ! : parmi les arguments de ces fonctions arithmétiques, il existe une option très importante : **na.rm =** (pour *"remove NA"*). En effet, il est possible que dans votre série de données il y ait des données manquantes (**NA**). L'option **na.rm = TRUE** permet de réaliser le calcul en retirant de la série les **NA**. Cette remarque est d'autant plus importante que, par défaut : *na.rm = FALSE*. Pensez à l'activer, cela vous évitera de mauvaises surprises !

Exemples d'application :

```
# Définition d'une série de données sans signification particulière
my_data <- c(NA, 10:20, NA, 100:200, NA, 1000:2000, NA)
# Nous y avons glissé des NA

# Calcul de la somme
sum(my_data, na.rm = TRUE)
```

```
## [1] 1516815
```

```
# Calcul de la moyenne
mean(my_data, na.rm = TRUE)
```

```
## [1] 1362.817
```

```
# Calcul de la médiane
median(my_data, na.rm = TRUE)
```

```
## [1] 1444
```

```
# Calcul des quantiles
quantile(my_data, na.rm = TRUE)
```

```
##   0%  25%  50%  75% 100%
##   10 1166 1444 1722 2000
```

```
# 0% = Minimum
# 25% = Premier quartile (Q1)
# 50% = Deuxième quartile = Médiane (Q2)
# 75% = Troisième quartile (Q3)
# 100% = Maximum
```

Les fonctions utiles

Dans cette section, voici quelques fonctions souvent utilisées dans les scripts :

- **print()** : permet de renvoyer tout ce qui est contenu dans les parenthèses comme argument de fonctions. Utile quand on a besoin qu'une commande nous renvoie une phrase après son exécution.
- **rm()** : permet de retirer de l'environnement l'objet donné en argument. Très pratique pour nettoyer partiellement votre environnement de travail. Il existe une commande généralement mise en début de script pour nettoyer tout l'environnement automatiquement, sans avoir à cliquer sur le raccourci *RStudio* (le petit balai). Pour cela, il suffit de donner à l'argument **list** = la valeur **ls()**. *Ex:* **rm(list = ls())**
- **rep()** : permet de répéter un certain nombre de fois la valeur x. Très partique pour construire un vecteur qui va contenir plusieurs fois une même valeur. *Ex:* **rep(x = 2, 10)**

```
# Construire un vecteur contenant 1000 fois la valeur 1
rep_1000 <- rep(x = 1, 1000)

# vérifier la longeur de notre vecteur
length(rep_1000)
```

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

- **round()** : permet d'arrondir des valeurs à l'unité la plus proche.

```
# Arrondir 2.672489023 à différent niveau unitaire
x <- 2.672489023
round(x, digits = 0) # Arrondi à l'unité la plus proche
```

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

```
round(x, digits = 1) # Arrondi à la première décimale la plus proche
```

```
## [1] 2.7
```

```
round(x, digits = 2) # Arrondi à la deuxième décimale la plus proche
```

```
## [1] 2.67
```

- **unique()** : prend un vecteur en entrée et renvoie un vecteur des différentes valeurs prises en enlevant les doublons. Très pratiques quand vous avez beaucoup de répétitions et que vous souhaitez rapidemment savoir quelle sont les différentes valeurs.</p

```
# Définition d'un vecteur simple avec plusieurs valeurs répétées
filieres <- c('S', 'L', 'ES', 'L', 'L', 'S', 'ES', 'S', 'L', 'S',
              'ES', 'ES', 'L', 'STG','S', 'L', 'STMG', 'ES', 'S')

# Obtenir les valeurs uniques du vecteur
unique(filieres)
```

```
## [1] "S"    "L"    "ES"   "STG"  "STMG"
```

- **table()** : permet d'obtenir le nombre exacte d'apparition de chaque valeur. Très utilisée pour faire des analyses de fréquences. Savoir quelle est la modalité la plus fréquente *v.s* la moins fréquente.

```
# Connaître le nombre d'apparitions des différentes filières du vecteur précédent
table(filieres)
```

```
## filieres
##   ES    L    S  STG STMG
##    5    6    6    1    1
```

- **runif()** : permet d'effectuer un tirage aléatoire de **n** valeurs comprises entre un **min** et un **max**. Il s'agit en réalité de l'utilisation d'une loi uniforme, loi selon laquelle tout les nombres d'un intervalle défini ont la même probabilité d'être tiré au sort. Cette fonction est très utile pour créer des algorithmes nécessitant d'utiliser des valeurs aléatoires.

```
# Simulation de 3 lancés de dés (compris entre 1 et 6)
# Pour cela il faut arrondir à l'unité la plus proche le résultat de runif

lance_de_1 <- round(runif(n = 3, min = 1, max = 6),
                    digits = 0)

# Afficher le résultat des 3 lancés de dés
lance_de_1
```

```
## [1] 4 2 4
```

- **set.seed()** : permet de bloquer l'aléa. On dit souvent dans le jargon "bloquer la graine" (trad. *seed* = graine). Cela permet de fixer un aléa. A chaque **graine** (seed) correspond un tirage aléatoire. Bloquer la graine permet de garantir la reproductibilité des résultats

```
# Simulation d'un lancé de dés en fixant une graine
# Tout le monde retrouvera les mêmes résultats
set.seed(1994)

lance_de_2 <- round(runif(n = 3, min = 1, max = 6),
                    digits = 0)

# Afficher le résultat des 3 lancés de dés
lance_de_2
```

```
## [1] 1 5 4
```

apply() : Cette fonction permet d'appliquer une certaine fonction à chaque ligne ou colonne d'une matrice ou d'un data frame. Cette fonction a 3 arguments :

- **X =** : Il s'agit d'une matrice ou d'un data frame
- **MARGIN =** : Mettre **1** si vous appliquez la fonction aux lignes et **2** pour les colonnes
- **FUN =** : Le nom de la fonction à utiliser

Souvenez-vous de l'évaluation n°3, vous pouvez construire un data frame avec les Pokémon en lignes et les statistiques de combat en colonnes. Vous souhaitez créer dans une nouvelle colonne, une 'moyenne', avec **apply()**, vous pouvez ! On essaye?

```
# Exemple d'utilisation de la fonction apply()

# On reprend les Pokémon donnés dans le sujet de l'interro 3

# On calcule une nouvelle variable : Moyenne
df_pokemon$Moyenne <- apply(X = df_pokemon[, c('HP', 'Attack', 'Defense', 'Sp_Atk', 'Sp_Def', 'Speed')],
                            MARGIN = 1,
                            FUN =  mean)
```

REMARQUE ! : Pour effectuer la fonction **mean()** sur plusieurs colonnes, il faut que ces colonnes soient de mode *numeric*. Aussi rappelez-vous dans le TD précédent, nous avons créé des variables **Nom** et **Type**. Ces deux variables ne sont pas compatibles avec la fonction **mean()**, car ce sont des *character*.

CONSEIL ! : Si la fonction **apply()** est une fonction régulièrement utilisée dans les scripts *R*, nous ferons remarquer que la maîtrise de cette fonction n'est pas, à ce stade, des plus obligatoires. C'est un élément à

connaître si vous souhaitez allez plus loin. A ce moment-là, vous découvrirez d'autres fonctions relativement similaires et plus complexes encore. Néanmoins, il se peut que vous en ayez besoin dans le projet.

Construire sa propre fonction

Si l'environnement de base de *R* propose de nombreuses fonctions, il est possible de se créer la sienne. Le plus souvent nous construisons des fonctions afin de répondre à un besoin spécifique et potentiellement récurrent. En effet, une fois que la fonction est définie, nous pouvons nous en servir à volonté dès lors qu'elle est chargée dans l'environnement de travail.

Pour définir une fonction, il faut utiliser un bloc dédié et enregistrer le tout dans ce qui sera le nom de la fonction.

function(*les arguments à définir*){*les opérations à exécuter*}

Par exemple, pour illustrer cela, nous allons construire une fonction personnalisée calculant les intérêts perçus au bout d'une période en année, selon un taux d'intérêt et un capital initial.

```
# Définition d'une fonction

interets <- function(capital = 0, periode = 0, taux = 0){

  capital_final <- capital*(1 + taux)**periode
  gain <- capital_final - capital
  arrondi_gain <- round(gain, 2)

  return(arrondi_gain)
}
```

Dans la fonction définie ci-dessus, nous retrouvons plusieurs éléments :

- Une liste d'arguments : c'est vous qui décidez du nom et du nombre d'arguments dont vous avez besoin pour exécuter votre fonction. Vous pouvez voir que nous avons affecté des valeurs **par défaut**. Les **= 0** permettent de dire à *R* que si la fonction vient à être utilisée sans qu'un des arguments ne soit explicité alors le calcul sera effectué avec la valeur par défaut, ici 0. Sans argument, la fonction **interets()** donnera toujours 0. Il s'agit d'une sorte de sécurité.
- Les accolades : les opérations effectuées par la fonction sont à coder entre les accolades.
- Le **return()** : très pratique pour indiquer l'objet de résultat. Ceci est d'autant plus vrai si, dans la fonction, de nombreux objets sont créés durant les opérations. Dans notre exemple ci-dessus, nous avons créé 3 objets : *capital_final*, *gain* et *arrondi_gain*. Vous remarquerez qu'aucun de ces objets n'est enregistré dans l'environnement de travail *R*. Ils sont définis temporairement au moment de l'usage de la fonction et aussitôt effacés. Seul l'objet contenu dans le **return** peut être sauvegardé.

```
# Utilisation de la fonction interets
interets(capital = 1000, periode = 10, taux = 0.0125)
```

```
## [1] 132.27
```

```
# Enregietrement du résultat
my_gain <- interets(capital = 1000, periode = 10, taux = 0.0125)
```

Blocs de codes conditionnels

"Avec des"Si", on mettrait Paris en bouteille" - Expression française populaire.

Le bloc if

Le **if** permet de construire des portions de code dont l'exécution n'aura lieu qu'à certaines conditions. Nous parlons d'exécution conditionnelle.

La structure est assez élémentaire :

if (*condition(s)*) {*code à exécuter si la/les conditions sont vérifiées*}

Les accolades **{ }** sont indispensables pour entrer dans le bloc de code spécial.

Il est tout à fait possible de créer une exécution alternative avec une branche **else** à écrire à la suite :

else { *code à éxécuter si les conditions testées ne sont pas vérifiées*}

En l'absence de **else**, si la condition testée dans le **if** n'est pas remplie, alors rien ne se passe.

Voici un exemple basique :

```
PSG <- 2
Napples <- 2

# Création d'un bloc de code à execution conditionnelle avec if :

if (PSG > Napples){
  print("Paris est magique !")
}else{
  print("La ligue des champions? Toujours pas pour cette année...")
  }
```

```
## [1] "La ligue des champions? Toujours pas pour cette année..."
```

Il est possible de cumuler les conditions et donc d'avoir encore plus d'exécutions possibles. Pour cela, il suffit d'insérer un bloc **else if**. Il se construit de la même façon que le bloc **if**.

ATTENTION ! : Si vous utilisez un bloc **else if**, il est impératif de refermer le tout par un **else**. Sans quoi, un message d'erreur vous attend...

```
France <- 4
Croatie <- 2

# Création un bloc if avec un else if :

if (France > Croatie){
  print("Champion du monde !!!!!!!!!!!")
}else if (France == Croatie){
  print('Pitié, pas les péno...')
}else{
  print("Bon bah je vais me coucher...")
}
```

```
## [1] "Champion du monde !!!!!!!!!!!"
```

La fonction ifelse()

Il est possible de définir des vecteurs selon des conditions. Pour cela, il existe sous *R* la fonction **ifelse()**. Elle fonctionne avec 3 arguments :

- **test** : Un test logique à réaliser
- **yes** : la valeur à donner si le résultat du test est **TRUE**
- **no** : la valeur à donner si le résultat du test est **FALSE**

Recoder une variable

On se sert souvent de la fonction **ifelse()** pour recoder les variables. Par recoder, on entend : changer la valeur affichée selon des règles précises. Par exemple, nous avons une variable dans notre base de données Pokémon qui se nomme **Legendary**, elle vaut 1 si le Pokémon est un Pokémon "Légendaire" et 0 sinon. Pour quelqu'un d'extérieur, 1 et 0 ne sont pas des informations très parlantes. On utilise alors **ifelse()** pour créer une variable récodée avec une valeur *character* "Légendaire" si et seulement si on avait des 1. On appelle souvent la variable recodée avec un **_r** à la fin pour indiquer que cette variable vient d'un recodage. Voici l'exemple en question :

```
# Cration d'une variable character recodée à partir de Legendary (1 ou 0)
df_pokemon$Legendary_r <- ifelse(df_pokemon$Legendary == 1, 'Légendaire', 'Commun')

#Afficher le résultat
df_pokemon$Legendary_r
```

```
## [1] "Commun"     "Commun"     "Commun"     "Commun"     "Légendaire"
## [6] "Commun"
```

Les années Bissextiles : Un exemple de bloc ifelse

Voici un exemple de **ifelse()** qui fait figure de cas d'école aujourd'hui : **Les années Bissextiles**. La règle veut qu'une année soit bissextile (366 jours au lieu de 365 - présence du 29 février) si et seulement si :

- L'année est un multiple de 4
- L'année n'est pas un multiple de 100 (exception pour les multiples de 400)

En pratique :

```
# Définir un vecteur un data frame colonne contenant une serie d'année
df_date <- data.frame(1900:2100)
names(df_date) <- 'Year'

# Définition d'une nouvelle variable 'Type_year' avec ifelse()

df_date$Type_year <- ifelse((df_date$Year %% 4 == 0 & df_date$Year %% 100 != 0) |
                            (df_date$Year %% 400 == 0), 'Bissextile', 'Base 365')

# Visualisation des premiers éléments de la variable "Type_year"
df_date$Type_year[1:5]
```

```
## [1] "Base 365"   "Base 365"   "Base 365"   "Base 365"   "Bissextile"
```

La première parenthèse dans le **ifelse()** contient la règle *multiple de 4 mais pas multiple de 100* **OU** la seconde possibilité *multiple de 400*

Rappel : le %% signifie **modulo**. Cela affiche le reste d'une division. Si le modulo de **A** par **B** vaut zéro (0), cela signifie que le A est divisible par B.

Les boucles for

La boucle **for** permet d'exécuter successivement une opération pour chaque élément d'une série.

Pour écrire une boucle **for**, il faut préciser un **compteur**, une **série** et une/des **opération(s)** à exécuter.

- La série permet de préciser pour quelles valeurs l'opération va succesivement être exécutée. La série est souvent un vecteur *numeric* avec des nombres qui se suivent (*Ex* : exécuter une opération pour chaque nombre allant de 1 à 10....). Attention, il n'est nécessaire que les nombres se suivent, ni même que la série soit *numeric*. On peut effectuer des opérations à partir des valeurs d'une série de *character*.
- Le compteur permet de changer la valeur à chaque itération de la boucle. On utilise souvent la lettre **i**. Par exemple, pour une série allant de 1 à 10 (**i in 1:10**), l'opération va s'exécuter avec **i** valant 1, puis à l'itération suivante **i** vaudra 2, etc..
- Comme pour les blocs **if**, il faut préciser les opérations que *R* doit lancer à chaque fois. Pour chaque valeur de **i**, *R* va exécuter la même opération. Encore ici, on groupera les expressions dans des accolades **{ }**

Voici un exemple simple avec une boucle for qui va effectuer 3 lancés de dés pour différentes valeurs de graines (aléa, voire fonction **set.seed()**) et dire si le joueur gagne si la somme des 3 est paire ou s'il perd avec un résultat impair.

```
# Une boucle for qui va effectuer des lancés de dés de manière
for (i in c(1939, 1994, 12, 45, 6, 2)){
  set.seed(i)
  # On bloque la graine (l'aléa) pour chaque valeur de i : 1939, puis 1994,...

  res_3_des <- round(runif(n = 3, min = 1, max =  6), 0)
  # On garde 3 nombres entre 1 et 6 et on arrondi le tout à l'unité proche

  sum_res <- sum(res_3_des)
  # On fait la somme des trois valeur de dés obtenus

  if (sum_res %% 2 == 0){
    # On teste si la somme est paire (divisible par 2)

    # Si c'est pair on affiche un petit mot
    print("Vous avez gagné !")

  }else {
    print("Sniff... Try again !")
    # On écrit un petit message de loser sinon...
  }
# Et on oublie pas de fermer les accolades...
}
```

```
## [1] "Sniff... Try again !"
## [1] "Vous avez gagné !"
## [1] "Vous avez gagné !"
## [1] "Sniff... Try again !"
## [1] "Vous avez gagné !"
## [1] "Sniff... Try again !"
```

Si *R* offre la possibilité de manipuler les boucles, la plupart des opérations utilisant des boucles peuvent être évitées en utilisant des fonctions. Cela permet de gagner du temps de calcul. Il y a peu de chance que vous ayez à utiliser énormément les boucles durant ces cours introductifs.

Exo de Synthèse

```r
# Bloquez un aléa (une graine)
set.seed(27)

# Avec runif(), créez un vecteur de 10 valeurs aléatoires comprises entre 1 et 6
my_vector <- runif(n = 10, min = 1, max = 6)

# Avec la fonction length(), vérifiez la longeur de votre vecteur
length(my_vector)
```

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

```r
# Utilisez la fonction round() pour les arrondir à l'unité la plus proche
my_vector <- round(my_vector, 0)

# Remplacez la 6ème valeur par un NA (valeur manquante)
my_vector[6] <- NA

# Utilisez la fonction mean pour calculer la moyenne de la série
mean(my_vector, na.rm = TRUE)
```

```
## [1] 2.555556
```

```r
# Utilisez la fonction quantile() pour connaître la distribution de la série
quantile(my_vector, na.rm = TRUE)
```

```
##   0%  25%  50%  75% 100%
##    1    1    2    3    6
```

```r
# Utilisez la fonction summary() pour avoir le détail de votre série
summary(my_vector, na.rm = TRUE)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.000   1.000   2.000   2.556   3.000   6.000       1
```

```r
# Utilisez un bloc for avec  if / else pour parcourir les 5 premières valeurs de my_vector
# Affichez "PAIR"" si le nombre est divisible par 2
# Affichez "IMPAIR" sinon

for (i in 1:5){
  if(my_vector[i] %% 2 == 0){
    print("PAIR")
  }else {
    print("IMPAIR")
  }
}
```

```
## [1] "PAIR"
## [1] "IMPAIR"
## [1] "IMPAIR"
## [1] "IMPAIR"
## [1] "PAIR"
```

Bibliographie

Ce cours a été construit conjointement par Joris FALIP (doctorant) et Morgan COUSIN (data analyst). Plusieurs éléments de documentation ont été utilisés pour construire ce support de cours. Vous les retrouverez ci-dessous en tant qu'éléments bibliographiques.

Vincent GOULET, (2016), *Introduction à la programmation en R*, Université de Laval, 5ème édition, 218 pages, disponible en ligne : https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf