FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Université Hassan II - Casablanca

Faculté des sciences - Ain Chock

Algorithmique & Programmation (Python)

Tronc Commun Physique Chimie (PC)
Filière Chimie - Semestre 3

Version 1.0

Pr. Y. OUASSIT

Date : 6 octobre 2025

Table des matières

I Algorithmique 1 **4**

1 Introduction à l'algorithmique **5**
1.1 Introduction . . . 5
1.1.1 Notion d'algorithme . . . 5
1.2 L'algorithmique . . . 6
1.2.1 La démarche de programmation et analyse descendante . . . 6
1.2.2 Étape d'analyse . . . 8
1.2.3 Pseudo langage . . . 9
1.3 Variables . . . 10
1.3.1 Déclaration des variables . . . 10
1.3.2 Noms de variables . . . 10
1.3.3 Types de variables . . . 11
1.4 Les instructions de base (Les primitives) . . . 12
1.4.1 L'affectation . . . 12
1.4.2 Les instructions de lecture / écriture . . . 13
1.4.2.1 L'instruction de lecture . . . 13
1.4.2.2 L'instruction de sortie/écriture . . . 15

2 Les structures de contrôle **18**
2.1 Introduction . . . 18
2.2 Les structures de contrôle conditionnelles . . . 18
2.2.1 La structure conditionnelle alternative . . . 18
2.2.1.1 Syntaxe . . . 18
2.2.1.2 Fonctionnement . . . 19
2.2.1.3 Organigramme . . . 19
2.2.1.4 Exemple . . . 19
2.2.2 La structure conditionnelle simple . . . 19
2.2.2.1 Syntaxe . . . 20
2.2.2.2 Fonctionnement . . . 20
2.2.2.3 Organigramme . . . 20
2.2.2.4 Exemple . . . 20
2.2.3 La structure conditionnelle imbriquée . . . 20
2.2.3.1 Exemple . . . 21
2.2.3.2 Structure conditionnelle multiple . . . 22
2.2.3.3 Syntaxe . . . 23
2.2.3.4 Fonctionnement . . . 23
2.3 Les structures répétitives . . . 23
2.3.1 La boucle Pour . . . 24

2.3.1.1 Syntaxe 24
2.3.1.2 Fonctionnement 24
2.3.1.3 Exemples 25
2.3.2 La boucle TantQue 25
2.3.2.1 Syntaxe 25
2.3.2.2 Organigramme 26
2.3.2.3 Fonctionnement 26
2.3.2.4 Exemples 26

II Programmation 1 **28**

3 Introduction à Python **29**
3.1 Introduction 29
3.1.1 Histoire et philosophie de Python 29
3.1.1.1 Origines de Python 29
3.1.1.2 Philosophie de Python 29
3.1.1.3 Évolution de Python 29
3.1.1.4 Caractéristiques, usage et popularité 30
3.1.2 Installation de Python et environnement de travail (IDE, Pyzo) . 31
3.1.2.1 Téléchargement 31
3.1.2.2 Installation 31
3.1.2.3 Choisir et Installer un IDE 31
3.2 Les Bases du Langage 31
3.2.1 Les deux modes d'exécution d'un code Python 31
3.2.2 Identificateurs 32
3.2.3 Style de nommage 32
3.2.4 Les mots réservés de Python 3 32
3.2.5 Notion d'expression 32
3.3 Les types de données primitives 33
3.3.1 Le type int (entiers) : 33
3.3.2 Le type float (réels) : 33
3.3.3 Le type complex (nombres complexes) : 33
3.3.4 Le type bool (booléens) : 34
3.3.5 Entrées et sorties (input/print) 34
3.3.5.1 Instruction de lecture 34
3.3.5.2 Instruction d'écriture 34
3.4 Structures de Contrôle 35
3.4.1 Structures conditionnelles (if, else, elif) 35
3.4.1.1 La structure conditionnelle alternative 35
3.4.1.2 La structure conditionnelle simple 36
3.4.1.3 La structure conditionnelle multiple 36
3.4.2 Les boucles 37
3.4.2.1 La boucle : for 37
3.4.2.2 Le boucle : while 38
3.4.3 Instructions de contrôle de boucle (break) 38
3.5 Fonctions 38
3.5.1 Définition et utilisation des fonctions (def) 38

3.5.1.1 Déclaration d'une fonction 38
3.5.1.2 Utilisation des fonctions 39
3.5.2 Paramètres, arguments et valeurs de retour 39
3.5.3 Portée des variables (globale vs locale) 40
3.5.4 Fonctions lambda 40
3.6 Structures de Données 41
3.6.1 Les listes 41
3.6.1.1 Création d'une liste 41
3.6.1.2 Accès aux éléments d'une liste 41
3.6.1.3 Modification des éléments 42
3.6.1.4 Opérations sur les listes 42
3.6.1.5 Ajout d'éléments 42
3.6.1.6 Insertion d'éléments 42
3.6.1.7 Suppression d'éléments 42
3.6.1.8 Méthodes de liste courantes 43
3.6.1.9 Slicing (slicing des liste) 43
3.6.2 Les tuples 44
3.6.2.1 Création d'un tuple 44
3.6.2.2 Accès aux éléments d'un tuple 45
3.6.2.3 Immuabilité des tuples 45
3.6.2.4 Opérations sur les tuples 45
3.6.2.5 Méthodes de tuple courantes 46
3.6.2.6 Parcours d'un tuple 46
3.6.2.7 Déballage (Unpacking) des tuples 46
3.6.2.8 Slicing des tuples 46
3.6.2.9 Quand utiliser un tuple plutôt qu'une liste ? 47
3.6.3 Les ensembles (sets) 47
3.6.3.1 Création d'un set 47
3.6.3.2 Accès aux éléments 48
3.6.3.3 Ajout et suppression d'éléments 48
3.6.3.4 Opérations sur les sets 48
3.6.3.5 Méthodes courantes 49
3.6.3.6 Conversion entre types 49
3.6.3.7 Quand utiliser un set ? 49
3.6.4 Manipulation des chaînes de caractères 50
3.6.4.1 Déclaration de chaînes 50
3.6.4.2 Accès aux caractères 50
3.6.4.3 Accès par index 50
3.6.4.4 Slicing (slicing des chaînes) 50
3.6.4.5 Opérations sur les chaînes 51
3.6.4.6 Méthodes de chaînes courantes 51
3.6.4.7 Immuabilité des chaînes 52

Première partie

Algorithmique 1

Chapitre 1

Introduction à l'algorithmique

1.1 Introduction

L'informatique, en tant que discipline moderne, repose fondamentalement sur la capacité à résoudre des problèmes à l'aide de méthodes systématiques et répétables. Ces méthodes sont souvent représentées sous la forme d'algorithmes. La notion d'algorithme et l'étude qui en découle, appelée algorithmique, occupent une place centrale dans l'évolution des sciences informatiques, influençant autant le développement de logiciels que la résolution de problèmes complexes dans divers domaines tels que les mathématiques, la physique, l'économie et la biologie. Dans cette première partie du cours, nous allons définir ces concepts essentiels, examiner leur importance, et poser les bases pour comprendre les principes de conception et d'analyse des algorithmes.

1.1.1 Notion d'algorithme

Un algorithme peut être défini comme une suite finie et ordonnée d'instructions ou d'étapes permettant de résoudre un problème ou de réaliser une tâche donnée. Il peut être représenté sous forme de texte, de diagrammes (comme les organigrammes) ou dans des langages de programmation. Le concept d'algorithme trouve son origine dans les travaux du mathématicien perse Al-Khwarizmi au IXe siècle, et est aujourd'hui au cœur de l'informatique moderne.

Plus précisément, un algorithme est une séquence d'instructions qui prend un ou plusieurs entrées (données d'entrée), effectue un ensemble d'opérations définies, et produit une ou plusieurs sorties (résultats).

Un algorithme est une notion attaché à notre vie quotidienne, nous utilisons chaque jour des algorithmes :

1. Une recette de cuisine est un algorithme !
2. Le mode d'emploi d'un magnétoscope est aussi un algorithme !
3. Indiqué un chemin à un touriste égaré ou faire chercher un objet à quelqu'un par téléphone c'est fabriquer - et faire exécuter - des algorithmes.

Une recette de cuisine, par exemple, est un algorithme : à partir des ingrédients, elle explique comment parvenir au plat. De même, un itinéraire routier explique comment, à partir d'une position initiale, rejoindre une position finale en un certain nombre d'étapes.

Un algorithme doit respecter certaines propriétés fondamentales :

1. **Finitude :** Un algorithme doit avoir un nombre fini d'étapes, et il doit se terminer après un temps raisonnable.
2. **Précision :** Chaque étape de l'algorithme doit être définie avec précision et ne doit pas prêter à confusion.
3. **Entrées et sorties :** Un algorithme doit accepter des entrées bien définies et produire des résultats.
4. **Efficacité :** Un bon algorithme doit être optimal, c'est-à-dire qu'il doit résoudre le problème avec un minimum de ressources (temps, espace mémoire, etc.).
5. **Déterminisme :** À partir des mêmes entrées, un algorithme doit toujours produire les mêmes sorties.

Les ordinateurs eux-mêmes ne sont fondamentalement capables d'exécuter que quatre opérations logiques :

1. l'affectation de variables
2. la lecture / écriture
3. les tests
4. les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques-unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes.

1.2 L'algorithmique

L'algorithmique est la branche de l'informatique qui étudie les algorithmes, leurs conceptions, leurs caractéristiques, et leurs performances. C'est une discipline qui combine des aspects théoriques et pratiques et s'intéresse aussi bien à la manière de concevoir un algorithme qu'à la manière d'analyser son efficacité.

L'algorithmique aborde plusieurs aspects essentiels, dont la conception et l'analyse des algorithmes. La conception d'algorithmes se concentre sur la manière de les construire pour résoudre un problème spécifique tout en suivant des principes qui garantissent leur bon fonctionnement et leur efficacité. L'analyse des algorithmes, quant à elle, vise à évaluer leur performance en termes de temps de calcul et d'espace mémoire utilisés, en examinant à la fois leur complexité temporelle, c'est-à-dire le nombre d'opérations nécessaires, et leur complexité spatiale, c'est-à-dire la quantité de mémoire requise.

1.2.1 La démarche de programmation et analyse descendante

La résolution d'un problème passe par toute une suite d'étapes :

— Phase d'analyse et de réflexion (algorithmique)

— Phase de programmation
  - choisir un langage de programmation
  - traduction de l'algorithme en programme

- programme (ou code) source
- compilation : traduction du code source en code objet
- traduction du code objet en code machine exécutable, compréhensible par l'ordinateur

— Phase de test

— Phase d'exécution

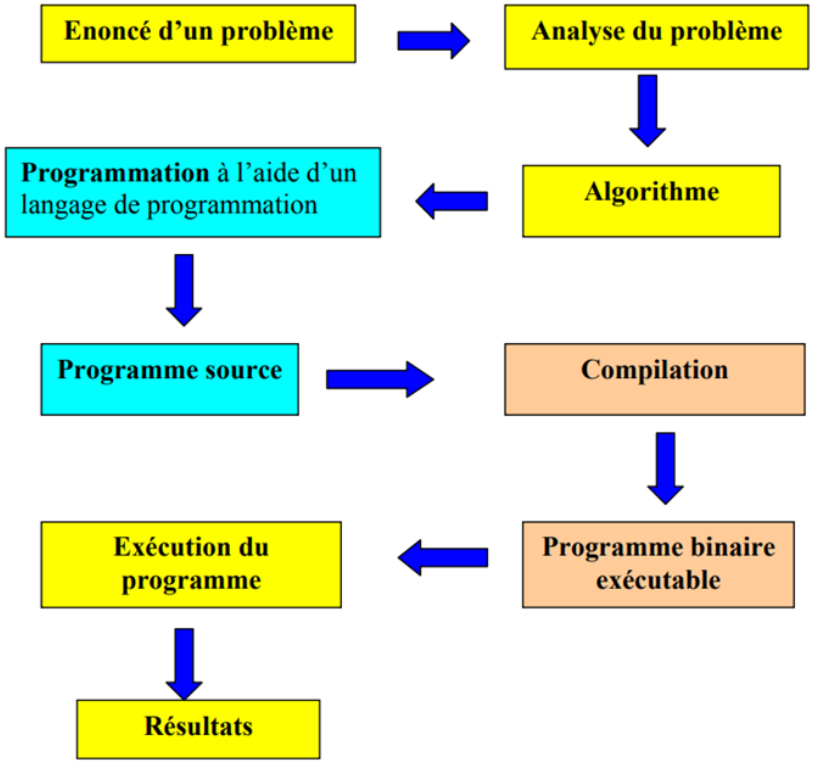

FIGURE 1.1 – Démarche de programmation et analyse descendante

Le travail est ici surtout basé sur l'analyse du problème et l'écriture de l'algorithme. La réalisation d'un programme passe par l'analyse descendante du problème : il faut réussir à trouver les actions élémentaires qui, en partant d'un environnement initial, nous conduisent à l'état final.

L'analyse descendante consiste à décomposer le problème donné en sous-problèmes, et ainsi de suite, jusqu'à descendre au niveau des primitives. Les étapes successives donnent lieu à des sous- algorithmes qui peuvent être considérés comme les primitives de machine intermédiaires.

Le travail de l'analyse est terminé lorsqu'on a obtenu un algorithme ne comportant que :

— Des primitives de la machine initiale,

— Des algorithmes déjà connus.

L'ordre des instructions est essentiel : la machine ne peut exécuter qu'une action à la fois et dans l'ordre donné ; c'est la propriété de séquentialité. Une fois ces actions déterminées, il suffit de les traduire dans le langage de programmation.

1.2.2 Étape d'analyse

L'étape d'analyse dans la démarche de conception des algorithmes consiste à bien comprendre le problème à résoudre avant de concevoir une solution. C'est une phase cruciale qui permet de s'assurer que l'algorithme sera efficace et adapté au problème.

Voici les principaux points de cette étape :

1. Comprendre le problème : Identifier les données d'entrée, les résultats attendus et les contraintes éventuelles. Il s'agit de bien définir ce que l'on cherche à résoudre.
2. Formuler le problème : Le problème doit être décrit de manière claire et précise, souvent sous forme d'énoncé ou de modélisation mathématique.
3. Choisir une méthode de résolution : Décider de l'approche à utiliser (itérative, récursive, diviser pour régner, etc.) pour répondre aux exigences du problème.
4. Vérifier la faisabilité : S'assurer que le problème peut être résolu avec les moyens disponibles (temps, ressources, matériel).

L'analyse permet de poser une base solide avant de passer à la conception proprement dite de l'algorithme. Elle garantit que toutes les spécificités du problème sont prises en compte pour éviter des erreurs ou des solutions inefficaces.

Exemple 1 :

1. **Énoncé du problème :**
   Nous devons calculer le montant total TTC (Toutes Taxes Comprises) d'un lot d'ordinateurs (PC) à partir du prix unitaire HT (Hors Taxes) de chaque PC, de la quantité achetée, et du taux de TVA applicable.
2. **Analyse du problème :**

   Données d'entrée : Le prix unitaire HT d'un PC. La quantité de PC achetée. Le taux de TVA (en pourcentage).

   Résultat attendu (sortie) : Le montant TTC total du lot de PC.

   Formulation du problème (Traitement) : Le montant total TTC est calculé à partir de la formule suivante :
   Montant TTC = (Prix unitaire HT × Quantité) × (1+TVA/100)

Cet exemple montre comment l'analyse permet de bien structurer le problème avant de passer à la conception de l'algorithme. Afin de simplifier l'expression des opérations mathématiques, nous allons associer à chaque donnée un identificateur. Il s'agit d'un nom symbolique qui représentera les données. Nous allons aussi spécifier le type de chaque données (domaine de valeurs) et la catégorie.

- Les données d'entré :

Données	Identificateur	Type	Catégorie
Le prix unitaire hors taxe	PUHT	Réel	Variable
Le nombre de PC	N	Entier	Variable
Le taux de TVA	TVA	Réel	Constante

- Les données de sortie :

Données	Identificateur	Type	Catégorie
Montant TTC	MTTC	Réel	Variable

- Traitement :

MHT = PUHT x N

MTVA = MHT x TVA

MTTC = MHT+ MTVA

1.2.3 Pseudo langage

Une fois l'étape d'analyse est effectuée, nous allons passer à la conception de notre algorithme. Pour cet étape nous allons adopter une structure pour l'écriture d'un algorithme, cette structure ne représente pas un standard mais son objectif c'est d'avoir une meilleure organisation des composants de l'algorithme. La structure générale d'un algorithme se compose de trois parties :

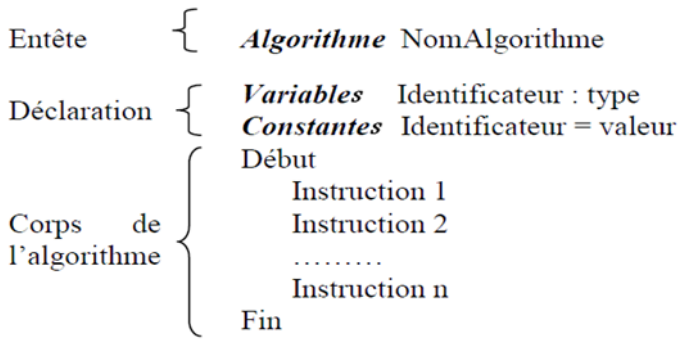

FIGURE 1.2 – Structure générale d'un algorithme

La partie entête :
Cette partie commence par le mot Algorithme et le nom de l'algorithme, ce dernier doit donner une idée sur l'objet de l'algorithme. Exemple : Pour un algorithme qui fait le calcul de la moyen des notes d'une classe, on peut lui donner comme nom (Moyen ; MoyenNote ; Moyen_Note ; ...).

La partie déclaration :
Dans cette partie on doit déclarer l'ensemble des données dont on aura besoin dans l'algorithme

La partie Corps :
Contient l'ensemble des instructions (ordres) à exécuter pour pouvoir résoudre le problème.

Nous allons continuer la conception de l'**Exemple 1** ci-dessous, et l'écrire en pseudo code :

Algorithme 1 Montant TTC

Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : TVA = 0.2
Début
$MHT = PUHT \times N$
$MTVA = MHT \times TVA$
$MTTC = MHT + MTVA$
Fin

L'algorithme ci-dessus 1 est une première version, que nous allons améliorer et compléter dans les sections suivantes.

1.3 Variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier). Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, etc. Dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une variable.

1.3.1 Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la déclaration des variables. Une variable ne peut être utilisée que s'elle est déclarée. La déclaration se fait par la donnée du nom de la variable et du type de la variable.

Syntaxe :

NomVariable1, NomVariable2 : TypeVariable

1.3.2 Noms de variables

Le nom de la variable (l'étiquette de la boîte) obéit à des règles qui changent selon le langage utiliser. Les principales règles à respecter sont :

- Le nom de variable peut comporter des lettres et des chiffres,
- On exclut la plupart des signes de ponctuation, en particulier les espaces.
- Un nom de variable doit commencer par une lettre.
- Le nombre maximal de caractères qui composent le nom d'une variable dépend du langage utilisé.

— Ne pas utiliser les mots clés du langage de programmation.

1.3.3 Types de variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires. Pour pouvoir accéder aux données, le programme (quel que soit le langage dans lequel il est écrit) fait usage d'un grand nombre de variables de différents types. Une VARIABLE possède :

— Un nom : Une variable apparaît en programmation sous un nom de variable.

— Un type : Pour distinguer les uns des autres les divers contenus possibles, on utilise différents types de variables (entiers, réels, chaîne de caractères ...).

— Une valeur : Une fois la variable définie, une valeur lui est assignée, ou affectée.

— Une adresse : C'est l'emplacement dans la mémoire de l'ordinateur, où est stockée la valeur de la variable.

Les Types simples (entier, flottant, caractère) :

— Type Entier : pour manipuler les nombres entiers positifs ou négatifs. Par exemple : 5, -15, etc.

— Type Réel : pour manipuler les nombres à virgule. Par exemple : 3.14, -15.5, etc.

— Type Chaîne : pour manipuler des chaînes de caractères permettant de représenter des mots ou des phrases. Par exemple : "bonjour, Monsieur"

— Type Booléen : pour les expressions logiques. Il n'y a que deux valeurs booléennes : vrai et faux.

A un type donné, correspond un ensemble d'opérations définies pour ce type :

Type	Opérations possibles	Symbole ou mot clé correspondant
Entier	Addition	+
	Soustraction	-
	Multiplication	*
	Division	/
	Division Entière	DIV
	Modulo	MOD
	Exposant (puissance)	^
	Comparaisons	<, =, >, <=, >=, !=
Réel	Addition	+
	Soustraction	-
	Multiplication	*
	Division	/
	Exposant	^
	Comparaisons	<, =, >, <=, >=, !=
Chaîne	Concaténation	<, =, >, <=, >=, !=, +
Booléen	Comparaison	<, =, >, <=, >=, !=
	Logiques	ET, OU, NON

FIGURE 1.3 – Opérations définies pour chaque type

1.4 Les instructions de base (Les primitives)

1.4.1 L'affectation

L'affectation est l'action élémentaire dont l'effet est de donner une valeur à une variable (ranger une valeur à une place). L'affectation est réalisée au moyen de l'opérateur ←. Elle signifie " prendre la valeur se trouvant du côté droit (souvent appelée rvalue) et la copier du côté gauche (souvent appelée lvalue) ". Une rvalue représente toute constante, variable ou expression capable de produire une valeur, mais une lvalue doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat).

Exemple : X ← 3
Signifie mettre la valeur 3 dans la case identifiée par X. A l'exécution de cette instruction, la valeur 3 est rangée en X (nom de la variable).
La valeur correspond au contenu : 3
La variable correspond au contenant : X

On peut représenter la variable X par une boite ou case, et quand elle prend la valeur 3, la valeur est dans la case X :

FIGURE 1.4 – Affectation

On remarque qu'une variable ne peut contenir à un instant donné qu'une seule valeur. La dernière instruction (Y ← X) signifie : copier dans Y la valeur actuelle de X.

Exercice 1: Un petit exercice instructif

Quelles sont les valeurs successives prises par les variables X et Y suit aux instructions suivantes : X ← 1 ; Y ← -4 ; X ← X+3 ; X ← Y-5 ; Y ← X+2 ; Y ← Y-6 ;

Solution :

X	1	1	4	-9	-9	-9
Y		-4	-4	-4	-7	-13

Exercice 2

Soit 2 variables quelconques (nombres ou caractères) x et y ayant respectivement comme valeur a et b ; quelles sont les affectations qui donneront à x la valeur b et à y la valeur a ?

Solution :
Analyse : la première idée est d'écrire : x ← y ; y ← x. Mais ça ne marche pas, les deux variables se retrouvent avec la même valeur b ! Il faut mettre la valeur de x de coté pour ne pas la perdre : on utilise une variable auxiliaire z et on écrit les instructions suivantes :
z ← x ;
x ← y ;
y ← z ;

Revenons à l'algorithme de l'**Exemple 1**. Nous allons le corriger, car dans le code, nous avons utilisé le symbole "=" dans les instructions telles que MHT = PUHT x N. Cette expression était censée demander à la machine de calculer PUHT x N et de stocker le résultat dans MHT. Cependant, comme nous l'avons vu, le symbole "=" est utilisé pour tester l'égalité entre deux expressions. Ainsi, nous devons le remplacer par le symbole d'affectation approprié. Voici la version corrigée de l'algorithme :

Algorithme 2 Montant TTC
Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : $TVA \leftarrow 0.2$
Début
$MHT \leftarrow PUHT \times N$
$MTVA \leftarrow MHT \times TVA$
$MTTC \leftarrow MHT + MTVA$
Fin

1.4.2 Les instructions de lecture / écriture

Les instructions de lecture et d'écriture sont des éléments fondamentaux en algorithmique, car elles permettent de créer une interaction entre la machine et l'utilisateur, ainsi que de manipuler et gérer les données. Comprendre ces instructions est crucial pour développer des algorithmes fonctionnels et efficaces.

Pour illustrer l'importance de ces instructions nous allons essayer de critiquer et améliorer l'algorithme suivant :

Algorithme 3 Carré
Variables : X,Y : Réel
Début
$X \leftarrow 12$
$Y \leftarrow X \hat{}\, 2$
Fin

1.4.2.1 L'instruction de lecture

L'algorithme présenté ci-dessus 3) calcule actuellement le carré du nombre 12. Bien que l'algorithme soit syntaxiquement correct, il manque d'interactivité. En effet, l'algorithme est figé pour un nombre spécifique (12) et, par conséquent, chaque exécution ne produit

que le carré de ce nombre. Pour calculer le carré d'un autre nombre, comme 20 par exemple, il est nécessaire de modifier l'algorithme en affectant la nouvelle valeur à la variable X et en relançant l'exécution.

Pour rendre cet algorithme plus flexible et interactif, nous allons ajouter une instruction permettant de récupérer la valeur de la variable X directement auprès de l'utilisateur à chaque exécution de l'algorithme. Ainsi, l'algorithme pourra être utilisé pour calculer le carré de n'importe quelle valeur sans nécessiter de modifications préalables. Pour ce faire, nous intégrerons une instruction de saisie utilisateur.

En algorithmique, l'instruction d'entrée est utilisée pour obtenir des données ou des valeurs de l'utilisateur ou d'une autre source externe. Cette instruction est cruciale pour rendre un algorithme interactif et dynamique.

L'instruction d'entrée ou de lecture permet à l'utilisateur de saisir des données au clavier pour qu'elles soient utilisées par l'algorithme. Cette instruction assigne (affecte) une valeur entrée au clavier dans une variable.

Syntaxe : *Lire(identificateur)*

Exemple : *Lire(A)*

Lire (A) : Cette instruction permet à l'utilisateur de saisir une valeur au clavier qui sera affectée à la variable A.

Remarque:. *Lorsque le programme rencontre cette instruction, l'exécution s'interrompt et attend que l'utilisateur tape une valeur. Cette valeur est rangée en mémoire dans la variable désignée.*

Voici la nouvelle amélioration de l'algorithme du calcule du carré :

Algorithme 4 Carré V2

Variables : X,Y : Réel
Début
Lire(X)
$Y \leftarrow X \frown 2$
Fin

Remarque:. *Généralement il faut prévoir à demander à l'utilisateur de saisir toutes les données d'entré du problème.*

Un exemple d'algorithme qui calcule la somme de deux nombres :

Algorithme 5 Somme

Variables : A,B,S : Réel
Début
Lire(A)
Lire(B)
$S \leftarrow A + B$
Fin

Dans l'algorithme ci-dessus 5 nous avons deux données d'entré (les variables A et B), nous avons donc fait appel à l'instruction de lecture Lire(A) et Lire(B).

Voici une nouvelle amélioration de l'algorithme de l'**Exemple 1**, où nous allons récupérer les valeurs des données d'entré auprès de l'utilisateur :

Algorithme 6 Montant TTC

Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : $TVA \leftarrow 0.2$
Début
Lire(PUHT)
Lire(N)
$MHT \leftarrow PUHT \times N$
$MTVA \leftarrow MHT \times TVA$
$MTTC \leftarrow MHT + MTVA$
Fin

1.4.2.2 L'instruction de sortie/écriture

instructions de sortie sont utilisées pour afficher des informations à l'utilisateur via le périphérique standard de sortie (L'écran). Elles permettent de communiquer des résultats, des messages ou des informations sur l'état de l'algorithme.

L'algorithme peut utiliser des instructions d'écriture pour afficher des résultats de calculs, des messages d'information, ou des erreurs. Par exemple, après avoir calculé la carré d'un nombre, l'algorithme peut afficher le résultat à l'utilisateur.

Syntaxe : *Ecrire(expressions)*

L'expression peut être :

— Un identificateur : Ecrire(A) $\Longrightarrow$ Affichera la valeur de A
— Un message : Ecrire("Bonjour") $\Longrightarrow$ Affichera le message Bonjour
— Une opération : Ecrire(A+B) $\Longrightarrow$ Affichera le résultat du calcul de A+B

Nous allons maintenant améliorer l'algorithme 4 afin qu'il puisse affiché à l'utilisateur le résultat du carré calculé :

Algorithme 7 Carré V3

Variables : X,Y : Réel
Début
Lire(X)
$Y \leftarrow X \hat{}\, 2$
Ecrire(X)
Fin

L'algorithme maintenant affichera à l'utilisateur le carré de la valeur saisit de X.

Avant de lire une variable, il est très fortement conseillé d'écrire des libellés à l'écran avec l'instruction "Ecrire" , afin de prévenir l'utilisateur de ce qu'il doit frapper :

Voici une autre amélioration du programme :

Algorithme 8 Carré V4

Variables : X,Y : Réel
Début
  Ecrire("Donner un nombre : ")
  Lire(X)
  $Y \leftarrow X \frown 2$
  Ecrire(X)
Fin

Il est aussi fortement conseillé d'écrire des libellés à l'écran avec l'instruction "Ecrire" avant d'afficher les résultat pour indiquer de quel information s'agit-t-il :

Encore une autre amélioration du programme :

Algorithme 9 Carré V5

Variables : X,Y : Réel
Début
  Ecrire("Donner un nombre : ")
  Lire(X)
  $Y \leftarrow X \frown 2$
  Ecrire("Le carré est : ")
  Ecrire(X)
Fin

Remarque:. *Lorsque plusieurs instructions de sortie successives sont présentes, il est possible de les regrouper en une seule instruction de sortie. Cela permet de simplifier le code et de rendre les messages affichés plus cohérents et organisés. En regroupant les instructions de sortie, on évite la répétition et on améliore la lisibilité du programme.*

Encore une autre amélioration du programme :

Algorithme 10 Carré V6

Variables : X,Y : Réel
Début
  Ecrire("Donner un nombre : ")
  Lire(X)
  $Y \leftarrow X \frown 2$
  Ecrire("Le carré est : ", X)
Fin

Retourne à l'algorithme de l'**Exemple 1**, que nous allons le compléter avec les instructions de sortie :

Algorithme 11 Montant TTC

Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier

Constantes : $TVA \leftarrow 0.2$

Début

Ecrire("Donner le prix HT d'un PC :")
Lire(PUHT)
Ecrire("Donner le nombre de PC : ")
Lire(N)
$MHT \leftarrow PUHT \times N$
$MTVA \leftarrow MHT \times TVA$
$MTTC \leftarrow MHT + MTVA$
Ecrire("Le montant TTC est : ", MTTC)

Fin

Chapitre 2

Les structures de contrôle

2.1 Introduction

Les structures de contrôle sont des constructions syntaxiques qui définissent l'ordre d'exécution des instructions dans un programme. Elles permettent de modifier le chemin d'exécution en fonction de conditions spécifiques ou de la répétition d'opérations. Les structures de contrôle jouent un rôle crucial dans la conception des algorithmes, car elles permettent de créer des programmes capables de réagir à des situations diverses et d'effectuer des tâches complexes.

Les Structures de contrôle permettent de contrôler l'ordre d'exécution des instructions (le séquencement) d'un algorithme. Il existe deux types de structures de contrôle fondamentales :

- Les Structures Conditionnelles : Permettent d'exécuter une suite d'instructions (bloc) si une condition est vérifiée (vrai).
- Les Structures Itératives (Boucle) : Permettent de répéter l'exécution d'une suite d'instructions (bloc) un certain nombre de fois.

2.2 Les structures de contrôle conditionnelles

Souvent les problèmes nécessitent l'étude de plusieurs situations qui ne peuvent pas être traitées par les séquences d'actions simples. Puisqu'on a plusieurs situations, et qu'avant l'exécution, on ne sait pas à quel cas de figure on aura à exécuter, dans l'algorithme on doit prévoir tous les cas possibles.

2.2.1 La structure conditionnelle alternative

2.2.1.1 Syntaxe

Si condition **Alors**
    Traitement 1
Sinon
    Traitement 2
Fin Si

2.2.1.2 Fonctionnement

La "condition" est un prédicat, qui peut être vrai ou faux, selon les valeurs des paramètres la constituant.

- Si la condition est vérifiée (sa valeur est vrai), c'est la ¡traitement 1¿ qui sera exécutée. Ensuite, le système passe à l'exécution juste après le FinSi.
- Dans le cas contraire, lorsque la condition n'est pas vérifiée (valeur de la condition est faux), le "traitement 2" qui s'exécute, en cas où celle ci existe (facultative). Si elle n'existe pas, le système passe directement à l'instruction qui suit le FinSi.
- Les suites traitements 1 et 2, peuvent être des actions simples ou même des structures conditionnelles.

2.2.1.3 Organigramme

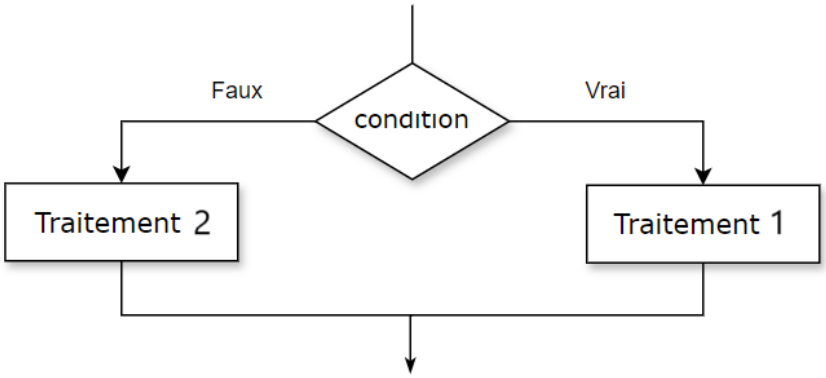

FIGURE 2.1 – Organigramme de la structure conditionnelle alternative

2.2.1.4 Exemple

Algorithme 12 POS-NEG

Variables : A : Réel
Début
  Afficher("Donner un nombre : ")
  Lire(A)
  Si A $<$ 0 **Alors**
    Afficher("Le nombre est strictement négatif")
  Sinon
    Afficher("Le nombre est positif")
  Fin Si
Fin

2.2.2 La structure conditionnelle simple

Lorsque nous n'avons pas de traitement à effectuer si la condition est fausse, il est possible d'omettre la partie "Sinon" dans la structure conditionnelle alternative. Dans ce

cas, on parle d'une structure conditionnelle simple, qui se limite à exécuter une action uniquement si la condition est vraie.

2.2.2.1 Syntaxe

Si condition **Alors**
  Traitement
Fin Si

2.2.2.2 Fonctionnement

- Si la condition est vérifiée (sa valeur est vrai), le "Traitement" qui sera exécutée. Ensuite, le système passe à l'exécution juste après le FinSi.
- Dans le cas contraire, lorsque la condition n'est pas vérifiée (valeur de la condition est faux), le système passe directement à l'instruction qui suit le FinSi car il y a aucun traitement à exécuté.

2.2.2.3 Organigramme

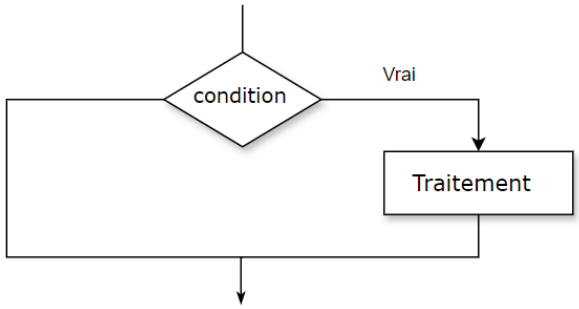

FIGURE 2.2 – Organigramme de la structure conditionnelle simple

2.2.2.4 Exemple

Algorithme 13 NUL
Variables : A : Réel
Début
  Afficher("Donner un nombre : ")
  Lire(A)
  Si A = 0 **Alors**
    Afficher("Le nombre est null")
  Fin Si
Fin

2.2.3 La structure conditionnelle imbriquée

Les structures conditionnelles imbriquées sont des instructions de contrôle dans lesquelles une instruction conditionnelle SI est placée à l'intérieur d'une autre instruction

conditionnelle. Elles permettent de tester plusieurs conditions successives et d'exécuter différentes instructions en fonction des résultats des tests.

Les structures conditionnelles imbriquées permettent de gérer des situations plus complexes où plusieurs décisions doivent être prises de manière séquentielle. Dans ce type de structure, une deuxième condition est évaluée uniquement si la première condition est vraie ou fausse (selon la logique de l'algorithme).

2.2.3.1 Exemple

Supposant que nous voulons écrire un algorithme qui fait la résolution des équation du premier degré de la forme $Ax + B = 0$.

Les données d'entré pour ce problème sont les coefficient A et B. Nous avons trois cas de figure à traiter.

- **Si** $A \neq 0$:
  - Si A est différent de 0, on peut résoudre directement l'équation. La solution est donnée par la formule suivante :

$$x = -\frac{B}{A}$$

  - L'algorithme calcule alors la valeur de x et l'affiche comme étant la solution unique.
- **Vérification si** $A = 0$:
  - Si $A = 0$, on examine la valeur de B à l'aide d'une structure **Si...Sinon** imbriquée :
    - **Si** $B = 0$: L'équation devient $0 = 0$, ce qui est toujours vrai. Cela signifie que **toutes les valeurs de** x **sont solutions** (équation indéterminée).
    - **Sinon** ($B \neq 0$) : L'équation devient $B = 0$, ce qui est faux. Il n'existe donc **aucune solution** (équation impossible).
- **Sortie des résultats** :
  - Selon les valeurs de A et B, l'algorithme affiche l'une des trois sorties possibles :
    - "L'équation est indéterminée : toutes les valeurs de x sont solutions."
    - "Pas de solution : l'équation est impossible."
    - "La solution est : x."

L'organigramme de la solution proposée est le suivant :

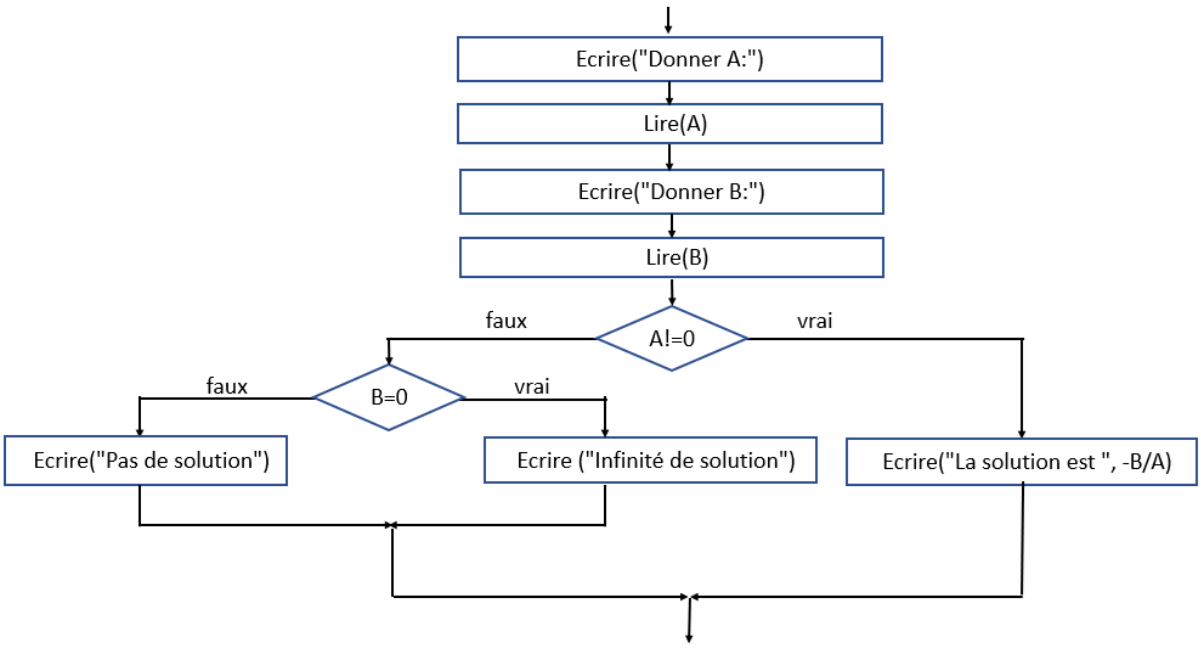

FIGURE 2.3 – Organigramme de la solution

Voici une première version en pseudo code de la solution :

Algorithme 14 EQUATION V1

Variables : A, B : Réel
Début
  Afficher(”Donner A : ”)
  Lire(A)
  Afficher(”Donner B : ”)
  Lire(B)
  Si A ! = 0 **Alors**
    Ecrire(”La solution est : ”, -B/A)
  Sinon
    Si B == 0 **Alors**
      Ecrire(”Infinité de solutions”)
    Sinon
      Ecrire(”Pas de solution !”)
    Fin Si
  Fin Si
Fin

2.2.3.2 Structure conditionnelle multiple

Lorsque notre solution nécessite de vérifier plusieurs conditions, l'utilisation de Si imbriqués à plusieurs niveaux peut rendre le code lourd et difficile à lire. Dans ce genre de situation, il est préférable d'utiliser une syntaxe plus claire appelée structure conditionnelle multiple. Cette construction algorithmique permet de tester plusieurs conditions de manière séquentielle et d'exécuter une action en fonction de la première condition vérifiée, améliorant ainsi la lisibilité et la gestion du code.

2.2.3.3 Syntaxe

Si condition 1 **Alors**
  Traitement 1
Sinon Si consition 2 **Alors**
  Traitement 2
Sinon Si consition 3 **Alors**
  Traitement 4
  ...
Sinon
  Traitement par défaut
Fin Si

2.2.3.4 Fonctionnement

La Condition 1 est vérifiée en premier lieu. Si elle est vraie, alors Traitement 1 est exécuté. Si elle est fausse, l'algorithme passe à la condition suivante.

Si Condition 1 est fausse, alors Condition 2 est évaluée. Si Condition 2 est vraie, Traitement 2 est exécuté. Sinon, l'algorithme passe à la condition suivante.

Ensuite, si Condition 2 est également fausse, Condition 3 est évaluée. Si Condition 3 est vraie, un ensemble de traitements (Traitement 4, etc.) est exécuté.

Si aucune des conditions précédentes n'est vraie, le bloc Else est activé, et Traitement par défaut est exécuté, ce qui permet de gérer les cas où aucune des conditions n'a été satisfaite.

Voici une deuxième version de la solution précédente :

Algorithme 15 EQUATION V2

Variables : A, B : Réel
Début
  Afficher("Donner A : ")
  Lire(A)
  Afficher("Donner B : ")
  Lire(B)
  Si A ! = 0 **Alors**
    Ecrire("La solution est : ", -B/A)
  Sinon Si B == 0 **Alors**
    Ecrire("Infinité de solutions")
  Sinon
    Ecrire("Pas de solution !")
  Fin Si
Fin

2.3 Les structures répétitives

Dans les problèmes quotidiens, on ne traite pas uniquement des séquences d'actions, sous ou sans conditions, mais il peut être fréquent d'être obligé d'exécuter un traitement (séquence d'actions), plusieurs fois. Ce problème est résolu à l'aide des structures

répétitives. Celles ci permettent de donner un ordre de répétition d'une action ou d'une séquence d'actions une ou plusieurs fois.

Les boucles sont des structures de contrôle très importantes en algorithmique. Elles permettent de répéter une séquence d'instructions un certain nombre de fois, soit de manière déterminée, soit jusqu'à ce qu'une condition soit remplie. Elles évitent la redondance de code et permettent d'automatiser des tâches répétitives. On distingue principalement deux types de boucles : les boucles conditionnelles (TantQue) et les boucles à itération fixe (Pour).

2.3.1 La boucle Pour

Une boucle "pour" est une structure de contrôle utilisée en programmation pour répéter un ensemble d'instructions un nombre déterminé de fois. Elle est généralement utilisée lorsque le nombre d'itérations est connu à l'avance.

2.3.1.1 Syntaxe

La syntaxe de la boucle est la suivante :

Pour *variable* =VI **à** VF **Faire**
  Traitement
Fin Pour

Explication :

- variable : La variable qui va prendre les différentes valeurs dans la boucle.
- VI : La valeur initial, c'est la première valeur que prendra la variable au début de la boucle.
- VF : La valeur final. C'est la dernière valeur que prendra la variable à la fin de la boucle.
- Traitement : Ce sont les actions qui seront exécutées à chaque itération.

Exemple : La boucle suivante permet d'afficher les nombre 1 jusqu'à 100.

Pour $i = 1$ **à** 100 **Faire**
  Ecrire(i)
Fin Pour

2.3.1.2 Fonctionnement

— Initialisation : La boucle commence par initialiser la variable de contrôle (i dans l'exemple) à la valeur initiale VI (1 dans l'exemple).

— Condition de continuation : Avant chaque itération, le programme vérifie si la variable de contrôle est inférieure ou égale à la valeur finale VF (100 dans l'exemple). Si cette condition est vraie, le programme entre dans la boucle.

— Exécution des instructions : Une fois la condition vérifiée, les instructions du traitement à l'intérieur de la boucle sont exécutées. Dans l'exemple, cela signifie afficher la valeur de i.

— Incrémentation : Après l'exécution des instructions, la variable de contrôle (i) est automatiquement incrémentée (généralement de 1, sauf indication contraire).
— Répétition : Le processus se répète : la condition est vérifiée à nouveau. Si elle est toujours vraie, le programme exécute à nouveau les instructions. Sinon, la boucle se termine.
— Fin de la boucle : Une fois que la variable de contrôle dépasse la valeur finale, le programme sort de la boucle et continue avec le code suivant.

2.3.1.3 Exemples

Exemple 1 : Une boucle qui affiche 'Bonjour' 10 fois

Pour $i = 1$ **à** 10 **Faire**
    Ecrire("Bonjour")
Fin Pour

Exemple 2 : Une boucle qui calcule la somme de 1 à 5

$s \leftarrow 0$
Pour $i = 1$ **à** 5 **Faire**
    $s \leftarrow s + 1$
Fin Pour

Exemple 3 : Une boucle qui affiche la table de multiplication de 7

$n \leftarrow 7$
Pour $i = 1$ **à** 10 **Faire**
    Ecrire(n, "x", i , "=", n*i)
Fin Pour

2.3.2 La boucle TantQue

Une boucle "tant que" est une structure de contrôle de flux utilisée en programmation pour répéter un bloc d'instructions tant qu'une condition spécifiée reste vraie. Elle est particulièrement utile lorsque le nombre d'itérations n'est pas connu à l'avance et dépend d'une condition dynamique.

2.3.2.1 Syntaxe

TantQue condition **Faire**
    Traitement
Fin TantQue

2.3.2.2 Organigramme

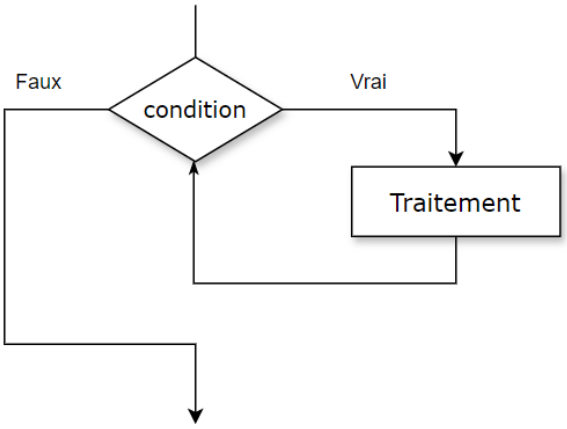

FIGURE 2.4 – La boucle TantQue

2.3.2.3 Fonctionnement

- Évaluation de la condition : Avant chaque itération, la condition est vérifiée. Si elle est vraie, le bloc d'instructions à l'intérieur de la boucle est exécuté.
- Exécution des instructions : Les instructions à l'intérieur de la boucle sont exécutées.
- Répétition : Après l'exécution des instructions, la condition est vérifiée à nouveau. Si elle reste vraie, la boucle continue de s'exécuter.
- Fin de la boucle : Lorsque la condition devient fausse, la boucle se termine et le contrôle passe à l'instruction suivante après la boucle.

Attention : Dans le corps de la boucle (instructions), au moins une instruction doit obligatoirement faire évoluer les valeurs des variables testés dans la condition. Sinon la boucle sera infinie (c-à-d qu'elle ne s'arrête jamais).

Exemple : Une boucle infinie qui affiche le mot Informatique à l'infinie

$i \leftarrow 1$
TantQue $i \leq 10$ **Faire**
  Ecrire("Informatique")
Fin TantQue

2.3.2.4 Exemples

Exemple 1 : Une boucle qui affiche 'Bonjour' 10 fois

$i \leftarrow 1$
TantQue $i \leq 10$ **Faire**
  Ecrire("Bonjour")
  $i \leftarrow i + 1$
Fin TantQue

Exemple 2 : Une boucle qui calcule la somme de 10 entier entré par l'utilisateur

$s \leftarrow 0$ ▷ Initialiser la variable somme
$i \leftarrow 1$ ▷ Initialiser le compteur de la boucle
TantQue $i \leq 10$ **Faire**
    Ecrire("Donner un nombre : ")
    Lire(a)
    $s \leftarrow s + a$ ▷ Accumuler les valeurs de a dans la variable s
    $i \leftarrow i + 1$ ▷ Incrémenter le compteur de la boucle i
Fin TantQue

Exemple 3 : Deviner un nombre de l'intervalle [1, 10].

Algorithme 16 Deviner un nombre

Variables : essai, N, nombre_secret : Entier
Début

```
nombre_secret ← 7
essai ← −1
TantQue essai ≠ nombre_secret Faire
    Lire essai
    Si essai < nombre_secret Alors
        Afficher "Trop petit"
    Sinon Si essai > nombre_secret Alors
        Afficher "Trop grand"
    Sinon
        Afficher "Bravo !"
    Fin Si
Fin TantQue
```

Deuxième partie

Programmation 1

Chapitre 3

Introduction à Python

3.1 Introduction

3.1.1 Histoire et philosophie de Python

3.1.1.1 Origines de Python

Python a été créé par **Guido van Rossum** à la fin des années 1980, et la première version officielle a été publiée en 1991. Guido voulait un langage qui soit facile à lire et à utiliser, tout en étant puissant et polyvalent. Il s'est inspiré de plusieurs langages existants, notamment ABC, un autre projet du CWI, ainsi que Modula-3 et C. Le nom "Python" ne vient pas du serpent, mais est inspiré de la troupe comique britannique **Monty Python**, reflétant l'humour et la simplicité que Van Rossum voulait intégrer dans le langage.

3.1.1.2 Philosophie de Python

La philosophie de Python se résume dans une série de principes appelés le **Zen de Python**, qui sont inclus directement dans l'interpréteur Python (vous pouvez les consulter en tapant 'import this' dans l'interpréteur). Certains de ces principes incluent :

— **Lisibilité du code** : Python est conçu pour être facile à lire, avec une syntaxe claire et simple. Le code doit être compréhensible par les humains avant tout.

— **Moins, c'est mieux** : Python privilégie une approche minimaliste, où il y a souvent une seule façon évidente de faire les choses, ce qui facilite l'apprentissage et la collaboration entre développeurs.

— **Simplicité** : Le langage favorise la simplicité et l'élégance, évitant les fonctionnalités trop complexes ou les syntaxes obscures.

— **Facilité d'utilisation** : Python est destiné à être accessible à tous, que vous soyez débutant ou expert. C'est pourquoi il est utilisé dans une grande variété de domaines, tels que le développement web, la science des données, l'intelligence artificielle, et bien plus encore.

3.1.1.3 Évolution de Python

Depuis sa création, Python a connu plusieurs versions majeures :

— **1991 :** Publication de Python 0.9.0, qui contenait déjà certaines caractéristiques essentielles, comme les fonctions, les classes, les exceptions, et un module pour le système d'exploitation.

— **1994 :** Python 1.0 introduit de nouvelles fonctionnalités comme les modules, les types de données intégrés (listes, dictionnaires).

— **2000 :** Lancement de Python 2.0 qui améliore la gestion des types de données et introduit la collecte de déchets (garbage collection). Python 2.x continuera à évoluer jusqu'à la version 2.7, supportée jusqu'en 2020.

— **2008 :** Sortie de Python 3.0, une version majeure non rétro-compatible avec Python 2, mais apportant des améliorations significatives en termes de clarté et de cohérence du langage. Le passage à Python 3 a été un grand défi pour la communauté, mais il a permis de moderniser le langage.

Python a évolué pour devenir l'un des langages les plus populaires au monde grâce à sa **communauté active** et à ses nombreuses **bibliothèques** pour le développement web, l'intelligence artificielle, l'automatisation, la science des données, et bien plus.

3.1.1.4 Caractéristiques, usage et popularité

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

— Il est multiplateforme. C'est-à-dire qu'il fonctionne sur de nombreux systèmes d'exploitation : Windows, Mac OS X, Linux, Android, iOS, depuis les mini-ordinateurs Raspberry Pi jusqu'aux supercalculateurs.

— Il est gratuit. Vous pouvez l'installer sur autant d'ordinateurs que vous voulez (même sur votre téléphone !).

— C'est un langage de haut niveau. Il demande relativement peu de connaissance sur le fonctionnement d'un ordinateur pour être utilisé.

— C'est un langage interprété. Un script Python n'a pas besoin d'être compilé pour être exécuté, contrairement à des langages comme le C ou le C++.

— Il est orienté objet. C'est-à-dire qu'il est possible de concevoir en Python des entités qui miment celles du monde réel avec un certain nombre de règles de fonctionnement et d'interactions.

— Il est relativement simple à prendre en main.

L'approche centrée sur la lisibilité et la simplicité a permis à Python de devenir un langage largement adopté dans plusieurs domaines :

— **Enseignement** : Python est souvent le premier langage enseigné aux étudiants en programmation en raison de sa syntaxe simple.

— **Développement web** : Avec des frameworks comme **Django** et **Flask**, Python est utilisé pour construire des applications web robustes.

— **Science des données et IA** : Grâce à des bibliothèques comme **NumPy**, **Pandas**, et **TensorFlow**, Python est un acteur majeur dans l'analyse de données et l'intelligence artificielle.

3.1.2 Installation de Python et environnement de travail (IDE, Pyzo)

Pour commencer à programmer en Python, vous aurez besoin d'installer Python et de choisir un environnement de travail (IDE). Voici un guide simple pour l'installation de Python et la configuration d'un environnement de travail comme Pyzo ou d'autres IDE populaires.

3.1.2.1 Téléchargement

Rendez-vous sur le site officiel de Python : `https://www.python.org/downloads/` et téléchargez la version correspondant à votre système d'exploitation (Windows, MacOS, ou Linux).

3.1.2.2 Installation

- **Sous Windows :** Exécutez le fichier téléchargé. Assurez-vous de cocher l'option **Add Python to PATH**. Puis, suivez les étapes d'installation.
- **Sous MacOS et Linux :** Double-cliquez pour installer (MacOS). Sous Linux, vérifiez si Python est déjà installé en tapant la commande suivante :

```
python3 --version
```

  Sinon, installez avec la commande :

```
sudo apt install python3
```

3.1.2.3 Choisir et Installer un IDE

- **Pyzo** : Téléchargez Pyzo depuis `https://pyzo.org/` et suivez les instructions d'installation.
- **PyCharm** : Téléchargez sur `https://www.jetbrains.com/pycharm/`.
- **Visual Studio Code (VS Code)** : Téléchargez sur `https://code.visualstudio.com/`. Installez ensuite l'extension Python pour bénéficier de la complétion automatique et du débogage.

3.2 Les Bases du Langage

3.2.1 Les deux modes d'exécution d'un code Python

- Soit on enregistre un ensemble d'instructions Python dans un fichier grâce à un éditeur (on parle alors d'un *script Python*) que l'on exécute par une commande ou par une touche du menu de l'éditeur ;
- soit on utilise l'interpréteur Python embarqué qui exécute la *boucle d'évaluation*.

```
>>> 5 + 3     Affichage d'une invite. L'utilisateur tape une expression.
8             Évaluation et affichage du résultat.
>>>           Réaffichage de l'invite.
```

FIGURE 3.1 – La boucle d'évaluation de l'interpréteur Python

3.2.2 Identificateurs

Comme tout langage, Python utilise des *identificateurs* pour nommer ses objets.

Un identificateur Python est une suite non vide de caractères, de longueur quelconque, formée d'**un** *caractère de début* et de **zéro ou plusieurs** *caractères de continuation.*

Sachant que :

— le *caractère de début* peut être n'importe quelle lettre Unicode, y compris le caractère souligné (_) ;

— un *caractère de continuation* est un caractère de début ou un chiffre.

Les identificateurs sont sensibles à la casse et ne doivent pas être un mot réservé de Python 3 (voir Les mots réservés de Python 3 ci-dessous).

3.2.3 Style de nommage

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Voici le style utilisé dans ce document :

— `nom_de_ma_variable` pour les variables ;

— `NOM_DE_MA_CONSTANTE` pour les constantes ;

— `maFonction`, `maMethode` pour les fonctions et les méthodes ;

— `MaClasse` pour les classes ;

— `UneExceptionError` pour les exceptions ;

— `nom_de_module` pour les modules et pour tous les autres identificateurs.

3.2.4 Les mots réservés de Python 3

Quelques mots clés de python :

```
and       del      from    None      True
as        elif     global  nonlocal  try
assert    else     if      not       while
break     except   import  or        with
class     False    in      pass      yield
continue  finally  is      raise     match
def       for      lambda  return
```

3.2.5 Notion d'expression

Une "expression" est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.

Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux représentant directement des valeurs, d'identificateurs et d'opérateurs.

3.3 Les types de données primitives

En Python, les types de données primitives sont des types de base qui permettent de manipuler des données simples et sont intégrés directement dans le langage. Voici un aperçu des types de données primitives les plus courants :

3.3.1 Le type int (entiers) :

Représente les nombres entiers, positifs ou négatifs, sans partie décimale.
Exemple : 5, -3, 42
En Python 3, les entiers peuvent être de taille illimitée (contrairement à d'autres langages).

Opérateur	Description	Exemple
`+`	Addition	`5 + 3 = 8`
`-`	Soustraction	`7 - 4 = 3`
`*`	Multiplication	`6 * 2 = 12`
`/`	Division réelle	`9 / 2 = 4.5`
`//`	Division entière (quotient)	`9 // 2 = 4`
`%`	Modulo (reste de la division entière)	`9 % 2 = 1`
`**`	Exponentiation (puissance)	`2 ** 3 = 8`
`+=`	Incrémente un entier par une valeur	`a += 1` (équivaut à `a = a + 1`)
`-=`	Décrémente un entier par une valeur	`a -= 1` (équivaut à `a = a - 1`)

Table 3.1 – Opérateurs du type `int` en Python

3.3.2 Le type float (réels) :

Représente les nombres réels avec une partie décimale.
Exemple : 3.14, -0.5, 2.0
Utilisé pour des calculs plus précis avec des nombres réels.

Opérateur	Description	Exemple
`+`	Addition	`5.5 + 3.2` = `8.7`
`-`	Soustraction	`10.5 - 2.3` = `8.2`
`*`	Multiplication	`4.1 * 3.0` = `12.3`
`/`	Division réelle	`7.0 / 2.0` = `3.5`
`//`	Division entière (résultat float si un opérande est float).	`7.0 // 2` = `3.0`
`%`	Modulo, reste de la division entière.	`7.5 % 2.0` = `1.5`
`**`	Exponentiation (puissance).	`2.0 ** 3.0` = `8.0`
`-` (unaire)	Négation, change le signe.	`-3.5` = `-3.5`

Table 3.2 – Opérateurs pour les flottants en Python

3.3.3 Le type complex (nombres complexes) :

Représente les nombres complexes, avec une partie réelle et une partie imaginaire.
Exemple : 3 + 4j, où j est la partie imaginaire.
Python gère directement les calculs avec les nombres complexes.

3.3.4 Le type bool (booléens) :

Représente les valeurs de vérité : True ou False.
Exemple : True, False
Utilisé dans les opérations logiques et les structures conditionnelles.
En interne, True est traité comme 1 et False comme 0.

Opérateur	Description	Exemple
`and`	`True` si les deux opérandes sont `True`, sinon `False`.	`True and False` donne `False`
`or`	`True` si au moins un des opérandes est `True`.	`True or False` donne `True`
`not`	Inverse la valeur de vérité d'un booléen.	`not True` donne `False`
`==`	Compare l'égalité de deux valeurs.	`5 == 5` donne `True`
`!=`	Compare la différence entre deux valeurs.	`5 != 3` donne `True`
$>$	Supérieur strictement.	`5 > 3` donne `True`
$<$	Inférieur strictement.	`5 < 3` donne `False`
`>=`	Supérieur ou égal.	`5 >= 5` donne `True`
`<=`	Inférieur ou égal.	`5 <= 6` donne `True`

TABLE 3.3 – Opérateurs pour les booléens en Python

3.3.5 Entrées et sorties (input/print)

3.3.5.1 Instruction de lecture

La fonction **input()** en Python est utilisée pour obtenir des données de l'utilisateur à partir de la console. Voici un aperçu détaillé de son fonctionnement et de son utilisation. La fonction **input()** retourne toujours une chaîne de caractères (str). Peu importe le type de données que l'utilisateur entre, tout est converti en chaîne.

Si vous devez obtenir un nombre (entier ou flottant) de l'utilisateur, vous devez convertir la chaîne résultante en utilisant les fonctions appropriées (**int()** ou **float()**).

Dans cet exemple, nous allons lire une chaîne de caractères, un entier et un nombre réel à partir des entrées de l'utilisateur :

```
1 # Lire une chaine de caractères
2 name = input("Entrez votre nom: ")
3
4 # Lire un entier
5 age = int(input("Entrez votre age: "))
6
7 # Lire un réel
8 height = float(input("Entrez votre taille en mètres: "))
```

3.3.5.2 Instruction d'écriture

La fonction **print** en Python est utilisée pour afficher des données à l'écran. Elle peut afficher des chaînes de caractères, des nombres, des variables et des expressions plus complexes. Elle permet également de formater la sortie à l'aide des f-strings pour une présentation plus lisible.

— 1. Cet exemple montre comment afficher un message simple :

```python
# Afficher un message simple
print("Bonjour, monde !")
```

— 2. La fonction `print()` peut afficher plusieurs valeurs séparées par des virgules :

```python
# Afficher plusieurs valeurs séparées par des virgules
nom = "Alice"
age = 25
print("Nom :", nom, "Âge :", age)
```

— 3. Les f-strings permettent d'intégrer des expressions à l'intérieur de chaînes de caractères à l'aide de accolades. Elles sont précédées par la lettre `f` :

```python
# Afficher avec des f-strings
taille = 1.75
print(f"Bonjour {nom}, vous avez {age} ans et mesurez {taille} m
    ètres.")
```

3.4 Structures de Contrôle

3.4.1 Structures conditionnelles (if, else, elif)

Les conditions permettent d'exécuter une ou plusieurs instructions seulement si une condition est vraie.

— if : Évalue une condition et exécute le bloc de code associé si la condition est vraie.

— elif : Permet de tester d'autres conditions si la précédente est fausse.

— else : Exécute un bloc de code si aucune des conditions précédentes n'est vraie.

On distingue trois formes de la structure conditionnelle :

1. La structure conditionnelle alternative
2. La structure conditionnelle simple
3. La structure conditionnelle multiple

3.4.1.1 La structure conditionnelle alternative

La structure conditionnelle alternative permet d'exécuter un bloc de code lorsque la condition est vraie et un autre bloc lorsque la condition est fausse. Cela se fait en utilisant les instructions if, elif et else.

Voici la syntaxe générale d'une structure conditionnelle alternative en Python :

```python
if condition:
    # Bloc de code exécuté si la condition est vraie
elif autre_condition:
    # Bloc de code exécuté si l'autre condition est vraie
else:
    # Bloc de code exécuté si aucune des conditions n'est vraie
```

Voici un exemple illustrant l'utilisation d'une structure conditionnelle alternative :

```python
# Exemple de structure conditionnelle alternative
age = int(input("Votre âge ?"))

if age < 18:
    print("Vous êtes mineur.")
elif age == 18:
    print("Vous venez d'atteindre la majorité.")
else:
    print("Vous êtes majeur.")
```

3.4.1.2 La structure conditionnelle simple

La structure conditionnelle simple permet d'exécuter un bloc de code uniquement si une condition est vraie. Si la condition est fausse, le programme continue sans exécuter ce bloc.

Voici la syntaxe générale d'une structure conditionnelle simple en Python :

```python
if condition:
    # Bloc de code exécuté si la condition est vraie
```

Voici un exemple illustrant l'utilisation d'une structure conditionnelle simple :

```python
# Exemple de structure conditionnelle simple
age = int(input("Votre âge ?"))

if age < 18:
    print("Vous êtes mineur.")
```

3.4.1.3 La structure conditionnelle multiple

La structure conditionnelle multiple permet de tester plusieurs conditions de manière séquentielle. Elle utilise les instructions if, elif, et else pour gérer plusieurs cas.

Voici la syntaxe générale d'une structure conditionnelle multiple en Python :

```python
if condition1:
    # Bloc de code exécuté si condition1 est vraie
elif condition2:
    # Bloc de code exécuté si condition2 est vraie
elif condition3:
    # Bloc de code exécuté si condition3 est vraie
else:
    # Bloc de code exécuté si aucune des conditions n'est vraie
```

Voici un exemple illustrant l'utilisation d'une structure conditionnelle multiple :

```python
# Exemple de structure conditionnelle multiple
note = int(input("Votre note ?"))

if note >= 16:
    print("Mention Très Bien")
elif note >= 14:
    print("Mention Bien")
elif note >= 12:
    print("Mention Assez Bien")
else:
    print("Mention Passable ou Insuffisante")
```

3.4.2 Les boucles

3.4.2.1 La boucle : for

La boucle for en Python permet de parcourir un ensemble d'éléments, comme une liste ou une chaîne de caractères. Elle est utilisée pour exécuter un bloc de code un nombre déterminé de fois, en itérant sur chaque élément de l'itérable.

Voici la syntaxe générale d'une boucle `for` en Python :

```
for élément in itérable:
    # Bloc de code à exécuter pour chaque élément
```

Voici un exemple illustrant l'utilisation d'une boucle `for` :

```
# Exemple de boucle for

for nombre in range(1,6):
    print("Le carré de", nombre, "est", nombre**2)
```

La fonction **range** en Python génère une séquence d'entiers. Elle est souvent utilisée dans les boucles for pour itérer un nombre déterminé de fois. La fonction peut prendre un, deux ou trois arguments pour définir les bornes et l'incrément :

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Les arguments de la fonction range() sont facultatifs, mais ils peuvent être utilisés pour contrôler le comportement de la séquence générée :

- start (optionnel) : spécifie la valeur de départ de la séquence. Par défaut, la valeur de départ est 0 ;
- stop : spécifie la valeur de fin de la séquence. La séquence générée par range() s'arrête juste avant cette valeur ;
- step (optionnel) : spécifie l'incrément entre les valeurs de la séquence. Par défaut, l'incrément est de 1.

Voici quelques exemples illustrant l'utilisation de la fonction `range` :

Exemple 1 : Utilisation de `range` avec un seul argument

```
# Exemple 1 : Utilisation de range avec un seul argument
for i in range(5):
    print(i)  # Affiche les nombres de 0 à 4
```

Exemple 2 : Utilisation de `range` avec deux arguments

```
# Exemple 2 : Utilisation de range avec deux arguments
for i in range(2, 6):
    print(i)  # Affiche les nombres de 2 à 5
```

Exemple 3 : Utilisation de `range` avec trois arguments

```
# Exemple 3 : Utilisation de range avec trois arguments
for i in range(0, 10, 2):
    print(i)  # Affiche les nombres pairs de 0 à 8
```

3.4.2.2 Le boucle : while

La boucle **while** en Python permet d'exécuter un bloc de code tant qu'une condition spécifiée est vraie. Elle est utilisée lorsque le nombre d'itérations n'est pas connu à l'avance et dépend d'une condition dynamique.

La syntaxe générale d'une boucle `while` en Python :

```
while condition:
    # Bloc de code à exécuter tant que la condition est vraie
```

Un exemple illustrant l'utilisation d'une boucle `while` :

```
# Exemple de boucle while
compteur = 0

while compteur < 5:
    print("Compteur =", compteur)
    compteur = compteur + 1  # Incrémente le compteur
```

3.4.3 Instructions de contrôle de boucle (break)

L'instruction `break` est utilisée pour sortir immédiatement d'une boucle. Elle est souvent utilisée lorsqu'une condition particulière est remplie, ce qui permet d'interrompre l'exécution de la boucle.

```
for element in iterable:
    if condition:
        break  # Sortie de la boucle
```

Exemple :

Dans cet exemple, la boucle s'arrête lorsque i atteint 5 :

```
for i in range(10):
    if i == 5:
        print("Sortie de la boucle à i =", i)
        break
```

3.5 Fonctions

3.5.1 Définition et utilisation des fonctions (def)

Les fonctions en Python sont des blocs de code réutilisables qui effectuent une tâche spécifique. Elles permettent de diviser un programme en sous-parties, ce qui facilite la gestion, la lecture et la maintenance du code. En définissant des fonctions, vous pouvez éviter la répétition de code et améliorer la modularité.

3.5.1.1 Déclaration d'une fonction

Pour déclarer(définir) une fonction en Python, on utilise le mot-clé `def`, suivi du nom de la fonction, d'une paire de parenthèses (dans laquelle on peut éventuellement indiquer

des paramètres), et d'un deux-points. Le corps de la fonction commence sur la ligne suivante, avec un retrait pour indiquer son niveau.

```
def nom_de_la_fonction(paramètres):
    # Corps de la fonction
    pass  # Cette instruction indique que la fonction ne fait rien
        pour l'instant
```

3.5.1.2 Utilisation des fonctions

Une fois définie, vous pouvez appeler une fonction en utilisant son nom et en fournissant les arguments nécessaires. Voici un exemple :

```
def saluer(nom):
    print("Bonjour,", nom)

# Appel de la fonction
saluer("Alice")  # Affiche : Bonjour, Alice
```

3.5.2 Paramètres, arguments et valeurs de retour

Les fonctions peuvent accepter des paramètres, qui sont des variables définies dans la déclaration de la fonction, et peuvent retourner des valeurs.

Paramètres et arguments

Les paramètres sont des valeurs qui sont passées à une fonction lors de son appel. Les arguments sont les valeurs réelles que vous passez à ces paramètres. Voici quelques types de paramètres :

- **Paramètres positionnels :** Ce sont les paramètres qui doivent être passés dans le même ordre que défini dans la fonction.
- **Paramètres nommés :** Vous pouvez également passer des arguments en utilisant le nom des paramètres, ce qui vous permet de spécifier l'ordre.

Exemple de paramètres positionnels :

```
def addition(a, b):
    return a + b

# Appel de la fonction
resultat = addition(3, 5)  # 3 et 5 sont des arguments positionnels
print("Le résultat est :", resultat)  # Affiche : Le résultat est : 8
```

Exemple de paramètres nommés :

```
def afficher_info(nom, age):
    print(f"Nom: {nom}, Age: {age}")

# Appel de la fonction avec des arguments nommés
afficher_info(age=25, nom="Alice")  # Affiche : Nom: Alice, Age: 25
```

Valeurs de retour

Une fonction peut renvoyer une valeur à l'aide du mot-clé `return`. Si aucune valeur n'est retournée, la fonction renvoie `None` par défaut.

Exemple d'une fonction avec valeur de retour :

```
def multiplication(a, b):
    return a * b

# Appel de la fonction
resultat = multiplication(4, 7)
print("Le produit est :", resultat)  # Affiche : Le produit est : 28
```

3.5.3 Portée des variables (globale vs locale)

La portée d'une variable détermine où elle peut être utilisée dans le code.

Variables locales

Une variable définie à l'intérieur d'une fonction est locale à cette fonction et ne peut pas être utilisée en dehors.

```
def fonction_locale():
    x = 10  # Variable locale
    print("x =", x)

fonction_locale()
# print(x)  # Erreur : x n'est pas défini ici
```

Variables globales

Une variable définie en dehors de toutes les fonctions est globale et peut être utilisée à l'intérieur des fonctions. On peut utiliser le mot-clé `global` pour modifier une variable globale à l'intérieur d'une fonction.

```
x = 5  # Variable globale

def fonction_globale():
    global x
    x += 1  # Modifie la variable globale

fonction_globale()
print("x =", x)  # Affiche 6
```

3.5.4 Fonctions lambda

Les fonctions lambda sont des fonctions anonymes que vous pouvez créer en utilisant le mot-clé `lambda`. Elles sont utiles pour créer des petites fonctions à la volée, souvent utilisées avec des fonctions comme `map()`, `filter()` et `sorted()`. Cette section explique ce qu'est une fonction lambda, son utilisation, ainsi qu'une comparaison avec les fonctions normales définies avec `def`.

Syntaxe

Une fonction lambda est définie avec le mot-clé `lambda`, suivi des arguments, d'un deux-points et d'une expression. La syntaxe générale est la suivante :

```
lambda arguments: expression
```

Les fonctions lambda peuvent prendre n'importe quel nombre d'arguments, mais elles ne peuvent avoir qu'une seule expression, ce qui les rend plus succinctes.

Exemple

Voici un exemple d'utilisation d'une fonction lambda pour ajouter deux nombres :

```
# Fonction lambda pour additionner deux nombres
addition = lambda x, y: x + y

resultat = addition(3, 4)
print("Le résultat de l'addition est :", resultat)  # Affiche : Le ré
    sultat de l'addition est : 7
```

3.6 Structures de Données

3.6.1 Les listes

Les listes sont des structures de données fondamentales en Python. Elles permettent de stocker une collection d'objets dans un ordre donné, ce qui en fait un outil puissant pour gérer des ensembles de données variés. Ce chapitre explore les caractéristiques, les opérations, et les méthodes associées aux listes en Python.

Une liste en Python est une collection ordonnée et modifiable d'objets. Elle peut contenir des éléments de types variés, tels que des nombres, des chaînes, des objets, et même d'autres listes. Les éléments d'une liste sont définis en les séparant par des virgules et en les plaçant entre crochets `[]`.

3.6.1.1 Création d'une liste

Nous pouvons créer des listes de différentes manière, voici quelques exemples de création de listes :

```
# Liste vide
liste_vide = []

# Liste avec des éléments
fruits = ["pomme", "banane", "cerise"]

# Liste avec des types mixtes
elements = [1, "deux", 3.0, True]
```

3.6.1.2 Accès aux éléments d'une liste

Les éléments d'une liste peuvent être accédés à l'aide d'indices. Les indices commencent à 0, ce qui signifie que le premier élément a un indice de 0, le deuxième un indice de 1, et ainsi de suite.

Voici comment accéder aux éléments d'une liste :

```
fruits = ["pomme", "banane", "cerise"]

# Accéder au premier élément
print(fruits[0])  # Affiche : pomme

# Accéder au deuxième élément
print(fruits[1])  # Affiche : banane

# Accéder au dernier élément
print(fruits[-1])  # Affiche : cerise
```

3.6.1.3 Modification des éléments

Les éléments d'une liste peuvent être modifiés en utilisant leur index :

```
fruits = ["pomme", "banane", "cerise"]
fruits[1] = "orange"
print(fruits)  # Affiche : ['pomme', 'orange', 'cerise']
```

3.6.1.4 Opérations sur les listes

Python fournit plusieurs opérations de base pour manipuler les listes.

3.6.1.5 Ajout d'éléments

Pour ajouter un élément à la fin d'une liste, vous pouvez utiliser la méthode `append()` :

```
fruits = ["pomme", "banane", "cerise"]
fruits.append("kiwi")
print(fruits)  # Affiche : ['pomme', 'orange', 'cerise', 'kiwi']
```

Pour ajouter plusieurs éléments, utilisez la méthode `extend()` :

```
fruits = ["pomme", "banane", "cerise"]
fruits.extend(["mangue", "ananas"])
print(fruits)  # Affiche : ['pomme', 'orange', 'cerise', 'kiwi', '
    mangue', 'ananas']
```

3.6.1.6 Insertion d'éléments

Pour insérer un élément à un index spécifique, utilisez la méthode `insert()` :

```
fruits = ["pomme", "banane", "cerise"]
fruits.insert(1, "fraise")  # Insère 'fraise' à l'index 1
print(fruits)  # Affiche : ['pomme', 'fraise', 'orange', 'cerise', '
    kiwi', 'mangue', 'ananas']
```

3.6.1.7 Suppression d'éléments

Pour supprimer un élément, vous pouvez utiliser `remove()` pour supprimer la première occurrence d'un élément ou `pop()` pour supprimer un élément à un index donné :

```python
fruits = ["pomme", "banane", "cerise", "kiwi"]
# Suppression par valeur
fruits.remove("kiwi")
print(fruits)  # Affiche : ['pomme', 'banane', 'cerise']

fruits = ["pomme", "banane", "cerise", "kiwi"]
# Suppression par index
fruit_supprime = fruits.pop(1)
print(fruit_supprime)  # Affiche : banane
print(fruits)  # Affiche : ["pomme", "cerise", "kiwi"]
```

3.6.1.8 Méthodes de liste courantes

Python fournit plusieurs méthodes intégrées pour manipuler les listes. Voici quelques-unes des méthodes les plus courantes :

Taille de la liste Pour obtenir le nombre d'éléments dans une liste, utilisez la fonction `len()` :

```python
nombres = [1, 2, 3, 4, 5]
taille = len(nombres)
print("La taille de la liste est :", taille)  # Affiche : La taille
    de la liste est : 5
```

Tri des éléments Pour trier les éléments d'une liste, utilisez la méthode `sort()` :

```python
nombres = [5, 3, 1, 4, 2]
nombres.sort()
print("Liste triée :", nombres)  # Affiche : Liste triée : [1, 2, 3,
    4, 5]
```

Inverser les éléments Pour inverser l'ordre des éléments d'une liste, utilisez la méthode `reverse()` :

```python
nombres = [1, 2, 3, 4, 5]
nombres.reverse()
print("Liste inversée :", nombres)  # Affiche : Liste inversée : [5,
    4, 3, 2, 1]
```

3.6.1.9 Slicing (slicing des liste)

En Python, le *slicing* ou découpage des listes est une technique permettant d'extraire des sous-parties d'une liste en utilisant une syntaxe simple : `liste[start:stop:step]`.

- **start** : L'indice de début (inclus).
- **stop** : L'indice de fin (exclus).
- **step** : Le pas (optionnel).

Voici quelques exemples de la manière d'utiliser le slicing en Python.

```python
# Exemple de liste
ma_liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Extraire les éléments de l'indice 2 à 5 (5 non inclus)
sous_liste = ma_liste[2:5]   # [2, 3, 4]
```

Slicing avec un pas (step) :
Vous pouvez également ajouter un **pas** pour extraire les éléments en sautant un nombre défini d'éléments.

```
# Extraire un élément sur deux
sous_liste = ma_liste[0:9:2]  # [0, 2, 4, 6, 8]
```

Omettre certains paramètres simplifie le slicing :

- **Omettre start** : commence au début de la liste.
- **Omettre stop** : va jusqu'à la fin de la liste.
- **Omettre step** : le pas est de 1 par défaut.

```
# Tous les éléments à partir de l'indice 3
sous_liste = ma_liste[3:]      # [3, 4, 5, 6, 7, 8, 9]

# Tous les éléments jusqu'à l'indice 4 (4 exclu)
sous_liste = ma_liste[:4]      # [0, 1, 2, 3]

# Tous les éléments dans l'ordre inverse
sous_liste = ma_liste[::-1]    # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

La table suivante résume les différentes syntaxes possibles.

Syntaxe	Signification
`liste[start:stop]`	Extraire les éléments de *start* à *stop-1*
`liste[start:stop:step]`	Extraire avec un pas de *step* entre chaque élément
`liste[:stop]`	Tous les éléments jusqu'à *stop-1*
`liste[start:]`	Tous les éléments à partir de *start*
`liste[::-1]`	Tous les éléments dans l'ordre inverse

TABLE 3.4 – Syntaxe du Slicing en Python

3.6.2 Les tuples

Les *tuples* sont une autre structure de données fondamentale en Python. Contrairement aux listes, les tuples sont **immuables**, c'est-à-dire que leurs éléments ne peuvent pas être modifiés après leur création. Ils sont souvent utilisés pour représenter des collections de données qui ne doivent pas changer, comme des coordonnées, des constantes, ou des ensembles de valeurs fixes.

Un tuple est une collection **ordonnée** et **non modifiable** d'éléments. Les éléments d'un tuple peuvent être de tout type : nombres, chaînes, objets, voire d'autres tuples. Les tuples sont définis en plaçant les éléments entre parenthèses `()` et en les séparant par des virgules.

3.6.2.1 Création d'un tuple

Il existe plusieurs façons de créer un tuple en Python :

```
# Tuple vide
tuple_vide = ()
```

```python

# Tuple avec des éléments
coordonnees = (10, 20)

# Tuple avec des types mixtes
informations = ("Alice", 25, True)

# Les parenthèses sont optionnelles
fruits = "pomme", "banane", "cerise"

# Tuple contenant un seul élément (notez la virgule)
un_element = ("unique", )
```

3.6.2.2 Accès aux éléments d'un tuple

Comme les listes, les tuples sont indexés. On peut accéder à leurs éléments en utilisant des indices, en commençant par 0 pour le premier élément.

```python
fruits = ("pomme", "banane", "cerise")

# Accéder au premier élément
print(fruits[0])  # Affiche : pomme

# Accéder au dernier élément
print(fruits[-1])  # Affiche : cerise
```

3.6.2.3 Immuabilité des tuples

Les tuples sont **immuables**, ce qui signifie qu'on ne peut ni modifier, ni ajouter, ni supprimer un élément après leur création. Toute tentative de modification provoquera une erreur :

```python
fruits = ("pomme", "banane", "cerise")
fruits[1] = "orange"  # Erreur : TypeError: 'tuple' object does not support item assignment
```

Pour modifier un tuple, il faut en créer un nouveau :

```python
fruits = ("pomme", "banane", "cerise")
nouveau_tuple = fruits + ("kiwi",)
print(nouveau_tuple)  # Affiche : ('pomme', 'banane', 'cerise', 'kiwi')
```

3.6.2.4 Opérations sur les tuples

Même si les tuples sont immuables, certaines opérations courantes sont possibles :

- **Concaténation** : Combiner deux tuples avec l'opérateur `+`.
- **Répétition** : Répéter un tuple plusieurs fois avec l'opérateur `*`.
- **Vérification d'appartenance** : Tester si un élément est présent avec `in`.

```python
t1 = (1, 2, 3)
t2 = (4, 5)

```

```
# Concaténation
resultat = t1 + t2
print(resultat)  # (1, 2, 3, 4, 5)

# Répétition
print(t1 * 2)  # (1, 2, 3, 1, 2, 3)

# Vérification d'appartenance
print(2 in t1)  # True
```

3.6.2.5 Méthodes de tuple courantes

Les tuples disposent de peu de méthodes, car ils sont immuables. Les deux principales sont :

- `count()` : retourne le nombre d'occurrences d'un élément.
- `index()` : retourne l'indice de la première occurrence d'un élément.

```
nombres = (1, 2, 2, 3, 4)

print(nombres.count(2))  # 2
print(nombres.index(3))  # 3
```

3.6.2.6 Parcours d'un tuple

Comme les listes, les tuples peuvent être parcourus à l'aide d'une boucle `for` :

```
fruits = ("pomme", "banane", "cerise")

for fruit in fruits:
        print(fruit)
```

3.6.2.7 Déballage (Unpacking) des tuples

Le *déballage* permet d'assigner les éléments d'un tuple à plusieurs variables en une seule instruction :

```
coordonnees = (10, 20)
x, y = coordonnees
print("x =", x)  # Affiche : x = 10
print("y =", y)  # Affiche : y = 20
```

On peut également utiliser l'opérateur `*` pour capturer plusieurs valeurs :

```
donnees = (1, 2, 3, 4, 5)
a, b, *reste = donnees
print(a, b)      # 1 2
print(reste)     # [3, 4, 5]
```

3.6.2.8 Slicing des tuples

Le *slicing* fonctionne de la même manière que pour les listes, et permet d'extraire des sous-parties d'un tuple :

```python
nombres = (0, 1, 2, 3, 4, 5, 6)

# Extraire les éléments d'indice 2 à 5 (5 non inclus)
sous_tuple = nombres[2:5]    # (2, 3, 4)

# Extraire un élément sur deux
sous_tuple = nombres[::2]    # (0, 2, 4, 6)

# Inverser le tuple
inverse = nombres[::-1]      # (6, 5, 4, 3, 2, 1, 0)
```

Syntaxe	Signification
`tuple[start:stop]`	Sous-tuple de *start* à *stop-1*
`tuple[start:stop:step]`	Sous-tuple avec un pas de *step*
`tuple[:stop]`	Tous les éléments jusqu'à *stop-1*
`tuple[start:]`	Tous les éléments à partir de *start*
`tuple[::-1]`	Tous les éléments dans l'ordre inverse

TABLE 3.5 – Syntaxe du Slicing sur les tuples en Python

3.6.2.9 Quand utiliser un tuple plutôt qu'une liste ?

Les tuples sont préférables aux listes lorsque :

— Les données ne doivent pas être modifiées.

— Les éléments sont hétérogènes (ex. coordonnées, constantes, etc.).

— Les performances sont importantes (les tuples sont légèrement plus rapides).

— Le tuple doit être utilisé comme clé dans un dictionnaire (les tuples sont hachables, contrairement aux listes).

3.6.3 Les ensembles (sets)

Les *ensembles* (`set`) sont une structure de données en Python qui permet de stocker des éléments **non ordonnés** et **uniques**. Ils sont particulièrement utiles lorsque l'on souhaite effectuer des opérations mathématiques classiques sur des collections, comme l'union, l'intersection, et la différence.

Un ensemble ne conserve pas l'ordre des éléments et ne permet pas de stocker des doublons. Les ensembles sont **modifiables** (on peut ajouter ou retirer des éléments), mais ne sont pas indexables.

3.6.3.1 Création d'un set

```python
# Ensemble vide
ensemble_vide = set()

# Ensemble avec des éléments
fruits = {"pomme", "banane", "cerise"}

# Les doublons sont automatiquement supprimés
```

```
fruits = {"pomme", "banane", "pomme"}
print(fruits)  # Affiche : {'pomme', 'banane'}

# Attention : {} crée un dictionnaire vide, pas un set
vide = {}  # dictionnaire vide
```

3.6.3.2 Accès aux éléments

Les ensembles ne sont pas indexés, donc on ne peut pas accéder aux éléments via un indice. Pour parcourir un set, on utilise une boucle `for` :

```
fruits = {"pomme", "banane", "cerise"}

for fruit in fruits:
        print(fruit)
```

3.6.3.3 Ajout et suppression d'éléments

```
fruits = {"pomme", "banane"}

# Ajouter un élément
fruits.add("cerise")
print(fruits)  # {'pomme', 'banane', 'cerise'}

# Ajouter plusieurs éléments
fruits.update(["kiwi", "mangue"])
print(fruits)  # {'pomme', 'banane', 'cerise', 'kiwi', 'mangue'}

# Supprimer un élément (provoque une erreur si l'élément n'existe pas
    )
fruits.remove("banane")
print(fruits)  # {'pomme', 'cerise', 'kiwi', 'mangue'}

# Supprimer un élément sans erreur si inexistant
fruits.discard("orange")
```

3.6.3.4 Opérations sur les sets

Python propose plusieurs opérations classiques pour les ensembles :

- **Union** : tous les éléments présents dans l'un ou l'autre des sets
- **Intersection** : éléments communs aux deux sets
- **Différence** : éléments présents dans le premier set mais pas dans le second
- **Différence symétrique** : éléments présents dans un set ou l'autre, mais pas dans les deux

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

# Union
print(A | B)  # {1, 2, 3, 4, 5, 6}

# Intersection
```

```
print(A & B)  # {3, 4}

# Différence
print(A - B)  # {1, 2}

# Différence symétrique
print(A ^ B)  # {1, 2, 5, 6}
```

3.6.3.5 Méthodes courantes

Quelques méthodes utiles pour les sets :

- `len(set)` : nombre d'éléments
- `clear()` : vide le set
- `copy()` : copie du set
- `isdisjoint(other)` : vérifie si les sets n'ont aucun élément commun
- `issubset(other)` et `issuperset(other)` : tests de sous-ensemble et sur-ensemble

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}

print(len(A))          # 3
print(A.copy())        # {1, 2, 3}
print(A.isdisjoint({4, 5}))  # True
print(A.issubset(B))         # True
print(B.issuperset(A))       # True
```

3.6.3.6 Conversion entre types

On peut convertir une liste ou un tuple en set pour supprimer les doublons :

```
liste = [1, 2, 2, 3, 3, 3]
ensemble = set(liste)
print(ensemble)  # {1, 2, 3}

tuple_ex = (4, 4, 5, 6)
ensemble2 = set(tuple_ex)
print(ensemble2)  # {4, 5, 6}
```

3.6.3.7 Quand utiliser un set ?

Les sets sont particulièrement utiles lorsque :

- On souhaite garantir que les éléments soient uniques.
- On veut effectuer des opérations d'ensemble (union, intersection, différence) facilement.
- L'ordre des éléments n'a pas d'importance.
- On a besoin de vérifier rapidement l'appartenance d'un élément (`in`) : la recherche est en moyenne plus rapide que dans une liste.

3.6.4 Manipulation des chaînes de caractères

Les chaînes de caractères sont des séquences de caractères utilisées pour représenter du texte en Python. Ce chapitre explore les caractéristiques, les méthodes et les opérations associées aux chaînes de caractères.

Une chaîne de caractères en Python est créée en entourant des caractères par des guillemets simples ' ou doubles ". Les chaînes peuvent contenir des lettres, des chiffres, des symboles et des espaces.

3.6.4.1 Déclaration de chaînes

Voici quelques exemples de création de chaînes de caractères :

```
# Chaînes de caractères simples
chaine1 = "Bonjour"
chaine2 = 'Python est génial!'

# Chaîne multi-lignes
chaine_multi_lignes = """Ceci est une chaîne
multi-lignes."""
```

3.6.4.2 Accès aux caractères

Les chaînes de caractères sont indexées, ce qui signifie que chaque caractère a un indice, commençant à 0.

3.6.4.3 Accès par index

Voici comment accéder à un caractère spécifique dans une chaîne :

```
texte = "Python"

# Accéder au premier caractère
print(texte[0])  # Affiche : P

# Accéder au dernier caractère
print(texte[-1])  # Affiche : n
```

3.6.4.4 Slicing (slicing des chaînes)

Le slicing est une technique puissante en Python qui permet d'extraire une sous-chaîne d'une chaîne de caractères. Cette fonctionnalité repose sur le fait que les chaînes sont indexées, ce qui signifie que chaque caractère a une position spécifique.

La syntaxe de base pour le slicing est la suivante :

`chaine[start:end:step]`

où :

- `start` est l'indice à partir duquel commencer l'extraction (inclus).
- `end` est l'indice où l'extraction s'arrête (exclu).
- `step` est l'intervalle entre les indices (facultatif).

```python
texte = "Python est génial!"

# Extraire les caractères de l'indice 0 à 5 (non inclus)
sous_chaine1 = texte[0:6]
print(sous_chaine1)  # Affiche : Python
```

Les indices négatifs permettent d'accéder aux caractères en partant de la fin de la chaîne. Par exemple, `-1` représente le dernier caractère :

```python
# Extraire les 6 derniers caractères
sous_chaine2 = texte[-6:]
print(sous_chaine2)  # Affiche : génial!
```

Vous pouvez également spécifier un `step` pour sauter certains caractères :

```python
# Extraire tous les deux caractères
sous_chaine3 = texte[0:18:2]
print(sous_chaine3)  # Affiche : Pto  eia
```

3.6.4.5 Opérations sur les chaînes

Python propose diverses opérations sur les chaînes de caractères.

Concaténation Pour combiner deux chaînes, vous pouvez utiliser l'opérateur `+` :

```python
greeting = "Bonjour" + " " + "tout le monde!"
print(greeting)  # Affiche : Bonjour tout le monde!
```

Répétition Vous pouvez répéter une chaîne plusieurs fois en utilisant l'opérateur `*` :

```python
mot = "ha"
cris = mot * 3
print(cris)  # Affiche : hahaha
```

Longueur de la chaîne Pour obtenir la longueur d'une chaîne, utilisez la fonction `len()` :

```python
longueur = len(greeting)
print("Longueur de la chaîne :", longueur)  # Affiche : Longueur de
    la chaîne : 24
```

3.6.4.6 Méthodes de chaînes courantes

Python fournit plusieurs méthodes intégrées pour manipuler les chaînes de caractères.

Conversion en majuscules et minuscules Vous pouvez convertir une chaîne en majuscules ou en minuscules avec les méthodes `upper()` et `lower()` :

```python
texte = "Python"
print(texte.upper())  # Affiche : PYTHON
print(texte.lower())  # Affiche : python
```

Remplacement de sous-chaînes Pour remplacer une partie d'une chaîne, utilisez la méthode `replace()` :

```python
nouveau_texte = texte.replace("Python", "Java")
print(nouveau_texte)  # Affiche : Java
```

Vérification de contenu Les méthodes `startswith()` et `endswith()` permettent de vérifier si une chaîne commence ou se termine par une sous-chaîne spécifique :

```
print(chaine1.startswith("Bon"))  # Affiche : True
print(chaine1.endswith("jour"))    # Affiche : False
```

f-strings (Python 3.6+) Les f-strings (formatted string literals) offrent une syntaxe plus concise pour le formatage :

```
message_f = f"{nom} a {age} ans."
print(message_f)  # Affiche : Alice a 30 ans.
```

3.6.4.7 Immuabilité des chaînes

Les chaînes de caractères en Python sont immuables, ce qui signifie qu'une fois créées, elles ne peuvent pas être modifiées. Toute opération qui semble modifier une chaîne renvoie en réalité une nouvelle chaîne.

```
chaine = "Bonjour"
chaine[0] = "b"  # Cela lèvera une erreur
# A la place, vous pouvez créer une nouvelle chaîne :
nouvelle_chaine = "b" + chaine[1:]
print(nouvelle_chaine)  # Affiche : bonjour
```