Introduction à R pour la Data Science

Fatoumata DIALLO

14 juillet 2025

Table des matières

Introduction

La **Data Science** consiste à analyser des données afin d'en extraire des informations utiles à la prise de décision. Elle repose sur des outils statistiques, informatiques et métiers.

Parmi les langages utilisés, **R** occupe une place centrale pour le traitement, l'analyse et la visualisation des données. Il est particulièrement apprécié pour ses capacités statistiques, sa richesse en bibliothèques spécialisées et ses outils graphiques puissants.

Ce support a pour objectif de présenter les bases du langage R, utiles pour manipuler des données, produire des analyses, et construire des visualisations claires et reproductibles.

1 Installation de R et RStudio

Pour utiliser le langage **R**, deux outils sont nécessaires :

— **R** : le moteur de calcul statistique.
— **RStudio** : une interface de développement (IDE) plus conviviale pour écrire et exécuter du code R.

1. Installer R

1. Aller sur le site officiel : https ://cran.r-project.org
2. Choisir le système d'exploitation (*Windows*, *macOS* ou *Linux*).
3. Télécharger la dernière version stable de R.
4. Suivre les étapes d'installation par défaut.

2. Installer RStudio

1. Aller sur le site : https ://posit.co/download/rstudio-desktop
2. Télécharger la version gratuite de **RStudio Desktop Open Source**.
3. Installer le logiciel comme n'importe quelle application.

3. Lancer RStudio

Une fois les deux outils installés, il suffit de lancer **RStudio**. Celui-ci utilise automatiquement l'interpréteur R installé en arrière-plan.

Remarque : Il est recommandé d'utiliser RStudio plutôt que l'interface de base de R, car il offre un environnement plus complet, avec éditeur de code, console, visualisation des données, historique des commandes, etc.

2 Présentation de l'interface RStudio

Une fois lancé, **RStudio** se présente sous la forme d'une interface divisée en quatre volets principaux :

- **Script (en haut à gauche)** : permet d'écrire et enregistrer du code R dans des fichiers.
- **Console (en bas à gauche)** : exécute les commandes R en temps réel.
- **Environnement / Historique (en haut à droite)** : affiche les objets en mémoire, les jeux de données, et l'historique des commandes.
- **Plots / Packages / Fichiers (en bas à droite)** : permet d'afficher les graphiques, gérer les packages, explorer les fichiers, etc.

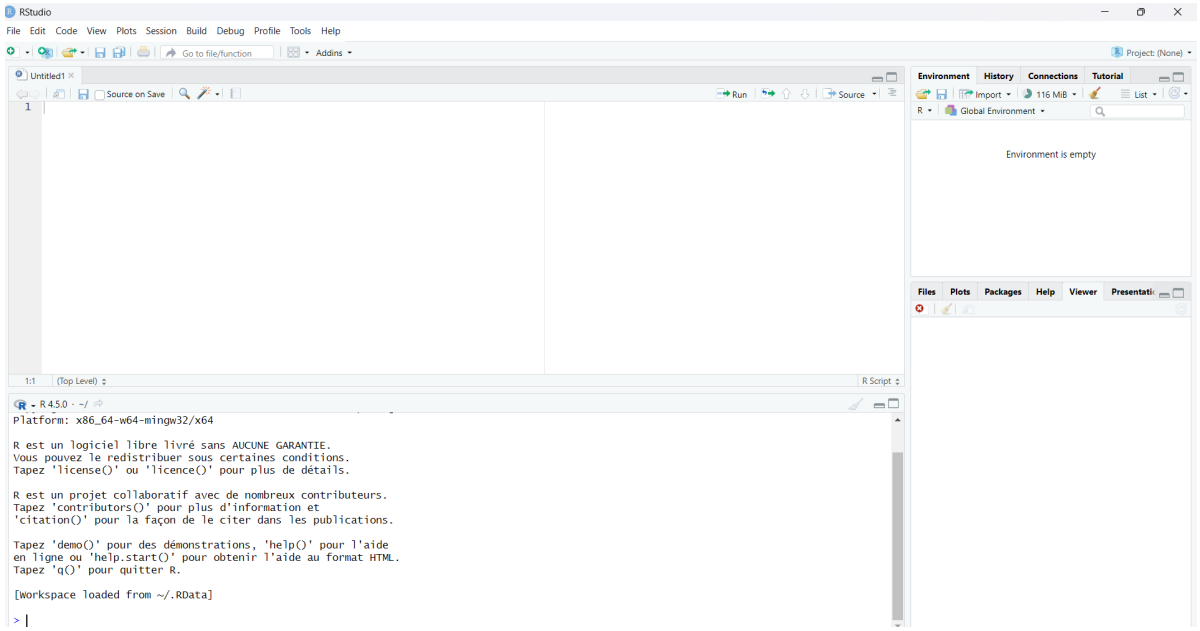

FIGURE 1 – Interface RStudio au démarrage

3 Créer et exécuter un script R dans RStudio

Dans RStudio, l'écriture du code se fait dans un fichier appelé **script R**, qui peut être sauvegardé et réutilisé.

1. Créer un nouveau fichier R

Pour créer un nouveau script R :

— Aller dans le menu `File` > `New File` > `R Script`.
— Un nouvel onglet s'ouvre dans la partie supérieure gauche de RStudio.
— Ce fichier peut être sauvegardé avec l'extension `.R`.

2. Types de fichiers courants

RStudio permet de créer différents types de fichiers selon les besoins :

— **R Script (.R)** : pour écrire du code R classique.
— **R Markdown (.Rmd)** : pour combiner code, texte et rendu dynamique.
— **Shiny App** : pour créer des applications interactives en R.
— **Quarto (.qmd)** : alternative moderne à R Markdown pour des rapports interactifs.

3. Exécuter du code R

Une fois un script ouvert, il est possible d'exécuter une ou plusieurs lignes de code de deux manières :

— **Ctrl** + **Entrée** (ou **Cmd** + **Entrée** sur Mac) : exécute la ligne sélectionnée.
— Cliquer sur le bouton `Run` en haut à droite de l'éditeur.

Le résultat de l'exécution s'affiche dans la console, en bas à gauche de l'interface.

4 Opérations de base

R peut être utilisé comme une calculatrice pour effectuer des opérations algébriques élémentaires. On peut également enregistrer des valeurs dans des variables et travailler ensuite avec celles-ci.

1. Calculs simples

Voici quelques exemples de calculs réalisés directement dans la console R :

```
3 + 2
3 - 2
```

```r
3 * 2
3 / 2
3^2

# Plusieurs instructions sur une seule ligne
3 + 2 ; 3 - 2 ; 3 * 2 ; 3^2
```

2. Affectations et expressions plus complexes

On peut stocker des valeurs dans des variables (ou scalaires) avec le symbole = ou ←. Ensuite, ces variables peuvent être combinées dans des expressions :

```r
x = 2
a = 2
b = 3
c = 4

x + c
b * x + c
a * x^2 + b * x + c
b / (x + c)^a
x ^ (a + b + c)
```

3. Opérations logiques

R permet également de manipuler des valeurs logiques :

```r
TRUE
FALSE

3 > 2       # TRUE
4 == 5      # FALSE
5 != 2      # TRUE
```

- `==` : égalité
- `!=` : différence
- `>` : supérieur
- `<` : inférieur
- `&` : et logique
- `|` : ou logique

5 Création de vecteurs

Les **vecteurs** sont les structures de base de R. Ce sont des séquences d'éléments du même type : numériques, chaînes de caractères ou logiques.

5.1 Vecteurs numériques

On peut créer un vecteur numérique à l'aide de la fonction `c()` :

```
vec1 = c(2.8, 2.4, 2.1, 3.6, 2.8)
vec1
```

Cela renvoie : `[1] 2.8 2.4 2.1 3.6 2.8`

On a ainsi créé le vecteur `vec1` en concaténant les éléments 2.8, 2.4, 2.1, 3.6 et 2.8.

5.2 Vecteurs de chaînes de caractères

```
vec2 = c("rouge", "vert", "vert", "vert", "jaune")
vec2
```

Cela renvoie : `[1] "rouge" "vert" "vert" "vert" "jaune"`

5.3 Vecteurs logiques

```
vec3 = c(TRUE, TRUE, FALSE, FALSE, FALSE)
vec3
```

Cela renvoie : `[1] TRUE TRUE FALSE FALSE FALSE`

5.4 Création par répétition

```
rep(4, 3)
```

Renvoie : `[1] 4 4 4`

On peut aussi répéter un vecteur complet :

```
vec4 = rep(vec1, 2)
vec4
```

Renvoie : `[1] 2.8 2.4 2.1 3.6 2.8 2.8 2.4 2.1 3.6 2.8`

Ou encore répéter chaque élément un certain nombre de fois :

```
vec5 = rep(vec1, c(2, 1, 3, 3, 2))
vec5
```

Renvoie : `[1] 2.8 2.8 2.4 2.1 2.1 2.1 3.6 3.6 3.6 2.8 2.8`

5.5 Création par suites

```
vec6 = 1:10
vec6
```

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

Avec des pas personnalisés :

```
vec7 = seq(from = 3, to = 5, by = 0.2)
vec7
```

5.6 Création élément par élément

On peut initialiser un vecteur vide et y ajouter des éléments :

```
vec8 = numeric()
vec8[1] = 41.8
vec8[2] = -0.3
vec8[3] = 92
vec8
```

Renvoie : `[1] 41.8 -0.3 92`

5.7 Création avec des nombres aléatoires

Il est possible de générer des vecteurs contenant des valeurs aléatoires à l'aide de la fonction `sample()`. Par exemple, pour simuler les moyennes de 20 élèves :

```
moyenne_de_la_classe <- sample(1:20, 20)
moyenne_de_la_classe
```

Renvoie (exemple) : `[1] 4 17 19 3 6 5 15 8 18 2 12 14 10 13 16 7 1 9 20 11`

Remarque : les valeurs générées changent à chaque exécution sauf si on fixe une graine avec `set.seed()`.

6 Opérations sur les vecteurs

R permet d'effectuer facilement des opérations arithmétiques, logiques ou d'indexation sur les vecteurs. Ces opérations s'appliquent élément par élément.

6.1 Opérations arithmétiques

On peut effectuer des opérations mathématiques directement sur les vecteurs :

```
vec1 = c(1, 2, 3)
vec2 = c(4, 5, 6)

vec1 + vec2    # Addition
vec1 - vec2    # Soustraction
vec1 * vec2    # Multiplication
vec2 / vec1    # Division
vec1 ^ 2       # Puissance (élévation au carré)
```

Renvoie :

- vec1 + vec2 : [1] 5 7 9
- vec1 - vec2 : [1] -3 -3 -3
- vec1 * vec2 : [1] 4 10 18
- vec2 / vec1 : [1] 4.0 2.5 2.0

— vec1 ^{2} : [1] 1 4 9

6.2 Opérations avec un scalaire

On peut combiner un vecteur avec un nombre seul (scalaire) : R applique alors l'opération à chaque élément.

```
vec1 + 10
vec1 * 2
```

6.3 Recyclage des vecteurs

Si deux vecteurs de tailles différentes sont combinés, R recycle les valeurs du plus petit :

```r
v1 = c(1, 2, 3, 4)
v2 = c(10, 100)

v1 + v2
```

Résultat :

```
[1] 11 102 13 104
```

Attention : R ne renvoie pas d'erreur, mais peut afficher un avertissement si la taille du plus grand n'est pas un multiple exact de la taille du plus petit.

6.4 Opérations logiques sur vecteurs

On peut effectuer des comparaisons entre les éléments :

```r
v = c(4, 7, 9, 3, 6)

v > 5         # [1] FALSE TRUE TRUE FALSE TRUE
v == 3        # [1] FALSE FALSE FALSE TRUE FALSE
v != 7        # [1] TRUE FALSE TRUE TRUE TRUE
```

6.5 Modification des éléments d'un vecteur

On peut modifier un ou plusieurs éléments d'un vecteur en utilisant leur position (indexation) :

```r
v = c(10, 20, 30, 40, 50)

v[2] = 99          # Remplace le 2e élément par 99
v[c(1, 3)] = 0     # Remplace les 1er et 3e éléments par 0
v
```

Résultat : `[1] 0 99 0 40 50`

6.6 Création à partir d'autres vecteurs

On peut créer de nouveaux vecteurs en combinant d'autres :

```r
a = c(1, 2, 3)
b = c(4, 5)

c1 = c(a, b)        # Concaténation
```

```r
c2 = a + 10          # Ajout  d un  scalaire
c3 = a * b           # Multiplication : recycle b
```

Résultats :

- c1 : [1] 1 2 3 4 5
- c2 : [1] 11 12 13
- c3 : [1] 4 10 12 (car b est recyclé)

6.7 Autres fonctions utiles sur les vecteurs

Voici un éventail de fonctions souvent utilisées avec des vecteurs :

```r
v = c(10, 20, 30, 40, 50)

sum(v)          # Somme des éléments
mean(v)         # Moyenne
min(v)          # Minimum
max(v)          # Maximum
length(v)       # Nombre  d  éléments
sort(v)         # Tri croissant
rev(v)          # Inversion de l'ordre
range(v)        # Renvoie le min et le max
unique(v)       # Valeurs uniques
duplicated(v)   # Indique les doublons
which(v > 30)   # Indices des éléments supérieurs à 30
any(v > 60)     # TRUE si au moins un élément > 60
all(v > 5)      # TRUE si tous les éléments > 5
cumsum(v)       # Somme cumulée
cumprod(v)      # Produit cumulé
diff(v)         # Différences successives
```

Chaque fonction permet de mieux comprendre, transformer ou manipuler les vecteurs selon le besoin.

7 Les listes

Une **liste** en R est une structure de données très flexible. Contrairement aux vecteurs ou aux matrices, une liste peut contenir des objets de types différents : un vecteur, une matrice, un data frame, une fonction, ou même une autre liste !

7.1 Créer une liste

Voici un exemple de liste contenant différents types d'objets :

```
# Créer une liste contenant différents objets
ma_liste <- list(
  nom = "Alice",
  notes = c(15, 18, 13),
  moyenne = mean(c(15, 18, 13)),
  validation = TRUE
)

# Afficher la liste
ma_liste
```

Renvoie :

```
$nom
[1] "Alice"

$notes
[1] 15 18 13

$moyenne
[1] 15.33333

$validation
[1] TRUE
```

7.2 Accéder aux éléments d'une liste

On peut accéder aux éléments d'une liste de plusieurs manières :

```
ma_liste$notes
```

Renvoie : `[1] 15 18 13`

```
ma_liste[[2]]
```

Renvoie : `[1] 15 18 13`

```
ma_liste[[3]]
```

Renvoie : `[1] 15.33333`

```
names(ma_liste)
```

Renvoie : `[1] "nom" "notes" "moyenne" "validation"`

```
ma_liste[2]
```

Renvoie (sous forme de sous-liste) :

```
$notes
[1] 15 18 13
```

Attention : `ma_liste[2]` renvoie une liste, tandis que `ma_liste[[2]]` renvoie l'objet lui-même.

7.3 Modifier une liste

On peut ajouter, modifier ou supprimer des éléments facilement :

```
# Ajouter un élément
ma_liste$remarque <- "Très bon travail"
```

Renvoie un nouveau composant :

```
$remarque
[1] "Très bon travail"
```

```
# Modifier un élément
ma_liste$validation <- FALSE
ma_liste$validation
```

Renvoie : `[1] FALSE`

```
# Supprimer un élément
ma_liste$nom <- NULL
ma_liste
```

Renvoie la liste sans l'élément `nom`.

7.4 Listes imbriquées

Une liste peut contenir une autre liste, créant une structure hiérarchique :

```r
# Liste contenant une autre liste
liste2 <- list(
  nom = "Projet",
  contenu = list(
    data = c(1, 2, 3),
    titre = "Analyse 2025"
  )
)
```

```r
liste2$contenu$data
```

Renvoie : `[1] 1 2 3`

Astuce : on peut aussi utiliser des crochets imbriqués :

```r
liste2[[2]][["data"]]
```

Renvoie également : `[1] 1 2 3`

7.5 Fonctions utiles pour les listes

```r
length(ma_liste)
```

Renvoie : `4` (nombre d'éléments)

```r
str(ma_liste)
```

Renvoie :

```
List of 4
 $ notes     : num [1:3] 15 18 13
 $ moyenne   : num 15.3
 $ validation: logi FALSE
 $ remarque  : chr "Très bon travail"
```

```r
names(ma_liste)
```

Renvoie : `[1] "notes" "moyenne" "validation" "remarque"`

```r
is.list(ma_liste)
```

Renvoie : `[1] TRUE`

8 Les matrices

Une **matrice** en R est une structure bidimensionnelle (lignes × colonnes) contenant uniquement des éléments du même type, généralement des valeurs numériques. Elle est souvent utilisée pour représenter des tableaux de notes, des résultats expérimentaux, ou pour effectuer des calculs statistiques et linéaires.

8.1 Créer une matrice à partir d'un vecteur

Méthode générique :

```
mat1 = matrix(vec4, ncol = 5)
mat1
```

Renvoie :

```
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.8  2.1  2.8  2.4  3.6
[2,]  2.4  3.6  2.8  2.1  2.8
```

Par défaut, les éléments sont insérés colonne par colonne.

Remplissage ligne par ligne :

```
mat2 = matrix(vec4, ncol = 5, byrow = TRUE)
mat2
```

Renvoie :

```
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.8  2.4  2.1  3.6  2.8
[2,]  2.8  2.4  2.1  3.6  2.8
```

8.2 Assembler des vecteurs en colonnes ou lignes

Concaténer par colonnes :

```
mat3 = cbind(vec1, 3:7)
mat3
```

Renvoie :

```
     vec1
[1,]  2.8   3
[2,]  2.4   4
[3,]  2.1   5
[4,]  3.6   6
[5,]  2.8   7
```

Concaténer par lignes :

```r
rbind(c(1,2,3), c(4,5,6))
```

Renvoie :

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

8.3 Extraire lignes, colonnes ou éléments

```r
mat1[ , 3]          # 3e colonne
mat1[2, ]           # 2e ligne
mat1[1, c(2, 4, 5)] # Sous-matrice
mat1[ , c(FALSE, TRUE, FALSE, TRUE, TRUE)]
```

Exemples de résultats :

- `mat1[, 3]` renvoie : `[1] 2.8 2.8` - `mat1[2, ]` renvoie : `[1] 2.4 3.6 2.8 2.1 2.8` - `mat1[1, c(2, 4, 5)]` renvoie : `[1] 2.4 3.6 2.8`

8.4 Créer une matrice diagonale

```r
mat4 = diag(c(2, 1, 5))
mat4
```

Renvoie :

```
     [,1] [,2] [,3]
[1,]    2    0    0
[2,]    0    1    0
[3,]    0    0    5
```

8.5 Exemple : matrice de notes

```r
notes <- sample(1:20, 15)
notes_des_eleves <- matrix(notes, ncol = 3, nrow = 5)
colnames(notes_des_eleves) <- c("SVT", "Maths", "Francais")
rownames(notes_des_eleves) <- c("A", "B", "C", "D", "E")
notes_des_eleves
```

Affiche une matrice avec noms de lignes et de colonnes.

```r
notes_des_eleves["A", "Francais"]
notes_des_eleves[c("A", "D"), "Francais"]
```

Renvoie par exemple :

- `notes_des_eleves["A", "Francais"]` : `12` - `notes_des_eleves[c("A", "D"), "Francais"]` : `[1] 12 15`

Modifier une valeur :

```r
notes_des_eleves[c("A", "D"), "Francais"] <- c(15, 10)
```

8.6 Opérations sur matrices

Matrice de pondération :

```r
notes_ponderees <- matrix(c(rep(0.5, 5), rep(1, 5), rep(0.9, 5)),
   ncol = 3)
notes_des_eleves * notes_ponderees
```

Fonctions utiles :

```r
rowSums(notes_des_eleves)      # Somme par élève
colMeans(notes_des_eleves)     # Moyenne par matière
t(notes_des_eleves)            # Transposition
dim(notes_des_eleves)          # Dimensions
```

9 Les data frames

9.1 Définition

Un **data frame** est l'une des structures de données les plus utilisées en R. C'est une table bidimensionnelle où chaque colonne peut contenir un type de données différent (numérique, caractère, logique, etc.), mais toutes les colonnes doivent avoir la même longueur.

C'est le format le plus courant pour manipuler des jeux de données.

9.2 Créer un data frame à partir d'une matrice

```r
mat = matrix(1:9, ncol = 3)
as.data.frame(mat)
```

Renvoie :

```
  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9
```

9.3 Créer un data frame à partir de vecteurs hétérogènes

```r
age   = c(24, 26, 22)
sexe  = c("H", "H", "F")
etud  = c(TRUE, FALSE, TRUE)

df = data.frame(age, sexe, etud)
df
```

Renvoie :

```
  age sexe etud
1  24    H TRUE
2  26    H FALSE
3  22    F TRUE
```

9.4 Nommer les colonnes et lignes

```r
colnames(df) = c(" ge ", "Sexe", "Étudiant")
rownames(df) = c("A", "B", "C")
df
```

Renvoie :

```
  Âge Sexe Étudiant
A  24    H     TRUE
B  26    H    FALSE
C  22    F     TRUE
```

9.5 Accéder aux éléments

```r
df$Sexe            # Accès à une colonne (vecteur)
df[1, ]            # Accès à la 1ère ligne
df["A", " ge "]     # Accès à une case précise
df[ , "Étudiant"] # Colonne Étudiant
```

Résultats typiques :

- `df$Sexe` : `[1] "H" "H" "F"` - `df[1, ]` : affiche la 1ère ligne - `df["A", "Âge"]` : `24` - `df[ , "Étudiant"]` : `[1] TRUE FALSE TRUE`

9.6 Modifier le contenu

```r
df["A", "Étudiant"] <- FALSE
df$Sexe[3] <- "H"
```

9.7 Ajouter une colonne ou une ligne

```r
# Ajouter une colonne
df$Ville <- c("Paris", "Lyon", "Caen")

# Ajouter une ligne
df["D", ] <- list(30, "F", TRUE, "Rennes")
```

9.8 Fonctions utiles sur les data frames

```r
str(df)            # Structure du data frame
dim(df)            # Dimensions (lignes, colonnes)
nrow(df)           # Nombre de lignes
ncol(df)           # Nombre de colonnes
colnames(df)       # Noms des colonnes
rownames(df)       # Noms des lignes
summary(df)        # Résumé statistique
```

10 Conditions, boucles et fonctions

Les langages de programmation incluent tous des mécanismes de contrôle de flux. En R, on utilise les instructions `if`, les boucles `for` et `while`, et bien sûr les `fonctions` personnalisées.

Comparateurs et opérateurs logiques

```
# Comparateurs simples : supérieur, inférieur
5 > 3          # vrai
4 < 6          # vrai

# Comparateurs inclusifs : inférieur ou égal, supérieur ou égal
4 <= 4         # vrai
4 >= 4         # vrai

# Égalité et différence
3 == 3         # égalité
3 != 3         # différence (faux ici)

# Opérateurs logiques : AND (&), OR (|), version scalaire (&&, ||)
2 == 2 & 1 == 1     # les deux conditions doivent être vraies
2 == 2 && 1 == 2    # ET logique mais retourne un seul booléen (faux
   ici)

# Comparaison entre vecteurs (élément par élément)
c(2, 2) == c(2, 2) & c(3, 2) == c(3, 3)

# OU logique : au moins une condition vraie
2 == 2 | 1 == 2       # vrai (car 2 == 2)
2 == 2 & 1 == 2       # faux (2e condition fausse)
```

Instruction `if`

```
# Structure conditionnelle de base
if (5 > 3 | 2 < 3) {
  print("OK")
} else {
  print("Pas vrai")
}
```

Boucle `for`

```
# Parcourir un vecteur et incrémenter les valeurs
for (valeur in c(1, 2, 3, 4, 5)) {
  print(valeur + 1)
}

# Afficher un texte avec chaque valeur
for (element in c(1, 2, 3, 4, 5)) {
```

```r
  print(paste("Mon chiffre :", element))
}

# Afficher uniquement les éléments compris entre 2 et 4
for (element in c(1, 2, 3, 4, 5)) {
  if (element > 1 & element < 5) {
    print(element)
  }
}
```

Boucle `while`

```r
# Répéter une opération tant qu'une condition est vraie
valeur <- 200

while (valeur / 5 > 1) {
  valeur <- valeur / 5
  print(valeur)
}
```

Application avec le jeu de données iris

```r
data(iris)

# Compter le nombre de longueurs de sépales >= 5
compteur_in_sup_5 <- 0
for (longueur in iris$Sepal.Length) {
  if (longueur >= 5) {
    compteur_in_sup_5 <- compteur_in_sup_5 + 1
  }
}
print(compteur_in_sup_5)

# Compter le nombre d'observations par espèce
compteur_set <- 0
compteur_ver <- 0
compteur_virg <- 0

for (species in iris$Species) {
  if (species == "setosa") {
    compteur_set <- compteur_set + 1
  } else if (species == "versicolor") {
    compteur_ver <- compteur_ver + 1
  } else {
```

```r
    compteur_virg <- compteur_virg + 1
  }
}

print(paste("nombre de setosa :", compteur_set))
print(paste("nombre de versicolor :", compteur_ver))
print(paste("nombre de virginica :", compteur_virg))

# Compter les setosa ayant une longueur de sépale > 5
nombre_setosa_supe_5 <- 0
for (ligne in 1:dim(iris)[1]) {
  individu <- iris[ligne, ]
  if (individu$Species == "setosa" & individu$Sepal.Length > 5) {
    nombre_setosa_supe_5 <- nombre_setosa_supe_5 + 1
  }
}
print(nombre_setosa_supe_5)
```

Créer ses propres fonctions

```r
# Fonction qui compte les longueurs > 5 dans Sepal.Length
total_sup_5 <- function(dataframe) {
  compteur <- 0
  for (val in dataframe$Sepal.Length) {
    if (val > 5) {
      compteur <- compteur + 1
    }
  }
  return(compteur)
}

# Fonction qui calcule la moyenne pour les setosa
moyenne_setosa <- function(dataframe) {
  total <- 0
  nb <- 0
  for (i in 1:nrow(dataframe)) {
    if (dataframe[i, "Species"] == "setosa") {
      total <- total + dataframe[i, "Sepal.Length"]
      nb <- nb + 1
    }
  }
  return(total / nb)
}

# Utilisation des fonctions créées
print(total_sup_5(iris))
```

```r
print(moyenne_setosa(iris))
```

11 Conditions, boucles et fonctions

Dans cette partie, nous allons découvrir les bases de la logique en programmation avec R : tester des conditions, répéter des actions, et créer des fonctions réutilisables.

Ce sont des outils essentiels pour rendre vos analyses automatiques, puissantes et intelligentes.

11.1 Tester des conditions avec `if`

L'instruction `if` permet d'exécuter un bloc de code **seulement si une condition est vraie**.

```r
# Exemple : tester si une condition est vraie
if (5 > 3) {
  print("C'est vrai !")
} else {
  print("C'est faux.")
}
```

On peut aussi utiliser plusieurs comparateurs :

```r
# Comparateurs classiques
4 < 6          # inférieur
4 <= 4         # inférieur ou égal
3 == 3         # égalité
3 != 4         # différence

# Comparaison multiple avec ET et OU
2 == 2 & 1 == 1    # les deux conditions doivent être vraies (ET)
2 == 2 | 1 == 2    # une seule condition suffit (OU)
```

11.2 Répéter des actions avec `for`

Une boucle `for` permet de **répéter un bloc de code** pour chaque élément d'un vecteur.

```r
# Ajouter 1 à chaque nombre de la liste
for (valeur in c(1, 2, 3, 4, 5)) {
  print(valeur + 1)
}
```

```r

# Afficher un texte pour chaque valeur
for (element in c(1, 2, 3, 4, 5)) {
  print(paste("Mon chiffre :", element))
}

# Afficher uniquement les chiffres entre 2 et 4
for (element in c(1, 2, 3, 4, 5)) {
  if (element > 1 & element < 5) {
    print(element)
  }
}
```

11.3 Répéter tant qu'une condition est vraie : `while`

La boucle `while` permet de répéter une action **tant qu'une condition reste vraie**.

```r
# Diviser la valeur par 5 tant que le résultat est supérieur à 1
valeur <- 200

while (valeur / 5 > 1) {
  valeur <- valeur / 5
  print(valeur)
}
```

11.4 Exemple pratique avec le jeu de données `iris`

```r
# Charger les données
data(iris)

# Compter combien de fleurs ont une longueur de sépale supérieure ou égale à 5
compteur_in_sup_5 <- 0
for (longueur in iris$Sepal.Length) {
  if (longueur >= 5) {
    compteur_in_sup_5 <- compteur_in_sup_5 + 1
  }
}
print(compteur_in_sup_5)
```

11.5 Boucler sur une colonne texte : les espèces

```r
# Compter le nombre de fleurs de chaque espèce
compteur_set <- 0
compteur_ver <- 0
compteur_virg <- 0

for (species in iris$Species) {
  if (species == "setosa") {
    compteur_set <- compteur_set + 1
  } else if (species == "versicolor") {
    compteur_ver <- compteur_ver + 1
  } else {
    compteur_virg <- compteur_virg + 1
  }
}

# Afficher les résultats
print(paste("Nombre de setosa :", compteur_set))
print(paste("Nombre de versicolor :", compteur_ver))
print(paste("Nombre de virginica :", compteur_virg))
```

11.6 Créer sa propre fonction en R

Une fonction est un petit programme réutilisable. Elle prend des arguments en entrée, exécute des calculs, et retourne un résultat.

```r
# Exemple : fonction qui compte combien de sépales > 5
total_sup_5 <- function(dataframe) {
  compteur <- 0
  for (val in dataframe$Sepal.Length) {
    if (val > 5) {
      compteur <- compteur + 1
    }
  }
  return(compteur)
}

# Exemple : fonction qui calcule la moyenne de Sepal.Length pour les
   setosa
moyenne_setosa <- function(dataframe) {
  total <- 0
  nb <- 0
  for (i in 1:nrow(dataframe)) {
    if (dataframe[i, "Species"] == "setosa") {
      total <- total + dataframe[i, "Sepal.Length"]
```

```
      nb <- nb + 1
    }
  }
  return(total / nb)
}

# Utiliser nos fonctions
print(total_sup_5(iris))
print(moyenne_setosa(iris))
```

12 Manipulation de données en R

Manipuler des données signifie les filtrer, les transformer, les résumer ou les grouper.

Cette section introduit des fonctions puissantes de R pour faire cela efficacement :

- Fonctions `apply()`, `by()`, `aggregate()`
- Manipulations avec le package `dplyr` : `select`, `filter`, `arrange`, `mutate`, `group_by`...
- Utilisation d'exemples concrets sur les jeux de données `iris` et `FastFood`

12.1 La fonction `apply`

`apply()` permet d'appliquer une fonction à chaque ligne ou colonne d'une matrice ou d'un data frame numérique.

```
# Charger les données
data(iris)

# Moyenne par colonne (2 = colonne)
apply(iris[, -5], 2, mean)

# Moyenne par ligne (1 = ligne)
apply(iris[, -5], 1, mean)

# Résumé statistique par colonne
apply(iris[, -5], 2, summary)
```

On peut aussi utiliser une fonction personnalisée :

```
# Compter les valeurs > 5
nombre_val_sup_a_5 <- function(v) {
  length(v[v > 5])
}

```

```r
apply(iris[, -5], 2, nombre_val_sup_a_5)
apply(iris[, -5], 1, nombre_val_sup_a_5)
```

12.2 Les fonctions `by()` et `aggregate()`

`by()` applique une fonction à chaque sous-groupe d'un facteur. `aggregate()` retourne un data frame résumant par groupe.

```r
# Statistiques par espèce
by(iris, iris$Species, summary)

# Corrélation par espèce
by(iris[, -5], iris$Species, cor)

# Moyenne de chaque variable par espèce
aggregate(iris[, -5], as.data.frame(iris$Species), mean)
```

12.3 Manipulations avec le package `dplyr`

`dplyr` est un package dédié à la manipulation de données, avec une syntaxe simple et fluide.

```r
library(dplyr)

# Convertir iris en tibble (plus lisible)
iris <- as_tibble(iris)

# Sélection de colonnes
select(iris, Sepal.Length, Petal.Length, Species)
select(iris, starts_with("Petal"))
select(iris, -Species)
select(iris, contains("al"))
```

```r
# Filtrer des lignes
filter(iris, Sepal.Length >= 5, Sepal.Width >= 2)
filter(iris, between(Sepal.Length, 4, 7))
filter(iris, Species == "setosa")
filter(iris, Species %in% c("setosa", "versicolor"))
```

```r
# Trier les données
iris %>% arrange(Sepal.Length)
iris %>% arrange(desc(Sepal.Length))
```

```r
# Résumés statistiques
iris %>%
  summarise(moyenne_petal = mean(Petal.Length),
```

```
              max_petal = max(Petal.Length),
              total = n())
```

```
# Grouper les données
iris %>%
  group_by(Species) %>%
  summarise(moyenne_sepal = mean(Sepal.Length),
              max_sepal = max(Sepal.Length),
              total = n())
```

```
# Ajouter une colonne calculée
iris %>% mutate(somme_petal = Petal.Length + Petal.Width)

# Supprimer une colonne
iris %>% mutate(Species = NULL)
```

Utilisation du pipe %>% : permet de chaîner les opérations de façon lisible :

```
iris %>%
  select(-Species) %>%
  filter_all(all_vars(. > 2))
```

12.4 Exemple : analyse des restaurants Fast Food (USA)

But : explorer un fichier CSV contenant les emplacements de fast foods.

```
# Charger le fichier CSV
fast_food <- read.csv("FastFoodRestaurants.csv")
fast_food <- as_tibble(fast_food)
```

1. Les 5 villes avec le plus de fast food

```
top_villes <- fast_food %>%
  group_by(city) %>%
  summarise(nb = n()) %>%
  arrange(desc(nb)) %>%
  head(5)
```

2. Marques les plus présentes dans ces villes

```
city_list <- pull(top_villes, city)

fast_food %>%
  filter(city %in% city_list) %>%
  group_by(name) %>%
  summarise(nombre = n()) %>%
  arrange(desc(nombre))
```

3. Fast food le plus fréquent aux USA

```
fast_food %>%
  group_by(name) %>%
  summarise(nb = n(), pourcentage = n() * 100 / nrow(fast_food)) %>%
  arrange(desc(nb))
```

4. Ville avec le plus de McDonald's

```
fast_food %>%
  filter(name == "McDonald's") %>%
  group_by(city) %>%
  summarise(nb = n()) %>%
  arrange(desc(nb))
```