CHAPITRE 2 : Syntaxe de Base

Introduction

La syntaxe de base en Python constitue le fondement de tout programme écrit dans ce langage. Elle regroupe les règles essentielles qui permettent de définir des variables, d'écrire des instructions et de structurer un code de manière claire et compréhensible.

L'un des principaux atouts de Python est sa syntaxe simple et lisible, proche du langage naturel, ce qui facilite son apprentissage pour les débutants. Contrairement à d'autres langages, Python utilise l'indentation pour structurer le code, ce qui améliore sa lisibilité et impose de bonnes pratiques de programmation.

Ainsi, la maîtrise de la syntaxe de base est une étape indispensable pour comprendre et développer des programmes Python efficaces.

I.Variables et types de données

En Python, les variables sont utilisées pour stocker des valeurs de données. Les variables n'ont pas besoin d'une déclaration explicite pour réserver un espace mémoire. La déclaration se fait automatiquement lorsqu'on affecte une valeur à une variable. Les variables peuvent contenir différents types de données, tels que des entiers, des nombres à virgule flottante, des chaînes de caractères, des listes, etc.

I.1. Création et affectation des variables

Vous pouvez créer et affecter une variable en utilisant l'opérateur égal '='. La valeur à droite est affectée à la variable située à gauche.

Exemple : Affectation de valeurs aux variables

```
x = 10

y = 3.14

name = "Ali"
```

I.2. Réaffectation des variables

Vous pouvez modifier la valeur d'une variable en lui attribuant une nouvelle valeur. Python est un langage à typage dynamique, ce qui signifie que vous pouvez affecter un type différent à la même variable.

Exemple : Réaffectation des variables

```
x = 100

print("x =", x)
```

Changement du type de la variable

```
x = "Hello"

print("x =", x)
```

Résultat :

x = 100

x = Hello

I.3. Affectation multiple

Vous pouvez attribuer des valeurs à plusieurs variables sur une seule ligne en utilisant l'affectation multiple.

Exemple : Affectation multiple

```
a, b, c = 5, 10, 15
```

Affichage des valeurs

```
print("a =", a)
print("b =", b)
print("c =", c)
```

Résultat :

```
a = 5
b = 10
c = 15
```

I.4. Variables globales et locales

- Les **variables locales** sont définies à l'intérieur d'une fonction et ne sont accessibles que dans cette fonction.
- Les **variables globales** sont définies en dehors des fonctions et sont accessibles dans tout le programme.

Exemple :

```
#Variable globale
x = "global"
def my_function():
    # Variable locale
    x = "local"
  print("À l'intérieur de la fonction :", x)
```

Appel de la fonction

```
my_function()
```

 # En dehors de la fonction

```
print("À l'extérieur de la fonction :", x)
```

Résultat :

```
À l'intérieur de la fonction : local
À l'extérieur de la fonction : global
```

Exemple avec le mot-clé global :

Vous pouvez utiliser le mot-clé global pour modifier une variable globale à l'intérieur d'une fonction.

Code Python :

```
x = "global"
def my_function():
    global x
    x = "modified global"
    print("À l'intérieur de la fonction :", x)
```

Appel de la fonction

```
my_function()
```

En dehors de la fonction

```
print("À l'extérieur de la fonction :", x)
```

Résultat :

À l'intérieur de la fonction : modified global

À l'extérieur de la fonction : modified global

I.5. Types de variables

5.1 Variables entières (int) : Les entiers sont des nombres sans partie décimale, positifs ou négatifs.

Exemple :

```
a = 10
b = -5
print("a =", a, "b =", b)
```

Résultat :

a = 10 b = -5

5.2 Variables à virgule flottante (float) : Les floats représentent des nombres décimaux.

Exemple :

```
x = 3.14
y = -0.001
print("x =", x, "y =", y)
```

Résultat :

x = 3.14 y = -0.001

5.3 Variables de type chaîne de caractères (str) :

Les chaînes sont des séquences de caractères, entourées par des guillemets simples ' ' ou doubles " "

Exemple :

Code Python

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print("Nom complet :", full_name)
```

Résultat :

Nom complet : John Doe

5.4 Variables de Type booléen (bool)

En Python, une variable booléenne ne peut contenir que deux valeurs :

- True (vrai)
- False (faux)

Exemple

```
is_active = True
is_finished = False
print("is_active =", is_active)            # True
print("is_finished =", is_finished)        # False
```

Résultat de l'exécution :

is_active = True

is_finished = False

I.6. Constantes

Python ne possède pas de type constant intégré, mais par convention, on définit les constantes en utilisant des **lettres majuscules**. Les constantes sont des variables dont les valeurs ne doivent pas être modifiées durant l'exécution du programme.

Exemple :

```
PI = 3.1416
GRAVITY = 9.8
print("PI =", PI)
print("Gravité =", GRAVITY)
```

Résultat :

PI = 3.1416

Gravité = 9.8

I.7. Vérification du type avec type()

Vous pouvez vérifier le type de données d'une variable en utilisant la fonction type().

Exemple :

```
x = 10
y = 3.14
name = "Alice"
print("Type de x :", type(x))
print("Type de y :", type(y))
print("Type de name :", type(name))
```

Résultat :

```
Type de x : <class 'int'>
Type de y : <class 'float'>
Type de name : <class 'str'>
```

I.8. Suppression des variables : Vous pouvez supprimer une variable en utilisant le mot-clé **« del »**.

Exemple :

```
x = 10
print("x =", x)
# Suppression de la variable x
del x
# Essayer d'accéder à la variable supprimée provoquera une erreur
# print(x)  # Décommentez cette ligne pour générer une erreur NameError
```

I.8. Typage dynamique en Python

Python est un langage à typage dynamique, ce qui signifie que vous n'avez pas besoin de déclarer explicitement le type d'une variable. Le type est déterminé en fonction de la valeur qui lui est affectée.

Exemple :

```
x = 10     # x est un entier
print("x =", x)
x = 3.14   # x devient un flottant
print("x =", x)
x = "Hello"  # x devient une chaîne de caractères
print("x =", x)
```

Résultat :

```
x = 10
x = 3.14
x = Hello
```

I.9. Règles de nommage des variables

Il existe certaines règles et conventions à respecter lors du nommage des variables :

- Les noms de variables doivent commencer par une lettre ou un underscore _.
- Le reste du nom peut contenir des lettres, des chiffres ou des underscores.
- Les noms de variables sont sensibles à la casse (**myVariable ≠ myvariable**).

Exemples de noms valides et invalides :

Noms de variables valides

```
age = 25
name2 = "Alice"
my_variable = 10
```

Noms de variables invalides (provoquent des erreurs de syntaxe)

```
2name = "Alice"    # Ne peut pas commencer par un chiffre
my-variable = 10  # Les tirets ne sont pas autorisés
```

I.10 Lecture des variables

En Python, la fonction **input()** permet de lire des données saisies par l'utilisateur au clavier.

Syntaxe générale :

```
variable = input("Message à afficher : ")
```

Exemple :

```
nom = input("Entrez votre nom : ")
```

Explication :

- input() affiche un message
- attend que l'utilisateur saisisse une valeur
- cette valeur est stockée dans une variable

Important:
La fonction input() retourne toujours une chaîne de caractères (str). Ainsi, en Python, toute donnée saisie par l'utilisateur via input() est automatiquement considérée comme une chaîne, même lorsqu'il s'agit d'un nombre. Pour pouvoir effectuer des opérations de calcul, il est donc nécessaire de convertir cette valeur vers un type numérique approprié.

Exemple : age = int(input("Entrez votre âge : "))

I.11. Affichage des variables

En Python, l'affichage des variables se fait à l'aide de la fonction **print().** Cette fonction permet d'afficher à l'écran le contenu d'une ou plusieurs variables.

On peut afficher directement la valeur d'une variable :

- **print(variable)**

Il est aussi possible d'afficher plusieurs variables :

- **print(var1, var2)**

Pour un affichage plus clair, on peut combiner du texte et des variables :

- **print("Valeur :", variable)**

Exemple 1:

```
nom = "Ali"
age = 20
print("Je m'appelle", nom, "et j'ai", age, "ans")
```

Une autre méthode moderne consiste à utiliser les **f-strings** (chaînes formatées), qui permettent d'intégrer facilement les variables dans du texte :

- print(f"Valeur = {variable}")

  f" ... " → indique une f-string

  {variable} → Python remplace automatiquement par la valeur

Exemple 2:

```
nom = "Ali"
age = 20
print(f"Je m'appelle {nom} et j'ai {age} ans")
```

Résultat :

Je m'appelle Ali et j'ai 20 ans

I.12 Les commentaires en Python

Les **commentaires** en Python sont des lignes de texte qui ne sont pas exécutées par le programme. Ils servent uniquement à **expliquer le code** et à le rendre plus compréhensible.

Syntaxe des commentaires

1. Commentaire sur une ligne

On utilise le symbole # :

Exemple :

```
# Ceci est un commentaire
print("Bonjour")
```

2. Commentaire sur plusieurs lignes : On utilise plusieurs # ou des guillemets triples :

Exemple :

```
"""
Ceci est un commentaire
sur plusieurs lignes
"""
```

II. Les Opérations de base :

II.1 Les opérations arithmétiques : Python permet de réaliser des opérations mathématiques de base telles que l'addition, la soustraction, la multiplication, la division, le modulo ou la puissance.

- **Ces opérations sont définies uniquement sur les types numériques (nombres) : les entiers (int) et les nombres décimaux (float).**

Opérateur	Description	Exemple
+	**Addition**	**x = 5 + 3 # x = 8**
-	**Soustraction**	**x = 10 - 4 # x = 6**
*	**Multiplication**	**x = 6 * 2 # x = 12**
/	**Division (flottante)**	**x = 7 / 2 # x = 3.5**
//	**Division entière**	**x = 7 // 2 # x = 3**
%	**Modulo (reste de la division)**	**x = 7 % 2 # x = 1**
**	**Puissance**	**x = 2 ** 3 # x = 8**

Exemple :

```
a = 10
b = 3
print("Addition :", a + b)
print("Soustraction :", a - b)
print("Multiplication :", a * b)
print("Division :", a / b)
print("Division entière :", a // b)
print("Modulo :", a % b)
print("Puissance :", a ** b)
```

Priorité des opérations arithmétiques (PEMDAS)

En Python, comme dans les mathématiques, certaines opérations sont prioritaires sur d'autres. La règle PEMDAS définit l'ordre d'exécution des opérations :

Parenthèses → Exposants → Multiplication/Division → Addition/Soustraction

- Les parenthèses sont toujours évaluées en premier.
- Ensuite viennent les exposants (puissances).
- Puis les multiplications et divisions, de gauche à droite.

- Enfin, les additions et soustractions, également de gauche à droite.

Cette règle permet d'éviter les ambiguïtés dans les calculs et garantit que Python renvoie le résultat correct.

 # Exemple 1 : Sans parenthèses

result1 = 5 + 3 * 2 # 5 + 6 = 11

print("5 + 3 × 2 =", result1)

 # Exemple 2 : Avec parenthèses

result2 = (5 + 3) * 2 # 8 × 2 = 16

print("(5 + 3) × 2 =", result2)

 # Exemple 3 : Calcul complexe

result3 = 10 + 2 ** 3 * 2 - 5 // 2

Étapes : 10 + (8 × 2) - (5 // 2)

10 + 16 - 2 = 24

print("10 + 2³ × 2 - 5 // 2 =", result3)

Utilisation des variables dans les calculs

x = 15

y = 4

sum_result = x + y

difference = x - y

product = x * y

quotient = x / y

print(f"{x} + {y} = {sum_result}")

print(f"{x} - {y} = {difference}")

print(f"{x} × {y} = {product}")

print(f"{x} ÷ {y} = {quotient}")

Mise à jour des variables avec des opérations arithmétiques

score = 100

score = score + 10 # Ajoute 10

score += 5 # Ajoute 5

score *= 2 # Multiplie par 2

print("Score final :", score)

Conversion de types (Type Conversion) : La conversion de types consiste à transformer une donnée d'un type à un autre (par exemple de chaîne de caractères en entier) afin de pouvoir l'utiliser correctement dans les calculs ou les traitements.

```
# Conversion entre int et float
number_int = 10
number_float = float(number_int)    # Convertir en float
print(number_float)                 # 10.0
decimal_num = 7.8
whole_num = int(decimal_num)        # Convertir en int (troncature)
print(whole_num)                    # 7
# Entrée utilisateur (toujours string)
age_input = "25"
age_number = int(age_input)
print(age_number + 5)               # 30
# Arrondir des nombres
pi = 3.14159
rounded_pi = round(pi, 2)           # 2 décimales
print(rounded_pi)                   # 3.14

large_num = 123.4567
rounded_large = round(large_num)    # Arrondi à l'entier le plus proche
print(rounded_large)                # 123
```

Fonctions mathématiques courantes (Common Math Functions)

```
# Fonctions intégrées
numbers = [4, 2, 9, 5, 1]
print("Maximum :", max(numbers))     # 9
print("Minimum :", min(numbers))     # 1
print("Valeur absolue :", abs(-10))  # 10
print("Puissance :", pow(2, 3))      # 8 (équivaut à 2**3)
# Module math pour les opérations avancées
import math
print("Racine carrée :", math.sqrt(25))  # 5.0
print("Plafond :", math.ceil(4.2))       # 5
```

```
print("Plancher :", math.floor(4.9))     # 4
print("Constante Pi :", math.pi)          # 3.141592653589793
```

Valeurs spéciales des nombres (Special Number Values)

```
# Infini
positive_inf = float('inf')
negative_inf = float('-inf')
print("Infini positif :", positive_inf)
print("Infini négatif :", negative_inf)
```

```
# Not a Number (NaN) - résultat d'opérations invalides
import math
result = math.sqrt(-1)  # Racine carrée d'un négatif impossible
print("Résultat NaN :", result)       # nan
# Vérification des valeurs spéciales
print("Est-ce infini ?", math.isinf(positive_inf))  # True
print("Est-ce NaN ?", math.isnan(result))          # True
```

II.2. Les opérations de comparaison

Les opérations de comparaison servent à **comparer deux valeurs**. Le résultat est toujours un **booléen** (True ou False), ce qui permet de savoir si une condition est vraie ou fausse. Ces opérations sont utilisées dans les conditions (if, while) et les boucles.

Opérateur	Signification
==	Égal à
!=	Différent de
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à

Exemple :

```
x = 10
y = 20

print("x == y :", x == y)  # False
```

```
print("x != y :", x != y)  # True
print("x > y :", x > y)    # False
print("x < y :", x < y)    # True
print("x >= 10 :", x >= 10) # True
print("y <= 15 :", y <= 15) # False
```

- Le type booléen est **automatiquement renvoyé** par ces opérations. Par exemple, 10 > 5 renvoie True.

II.3 Opérations logiques :

Les opérations logiques permettent de **combiner ou inverser des expressions booléennes**. Elles s'appliquent **uniquement sur le type booléen (bool)**. Elles sont souvent utilisées avec les comparaisons pour créer des conditions plus complexes.

Opérateurs logiques

Opérateur	Signification
and	ET logique : True si toutes les conditions sont vraies
or	OU logique : True si au moins une condition est vraie
not	Négation : inverse la valeur booléenne

Exemple :

```
x = True
y = False
# ET logique
print("x AND y :", x and y)  # False
# OU logique
print("x OR y  :", x or y)   # True
# Négation
print("NOT x   :", not x)    # False
```

Exemple avec des comparaisons :

```
a = 10
b = 5
print((a > b) and (b < 10))  # True
print((a < b) or (b < 10))   # True
print(not(a == b))           # True
```

Les opérations logiques utilisent les **valeurs booléennes**, que ce soit des **variables booléennes explicites** ou le **résultat d'une comparaison**.

III. Structures de contrôle :

Les instructions Python sont des commandes que l'interpréteur Python exécute. Un programme Python est composé de plusieurs instructions, qui permettent d'effectuer différentes tâches telles que l'affectation de valeurs à des variables, le contrôle du flux du programme, l'exécution de boucles, la gestion des exceptions, etc.

Voyons les différentes instructions Python avec des explications et des exemples.

III.1. Instructions conditionnelles (if, elif, else)

Les instructions conditionnelles permettent de contrôler le déroulement du programme en fonction de conditions. L'instruction if évalue une condition, et si celle-ci est vraie, le bloc de code correspondant est exécuté.

On peut enchaîner plusieurs conditions avec elif, et prévoir un cas par défaut avec else.

Exemple :

```
age = 18
if age >= 18:
    print("Vous êtes adulte.")
elif age >= 13:
    print("Vous êtes adolescent.")
else:
    print("Vous êtes enfant.")
```

Résultat :

Vous êtes adulte.

Explication :

- Le programme vérifie la valeur de age.
- Comme age = 18, la condition age >= 18 est vraie.
- Donc, le programme affiche "Vous êtes adulte."

II.2 Instructions de boucle

1. Boucle for

La **boucle for en Python** est utilisée pour **parcourir une séquence** (comme une liste, une chaîne de caractères, un tuple, ou une plage de valeurs.) et exécuter un bloc de code pour chaque élément.

La boucle for est très utilisée en Python pour :

- parcourir des données
- répéter des actions
- traiter des listes ou des chaînes facilement

🔹 **Syntaxe générale**

for variable **in** sequence:

    # instructions

🔹 **Exemple 1 : Parcourir une liste**

```
nombres = [1, 2, 3, 4, 5]
for n in nombres:
    print(n)
```

Résultat :

1

2

3

4

5

Explication :

- n prend chaque valeur de la liste nombres une par une
- Le print(n) s'exécute à chaque itération

🔹 **Exemple 2 : Utilisation avec range()**

La fonction range() en Python

La fonction range() est utilisée pour générer une suite de nombres entiers. Elle est très souvent utilisée avec la boucle for pour répéter une action un certain nombre de fois.

🔸 **Syntaxe générale de range()**

    range(debut, fin, pas)

🔸 **Les différentes formes**

a) range(fin)

range(5)

Résulat : Génère : 0, 1, 2, 3, 4

Commence à 0 par défaut et s'arrête avant fin

b) range(debut, fin)

range(2, 6)

Résultat : Génère : 2, 3, 4, 5
Commence à debut et s'arrête avant fin

c) range(debut, fin, pas)

range(0, 10, 2)

Génère : 0, 2, 4, 6, 8
Avance avec un pas de 2

🔸 **remarques importantes pour range()**

- La valeur de fin n'est jamais incluse
- Le pas (step) est optionnel (par défaut = 1)

- On peut utiliser un pas négatif :

Exemple : range(5, 0, -1)

Génère : 5, 4, 3, 2, 1

🔸 **Utilisation typique avec for**

```
for i in range(5):
    print(i)
```

Résultat :

0

1

2

3

4

➢ range(5) génère les nombres de 0 à 4

🔹 **Exemple 3 : Parcourir une chaîne de caractères**

```
mot = "Python"

for lettre in mot:
    print(lettre)
```

Résultat :

P

y

t

h

o

n

- **Exemple 4 : Boucle avec condition**

```
for i in range(10):
    if i % 2 == 0:
        print(i)
```

  - Affiche uniquement les nombres pairs

- **Exemple 5 :**

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Résultat :

apple

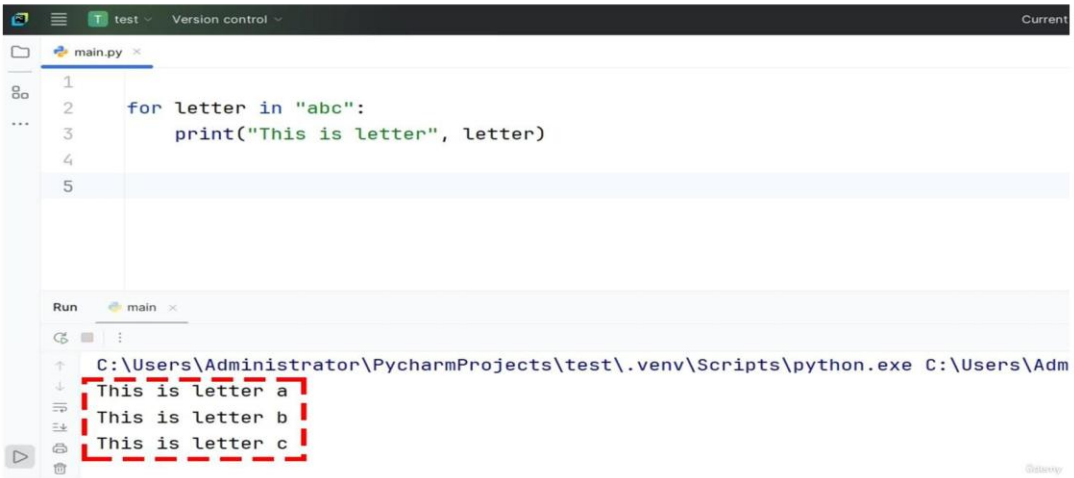
banana

cherry

Explication :

- La boucle for parcourt la liste fruits.
- À chaque itération, un élément de la liste est affecté à la variable fruit.
- Chaque élément est ensuite affiché sur une nouvelle ligne.

- **Exemple 6**



2. Boucle while

La boucle while en Python permet de **répéter un bloc d'instructions tant qu'une condition est vraie**.

La boucle while est utile lorsque :

- le nombre d'itérations n'est pas connu à l'avance
- on veut répéter une action jusqu'à ce qu'une condition change

🔸 Syntaxe générale

while condition:

    # instructions

- ➢ Tant que la **condition est vraie**, la boucle continue.

🔹 Différence avec for

- for → utilisé quand on connaît le nombre de répétitions
- while → utilisé quand la condition dépend d'une situation

Exemple 1:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Résultat :

1

2

3

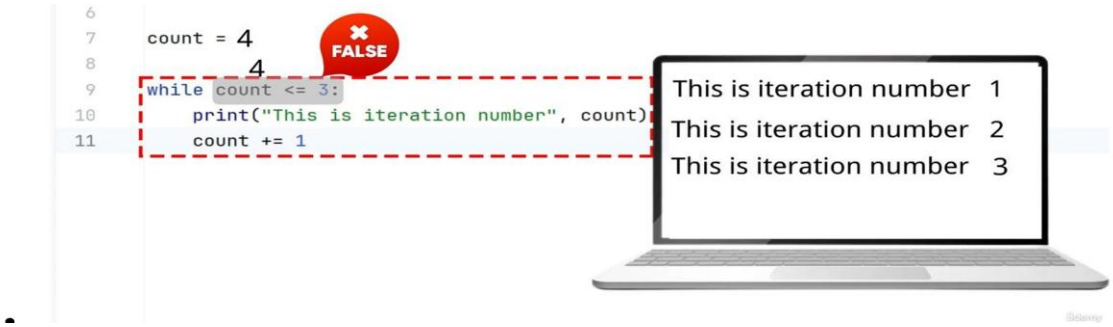
4

5

Explication :

- La boucle while affiche la valeur de i tant que i est inférieur ou égal à 5.
- Après chaque itération, i est incrémenté de 1 grâce à l'instruction i += 1.
- La boucle s'arrête lorsque la condition devient fausse.

Exemple 2 :



Exemple 3 — Parcourir une chaîne de caractères

word = "Python"

for letter in word:

print(letter)

Résultat :

P

y

t

h

Exemple — Boucles imbriquées (حلقة داخل حلقة)

for i in range(2): # boucle externe

for j in range(3): # boucle interne

print(i, j)

Sortie (résultat de l'exécution) :

0 0

0 1

0 2

1 0

1 1

1 2

3. Instructions break et continue

3.1. Instruction break

L'instruction **break** est utilisée pour quitter une boucle de manière anticipée lorsqu'une condition spécifique est satisfaite.

Exemple :

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

Sortie :

1

2

3

4

Explication :
• La boucle s'arrête lorsque **i** devient égal à 5 à cause de l'instruction **break**, donc les nombres de 1 à 4 sont affichés.

3.2. Instruction continue

L'instruction **continue** permet de sauter l'itération courante de la boucle et de passer directement à la suivante.

Exemple :

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Explication :
• Lorsque **i** est égal à 3, l'instruction **continue** est exécutée, ce qui saute l'instruction **print** pour cette itération. Les nombres 1, 2, 4 et 5 sont affichés.

Conclusion
Les instructions Python constituent les éléments fondamentaux d'un programme Python. Elles incluent notamment l'affectation, le contrôle du flux, les boucles, la définition de fonctions et la gestion des exceptions. Chaque type d'instruction remplit un rôle spécifique et permet de contrôler le déroulement et le comportement de votre programme.