Module 1: Introduction à la programmation R

Table of contents

Objectif du module

Ce module vise à initier les apprenants aux bases du langage R, en couvrant les concepts fondamentaux, l'environnement de travail, et les manipulations de données simples.

1. Introduction à R et RStudio

Présentation du langage R : historique et applications

R est un **langage de programmation** et un **logiciel de statistique** développé par **Ross Ihaka** et **Robert Gentleman** dans les années 1990. Il constitue à la fois un **environnement**

interactif permettant la manipulation de données et un langage de programmation conçu pour **l'analyse statistique avancée**. Les commandes y sont exécutées sous forme de scripts clairs et intuitifs, avec une gestion efficace des résultats sous forme de **tableaux, graphiques et rapports détaillés**.

Origine et évolution

R est issu du langage **S**, développé par **AT&T Bell Laboratories** en 1976 et rendu disponible sous la version commerciale **S-Plus** en 1988. Sa création visait à offrir un environnement puissant pour les analyses statistiques et les simulations complexes. Grâce à son architecture open-source, R a rapidement gagné en popularité et est devenu l'un des outils incontournables dans de nombreux **domaines scientifiques et industriels**.

Un logiciel libre et multiplateforme

L'un des atouts majeurs de R est qu'il est **gratuit** et disponible sous **licence libre** (*open-source*). Il fonctionne sur **UNIX/Linux, Microsoft Windows et MacOS**, ce qui lui confère une large accessibilité. Sa nature open-source permet aux utilisateurs de développer leurs propres extensions, d'améliorer les fonctionnalités et de partager leurs contributions au sein d'une communauté mondiale dynamique.

Applications et domaines d'utilisation

R est aujourd'hui utilisé dans **divers secteurs**, notamment :
- **Statistique et recherche scientifique** : Analyse de données, modélisation statistique et tests d'hypothèses.
- **Finance et économie** : Prévisions financières, analyse des risques et optimisation des portefeuilles.
- **Biostatistique et médecine** : Études cliniques, analyses génétiques et traitement des données biomédicales.
- **Data science et intelligence artificielle** : Apprentissage automatique, traitement du big data et visualisation avancée.
- **Géographie et environnement** : Modélisation climatique, analyse spatiale et cartographie des données.

Grâce à ses bibliothèques spécialisées, R permet de traiter des volumes de données considérables et de produire des analyses précises et fiables. Son interface graphique avancée en fait un outil puissant pour la visualisation et la présentation des résultats sous forme de graphiques interactifs et tableaux dynamiques.

Une communauté dynamique

Développé dans l'esprit du **logiciel libre**, R bénéficie du soutien d'une communauté mondiale **active et engagée**. Des milliers de bénévoles contribuent en permanence à son amélioration, en proposant de nouvelles méthodes d'analyse, des packages innovants et des outils intégrés facilitant son usage. Cette évolution rapide assure à R une place de choix parmi les **meilleurs langages pour l'analyse statistique et le traitement des données**.
Avec son évolutivité et son adaptabilité, R continue d'être un **pilier incontournable** pour les scientifiques, analystes et ingénieurs à travers le monde.

Installation de R et RStudio

Installer R

- Rendez-vous sur le site officiel de R : https://cran.r-project.org.
- Choisis le système d'exploitation correspondant (**Windows**, **MacOS**, ou **Linux**).
- Télécharge la dernière version disponible.
- Une fois le fichier téléchargé, ouvre-le et suis les instructions d'installation.
- Choisis les options par défaut, sauf si tu as des besoins spécifiques.
- Une fois installé, vérifie en lançant **R GUI** (l'interface native de R).

Installer RStudio

RStudio est un environnement de développement interactif qui facilite l'utilisation de R. Pour l'installer :

- Accède au site officiel : https://posit.co/download/rstudio-desktop.
- Sélectionne la version adaptée à ton système d'exploitation (**Windows**, **MacOS**, ou **Linux**).
- Ouvre le fichier d'installation téléchargé et suis les étapes.
- Lance **RStudio** et vérifie que R est bien détecté (si tout est correct, la console affichera `[R version x.x.x]`).

Tester l'installation

Ouvrir RStudio et repérer la console

Exécuter une commande simple dans la console, par exemple :

```
    print("R est mon ami !")
```

```
[1] "R est mon ami !"
```

Si la phrase s'affiche, tout est bien installé et fonctionnel !

Installation de packages essentiels

RStudio propose des fonctionnalités enrichies grâce aux **packages**. Installe quelques indispensables avec ce code:

```
install.packages("tidyverse")  # Pour la manipulation de données
install.packages("ggplot2")    # Pour la visualisation graphique
install.packages("dplyr")      # Pour le traitement des données
```

Une fois ces étapes terminées, tu es prêt(e) à explorer **R** et **RStudio** !

2. Bases de la programmation en R

R, comme beaucoup d'autres langages de ce type, remplace aisément les fonctionnalités d'une calculatrice (très sophistiquée !). Sa grande force réside également dans sa capacité à effectuer des calculs sur des vecteurs. Voici quelques exemples très simples :

```
x <- 5 * (3.2)  # Attention, le séparateur décimal doit être un point (.)
x
```

```
[1] 16
```

```
5 * (-3,2)  # Sinon, l'erreur suivante est générée :
```

```
Error in parse(text = input): <text>:1:8: ',' inattendu(e)
1: 5 * (-3,
           ^
```

```r
5^2  # Identique à 5**2.
```

```
[1] 25
```

```r
sin(2 * pi / 3)
```

```
[1] 0.8660254
```

```r
sqrt(4)  # Racine carrée de 4.
```

```
[1] 2
```

```r
c(1, 2, 3, 4, 5)  # Crée un vecteur contenant les cinq premiers entiers.
```

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

Les variables et affectations (<-)

Comme vous l'avez sans doute remarqué, R répond à vos requêtes en affichant le résultat obtenu après évaluation. Ce résultat est affiché puis perdu. Lors d'une première utilisation, cela peut sembler pratique, mais dans un usage plus avancé, il devient plus intéressant de rediriger la sortie de votre requête en la stockant dans une variable. Cette opération s'appelle l'affectation du résultat dans une variable. Une affectation évalue ainsi une expression, mais n'affiche pas le résultat qui est en réalité stocké dans un objet. Pour afficher ce résultat, il suffit de taper le nom de cet objet, suivi de la touche Entrée. Affectation d'une variable Pour réaliser cette opération, on utilise la flèche d'affectation <-. Elle s'obtient en tapant le signe inférieur < suivi du signe moins -. La syntaxe pour créer un objet dans R est la suivante : Nom_objet_a_creer <- instructions

```r
Nom_objet_a_creer <- "instructions"
x <- 6  # Affectation
x       # Affichage
```

```
[1] 6
```

On dit alors que x vaut 6, ou que l'on affecte 6 à x, ou encore que l'on stocke la valeur 6 dans x ### Affectation dans l'autre sens: Notez que l'on peut aussi utiliser l'opération d'affectation dans l'autre sens -> de la manière suivante :

```r
12 -> y
y
```

```
[1] 12
```

Règles pour choisir un nom de variable dans R

Dans R, un nom de variable doit respecter les conventions suivantes : Caractères autorisés : Un nom de variable ne peut être constitué que de caractères alphanumériques (a-z, A-Z, 0-9) ainsi que du point (.). Sensible à la casse : R distingue les majuscules des minuscules (maVariable et mavariable sont considérés comme différents). **Interdictions :** Un nom de variable ne peut pas contenir d'espaces. Il ne peut pas commencer par un chiffre, sauf s'il est encadré de guillemets (“ “). Exemples de noms valides et invalides

```r
# Valides
ma_variable <- 10
MaVariable <- "texte"
var2 <- 5.6
taux.de.croissance <- 0.03  # Utilisation du point

#  Invalides
2variable <- "erreur"  # Commence par un chiffre
ma variable <- 100      # Contient un espace
mavariable! <- TRUE     # Contient un caractère spécial
```

Prise en main de R:

Exercice 1

Commencez par créer un dossier nommé TravauxR dans votre compte. Ensuite, tapez et enregistrez dans un script R les instructions précédentes. Le fichier contenant le script R sera nommé monscript.R et placé dans TravauxR. Maintenant, fermez puis rouvrez R. Enfin, modifiez votre répertoire de travail courant pour qu'il pointe vers TravauxR. exécuter dans la console la commande: source(“monscript.R”) Notez que la fonction source() permet d'exécuter votre script. Vous aurez peut-être aussi remarqué que les calculs qui n'ont pas été redirigés dans des variables ne sont pas affichés dans la console. Ainsi, leur résultat est perdu. Modifiez votre script et ajoutez-y, à la fin, les instructions suivantes :

```r
print(2*3)
print(x)
```

Sauvegarder-le, puis sourcez-le de nouveau. Que s'est-il passé ?

Prenez l'habitude d'utiliser le système d'aide en ligne de R aussi souvent que possible. Cette aide, très complète (mais uniquement disponible en anglais), est accessible via la fonction help(). Par exemple, vous pouvez taper :

```
help(source)
```

Cela vous permettra d'obtenir des informations détaillées sur la fonction source().

Exercice 2:

L'Indice de Masse Corporelle (IMC) permet de déterminer la corpulence d'une personne. Il se calcule à l'aide de la formule suivante : ** `IMC = Poids (kg)/Taille^2` ** Calculons notre IMC Pour effectuer ce calcul, il suffit de taper les lignes suivantes dans votre fenêtre de script :

```
# Il est possible d'écrire plusieurs instructions sur la même ligne grâce au signe ";"
Mon.Poids <- 63 ; Ma.Taille <- 1.70
Mon.IMC <- Mon.Poids / Ma.Taille^2
Mon.IMC  # Affichage du résultat
```

```
[1] 21.79931
```

Lancez ce script en suivant la stratégie de travail vue précédemment. Vous pouvez ensuite modifier ce script pour calculer votre propre IMC en adaptant les valeurs de `Mon.Poids` et `Ma.Taille`

Utilisation des fonctions dans R

R propose de nombreuses fonctions intégrées, comme sin(), sqrt(), exp() et log(), et permet d'en ajouter des milliers d'autres via des packages ou en créant ses propres fonctions. #### Définition et appel de fonction Une fonction est caractérisée par son nom et une liste de paramètres. La plupart des fonctions renvoient une valeur pouvant être un nombre, un vecteur, une matrice, etc. L'utilisation d'une fonction se fait en tapant son nom, suivi d'une paire de parenthèses contenant les paramètres. Les paramètres sont séparés par des virgules, et peuvent être précisés avec = pour leur attribuer une valeur :

```
nomfonction(par1 = valeur1, par2 = valeur2, par3 = valeur3)
```

Cependant, il est possible d'appeler une fonction sans nommer explicitement les paramètres, tant que leur ordre est respecté. # Paramètres obligatoires et facultatifs Dans R, certaines fonctions nécessitent des paramètres obligatoires, tandis que d'autres acceptent des valeurs par défaut. À noter que R utilise le terme argument pour désigner ce que l'on appelle ici paramètre.

Appel de fonctions et manipulation des paramètres dans R

Dans R, une même fonction peut être appelée de différentes façons, en jouant sur l'ordre et la définition explicite des paramètres. Cette flexibilité rend l'utilisation de R très intuitive. Exemple : Calcul du logarithme népérien de 3 Les expressions suivantes sont équivalentes et permettent d'obtenir log(3), qui correspond au logarithme népérien de 3 :

```
log(3)
```

```
[1] 1.098612
```

```
log(x = 3)
```

```
[1] 1.098612
```

```
log(x = 3, base = exp(1))
```

```
[1] 1.098612
```

```
log(x = 3, exp(1))
```

```
[1] 1.098612
```

```
log(3, base = exp(1))
```

```
[1] 1.098612
```

```
log(3, exp(1))
```

```
[1] 1.098612
```

```r
log(base = exp(1), 3)
```

```
[1] 1.098612
```

```r
log(base = exp(1), x = 3)
```

```
[1] 1.098612
```

Cependant, l'expression suivante modifie la base du logarithme, ce qui change le résultat :

```r
log(exp(1), 3)  # Logarithme de exp(1) en base 3.
```

```
[1] 0.9102392
```

Création de fonctions personnalisées

R permet aussi de définir ses propres fonctions très facilement. Par exemple, la fonction factorial() est définie ainsi dans R :

```r
factorial <- function(x) gamma(x + 1)
```

Un exemple de fonction personnalisée pour calcUler l'IMC :

```r
MyIMC <- function(Poids, Taille) {
  Poids / (Taille)^2
}
MyIMC(80,1.65)
```

```
[1] 29.38476
```

La capacité de R à gérer les paramètres de plusieurs manières et à permettre la création de fonctions personnalisées le rend très puissant et modulable pour tout type d'analyse.

3. Types de données dans R

R reconnaît automatiquement le type des données saisies et permet de les organiser de manière structurée, ce qui est essentiel pour les analyses statistiques.

Gestion des types de données

- `typeof()` : Détermine le **type** d'un objet.
- `mode()` : Fonction similaire à `typeof()`, avec quelques subtilités.
- `class()` : Plus générale, elle permet de gérer le **type et la structure** des données. ### **Types numériques (`numeric`)** R distingue deux sous-types :
- **Entiers (`integer`)** : Moins gourmands en mémoire.
- **Réels (`double`)** : Par défaut, toutes les valeurs numériques sont considérées comme des `double`.

```
a <- 1
b <- 3.4
c <- as.integer(a)  # Conversion en entier
typeof(c)  # Renvoie "integer"
```

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

Types complexes (`complex`)

Les nombres complexes sont créés avec la lettre **i**. **Fonctions utiles :**
- `Re(x)` : Partie réelle.
- `Im(x)` : Partie imaginaire.
- `Mod(x)` : Module.
- `Arg(x)` : Argument.

```
z <- 1 + 2i
  Re(z)  # Partie réelle → [1] 1
  Im(z)  # Partie imaginaire → [1] 2
  Mod(z) # Module → [1] 2.236
  Arg(z) # Argument → [1] 1.107
  typeof(z)
  is.complex(z)
```

Type booléen (`logical`)

Résultat d'une condition logique, il prend les valeurs `TRUE` ou `FALSE`.
- Vérifier le type : `is.logical(x)` - Comparaisons logiques :

```r
a <- 1; b <- 3.4
 a < b  # [1] TRUE
```

```
[1] TRUE
```

```r
 a == b # [1] FALSE
```

```
[1] FALSE
```

- **Attention** : `TRUE` = 1, `FALSE` = 0

```r
TRUE + FALSE  # [1] 1
```

```
[1] 1
```

Données manquantes (`NA`)

Les valeurs manquantes sont représentées par **NA**. - Vérifier si une donnée est manquante : `is.na(x)` - Calculer en ignorant les `NA` :

```r
 x <- c(3, NA, 6)
 mean(x, na.rm=TRUE)
```

```
[1] 4.5
```

- **Différence entre `NA`, `NaN` et `Inf`** :

```r
0/0  # NaN (Not a Number)
```

```
[1] NaN
```

```r
 3/0  # Inf (Infini)
```

```
[1] Inf
```

Type chaîne de caractères (`character`)

Toute valeur entre **guillemets** (`" "` ou `' '`) est une chaîne de caractères.
- Vérifier le type : `is.character(a)` - Convertir entre types :

```r
 as.character(2.3)  # "2.3"
as.numeric("2.3")  # 2.3
as.integer("3.4")  # 3
```

Données brutes (raw) dans R

R permet de manipuler directement des octets, affichés sous forme hexadécimale. Cette fonctionnalité est particulièrement utile pour lire et traiter des fichiers au format binaire.

```r
x <- as.raw(15)
x  # Affichage en hexadécimal → [1] 0f
```

```
[1] 0f
```

```r
mode(x)  # Vérification du type → [1] "raw"
```

```
[1] "raw"
```

Ce type de données est moins courant mais essentiel pour certaines opérations avancées. En somme, R offre une grande flexibilité pour manipuler et convertir les données, ce qui est essentiel pour le traitement statistique. ## **Récapitulatif**

Les differents types de données en R

Type de donées	Type sous R	Présentation
réel (entier ou non)	numeric	5.7
complexe	complex	7+2i
logique (vrai/faux)	logical	TRUE ou FALSE
manquant	logical	NA
texte (Chaine de caractères)	character	“texte”
binaires	raw	1c

4. Structures de données de base :

R offre la possibilité d’organiser et de structurer les différents types de données définis précédemment. La fonction `class()` permet de manipuler ces structures. Nous présentons ici les plus utiles.

Les vecteurs (`vector`)

Le vecteur est la structure de données la plus simple. Il représente une suite de données de même type. La fonction `c()` (pour collection ou concaténation) permet de créer des vecteurs. D'autres fonctions comme `seq()` ou l'utilisation des deux points (`:`) permettent également de générer des vecteurs. À noter que lors de la création d'un vecteur, il est possible de mélanger des données de différents types. Dans ce cas, R effectue une conversion implicite vers le type le plus fréquent, comme illustré ci-dessous :

```r
# Création de vecteurs
vec1 <- c(3, 1, 7)
print(vec1)  # Affichage du vecteur
```

```
[1] 3 1 7
```

```r
# Mélange de types : R effectue une conversion implicite
vec2 <- c(3, TRUE, 7)  # TRUE est converti en 1
print(vec2)
```

```
[1] 3 1 7
```

```r
vec3 <- c(3, T, "7")  # Tous les éléments sont convertis en caractères
print(vec3)
```

```
[1] "3"    "TRUE" "7"
```

```r
# Génération de séquences numériques
seq1 <- seq(from = 0, to = 1, by = 0.1)
print(seq1)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```r
seq2 <- seq(from = 0, to = 20, length.out = 5)  # Utilisation de length.out pour garantir le
print(seq2)
```

```
[1]  0  5 10 15 20
```

```r
# Création d'un vecteur avec l'opérateur `:`
vec <- 2:36
print(vec)  # Affichage du vecteur
```

```
 [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
[26] 27 28 29 30 31 32 33 34 35 36
```

Il est également possible de **nommer** les éléments d'un vecteur grâce à la fonction `names()`.

```r
vec <- c(1, 3, 6, 2, 7, 4, 8, 1, 0)
 names(vec) <- letters[1:9] # Attribution des 9 premières lettres de l'alphabet.
vec
```

```
a b c d e f g h i
1 3 6 2 7 4 8 1 0
```

Attention
Les indications `[1]` et `[26]` affichées dans la console correspondent à l'index du premier élément de chaque ligne dans le vecteur.

```r
# Vérification si vec est un vecteur
vec <- 2:36
print(is.vector(vec))  # Affiche TRUE
```

```
[1] TRUE
```

```r
# Comparaison entre integer et numeric
x <- 1:3  # Stocké comme integer
print(x)
```

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

```r
y <- c(1, 2, 3)  # Stocké comme numeric
print(y)
```

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

```r
# Affichage du type de chaque variable
print(class(x))  # Affiche "integer"
```

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

```r
print(class(y))  # Affiche "numeric"
```

```
[1] "numeric"
```

Bien que l'on puisse s'attendre à voir **"vector of doubles"** ou **"vector of integers"** affiché en sortie, R utilise plutôt les termes **"numeric"** et **"integer"** pour désigner ces types.

Les instructions `c()` et `:` produisent un affichage identique, mais en interne, `x` et `y` sont stockés différemment. Le type **integer** est plus économe en mémoire que le type **numeric**.

Les matrices (`matrix`) et les tableaux (`array`)

Les matrices et les tableaux généralisent la notion de vecteur. Une **matrice** représente une structure de données à **deux indices** (lignes et colonnes), tandis qu'un **tableau** peut avoir plusieurs indices (dimensions). Comme pour les vecteurs, tous les éléments doivent être du **même type**, sinon R effectue des conversions implicites.

Création de matrices

La fonction `matrix()` permet de créer une matrice. L'exemple suivant génère une matrice **remplie ligne par ligne** (`byrow = TRUE`) avec les valeurs de `1:12` :

```r
X <- matrix(1:12, nrow = 4, ncol = 3, byrow = TRUE)
print(X)
```

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

La matrice `X` contient **4 lignes** et **3 colonnes**, remplies **ligne par ligne**.

De la même manière, il est possible de **remplir une matrice colonne par colonne** (`byrow = FALSE`) :

```
Y <- matrix(1:12, nrow = 4, ncol = 3, byrow = FALSE)
print(Y)
```

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

La matrice `Y` est remplie **colonne par colonne**.

Pour vérifier le type de `Y`, on utilise :

```
print(class(Y))  # Affiche "matrix"
```

```
[1] "matrix" "array"
```

Création de tableaux multidimensionnels (`array`)

Les tableaux multidimensionnels permettent de **ajouter des dimensions supplémentaires**. La fonction `array()` permet de créer des structures à **plus de deux dimensions** :

```
X <- array(1:12, dim = c(2, 2, 3))
print(X)
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

La variable `X` contient **3 matrices (couches)** de **2 lignes et 2 colonnes** chacune.

Pour vérifier le type de `X` :

```r
print(class(X))  # Affiche "array"
```

```
[1] "array"
```

Attention

Il est possible de créer des **tableaux à plus de trois dimensions** grâce au **paramètre `dim`**, qui peut prendre **plus de trois valeurs** :

```r
multi_array <- array(1:24, dim = c(2, 3, 4))  # Tableau avec 4 dimensions
print(multi_array)
```

```
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18
```

```
, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

Les listes (`list`)

La structure **la plus souple** et **la plus riche** du langage R est celle des **listes**. Contrairement aux structures précédentes, les listes permettent de **regrouper dans une même structure** des données **de types différents** sans les altérer.
Chaque élément d'une liste peut être **un vecteur, une matrice, un tableau (`array`) ou même une autre liste**.

```r
A <- list(
  TRUE,
  -1:3,
  matrix(1:4, nrow = 2),
  c(1 + 2i, 3),
  "Une chaîne de caractères"
)
print(A)
```

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

[[2]]
[1] -1  0  1  2  3

[[3]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[4]]
[1] 1+2i 3+0i

[[5]]
[1] "Une chaîne de caractères"
```

On peut vérifier le type de l'objet `A` :

```r
print(class(A))  # Affiche "list"
```

```
[1] "list"
```

Nommer les éléments d'une liste

Les listes sont **hétérogènes** en types de données, et l'ordre des éléments peut être **arbitraire**. C'est pourquoi il est possible de **nommer explicitement** chaque élément afin d'améliorer la lisibilité des sorties. Un exemple de liste avec des **noms explicites** :

```r
B <- list(
  une.matrice = matrix(1:4, nrow = 2),
  des.complexes = c(1 + 2i, 3)
)
print(B)
```

```
$une.matrice
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$des.complexes
[1] 1+2i 3+0i
```

Listes imbriquées

Une liste peut **elle-même contenir d'autres listes**, ce qui permet de créer **des structures complexes et organisées** :

```r
liste1 <- list(complexe = 1 + 1i, logique = FALSE)
liste2 <- list(chaine = "J'apprends R", vecteur = 1:2)

C <- list(
  "Ma première liste" = liste1,
  Ma.seconde.liste = liste2
)
print(C)
```

```
$`Ma première liste`
$`Ma première liste`$complexe
[1] 1+1i

$`Ma première liste`$logique
[1] FALSE


$Ma.seconde.liste
$Ma.seconde.liste$chaine
[1] "J'apprends R"

$Ma.seconde.liste$vecteur
[1] 1 2
```

Nommer les éléments permet de **faciliter leur extraction**.

Le tableau individus × variables (`data.frame`)

Le tableau **individus × variables** est la structure **fondamentale en statistique**. Dans R, cette notion est représentée par le **`data.frame`**, qui fonctionne comme une **matrice**, où :
- **Les lignes** correspondent aux **individus**
- **Les colonnes** correspondent aux **variables** mesurées Chaque colonne représente **une variable unique** et tous ses éléments doivent être **du même type**. Les colonnes peuvent être **nommées** pour une meilleure lisibilité.

Création d'un `data.frame`

Voici un exemple de création d'un `data.frame` avec les informations de taille et poids selon le sexe :

```r
IMC <- data.frame(
  Sexe = c("H", "F", "H", "F", "H", "F"),
  Taille = c(1.83, 1.76, 1.82, 1.60, 1.90, 1.66),
  Poids = c(67, 58, 66, 48, 75, 55),
  row.names = c("Rémy", "Lol", "Pierre", "Domi", "Ben", "Cécile")
)

print(IMC)
```

```
       Sexe Taille Poids
Rémy      H   1.83    67
Lol       F   1.76    58
Pierre    H   1.82    66
Domi      F   1.60    48
Ben       H   1.90    75
Cécile    F   1.66    55
```

Vérification du type et de la structure du `data.frame`

```r
print(is.data.frame(IMC))  # Vérifie si IMC est bien un data.frame (TRUE)
```

```
[1] TRUE
```

```r
print(class(IMC))          # Affiche "data.frame"
```

```
[1] "data.frame"
```

```r
print(str(IMC))            # Affiche la structure détaillée du data.frame
```

```
'data.frame':   6 obs. of  3 variables:
 $ Sexe  : chr  "H" "F" "H" "F" ...
 $ Taille: num  1.83 1.76 1.82 1.6 1.9 1.66
 $ Poids : num  67 58 66 48 75 55
NULL
```

La fonction `str()` permet **d'afficher la structure** des colonnes du `data.frame`, y compris les **types de données** et **leurs niveaux**.
Un `data.frame` peut être **vu comme une liste** de **vecteurs de même longueur**. En effet, R **structure ses `data.frame` de cette façon** en interne :

```r
print(is.list(IMC))  # Vérifie si IMC est une liste (TRUE)
```

```
[1] TRUE
```

Cela signifie que chaque colonne d'un `data.frame` est en réalité **un vecteur** que l'on peut manipuler individuellement.

Les facteurs (`factor`) et les variables ordinales (`ordered`)

R permet d'organiser les **chaînes de caractères** de manière plus efficace grâce à la fonction `factor()`.

Création d'un facteur

L'exemple suivant crée un facteur (`factor`) contenant des couleurs :

```
x <- factor(c("bleu", "vert", "bleu", "rouge", "bleu", "vert", "vert"))
print(x)
```

```
[1] bleu  vert  bleu  rouge bleu  vert  vert
Levels: bleu rouge vert
```

Pour connaître les **niveaux (`levels`)** du facteur :

```
print(levels(x))  # Affiche : "bleu" "rouge" "vert"
```

```
[1] "bleu"  "rouge" "vert"
```

Pour vérifier le type de l'objet `x` :

```
print(class(x))  # Affiche "factor"
```

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

Recodage d'une variable continue en facteur

La fonction `cut()` permet de **transformer une variable continue** en **facteurs**.

```
Poids <- c(55, 63, 83, 57, 75, 90, 73, 67, 58, 84, 87, 79, 48, 52)
Poids_categorise <- cut(Poids, 3)
print(Poids_categorise)
```

```
 [1] (48,62] (62,76] (76,90] (48,62] (62,76] (76,90] (62,76] (62,76] (48,62]
[10] (76,90] (76,90] (76,90] (48,62] (48,62]
Levels: (48,62] (62,76] (76,90]
```

R regroupe les valeurs de `Poids` en **trois intervalles**, facilitant l'analyse statistique.

Facteurs et `data.frame`

Il est possible d'inclure des **facteurs** dans un `data.frame`. R indique automatiquement les **différents niveaux (`levels`)** de la variable qualitative.

```
IMC <- data.frame(
  Sexe = factor(c("H", "F", "H", "F", "H", "F")),
  Taille = c(1.83, 1.76, 1.82, 1.60, 1.90, 1.66),
  Poids = c(67, 58, 66, 48, 75, 55)
)
print(IMC)
```

```
  Sexe Taille Poids
1    H   1.83    67
2    F   1.76    58
3    H   1.82    66
4    F   1.60    48
5    H   1.90    75
6    F   1.66    55
```

```
print(str(IMC))  # Affiche la structure du data.frame
```

```
'data.frame':   6 obs. of  3 variables:
 $ Sexe  : Factor w/ 2 levels "F","H": 2 1 2 1 2 1
 $ Taille: num  1.83 1.76 1.82 1.6 1.9 1.66
 $ Poids : num  67 58 66 48 75 55
NULL
```

Variables ordinales avec `ordered()`

Les variables **ordinales** ont un **ordre défini** entre leurs catégories. Il est recommandé d'utiliser `ordered()` pour ces variables :

```
z <- ordered(
  c("Petit", "Grand", "Moyen", "Grand", "Moyen", "Petit", "Petit"),
  levels = c("Petit", "Moyen", "Grand")
)
print(class(z))  # Affiche "ordered" "factor"
```

```
[1] "ordered" "factor"
```

L'option `levels` permet de **spécifier l'ordre** des modalités de la variable.

Création rapide de facteurs avec `gl()`

La fonction `gl()` permet de **générer des facteurs** en spécifiant le **nombre de niveaux** et **le nombre de répétitions** :

```
facteur_exemple <- gl(n = 2, k = 8, labels = c("Control", "Treat"))
print(facteur_exemple)
```

```
 [1] Control Control Control Control Control Control Control Control Treat
[10] Treat   Treat   Treat   Treat   Treat   Treat   Treat
Levels: Control Treat
```

Dans `gl(n = 2, k = 8, labels = c("Control", "Treat"))` :
- `n = 2` → Nombre de **niveaux** (ici, **"Control" et "Treat"**)
- `k = 8` → Nombre de **répétitions** pour chaque niveau

R **optimise la mémoire** en codant les facteurs sous la forme **d'entiers**, réduisant l'espace utilisé lorsque les valeurs se répètent fréquemment.

Les dates en R

R permet de manipuler les **données temporelles** grâce à la fonction `as.Date()`, qui convertit une **chaîne de caractères** en **objet Date**.

Conversion de chaînes de caractères en dates

L'exemple suivant illustre la conversion d'un vecteur de dates au format `"jour/mois/année"` en **format Date** :

```
# Vecteur de dates sous forme de chaînes de caractères
dates <- c("27/02/92", "27/02/92", "14/01/92", "28/02/92", "01/02/92")

# Conversion en format Date avec as.Date()
dates <- as.Date(dates, format = "%d/%m/%y")

# Affichage des dates converties
print(dates)
```

```
[1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

```r
# Vérification du type de l'objet
print(class(dates))  # Affiche "Date"
```

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

R a correctement interprété les dates et les a converties en **format standard ISO (`YYYY-MM-DD`)**, facilitant leur manipulation dans des analyses temporelles.

Explication du format `"%d/%m/%y"`

Dans `as.Date(dates, format = "%d/%m/%y")` :
- **`%d`** → Jour (exemple : `"27"`)
- **`%m`** → Mois (exemple : `"02"`)
- **`%y`** → Année à **deux chiffres** (`92` devient `1992`) Si l'année avait **quatre chiffres**, on utiliserait **`%Y`** :

```r
dates_complet <- as.Date("27/02/1992", format = "%d/%m/%Y")
print(dates_complet)  # Affiche "1992-02-27"
```

```
[1] "1992-02-27"
```

Les séries temporelles en R

Lorsque les données sont indexées par le **temps**, il peut être utile de les organiser dans une structure qui reflète cet aspect temporel. R offre la fonction **`ts()`** pour créer des **séries temporelles**, facilitant l'analyse et la modélisation des données chronologiques.

Création d'une série temporelle avec `ts()`

L'exemple suivant montre comment **générer une série temporelle** avec une fréquence de **4 observations par an** (trimestres) et un point de départ **au deuxième trimestre de 1959** :

```r
serie_temporelle <- ts(1:10, frequency = 4, start = c(1959, 2))
print(serie_temporelle)
```

```
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
```

Explication des paramètres

Dans `ts(1:10, frequency = 4, start = c(1959, 2))` :
- `1:10` : Données de la série temporelle
- **`frequency = 4`** : Indique **4 périodes par an** (trimestres)
- **`start = c(1959, 2)`** : Démarre la série **au deuxième trimestre de 1959**
Si les données étaient **mensuelles**, on utiliserait **`frequency = 12`** (12 mois par an) :

```r
serie_mensuelle <- ts(1:24, frequency = 12, start = c(2000, 1))
print(serie_mensuelle)
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2000   1   2   3   4   5   6   7   8   9  10  11  12
2001  13  14  15  16  17  18  19  20  21  22  23  24
```

Applications des séries temporelles

Les **séries temporelles** sont largement utilisées en :
- Analyse économique et financière
- Prédiction des tendances climatiques
- Études de consommation et de comportement

Termes à retenir

Affectation des variables
- `<-`, `->` : Flèches d'affectation permettant d'assigner une valeur à une variable

Types d'objets et structures de données
- `mode()`, `typeof()` : Identifie **la nature d'un objet**
- `is.numeric()` : Vérifie si un objet est **numérique**
- `TRUE`, `FALSE`, `is.logical()` : Valeurs **booléennes** et test logique
- `is.character()` : Vérifie si un objet est **une chaîne de caractères**
- `NA`, `is.na()` : **Valeur manquante**, et test de présence de **valeurs manquantes**
- `class()` : Détermine **la structure d'un objet**

Création de structures de données
- `c()` : Crée un **vecteur** (suite d'éléments de même type)
- `matrix()`, `array()` : Génère une **matrice** ou un **tableau multidimensionnel**
- `list()` : Crée une **liste**, permettant de contenir des **éléments de types différents**
- `data.frame()` : Crée un **tableau individus $\times$ variables** (structure centrale en **statistique**)
- `factor()` : Génère un **facteur**, utile pour stocker **des variables qualitatives**

5. EXERCICES & TRAVAUX PRATIQUES

EXERCICE:

1.1 - Que renvoie cette instruction:

```
1:3^2
```

1.2 - Que renvoie cette instruction/

```
(1:5) * 2
```

1.3 - Que renvoie cette instruction :

```
var <- 3
Var * 2
```

1.4 -Que renvoie cette instruction:

```
x <- 2
2x <- 2 * x
```

1.5 - Que renvoie cette instruction :

```
racine.de.quatre <- sqrt(4)
racine.de.quatre
```

1.6 - Que renvoie cette instruction :

```
x <- 1
x < -1
```

1.7 - Que renvoie cette instruction ?

```
Un chiffre pair <- 16
```

1.8 - Que renvoie cette instruction:

```
"Un chiffre pair" <- 16
```

1.9 - Que renvoie cette instruction :

```
"2x" <- 14
```

1.10 - Que renvoie cette instruction ?

```
Un chiffre pair
```

1.11 - Complétez cette sortie où deux symboles ont été omis :

```
> 2
+
[1] 6
```

1.12 - Que renvoie cette instruction :

```
TRUE + T + FALSE * F + T * FALSE + F
```

1.13 - Quels sont les cinq types de données en R ?

1.14 - Écrivez l'instruction R permettant d'obtenir cet affichage :

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

1.15 - Quelles sont les structures (`classes`) de données disponibles en R ?

TP

Étude sur l'indice de masse corporelle

Un échantillon de dossiers d'enfants a été collecté. Il concerne des enfants examinés lors d'une visite en première section de maternelle au cours de l'année 1996-1997 dans des écoles de Bordeaux, en Gironde, France.
L'échantillon présenté ici est composé de dix enfants âgés de trois ou quatre ans.
Données disponibles pour chaque enfant
- Sexe : F pour fille et G pour garçon
- École en ZEP (zone d'éducation prioritaire) : O pour oui et N pour non
- Âge : deux variables distinctes, une pour le nombre d'années et une pour le nombre de mois
- Poids : exprimé en kilogrammes, arrondi à 100 grammes près
- Taille : exprimée en centimètres, arrondie à 0,5 centimètre près

Prénom	Sexe	ZEP	Poids (kg)	Âge (ans)	Mois	Taille (cm)
Erika	F	O	16.0	3	5	100.0
Célia	F	O	14.0	3	10	97.0
Eric	G	O	13.5	3	5	95.5
Eve	F	O	15.4	4	0	101.0
Paul	G	N	16.5	3	8	100.0
Jean	G	O	16.0	4	0	98.5
Adam	G	N	17.0	3	11	103.0
Louis	G	O	14.8	3	9	98.0
Jules	G	O	17.0	4	1	101.5
Léo	G	O	16.7	3	3	100.0

En statistique, il est essentiel de connaître le type des variables étudiées, qu'elles soient qualitatives, ordinales ou quantitatives. R permet de spécifier explicitement ces types grâce aux fonctions de structure abordées dans ce chapitre. Voici quelques manipulations à effectuer avec R. Assurez-vous d'utiliser la stratégie de travail introduite en début de chapitre.

1.1- Choisissez la fonction R appropriée pour enregistrer les données de chacune des variables précédentes dans des vecteurs nommés Individus, Poids, Taille et Sexe.
1.2- Calculez la moyenne des variables pour lesquelles cela est possible.
1.3- Calculez l'IMC des individus et regroupez les valeurs obtenues dans un vecteur nommé IMC, en veillant aux unités.

1.4- Regroupez ces variables dans la structure R qui vous semble la plus adaptée.
1.5- Consultez l'aide en ligne de R afin d'obtenir des informations sur la fonction plot().
1.6- Tracez le nuage de points du Poids en fonction de la Taille. Pensez à ajouter un titre à votre graphique et à annoter les axes.

---

3. Manipulation de données

- Importation et exportation de données (`read.csv()`, `write.csv()`)
- Fonctions de base pour manipuler les tableaux (`head()`, `tail()`, `summary()`)
- Indexation et filtrage des données

4. Introduction aux fonctions et opérations

- Création et utilisation de fonctions (`function()`)
- Opérations mathématiques et logiques
- Structures conditionnelles (`if`, `else`) et boucles (`for`, `while`)