

Secteur : **Digital & IA**

Manuel de cours

M102 : Acquérir les bases de l'algorithmique

1^{ère} Année

Filière :

Développement
Digital
(Tronc comun)



SOMMAIRE

01 - MODÉLISER UN PROBLÈME

Analyser un problème
Identifier les approches d'analyse d'un problème

02 - FORMULER UN TRAITEMENT

Reconnaitre la structure d'un algorithme
Connaitre les bases
Structurer un algorithme
Structurer les données

03 - PROGRAMMER EN PYTHON

Transformer une suite d'étapes algorithmiques
en une suite d'instructions Python
Manipuler les données

04 - DÉPLOYER LA SOLUTION PYTHON

Débloquer le code Python
Déployer une solution Python



MODALITÉS PÉDAGOGIQUES



WEBFORCE
BE THE CHANGE



1

LE GUIDE DE SOUTIEN

Il contient le résumé théorique et le manuel des travaux pratiques



2

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life



3

DES CONTENUS TÉLÉCHARGEABLES

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life



4

DU CONTENU INTERACTIF

Vous disposez de contenus interactifs sous forme d'exercices et de cours à utiliser sur WebForce Life



5

DES RESSOURCES EN LIGNES

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage



PARTIE 1

MODÉLISER UN PROBLÈME

Dans ce module, vous allez :

- Comprendre comment analyser un problème
- Différencier les approches d'analyse d'un problème



06 heures

CHAPITRE 1

ANALYSER UN PROBLÈME

Ce que vous allez apprendre dans ce chapitre :

- Acquérir une compréhension de la méthode d'analyser d'un problème
- Identifier le contexte et les entrées/sorties d'un problème
- Comprendre les différents types de traitement de données



03 heures



CHAPITRE 1

ANALYSER UN PROBLÈME

1. Définition du problème (contexte, entrées/sorties, traitements)

2. Types de traitement des données



01 – ANALYSER UN PROBLÈME

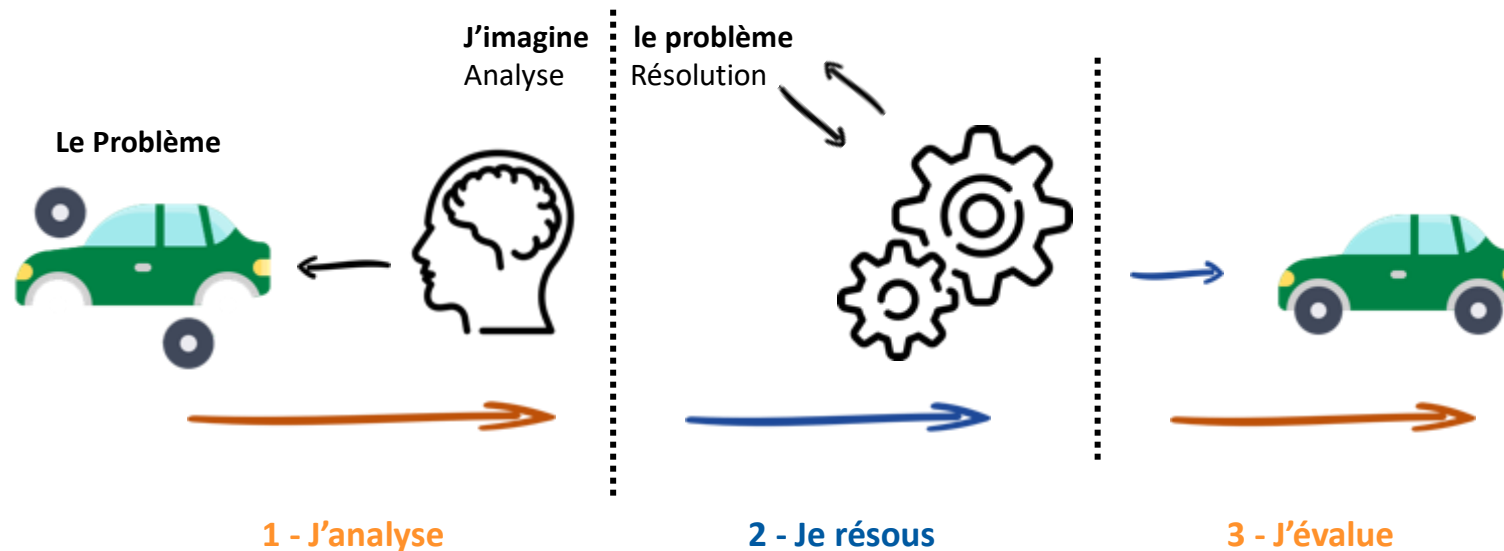
Définition du problème

Définition du problème

Le développement d'un logiciel est la transformation d'une idée ou d'un besoin (problème) en un logiciel fonctionnel.

Le processus de résolution d'un problème peut être décrit en 3 phases :

- Analyse du problème
- Résolution du problème (conception et réalisation de la solution)
- Évaluation de la solution



Processus de résolution d'un problème

01 – ANALYSER UN PROBLÈME

Définition du problème

Définition du problème

- Un problème est une difficulté qu'il faut résoudre afin d'obtenir un résultat souhaité

Exemple 1 :

- Un magasin d'électroménager possède en stock 380 aspirateurs.
- Le magasin fait l'acquisition de 40 autres appareils.
- La magasin a vendu 5 aspirateurs.

Quel est le stock actuel ?

- Un problème peut aussi se définir par un écart entre ce qui existe et ce qui devrait ou pourrait exister.



Entrée



Algorithme



Résultat

Entrée/résultat d'un algorithme

L'identification du problème est une étape cruciale dans le processus de modélisation et de résolution de celui-ci.

01 – ANALYSER UN PROBLÈME

Définition du problème



Définition du problème

L'analyse d'un problème est une étape préalable indispensable dans le processus de résolution de problème.

En effet, un problème bien analysé est un problème à moitié résolu.

L'analyse des problèmes s'intéresse aux éléments suivants :

- Les résultats souhaités (sorties)
- Les traitements (actions réalisées pour atteindre le résultat)
- Les données nécessaires aux traitements (entrées)

L'objectif de cette étape est de :

- Bien comprendre l'énoncé du problème
- Déterminer les dimensions du problème (entrées et sorties)
- Déterminer la méthode de sa résolution par décomposition et raffinements successifs
- Déterminer les formules de calculs, les règles de gestion, etc.

Le problème posé est souvent en langue naturelle et comporte plusieurs ambiguïtés d'où la nécessité de :

- Lecture entière et itérative pour comprendre et délimiter le problème à résoudre
- Reformulation du problème sous la forme d'une question ou bien en utilisant un modèle mathématique
- il est impératif de relire ensuite le sujet pour bien vérifier qu'il n'y manque rien d'important

01 – ANALYSER UN PROBLÈME

Définition du problème



Bien comprendre le problème à résoudre

Exemple 2 :

On se donne un ensemble d'entiers positifs, on souhaite calculer la moyenne ('K') de ces entiers



Quelle est la moyenne de 'K' entiers positifs ?

Conseils :

- Se consacrer entièrement à la compréhension du sujet
- Éviter de chercher des idées de résolution (le comment faire ?)
- Identifier les données d'entrée et les résultats attendus

01 – ANALYSER UN PROBLÈME

Définition du problème



Bien comprendre le problème à résoudre

Données d'entrée :

- Les données d'entrée représentent l'ensemble des données à partir desquelles on doit calculer puis afficher un résultat .
- Il s'agit du jeu de données que l'algorithme doit recevoir.



Entrée



Algorithme



Résultat

Entrée/résultat d'un algorithme

Conseils :

Décrire précisément et avoir bien en tête les valeurs qu'elles peuvent prendre.

01 – ANALYSER UN PROBLÈME

Définition du problème

Bien comprendre le problème à résoudre

Exemple 1 :

- Un magasin d'électroménager possède en stock 380 aspirateurs.
- Le magasin fait l'acquisition de 40 autres appareils.
- La magasin a vendu 5 aspirateurs.

Quel est le stock actuel ?



Entrées :

- Valeur du stock initial
- Quantité d'approvisionnement
- Quantités vendues

Exemple 2 :

On se donne un ensemble d'entiers positifs, on souhaite calculer la moyenne ('K') de ces entiers.



Entrées :

- Le nombre des entiers
- Les entiers positifs

01 – ANALYSER UN PROBLÈME

Définition du problème

Bien comprendre le problème à résoudre

Les résultats ou sorties :

Ils correspondent à ce que l'on demande de calculer ou de déterminer pour pouvoir obtenir le résultat.



Entrée



Algorithme



Résultat

Entrée/résultat d'un algorithme

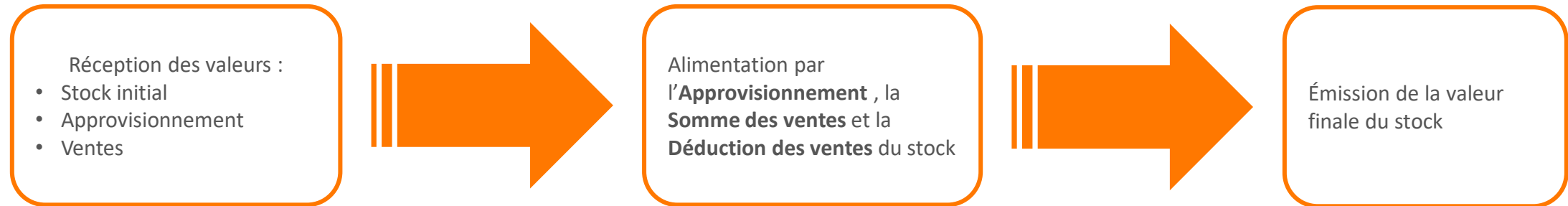
01 – ANALYSER UN PROBLÈME

Définition du problème

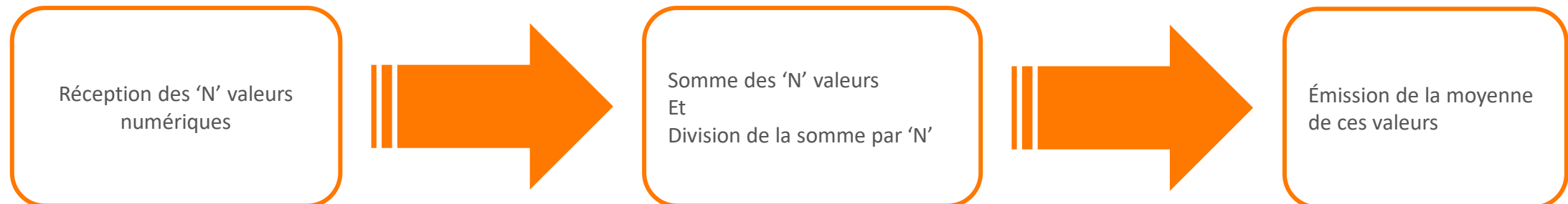


Bien comprendre le problème à résoudre

Exemple 1 :



Exemple 2 :



01 – ANALYSER UN PROBLÈME

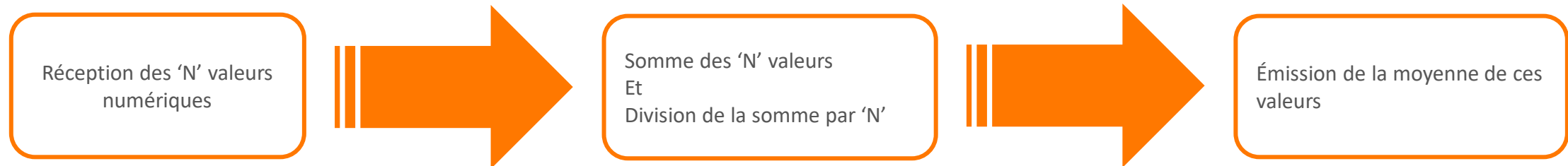
Définition du problème



Le traitement des données

- L'analyse d'un problème se base aussi sur la spécification de toutes les relations liant les résultats aux données et éventuellement les résultats entre eux
- La spécification des relations est la partie liée aux traitements à développer afin de résoudre le problème
- Le traitement est décrit à travers une suite finie et ordonnées de règles opératoires à suivre en vue de résoudre un problème

Exemple 2 :



Le traitement des données est donc la formulation d'une solution imaginée par :

- Analogie: recherche des ressemblances, des rapprochements à partir d'idées déjà trouvées pour un problème précédent plus ou moins similaire
- Contraste: recherche des différences, des oppositions, des arguments antagonistes
- Contigüité: recherche des faits se produisant en même temps, des parallélismes, des simultanités et autres concomitances

Il est nécessaire d'avoir du bon sens, d'adopter une démarche rigoureuse et d'utiliser des outils adaptés

CHAPITRE 1

ANALYSER UN PROBLÈME

1. Définition du problème (contexte, entrées/sorties, traitements)

2. Types de traitement des données



01 – ANALYSER UN PROBLÈME

Le traitement des données



Le traitement des données

- Tout traitement est effectué par l'exécution séquencée d'opérations appelées *instructions*
- Selon la nature du problème, un traitement est classé en 4 catégories

Traitement séquentiel

Traitement
conditionnel

Traitement itératif

Traitement récursif

01 – ANALYSER UN PROBLÈME

Le traitement des données



Le traitement séquentiel

Le traitement est décrit à travers l'enchaînement d'une suite d'actions primitives
La séquence des actions sera exécutée dans l'ordre

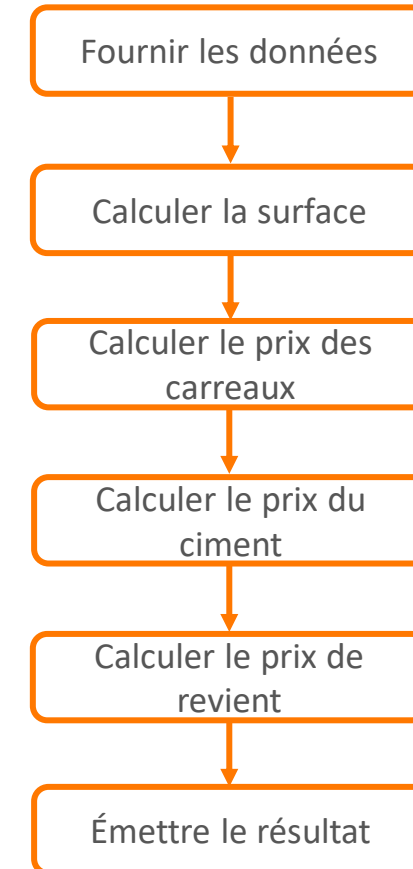
Traitement séquentiel

Traitement
conditionnel

Traitement itératif

Traitement récursif

Exemple :



Exemple d'enchaînement d'une suite d'actions primitives

01 – ANALYSER UN PROBLÈME

Le traitement des données



Le traitement conditionnel

Le traitement est utilisé pour résoudre des problèmes dont la solution ne peut être décrite par une simple séquence d'actions mais **implique un ou plusieurs choix entre différentes possibilités**

Traitement
séquentiel

**Traitement
conditionnel**

Traitement itératif

Traitement récursif

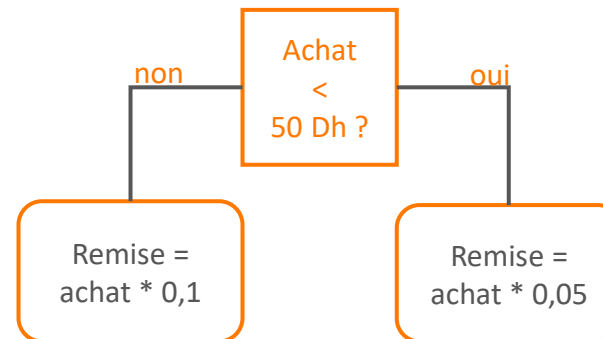
Exemple :

Un magasin accorde une remise sur les achats de ses clients.

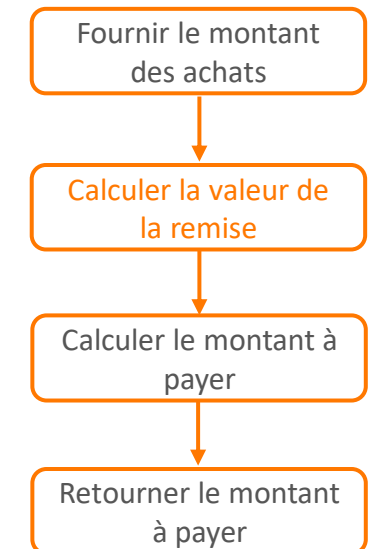
Le taux de la remise est de 5 % si le montant de l'achat est inférieur à 50 Dh

Le taux de la remise est de 10 % si le montant dépasse 50 Dh

Connaissant le montant d'achat d'un client, on souhaite déterminer la valeur de la remise et calculer le montant à payer



L'action « Calculer la valeur de la remise » aura un résultat différent selon la valeur de la donnée « montant de l'achat » = **Traitement conditionnel**



Traitement conditionnel du calcul de la remise

Exemple d'enchaînement d'une suite d'actions primitives

01 – ANALYSER UN PROBLÈME

Le traitement des données



Le traitement itératif

L'analyse d'un problème peut révéler le besoin de répéter un même traitement plus d'une fois. Recours à des outils permettant d'exécuter ce traitement un certain nombre de fois sans pour autant le réécrire autant de fois.

Traitement séquentiel

Traitement
conditionnel

Traitement itératif

Traitement récursif

Exemple :

Considérons le problème qui consiste à calculer la somme de 10 entiers positifs donnés.

Entrer un entier
Ajouter l'entier à la somme
Répéter 1 et 2 10 fois
Afficher le résultat

01 – ANALYSER UN PROBLÈME

Le traitement des données



Le traitement récursif

Un problème peut être exprimé en fonction d'un ou de plusieurs sous-problèmes tous de même nature que lui mais de complexité moindre

Traitement séquentiel

Traitement
conditionnel

Traitement itératif

Traitement récursif

Exemple :

Considérons le problème qui consiste à calculer la factorielle d'un entier N positif ou nul

On peut formuler le problème de cette manière :

Si $N > 0$

Factorielle (N) = $N * \text{Factorielle}$

Si $N = 0$

Factorielle (N) = 1

CHAPITRE 2

IDENTIFIER LES APPROCHES D'ANALYSE D'UN PROBLÈME

Ce que vous allez apprendre dans ce chapitre :

- Différencier les différentes approches d'analyse d'un problème
- Maîtriser chacune des approches d'analyse d'un problème



3 heures

CHAPITRE 2

IDENTIFIER LES APPROCHES D'ANALYSE D'UN PROBLÈME

1. Approche descendante

2. Approche ascendante

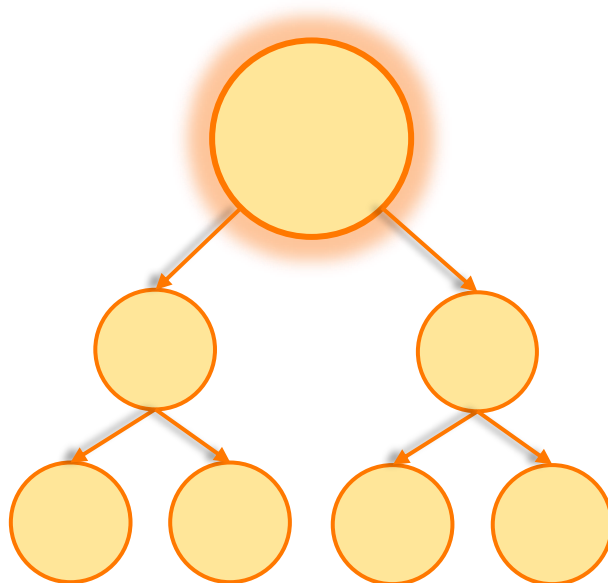


02 – IDENTIFIER LES APPROCHES D'ANALYSE D'UN PROBLÈME

Approche descendante

Approche descendante

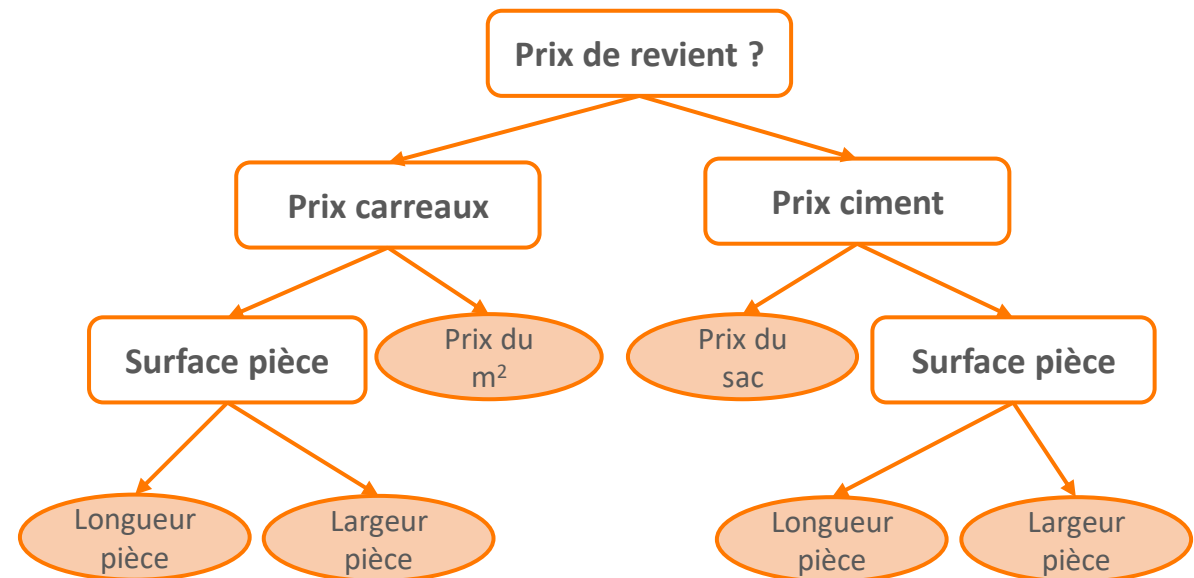
- L'approche descendante divise un problème complexe en plusieurs parties plus simples et plus petites (modules) pour organiser le traitement de manière efficace
- Ces modules sont ensuite décomposés jusqu'à ce que le module résultant constitue l'action primitive comprise et ne puisse plus être décomposée



Principe de l'approche descendante

Exemple :

Une pièce rectangulaire de 4 sur 3 mètres doit être carrelée. Le carrelage d'un m^2 nécessite 1 sac de ciment. On cherche le prix de revient du carrelage de cette pièce sachant que le prix des carreaux est de 58 Dh / m^2 et le prix d'un sac de ciment est de 75 Dh



Exemple d'application d'une approche descendante

CHAPITRE 2

IDENTIFIER LES APPROCHES D'ANALYSE D'UN PROBLÈME

1. Approche descendante

2. Approche ascendante



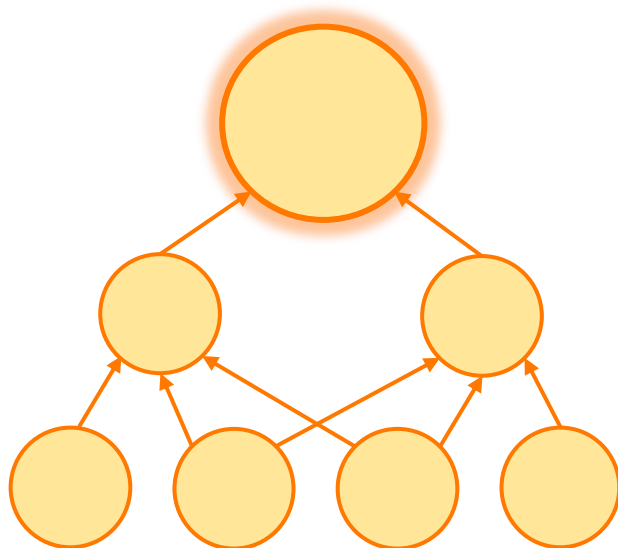
02 – IDENTIFIER LES APPROCHES D'ANALYSE D'UN PROBLÈME

Approche ascendante

Approche ascendante

L'approche ascendante fonctionne de manière inverse, les actions primitives étant d'abord conçues puis poursuivies au niveau supérieur

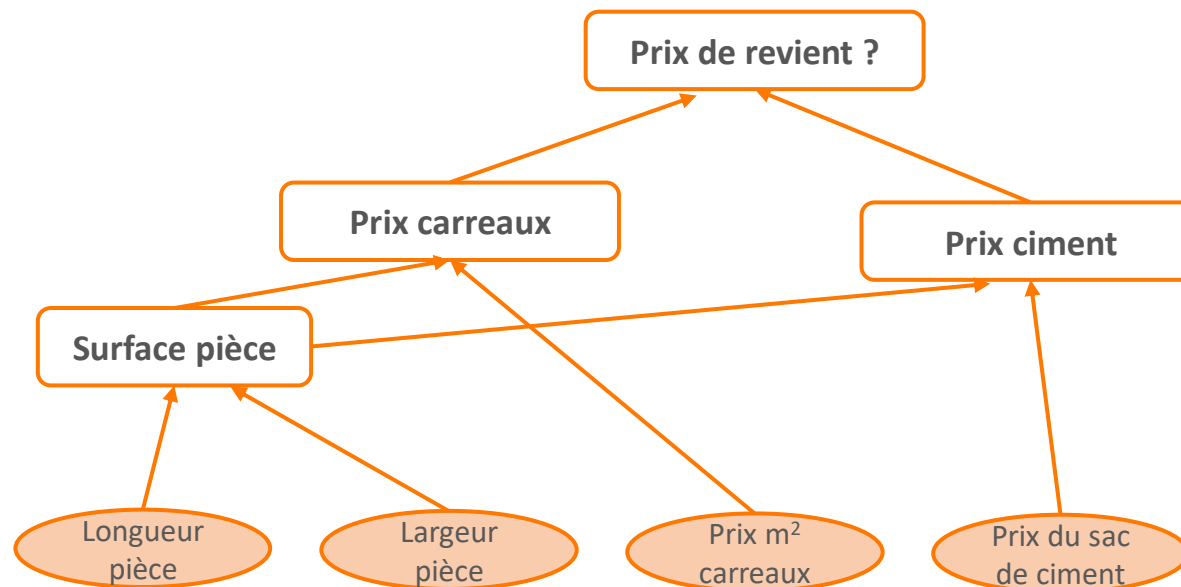
- Conception des pièces les plus fondamentales qui sont ensuite combinées pour former le module de niveau supérieur
- L'intégration de sous-modules et de modules dans le module du niveau supérieur est répétée jusqu'à l'obtention de la solution complète requise



Principe de l'approche ascendante

Exemple :

Une pièce rectangulaire de 4 sur 3 mètres doit être carrelée. Le carrelage d'un m^2 nécessite 1 sac de ciment. On cherche le prix de revient du carrelage de cette pièce sachant que le prix des carreaux est de 58 Dh / m^2 et le prix d'un sac de ciment est de 75 Dh.



Exemple d'application d'une approche ascendante



PARTIE 2

FORMULER UN TRAITEMENT

Dans ce module, vous allez :

- Maitriser la structure d'un algorithme
- Connaitre les différents types de traitement
- Maitriser la programmation structurée
- Manipuler les structures de données



48 heures

CHAPITRE 1

RECONNAITRE LA STRUCTURE D'UN ALGORITHME

Ce que vous allez apprendre dans ce chapitre :

- Comprendre la notion d'algorithme
- Connaître les différents types d'objets informatiques
- Différencier la notion de variable et de constante
- Maîtriser la structure d'un algorithme



7 heures



CHAPITRE 1

RECONNAITRE LA STRUCTURE D'UN ALGORITHME

1. Définition d'un algorithme

2. Objets informatiques (variable, constante, type)

3. Structure d'un algorithme



01 – STRUCTURE D'UN ALGORITHME

Définition d'un algorithme

Définition d'un algorithme

Un algorithme est une **suite d'instructions détaillées** qui, si elles sont **correctement exécutées**, conduit à un **résultat donné**.

"détaillées" signifie que les instructions sont suffisamment précises pour pouvoir être mises en œuvre correctement par l'exécutant (homme ou machine)

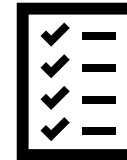
En algorithmique, nous utiliserons un langage situé à mi-chemin entre le langage courant et un langage de programmation appelé pseudo-code



Ingédients



Ustensiles & Récipients



Une recette



Le résultat

Exemple de fabrication de fromage

Explication : Lorsque nous cuisinons, nous utilisons des ingrédients et ustensiles, dans un ordre précis qui est régi par notre recette (une liste d'instruction dans un ordre donné), afin d'obtenir un résultat précis.

CHAPITRE 1

RECONNAITRE LA STRUCTURE D'UN ALGORITHME

1. Définition d'un algorithme

2. Objets informatiques (variable, constante, type)

3. Structure d'un algorithme



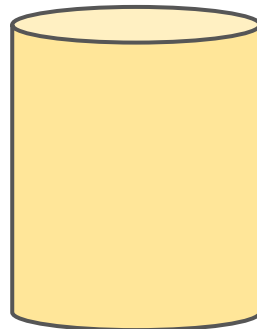
01 – STRUCTURE D'UN ALGORITHME

Objets informatiques (variable, constante, type)

Un algorithme manipule des objets (données) pour obtenir un résultat

Un objet est composé de :

- Un identificateur : Il s'agit du nom unique qui désigne cet objet.
- Un type : Il sert à déterminer la nature de l'objet qu'il soit simple (entier, caractère, etc.) ou composé (tableau,...), en particulier les valeurs possibles de l'objet, la taille mémoire réservée à l'objet et les opérations primitives applicables à l'objet.
- Une valeur : détermine le contenu unique de l'objet



Un objet



Un objet composé d'un identificateur



Un objet composé d'un identificateur + type



Un objet composé d'un identificateur + type + valeur

Composition d'un objet informatique

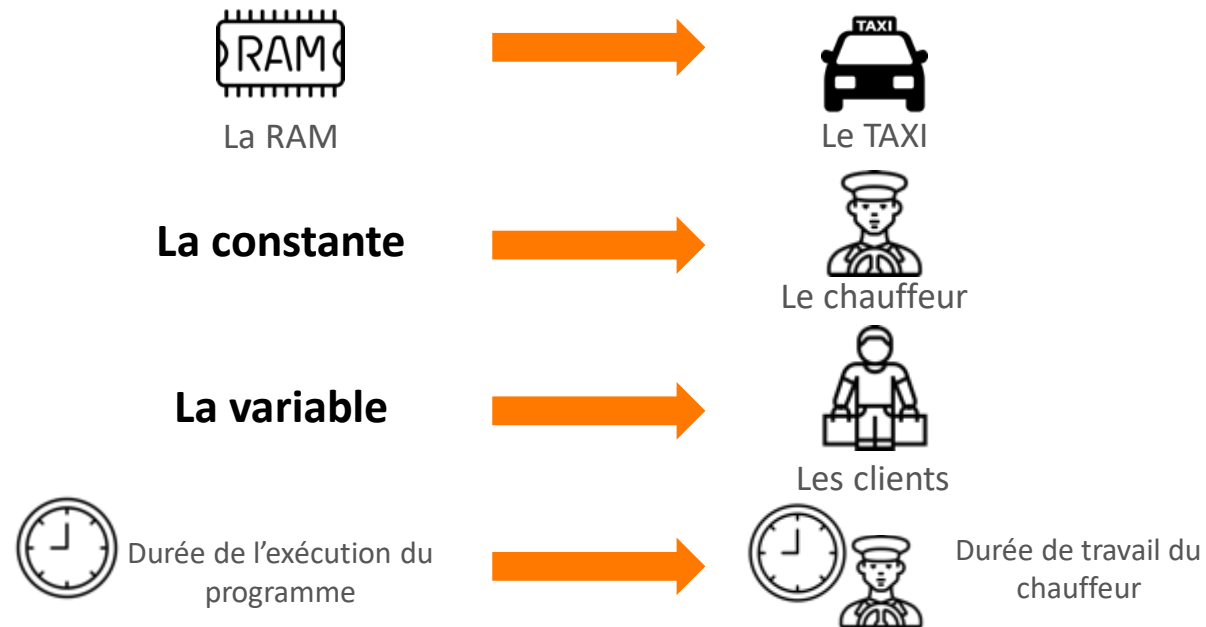
01 – STRUCTURE D'UN ALGORITHME

Objets informatiques (variable, constante, type)

Les objets sont de deux natures: les constantes et les variables

- Une **constante** est un objet dont l'état **reste inchangé** durant toute l'exécution d'un programme. On ne peut jamais modifier sa valeur et celle-ci doit donc être précisée lors de la définition de l'objet
- Une **variable** est un objet dont le contenu (sa valeur) **peut être modifié** par une action

Exemple:



Types des objets informatiques

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Types des objets

- À chaque variable utilisée dans le programme, il faut associer un type qui permet de définir :
 - L'ensemble des valeurs que peut prendre la variable
 - L'ensemble des opérations qu'on peut appliquer sur la variable

Les principaux types utilisés en algorithmique sont :

- Entier
- Réel
- Caractère
- Chaîne de caractères
- Logique ou booléen

Type entier

- Une variable est dite entière si elle prend ses valeurs dans \mathbb{Z} (ensemble des entiers relatifs)

Elle peut supporter les opérations suivantes :

Opération	Notation
Addition	+
Soustraction	-
Multiplication	*
Division entière	div
Modulo (reste de la division)	mod
Puissance	^

Exemples:

$$13 \text{ div } 5 = 2$$
$$13 \text{ mod } 5 = 3$$

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Type réel ou décimal

Il existe plusieurs types de réels représentant chacun un ensemble particulier de valeurs prises dans \mathbb{R} (ensemble des nombres réels)

Il existe deux formes de représentations des réels :

- La forme usuelle avec le point comme symbole décimal :

Exemples :

-3.2467 2 12.7 +36.49

- La notation scientifique selon le format aEb , où : a est la mantisse, qui s’écrit sous une forme usuelle, b est l’exposant représentant un entier relatif :

Exemples :

347 = 3.47E2 = 0.347E+3 = 3470E-1

Les opérations définies sur les réels sont :

Opération	Notation
Addition	+
Soustraction	-
Multiplication	*
Division (réelle)	/
Puissance	^

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Type caractère

- Un caractère peut appartenir au domaine des chiffres de "0" à "9", des lettres (minuscules et majuscules) et des caractères spéciaux ("*", "/", "{", "\$", "#", "%" ...)
- Un caractère sera toujours noté entre des guillemets.
- Le caractère espace (blanc) sera noté " "
- Les opérateurs définis sur les données de type caractère sont :

Opération	Notation
Égal	=
Différent	#
Inférieur	<
Inférieur ou égal	<=
Supérieur	>
Supérieur ou égal	>=

- La comparaison entre les caractères se fait selon leur codes ASCII : Le code ASCII est une norme informatique de codage de caractères, dans laquelle chaque caractère alphabétique, numérique ou spécial est représenté par un nombre binaire sur 7 bits (une chaîne composée de sept 0 ou 1).

Exemple :

" " < "0" < "1" < "A" < "B" < "a" < "b" < "{"

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Type logique ou booléen

- Une variable logique ne peut prendre que les valeurs "Vrai" ou "Faux"
- Elle intervient dans l'évaluation d'une condition
- Les principales opérations définies sur les variables de type logique sont : la négation (NON), l'intersection (ET) et l'union (OU)
- L'application de ces opérateurs se fait conformément à la table de vérité suivante :

A	B	NON (A)	A et B	A ou B
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

Table de vérité des opérateurs logiques

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Expressions

- Ce sont des combinaisons entre des variables et des constantes à l'aide d'opérateurs
- Elles expriment un calcul (expressions arithmétiques) ou une relation (expressions logiques)

Les expressions arithmétiques:

Ce sont des expressions simples construites avec des opérateurs arithmétiques et des constantes ou des références à des cellules.

Exemple: $x * 53.4 / (2 + \text{Pi})$

- L'ordre selon lequel se déroule chaque opération de calcul est important
- Afin d'éviter les ambiguïtés dans l'écriture, on se sert des parenthèses et des relations de priorité entre les opérateurs arithmétiques :

Priorité	Opérateurs
1	- Signe négatif (opérateur unaire)
2	() Parenthèses
3	^ Puissance
4	* Et / Multiplication et division
5	+ et – addition et soustraction

En cas de conflit entre deux opérateurs de même priorité, on commence par celui situé le plus à gauche

Ordre de priorité des opérateurs arithmétiques

01 – STRUCTURE D’UN ALGORITHME

Objets informatiques (variable, constante, type)



Expressions

Les expressions logiques:

- Ce sont des expressions de type booléen, c'est à dire des expressions pouvant prendre la valeur vrai ou faux.
- Elles sont formées à partir des combinaisons entre des variables et des constantes à l'aide des opérateurs relationnels et logiques.
- Les opérateurs relationnels sont (=, <, <=, >, >=, #) et les opérateurs logiques sont (NON, ET, OU, etc.)
- On utilise les parenthèses et l'ordre de priorité entre les différents opérateurs pour résoudre les problèmes de conflits

Priorité	Opérateur
1	NON
2	ET
3	OU

Opérateurs logiques

Priorité	Opérateur
1	la parenthèse () la plus interne
2	>
3	>=
4	<
5	<=
6	=
7	#

Opérateurs relationnels

Exemple :

$$5 + 2 * 6 - 4 + (8 + 2 ^ 3) / (2 - 4 + 5 * 2) = 15$$

On commence par calculer ce qui est entre les parenthèses : $(8 + 2 ^ 3) = 16$ et $(2 - 4 + 5 * 2) = 2$

L'expression deviant : $5 + 2 * 6 - 4 + 8 = 15$.

01 – STRUCTURE D'UN ALGORITHME

Objets informatiques (variable, constante, type)



Déclaration d'une variable

- Toute variable utilisée dans un programme doit être l'objet d'une déclaration préalable
- En pseudo-code, la déclaration de variables est effectuée par la forme suivante :

Var liste d'identificateurs : type

Exemple:

Var
i, j, k : Entier x, y : Réel
OK: Booléen
C1, C2 : Caractère

Déclaration d'une constante

- En pseudo-code, la déclaration des constantes est effectuée par la forme suivante :
- Par convention, les noms de constantes sont en majuscules
- Une constante doit toujours recevoir une valeur dès sa déclaration

Const identificateur=valeur : type

Exemple: Const PI=3.14 : réel

Pour calculer la surface des cercles, la valeur de Pi est une constante mais le rayon est une variable

CHAPITRE 1

RECONNAITRE LA STRUCTURE D'UN ALGORITHME

1. Définition d'un algorithme
2. Objets informatiques (variable, constante, type)

3. Structure d'un algorithme



01 – STRUCTURE D'UN ALGORITHME

Structure d'un algorithme



Structure d'un algorithme

<NOM_ALGORITHME>

Const

Const1 = val1 : type

Const2 = val2 : type

.....

Var

v1 : type

v2 : type

.....

Début

Instruction 1

Instruction 2

.....;

Fin

Liste des constantes

Liste des variables

Corps de l'algorithme

Cercle

Const

Pi = 3.14

Var

r, p, s : Réel

Début

Écrire ("Entrer le rayon du cercle : ")

Lire (r)

$p := 2 * \text{Pi} * r$

$s := \text{Pi} * r^2$

Écrire ("Périmètre = ", p)

Écrire ("Surface = ", s)

Fin

CHAPITRE 2

RECONNAITRE LES BASES

Ce que vous allez apprendre dans ce chapitre :

- Maîtriser les instructions d'affectation et les instructions d'entrée/sortie
- Reconnaître les différents types de traitement des instructions dans un algorithme



15 heures



CHAPITRE 2

RECONNAITRE LES BASES

1. Traitement séquentiel (affectation, lecture et écriture)

2. Traitement alternatif (conditions)

3. Traitement itératif (boucles)



02 – RECONNAITRE LES BASES

Traitement séquentiel (affectation, lecture et écriture)



Instruction d'affectation

- L'affectation consiste à attribuer une valeur à une variable (c'est-à-dire remplir ou modifier le contenu d'une zone mémoire)
- En pseudo-code, l'affectation est notée par le signe :=

Var1:= e attribue la valeur de e à la variable Var1

- **e** peut être une valeur, une autre variable ou une expression
- **Var1 et e** doivent être de même type ou de types compatibles . Le type est dit compatible s'il est inclus dans le type de la variable réceptrice.

Exemple : REEL (reçoit) ENTIER est possible mais pas l'inverse.

- L'affectation ne modifie que ce qui est à gauche de la flèche

Exemple :

- L'instruction : $A := 6$ signifie « mettre la valeur 6 dans la case mémoire identifiée par A »
- L'instruction : $B := (A + 4) \text{ Mod } 3$ range dans B la valeur 1 ($A = 6$ d'après le premier exemple. Donc l'expression $(A+4) \text{ Mod } 3 = 10 \text{ Mod } 3$ qui est égal à 1)

02 – RECONNAITRE LES BASES

Traitement séquentiel (affectation, lecture et écriture)



Instruction de lecture

- Les instructions de lecture et d'écriture (entrée/sortie) permettent à la machine de communiquer avec l'utilisateur
- La lecture permet d'entrer des données à partir du clavier
- En pseudo-code, on note :

lire (var1)

- La machine met la valeur entrée au clavier dans la zone mémoire nommée **var1**
- Le programme s'arrête lorsqu'il rencontre une instruction Lire et ne se poursuit qu'après la frappe d'une valeur au clavier et de la touche **Entrée**.

Exemple:

Soit A une variable de type entier :

Lire(A).

L'utilisateur entre la valeur 5 sur son clavier puis clique sur entrée.

Le système Affecte la valeur 5 à la variable A.

Instruction d'écriture

- L'écriture permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
- En pseudo-code, on note :

Écrire (paramètre1,..)

- Paramètre1 peut être une variable, une expression ou une constante.
la constante peut être un nombre ou un message.
- La machine affiche le contenu de la zone mémoire **paramètre1**

Exemple:

Écrire (" La valeur de 3*2 est égale à ", 3*2).

Il s'agit d'un message + une expression. Le système va afficher le résultat suivant:

La valeur de 3*2 est égale à 6

CHAPITRE 2

RECONNAITRE LES BASES

1. Traitement séquentiel (affectation, lecture et écriture)

2. Traitement alternatif (conditions)

3. Traitement itératif (boucles)



02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)



Traitement alternatif (conditions)

Rappel:

- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée.

Ces structures sont utilisées pour décider de l'exécution d'un bloc d'instruction : est-ce que ce bloc est exécuté ou non. Ou bien pour choisir entre l'exécution de deux ou plusieurs blocs différents.

Il existe différents types de traitements alternatifs :

1- Forme simple: **SI_ALORS**

2- Forme Alternative : **SI_ALORS_SINON**

3- Schéma conditionnel à choix multiple : **SELONQUE**

02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)

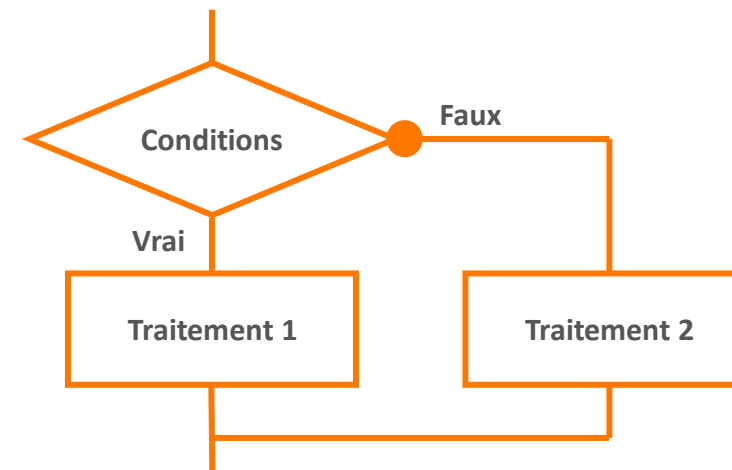


Forme simple SI_ALORS

Syntaxe: forme simple

```
Si <Condition> Alors  
    <Séquence d'instructions>  
FinSi
```

- Cette primitive a pour effet d'exécuter la séquence d'instructions si et seulement si la condition est vérifiée
- L'exécution de cette instruction se déroule selon l'organigramme suivant :



Organigramme de l'exécution de la structure Si..Alors..FinSi

02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)

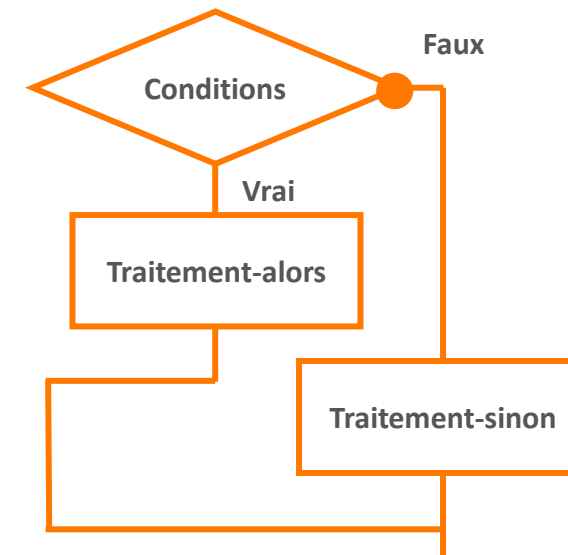


Forme Alternative: SI_ALORS_SINON

Syntaxe : forme alternative

```
Si <Condition> Alors
    <Séquence d'instructions 1>
Sinon
    <Séquence d'instructions 2>
FinSi
```

- Cette primitive a pour effet d'exécuter la première séquence d'instructions si la condition est vérifiée ou bien la deuxième séquence d'instructions dans le cas contraire
- L'exécution de cette instruction se déroule selon l'organigramme suivant :



Organigramme de l'exécution de la structure Si..Alors... Sinon ..FinSi

02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)



Forme Alternative: SI_ALORS_SINON

Exemple 1 :

```
Si ( a ≠ 0 ) Alors  
  a := 0  
Sinon  
  a := b  
  c := d  
FinSi
```

- Si la condition est vraie c'est à dire la variable a est différente de 0 alors on lui affecte la valeur 0, sinon on exécute le bloc.

Exemple 2 :

```
Si ( a - b ≠ c ) Alors  
  a := c  
Sinon  
  a := d  
FinSi
```

- Si la condition est vraie, la seule instruction qui sera exécutée est l'instruction d'affectation **a := c**.
- Sinon la seule instruction qui sera exécutée est l'instruction d'affectation **a := d**.

02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)



Schéma conditionnel à choix multiple

Syntaxe : Schéma conditionnel à choix multiple

Cas <var> de:

<valeur 1> : <action 1>

< valeur 2> : <action 2>

...

< valeur n> : <action n>

Sinon : <action_sinon>

FinCas

la partie **action-sinon** est facultative

Exemple:

On dispose d'un ensemble de tâches que l'on souhaite exécuter en fonction de la valeur d'une variable de choix de type entier, conformément au tableau suivant :

Valeur de choix	Tâche à exécuter
1	Commande
2	Livraison
3	Facturation
4	Règlement
5	Stock
Autre valeur	ERREUR

02 – RECONNAITRE LES BASES

Traitement alternatif (conditions)



Soient les blocs d'instructions : Commande, Livraison, Facturation et Règlement. L'exemple suivant illustre la différence entre les formes alternative et le schéma conditionnel à choix multiples.

Forme alternative

```
Si choix = 1 Alors  
  Commande  
Sinon  
  si choix = 2 Alors  
    Livraison  
  Sinon  
    Si choix = 3 Alors  
      Facturation  
    Sinon  
      si choix = 4 Alors  
        Règlement  
      Sinon  
        si choix = 5 Alors  
          Stock  
        Sinon  
          Ecrire ("Erreur")  
        FinSi  
      FinSi  
    FinSi  
  FinSi  
FinSi
```

Schéma conditionnel à choix multiple

Cas choix de :

- 1: Commande
- 2: Livraison
- 3: Facturation
- 4: Règlement

sinon Ecrire ("Erreur")

FinCas

CHAPITRE 2

RECONNAITRE LES BASES

1. Traitement séquentiel (affectation, lecture et écriture)
2. Traitement alternatif (conditions)
- 3. Traitement itératif (boucles)**



02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Traitement itératif (boucles)

- Une boucle permet de parcourir une partie d'un programme un certain nombre de fois. Une itération est la répétition d'un même traitement plusieurs fois. Un indice de boucle varie alors de la valeur minimum (initiale) jusqu'à la valeur maximum (finale).

Il existe trois types de structures itératives:

La structure « POUR FAIRE »

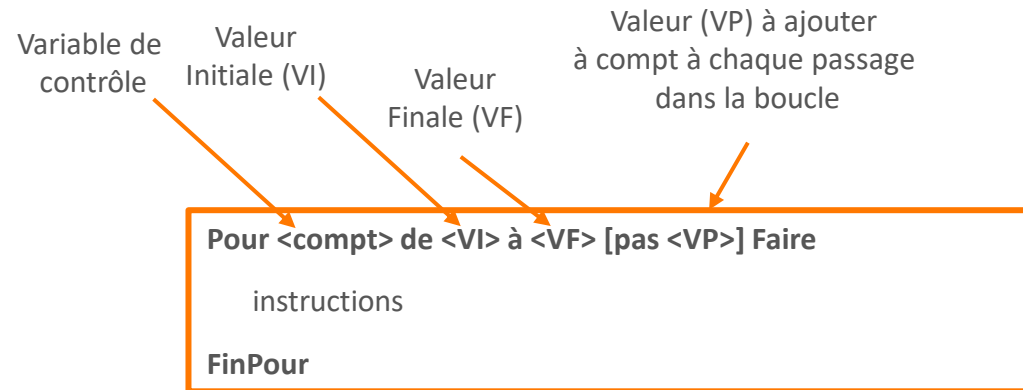
La structure « TANT QUE FAIRE »

La structure « REPETER ... JUSQUA »

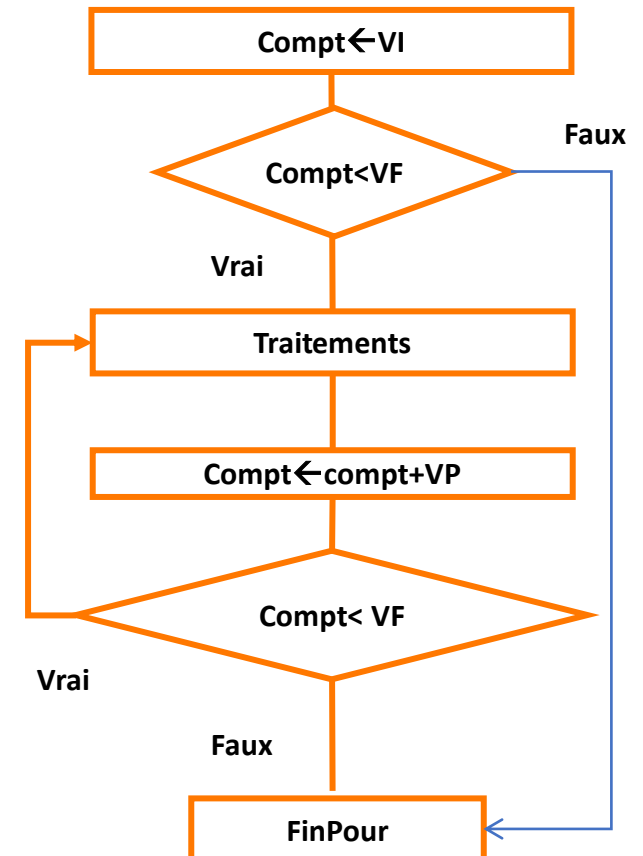
02 – RECONNAITRE LES BASES

Traitement itératif (boucles)

Structure « PourFaire »



- Cette structure consiste à répéter un certain traitement un nombre de fois fixé à l'avance en utilisant une variable de contrôle (compteur) d'itérations caractérisée par : sa valeur initiale, sa valeur finale et son pas de variation.
- Le compteur prend la valeur initiale au moment d'accès à la boucle puis, à chaque parcours, il passe automatiquement à la valeur suivante dans son domaine selon le pas jusqu'à atteindre la valeur finale
- L'exécution de cette instruction se déroule selon l'organigramme suivant :



Organigramme de l'exécution de la structure Pour..faire

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Structure « Pour.....Faire »

Exemple:

un algorithme permettant de lire N réels, de les calculer et d'afficher leur moyenne

moyenne

var n, i, x, s : réel

Début

lire(n)

Si n=0

Ecrire ("Impossible de calculer")

Fin

Sinon

s := 0

Pour i de 1 **à** n **Faire**

lire(x)

s := s + x

FinPour

Ecrire("la moyenne est :", s / n)

FinSi

Fin

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



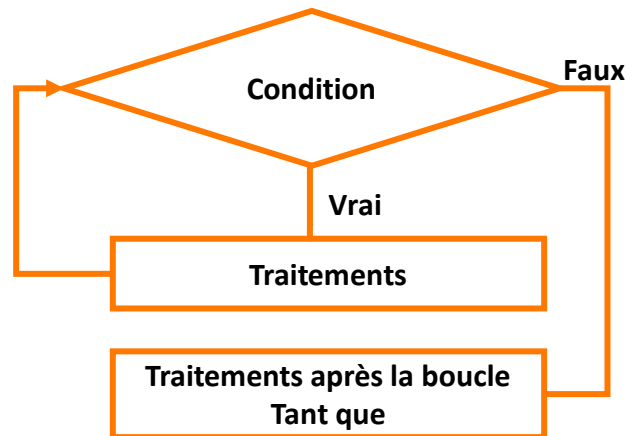
Structure « TantQueFaire »

- Le traitement est exécuté aussi longtemps que la condition est vérifiée. Si dès le début cette condition est fausse, le traitement ne sera exécuté aucune fois.

Une boucle « TantQue » peut s'exécuter 0, 1 ou n fois.

```
TantQue <Condition> Faire  
    <Séquence d'instructions>  
FinTQ
```

- L'exécution de cette instruction se déroule selon l'organigramme suivant :



Organigramme de l'exécution de la structure TantQue..faire

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Structure « TantQueFaire »

Exemple:

Reprenons le même exemple de l'algorithme de calcul de moyenne d'un nombre N de réels en utilisant la boucle TANQUE.

```
moyenne
var i, x, s : réel

Début
  lire( x )
  s := 0
  i := 0
  TantQue i <= N faire
    i := i + 1
    s := s + x
    lire( x )
  FinTQ
  si i ≠ 0
    Alors écrire( "la moyenne est :", s / i )
    Sinon
      écrire( "Impossible de calculer",)
  Finsi
Fin
```

Condition obligatoire pour éviter de diviser par 0 si N=0

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Structure « Répéter..... Jusqu' à »

La séquence d'instructions est exécutée une première fois, puis l'exécution se répète jusqu'à ce que la condition de sortie soit vérifiée.

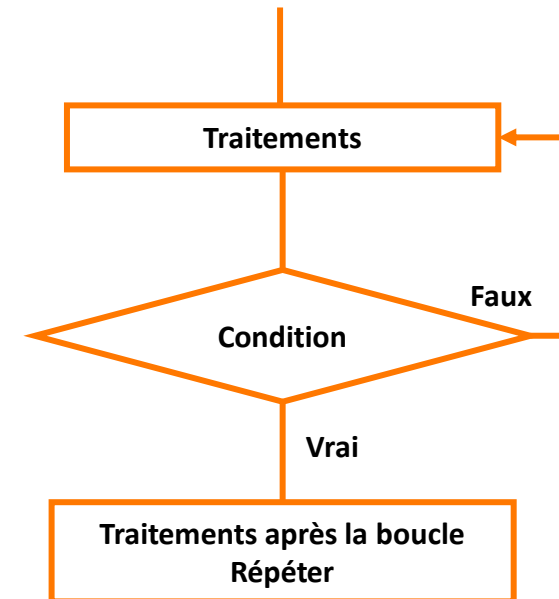
Une boucle « répéter » **s'exécute toujours au moins une fois.**

Répéter

<Séquence d'instructions>

Jusqu'à <condition>

L'exécution de cette instruction se déroule selon l'organigramme suivant :



Organigramme de l'exécution de la structure Répéter..jusqu'à

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Structure « Répéter.....Jusqu'à »

Exemple :

un algorithme permettant de lire deux entiers, de calculer et d'afficher le résultat de la division du premier par le second (quotient). L'algorithme doit s'assurer que le deuxième entier entré soit différent de 0. On utilise donc une boucle « Répéter.. Jusqu'à » pour vérifier la valeur de Y avant de passer au calcul. La condition d'arrêt de cette boucle sera $Y > 0$.

```
quotient
var x, y : entier

Début

    lire( x )
    Répéter
        lire( y )
    jusqu'à y > 0
    écrire( x / y )

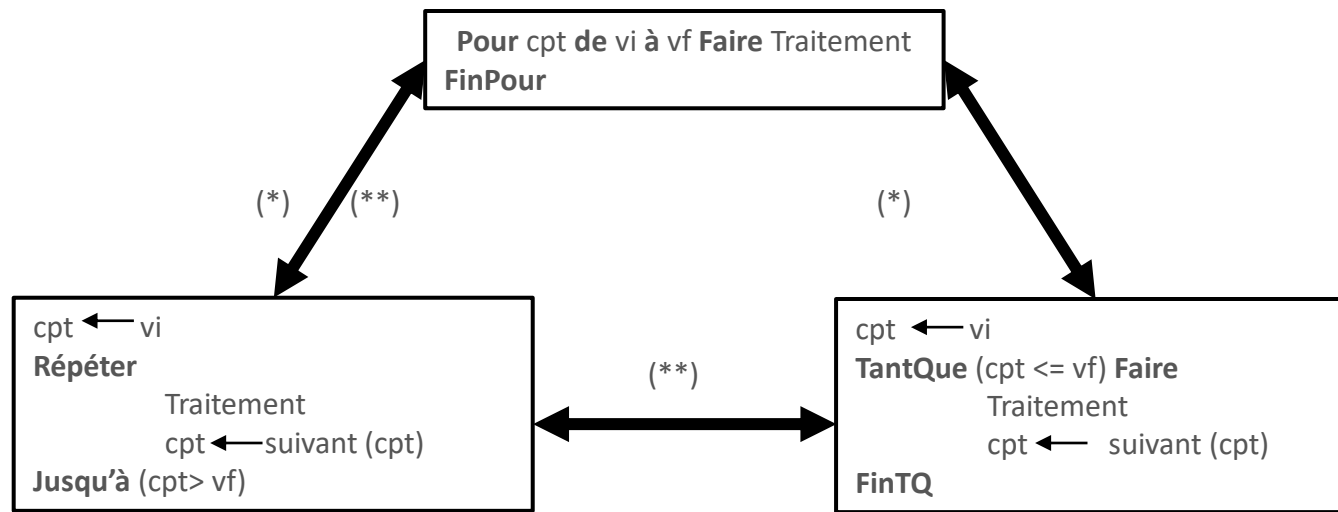
Fin
```

Un contrôle obligatoire doit être effectué lors de la lecture de la deuxième valeur.

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)

Passage d'une structure itérative à une autre



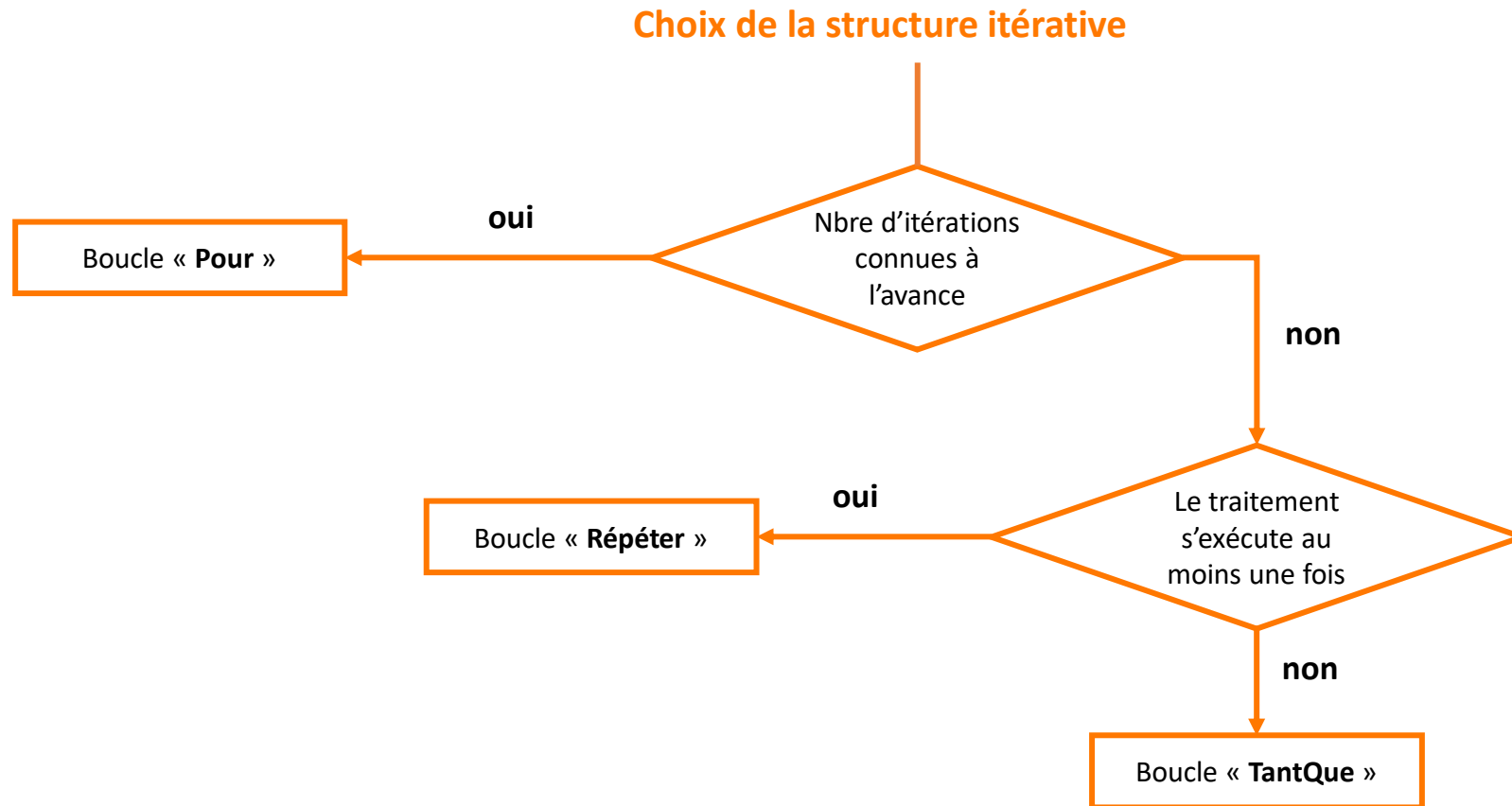
Passage d'une structure itérative à une autre

(*): Le passage d'une boucle « Répéter » ou « TantQue » à une boucle « Pour » n'est possible que si le nombre de parcours est connu à l'avance

()**: Lors du passage d'une boucle « Pour » ou « TantQue » à une boucle « Répéter », faire attention aux cas particuliers (le traitement sera toujours exécuté au moins une fois)

02 – RECONNAITRE LES BASES

Traitement itératif (boucles)



Organigramme de Choix de la structure itérative appropriée

CHAPITRE 3

STRUCTURER UN ALGORITHME

Ce que vous allez apprendre dans ce chapitre :

- Maîtriser la définition des procédures et des fonctions
- Maîtriser les notions de paramètres formels et paramètres effectifs
- Définir les différents types de passage des paramètres
- Connaître la notion de variable locale et de variable globale



10 heures



CHAPITRE 3

STRUCTURER UN ALGORITHME

1. Procédures et Fonctions

2. Portée d'une variable



03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions

- La résolution d'un problème complexe peut engendrer des milliers de lignes de code :

Algorithme long ;

Algorithme difficile à écrire ;

Algorithme difficile à interpréter ;

Algorithme difficile à maintenir.

- Solution : utiliser une méthodologie de résolution .**

Programmation Structurée : Il s'agit d'une méthodologie de résolution qui se base sur le découpage d'un problème en des sous-problèmes moins complexes.

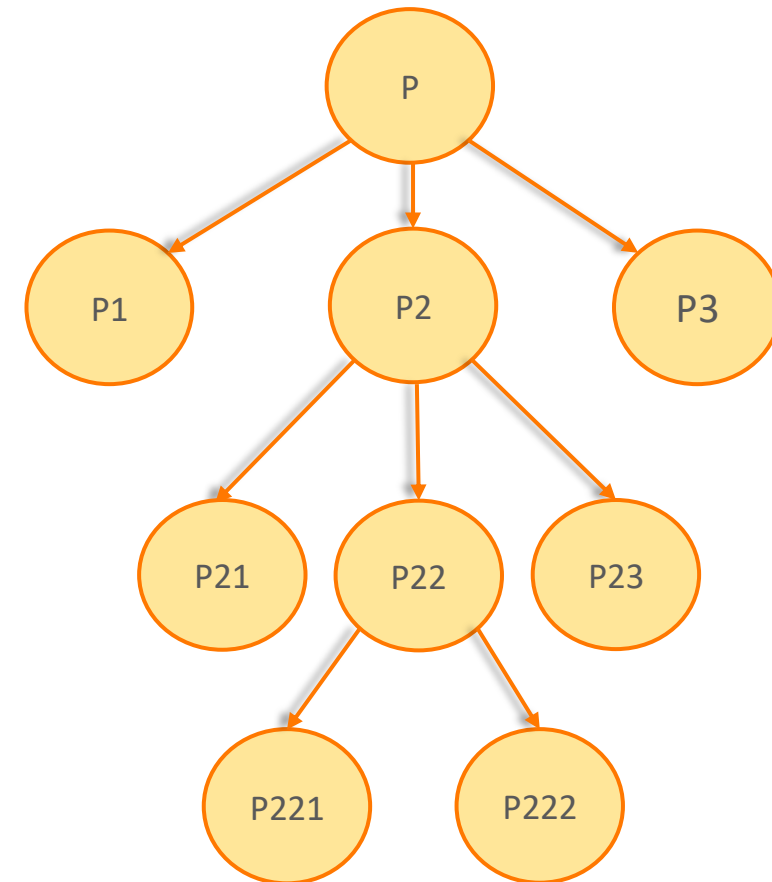
- Avantages :**

Clarté de l'algorithme ;

Lisibilité de la lecture d'un algorithme ;

Facilité de maintenance ;

Réutilisation des sous-algorithmes.



Exemple de décomposition d'un problème P en sous-problèmes

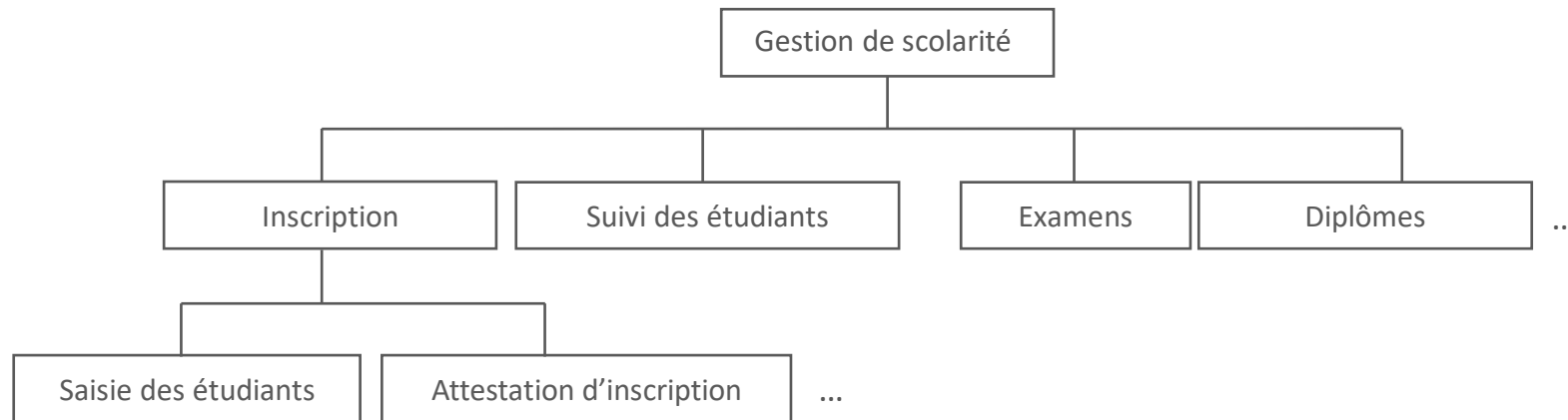
03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Programmation structurée

Exemple: un programme de gestion de scolarité peut être découpé en plusieurs modules : inscription, suivi des absences, examens, diplômes, etc.



Décomposition du problème de gestion de scolarité

Les modules développés peuvent être réutilisés plusieurs fois dans le même programme ou dans d'autres programmes une fois intégrés à des bibliothèques. une bibliothèque étant un ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

- Chaque module développé est un sous-programme permettant de résoudre un sous-problème
- Un sous-programme peut être une procédure ou une fonction.

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Programmation structurée

- Deux types de sous-algorithmes/sous-programmes sont possibles :
 - Procédures
 - Fonctions
- Une procédure/ une fonction est un bloc d'instructions regroupés sous un nom permettant de réaliser des actions particulières
 - Une procédure **renvoie plusieurs valeurs** ou **aucune valeur**
 - Une fonction **renvoie une seule valeur**
- Un sous-algorithme (procédure ou fonction) prend des données par l'intermédiaire de ses **paramètres**
- La communication entre les sous- algorithmes (procédure ou fonction) se fait via des **paramètres** suite à un appel :



- Il existe 3 types de paramètres :
 - Paramètres données : **les entrées**
 - Paramètres résultats : **les sorties**
 - Paramètres données/résultats : **des paramètres que l'action paramétrée utilise, modifie et retourne en sortie.**

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Paramètre formel/paramètre effectif

- **Paramètres formels** : objets utilisés pour la description d'un sous-algorithme
- **Paramètres effectifs** : objets utilisés lors de l'appel d'un sous-algorithme
- Un **paramètre formel** est toujours une **variable**
- Un **paramètre effectif** peut être :
 - Une **variable**
 - Une **constante**
 - Une **expression arithmétique**
 - Un **appel de fonction**
- Pour tout paramètre formel on fait correspondre un paramètre effectif



- Le paramètre formel et le paramètre effectif correspondant doivent avoir le même type ou être de types compatibles
- La correspondance entre paramètres formels et paramètres effectifs se fait selon l'ordre de leurs apparitions dans la définition et dans l'utilisation de la procédure ou la fonction

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Syntaxe de définition d'une procédure

Pour définir une procédure, on adoptera la syntaxe suivante :

```
<Nom_proc> (<liste_par_form>)
```

```
Var <declaration_variables>
```

```
Debut
```

```
    <Corps_procedure>
```

```
Fin
```

Nom_proc : désigne le nom de la procédure.

<liste_par_form> : la liste des paramètres formels. Un paramètre résultat ou donnée/résultat doit être précédé par le mot clé var.

<declaration_variables> : la liste des variables.

<Corps_procedure> : la suite des instructions décrivant le traitement à effectuer.

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Syntaxe de définition d'une procédure

Exemple 1:

- La procédure suivante permet de lire N valeurs entières et de calculer la plus petite et la plus grande parmi ces N valeurs
- Les entiers saisis doivent être supérieurs à 0 et inférieurs à 100
 - Le nom de cette procédure est **Min_Max**
 - Les paramètres formels sont : l'entier N comme paramètre donné, les entiers min et max comme paramètres résultats (précédés par le mot clé var)
 - 2 variables locales de type entier : **i** et **x**

Min_Max (N : entier ; var min: entier, var max : entier)

Var i, x : entier

Début

min := 100

max := 0

pour i de 1 à N faire

Répéter

Lire (x)

Jusqu'à (x > 0) et (x < 100)

Si x < min **Alors**

min := x

FinSi

Si x > max **Alors**

max := x

FinSi

FinPour

Fin

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Syntaxe de définition d'une fonction

Pour définir une fonction, on adoptera la syntaxe suivante :

Nom_fonction<liste_par_form> : <Type-fonction>

Var <declarat_var_locales>

Début

<Corps_fonction>

retourner <valeur>

Fin

Nom_fonction : désigne le nom de la fonction.

<liste_par_form> : désigne la liste des paramètres formels de la fonction.

<declarat_var_locales> : définissent les mêmes concepts que pour la procédure.

<Type-fonction> : est le type de la valeur retournée par la fonction.

<Corps_fonction> : en plus des instructions décrivant le traitement à effectuer, une instruction d'affectation du résultat que devrait porter la fonction au nom de la fonction elle-même.

retourner <valeur> : est la valeur retournée par la fonction.

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Syntaxe de définition d'une fonction

Exemple :

- La fonction suivante permet de lire N valeurs entières et de retourner la plus petite parmi ces valeurs.
- Les entiers saisis doivent être supérieurs à 0 et inférieurs à 100
 - Le nom de cette fonction est **Min**
 - Le retour de la fonction est un entier
 - Les paramètres formels sont : l'entier **N** comme paramètre donné
 - 3 variables locales de type entier : **i** , **x** et **min**

Min (N : entier): entier

Var i, x, min : entier

Début

min := 100

pour i de 1 à N **Faire**

Répéter

Lire (x)

Jusqu'à (x > 0) et (x < 100)

Si x < min **Alors**

min := x

FinSi

FinPour

retourner min

Fin

L'affectation de la valeur du retour

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Appel d'une procédure/une fonction

- Lors de l'appel d'un sous-algorithme (procédure ou fonction) à partir d'un algorithme appelant, on utilisera le nom de la procédure ou la fonction suivi par la liste de ses paramètres effectifs

`<Nom>(<liste_par_effectif>)`

- **<Nom>** : est le nom de la procédure ou la fonction
 - **<liste_par_effectif>** : une suite d'objets désignant les paramètres effectifs séparés par des virgules (',')
-
- Les paramètres effectifs et les paramètres formels doivent être compatibles en nombre et en type
 - La correspondance entre les 2 types de paramètres se fait selon l'ordre d'apparition
 - **Utilisation des fonctions et procédures** : Une fonction s'utilise telle une valeur. La procédure diffère de la fonction par le fait qu'elle ne produit pas de résultat, mais plutôt s'utilise telle qu'une instruction.

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Appel d'une procédure/une fonction

Exemple :

- On souhaite écrire un algorithme qui lit un entier N supérieur à 3, puis saisit N valeurs entières et affiche la plus petite parmi ces N valeurs. Les entiers saisis doivent être supérieurs à 0 et inférieurs à 100

Programme principal

```
Affiche_min
Var N, min: entier
Début
    lire_nombre(N)
    min := Min(N)
    écrire( "la plus petite valeur est :", min)
Fin
```

Procédure lire_nombre

```
lire_nombre(var N)
Début
    Répéter
        Lire ( N )
    Jusqu'à ( N > 3)
Fin
```

Fonction Min

```
Min (N : entier): entier
Var i, x, min : entier
Début
    min := 100
    Pour i de 1 à N faire
        Répéter
            lire ( x )
        Jusqu'à (x > 0) et (x < 100)
        Si x < min Alors
            min := x
        FinSi
    FinPour
    retourner min
Fin
```

On a fait appel à la procédure lire_nombre() qui exécute une série d'instructions, alors que l'appel à la fonction Min() se fait via la variable min. Celle-ci va recevoir la valeur retournée par la fonction Min().

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Passage de paramètre

- **Passage par valeur** : valeur du paramètre effectif transmise au paramètre formel :
Il s'agit d'un paramètre donnée
- **Passage par adresse** : l'adresse du paramètre effectif transmise au paramètre formel :
Il s'agit d'un paramètre résultat ou d'un paramètre donnée/résultat

Appelant

p.formel

Appelé

p.effectif

Passage par valeur

- Une copie est transmise
- Toute modification sur le paramètre formel n'altère pas le paramètre effectif

Appelant

p.formel

Appelé

p.effectif

Passage par adresse

Zone mémoire

- Une adresse est transmise
- Toute modification sur le paramètre formel altère le paramètre effectif

03 – STRUCTURER UN ALGORITHME

Procédures et Fonctions



Passage de paramètre

Exemple :

Passage de paramètres par valeur:

```
ajoute_un (a : entier)
```

Debut

```
    a := a+1
```

Fin

Appel :

Programme Principal

```
var x : entier
```

Debut

```
    x := 9
```

```
    ajoute_un(x)
```

```
    ecrire(x)
```

Fin

Valeur affichée 9

Passage de paramètres par adresse:

```
inc(var x : entier)
```

Debut

```
    x := x+1
```

Fin

Appel :

Programme Principal

```
var y : entier
```

Debut

```
    y := 9
```

```
    inc(y)
```

```
    ecrire(y)
```

Fin

Valeur affichée 10

CHAPITRE 3

STRUCTURER UN ALGORITHME

1. Procédures et Fonctions

2. Portée d'une variable



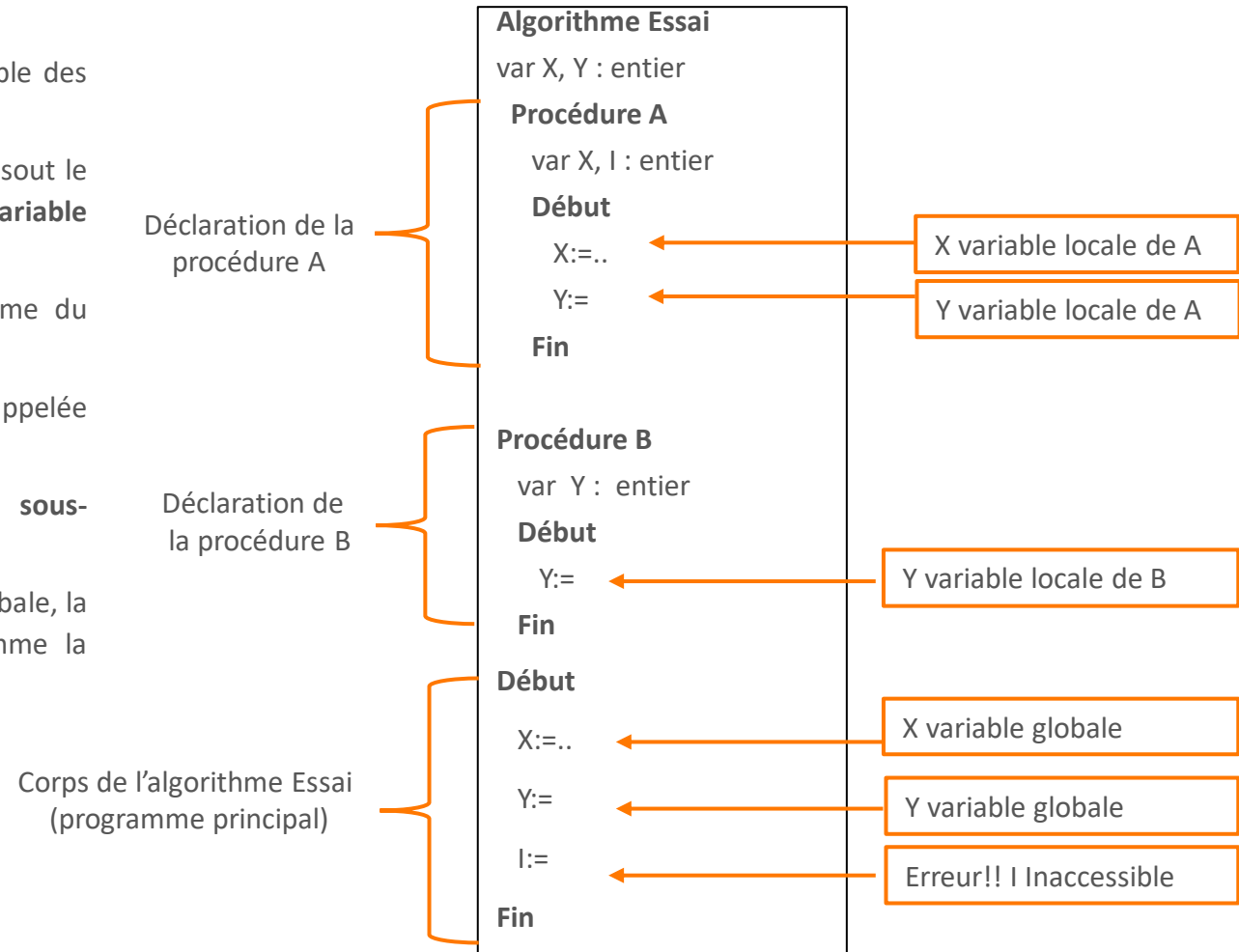
03 – STRUCTURER UN ALGORITHME

Portée d'une variable



Variable globale/variable locale

- La portée d'une variable désigne la partie du programme (l'ensemble des sous-algorithmes) dans laquelle on peut l'utiliser.
- Une **variable définie** au niveau du programme principal (celui qui résout le problème initial, le problème de plus haut niveau) est appelée **variable globale**
- La **portée d'une variable globale est totale** : tout sous-algorithme du programme principal peut utiliser cette variable
- Une variable définie au sein d'un sous-algorithme ou un bloc est appelée **variable locale**
- La **portée d'une variable locale est restreinte à l'intérieur du sous-algorithme dans lequel elle est déclarée** :
- Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale est localement masquée. Dans ce sous-programme la variable globale devient inaccessible :



CHAPITRE 4

STRUCTURER LES DONNÉES

Ce que vous allez apprendre dans ce chapitre :

- Maîtriser la manipulation d'un tableau vecteur et d'un tableau multidimensionnel
- Connaître les principaux algorithmes de tri d'un tableau
- Maîtriser la manipulation des chaînes de caractères



16 heures



CHAPITRE 4

STRUCTURER LES DONNÉES

1. Différents types de tableaux

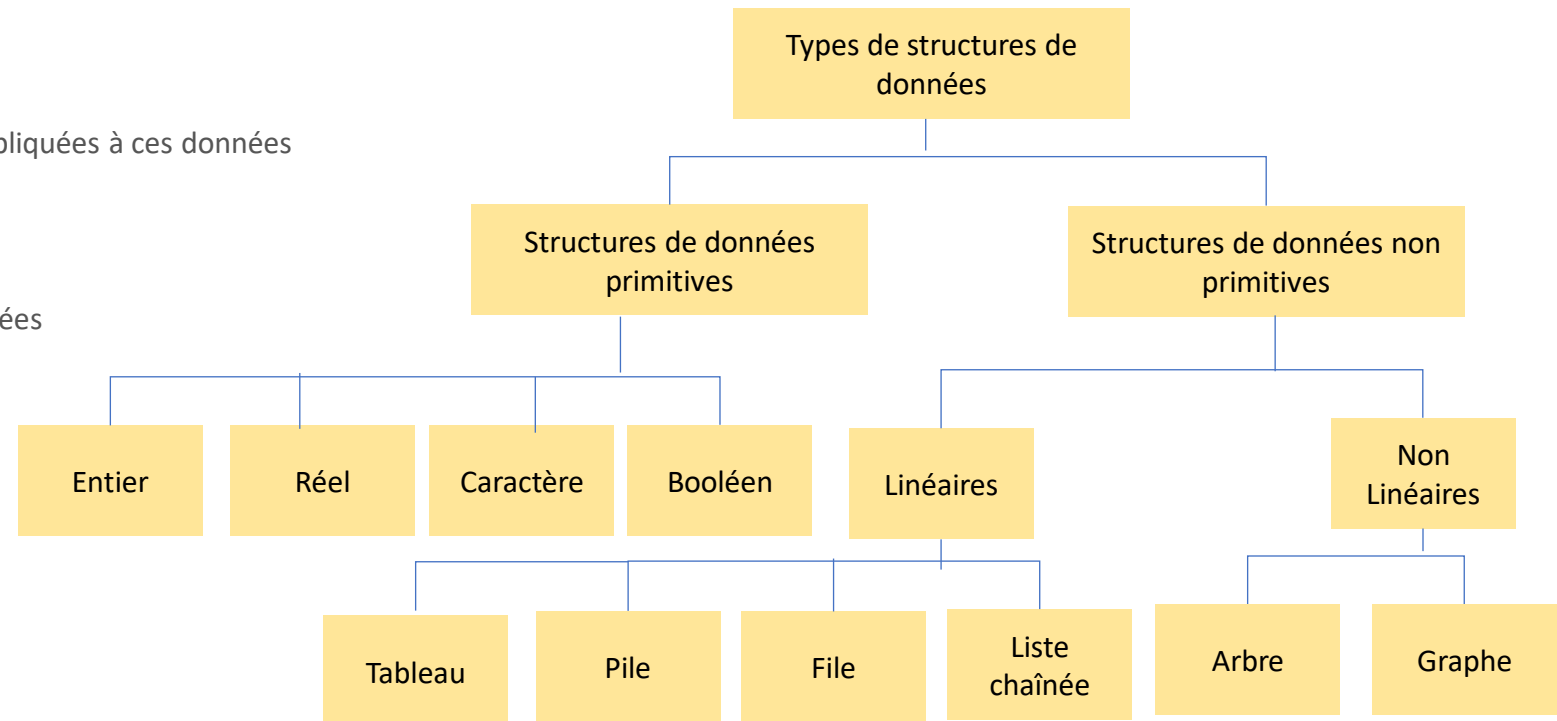
2. Chaines de caractères



Structure de données

Une structure de données est une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir les utiliser efficacement.

- **Une structure de données regroupe :**
 - Un certain nombre de données à gérer
 - Un ensemble d'opérations pouvant être appliquées à ces données
- **Dans la plupart des cas, il existe :**
 - Plusieurs manières de représenter les données
 - Différents algorithmes de manipulation



04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Structure Tableau Vecteur

- Un tableau est une structure de données qui permet de stocker à l'aide d'une seule variable un ensemble de valeurs de même type
- Un tableau unidimensionnel est appelé vecteur
- Syntaxe de déclaration d'un tableau vecteur :

Type vecteur = TABLEAU[min_indice..max_indice] DE <type_predefini>

De telle façon que :

- les éléments du tableau ont pour type le « type_predefini »
- les indices des éléments vont de **min_indice** à **max_indice** et **min_indice < max_indice**,

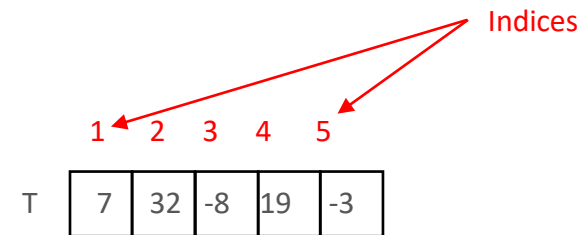
Exemple: déclaration d'un **tableau de 5 cases**

T : tableau [1..5] d'entier

- L'accès à un élément du tableau se fait via la position de cet élément dans le tableau :

nom_tableau [indice]

avec **indice** est la position de l'élément dans le tableau



Exemple de tableau

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Structure Tableau Vecteur

Caractéristiques :

- Un tableau vecteur possède un nombre maximal d'éléments défini lors de l'écriture de l'algorithme (les bornes sont des constantes explicites, par exemple MAX, ou implicites, par exemple 10)
- Le nombre d'éléments maximal d'un tableau est différent du nombre d'éléments significatifs dans un tableau

Exemple :

d'un algorithme permettant de lire un tableau vecteur de 12 entiers

```
LectureTabVecteur  
var i : entier  
      T : tableau[1..12] de Réel  
Debut  
      Pour i de 1 à 12 Faire  
          lire(T[i])  
      FinPour  
Fin
```

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Structure de tableau multi-dimensions

- Par extension, on peut définir et utiliser des tableaux à n dimensions
- Syntaxe de déclaration d'un tableau à n dimensions :

tableau [intervalle1,intervalle2,...,intervallen] de type des éléments

- Les tableaux à deux dimensions permettent de représenter les matrices
- Syntaxe de déclaration d'une matrice :

tableau [intervalle1,intervalle2] de type des éléments

- Chaque élément de la matrice est repéré par deux indices :
 - Le premier indique le numéro de la ligne
 - Le second indique le numéro de la colonne
- **Exemple:** Déclaration d'un tableau à 2 dimensions : **T : tableau [1..2,1..M] d'entier**

T[2,i] désigne l'élément situé à la 2ème ligne et la ième colonne.

1						
2	1	2		i	...	M

T[2,i] ← 10

Exemple de tableau à 2 dimensions

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Structure de tableau multi-dimensions

Exemple: un algorithme permettant de lire un tableau matrice d'entiers de 12 lignes et 8 colonnes

LectureTabMatrice

var i, j : entier

T: tableau[1..12, 1..8] de Réel

Debut

Pour i **de** 1 **à** 12 **Faire**

Pour j **de** 1 **à** 8 **Faire**

Lire(T[i, j])

FinPour

FinPour

Fin

indices

i
de 1 à 12

j de 1 à 8

:

--	--	--	--	--	--	--	--

04 – STRUCTURER LES DONNÉES

Différents types de tableaux

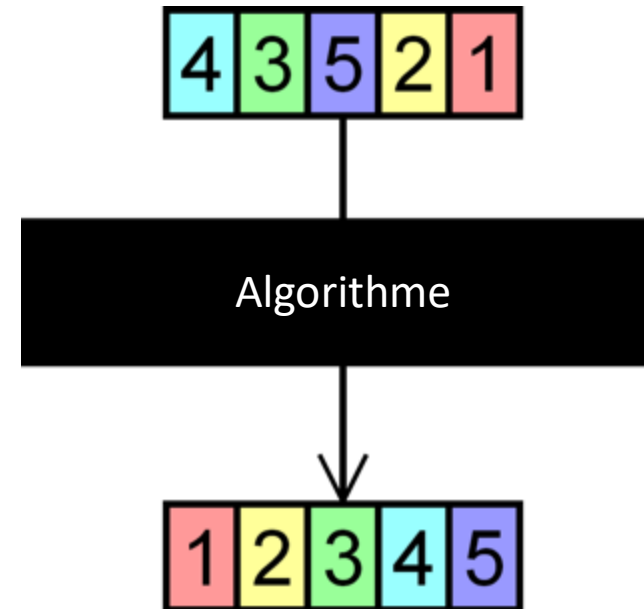
Tri d'un tableau

- Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur
- Parmi les méthodes de tri d'un tableau on cite :

Tri par insertion

Tri à bulles

Tri par sélection



Entrée/sortie d'un algorithme de tri

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Tri par sélection

Le tri par sélection est la méthode de tri la plus simple. Elle consiste à :

- Chercher l'indice du plus petit élément du tableau $T[1..n]$ et permuter l'élément correspondant avec l'élément d'indice 1
- Chercher l'indice du plus petit élément du tableau $T[2..n]$ et permuter l'élément correspondant avec l'élément d'indice 2
-
- Chercher l'indice du plus petit élément du tableau $T[n-1..n]$ et permuter l'élément correspondant avec l'élément d'indice (n-1)

Tableau initial

6	4	2	3	5
---	---	---	---	---

Après la 1ère itération

2	4	6	3	5
---	---	---	---	---

Après la 2^{ème} itération

2	3	6	4	5
---	---	---	---	---

Après la 3^{ème} itération

2	3	4	6	5
---	---	---	---	---

Après la 4^{ème} itération

2	3	4	5	6
---	---	---	---	---

Exemple d'exécution d'un tri par sélection

Tri_Selection(Var T : Tab)

var

i, j, x, indmin : Entier

Début

Pour i de 1 à (n-1) Faire

indmin := i

Pour j de (i+1) à n Faire

Si (T[j] < T[indmin]) Alors

indmin := j

FinSi

FinPour

x := T[i]

T[i] := T[indmin]

T[indmin] := x

FinPour

Fin

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Tri à bulles

La méthode de tri à bulles nécessite deux étapes :

- Parcourir les éléments du tableau de 1 à $(n-1)$; si l'élément i est supérieur à l'élément $(i+1)$, alors on les permute
- Le programme s'arrête lorsqu'aucune permutation n'est réalisable après un parcours complet du tableau

Tableau initial

6	4	3	5	2
---	---	---	---	---

Après la 1^{ère} itération

4	3	5	2	6
---	---	---	---	---

Après la 2^{ème} itération

3	4	2	5	6
---	---	---	---	---

Après la 3^{ème} itération

3	2	4	5	6
---	---	---	---	---

Après la 4^{ème} itération

2	3	4	5	6
---	---	---	---	---

Exemple d'exécution d'un tri à bulles

Tri_Bulle (Var T : Tab)

var

i, x : Entier

échange : Booléen

Début

Répéter

échange := Faux

Pour i **de** 1 **à** $(n-1)$ **Faire**

Si $(T[i] > T[i+1])$ **Alors**

$x := T[i]$

$T[i] := T[i+1]$

$T[i+1] := x$

échange := Vrai

FinSi

FinPour

Jusqu'à (échange = Faux)

Fin

04 – STRUCTURER LES DONNÉES

Différents types de tableaux



Tri par insertion

Le tri par insertion consiste à prendre les éléments de la liste un par un et insérer chacun dans sa bonne place de façon à ce que les éléments traités forment une sous-liste triée.

Pour ce faire, on procède de la façon suivante :

- Comparer et permuter si nécessaire $T[1]$ et $T[2]$ de façon à placer le plus petit dans la case d'indice 1
- Comparer et permuter si nécessaire l'élément $T[3]$ avec ceux qui le précèdent dans l'ordre ($T[2]$ puis $T[1]$) afin de former une sous-liste triée $T[1..3]$
-
- Comparer et permuter si nécessaire l'élément $T[n]$ avec ceux qui le précèdent dans l'ordre ($T[n-1]$, $T[n-2]$, ...) afin d'obtenir un tableau trié

Tableau initial

6	4	3	5	2
---	---	---	---	---

Après la 1^{ère} itération

4	6	3	5	2
---	---	---	---	---

Après la 2^{ème} itération

3	4	6	5	2
---	---	---	---	---

Après la 3^{ème} itération

3	4	5	6	2
---	---	---	---	---

Après la 4^{ème} itération

2	3	4	5	6
---	---	---	---	---

Exemple d'exécution d'un tri par insertion

Tri_Insertion(Var T : Tab)

var

i, j, x, pos : Entier

Début

Pour i de 2 à n Faire

pos := i - 1

TantQue (pos >= 1) et ($T[pos] > T[i]$) **Faire**

pos := pos - 1

FinTQ

pos := pos + 1

x := $T[i]$

Pour j de (i-1) à pos [pas = -1] Faire

$T[j+1] := T[j]$

FinPour

$T[pos] := x$

FinPour

Fin

[Pas = -1] signifie que le parcours se fait dans le sens décroissant

CHAPITRE 4

STRUCTURER LES DONNÉES

1. Différents types de tableaux

2. Chaines de caractères



04 – STRUCTURER LES DONNÉES

Chaines de caractères



Chaines de caractères

- Une chaîne de caractères est une suite de caractères. La chaîne ne contenant aucun caractère est appelée chaîne vide
- **Syntaxe** de déclaration d'un chaîne :

ch : Chaîne
chn : Chaîne[Max]

La variable ch peut contenir jusqu'à 255 caractères alors que chn peut contenir au maximum Max caractères

- **Les opérations sur les chaînes de caractères**
 - La **concaténation** est l'assemblage de deux chaînes de caractères en utilisant l'opérateur « + »

Exemple:

chn1 := "Structure"

chn2 := "de données"

chn3 := chn1+ " "+chn2

la variable **chn3** contiendra "Structure de données"

Les opérations sur les chaines de caractères

- Les opérateurs relationnels (>, >=, <, <=, =, #):

Il est possible d'effectuer une comparaison entre deux chaînes de caractères, le résultat est de type booléen. La comparaison se fait caractère par caractère de la gauche vers la droite selon le code ASCII

Exemples:

- L'expression ("a" > "A") est vraie puisque le code ASCII de "a" (97) est supérieur à celui de "A" (65)
- L'expression ("programme" < "programmation") est fausse puisque "e" > "a" »
- L'expression (" " = " ") est fausse (le vide est différent du caractère espace)

- Accès à un caractère dans une chaîne :

Il suffit d'indiquer le nom de la chaîne suivi d'un entier entre crochets qui indique la position du caractère dans la chaîne, de la même manière pour un élément d'un vecteur.

Exemples:

- chn:= "Turbo Pascal"
- c:= chn[7] (la variable c contiendra le caractère "P")

En général, ch[i] désigne le i^{ème} caractère de la chaîne ch

04 – STRUCTURER LES DONNÉES

Chaines de caractères



Procédures et Fonctions standards sur les chaines de caractères

Procédures standards sur les chaines de caractères :

Procédure	Rôle	Exemple
Efface (Chaîne, P, N)	Enlève N caractères de Chaîne à partir de la position P donnée.	Chn ← "Turbo Pascal" Efface(chn,6,7) → Chn contiendra "Turbo"
Insert (Ch1, Ch2, P)	Insère la chaîne Ch1 dans la chaîne Ch2 à partir de la position P.	Ch1 ← " D" Ch2 ← "AA" Insert'(ch1,ch2,2) → ch2 contiendra "ADA"
Convch(Nbr, Ch)	Converti le nombre Nbr en une chaîne de caractères Ch.	N = 1665 Convch(n,chn) → Chn contiendra la chaîne "1665"

Fonctions standards sur les chaines de caractères :

Fonction	Rôle	Exemple
Long(Chaîne)	Retourne la longueur de la chaîne	Chn ← " Turbo Pascal " N ← Long(chn) → n contiendra 12
Copie(Chaîne, P, N)	Copie N caractères de Chaîne à partir de la position P donnée.	Ch1 ← "Turbo Pascal " Ch2 ← contiendra "Pascal"
Position(Ch1, Ch2)	Retourne la position de la première occurrence de la chaîne Ch1 dans la chaîne Ch2	Ch1 ← "as" Ch2 ← " Turbo Pascal " N ← Position(ch1,ch2) → n contiendra 8



PARTIE 3

PROGRAMMER EN PYTHON

Dans ce module, vous allez :

- Maitriser les bases de la programmation Python
- Appliquer les bonnes pratiques de la programmation Python
- Manipuler les fonctions en Python
- Maitriser la manipulation des données en Python



48 heures

CHAPITRE 1

TRANSFORMER UNE SUITE D'ÉTAPES ALGORITHMIQUES EN UNE SUITE D'INSTRUCTIONS PYTHON

Ce que vous allez apprendre dans ce chapitre :

- Connaître les Critères de choix d'un langage de programmation
- Connaître les caractéristiques du langage Python
- Maîtriser la structure générale d'un programme Python
- Traduire un algorithme en langage Python
- Appliquer les bonnes pratiques du codage en Python



20 heures



CHAPITRE 1

TRANSFORMER UNE SUITE D'ÉTAPES ALGORITHMIQUES EN UNE SUITE D'INSTRUCTIONS PYTHON

1. Critères de choix d'un langage de programmation

2. Blocs d'instructions

3. Conversion de l'algorithme en Python

4. Optimisation du code (bonnes pratiques de codage, commentaires, etc.)



Langage de programmation

- Le langage machine est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique
- Le langage machine est peu compréhensible par un humain d'où le besoin au recours à un langage de programmation de haut niveau
- Le langage de programmation est un outil à l'aide duquel le programmeur écrit des programmes exécutables sur un ordinateur

Exemples de langages de programmation:

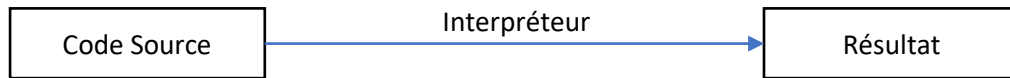
FORTRAN, COBOL, Pascal, Ada, C, Java, Python

Un code source ou programme écrit en un langage de programmation de haut niveau qui est ensuite traduit vers un langage machine.

Langage de programmation

Il existe 2 techniques pour effectuer la traduction en langage machine :

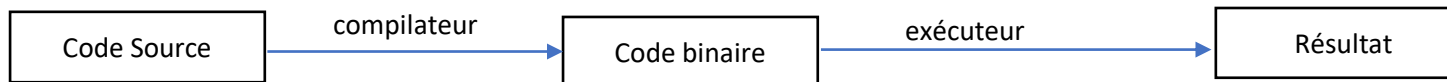
Langage interprété :



Passage d'un code source à un résultat dans un langage interprété

Un interpréteur est un **programme informatique** qui traite le code source d'un projet logiciel pendant son fonctionnement – c'est-à-dire pendant son exécution

Langage compilé :

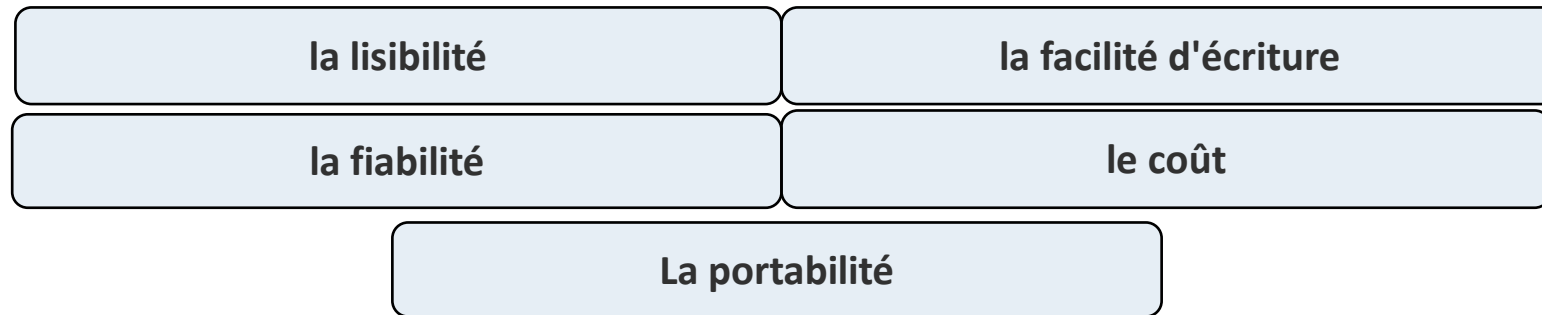


Passage d'un code source à un résultat dans un langage compilé

Un compilateur est un **programme informatique** qui traduit l'ensemble du code source d'un projet logiciel en code machine avant son exécution.

Langage de programmation

Il est important de pouvoir évaluer des langages de programmation afin de pouvoir les choisir de manière appropriée et de les améliorer. Cinq critères d'évaluation sont généralement utilisés :



Critères d'évaluation d'un langage de programmation

Critères d'évaluation

Critères affectant la
lisibilité

Critères affectant la
facilité d'écriture

Critères affectant la
fiabilité

Critères affectant
le coût

Critères affectant la
portabilité

La simplicité :

- S'il y a beaucoup de composantes de base (telles que les mots clés), il est difficile de toutes les connaître
- S'il existe plusieurs façons d'exprimer une commande, il est aussi difficile de toutes les connaître
- Trop de simplicité cause la difficulté de lecture
- Il faut alors trouver un compromis entre la simplicité et la facilité d'écriture

L'orthogonalité :

- L'orthogonalité est la propriété qui signifie "Changer A ne change pas B"
- Dans les langages de programmation, cela signifie que lorsque vous exécutez une instruction, seule cette instruction se produit
- De plus, la signification d'un élément du langage doit être indépendante du contexte dans lequel il apparaît

Instructions de contrôle :

- Pour la lisibilité d'un langage de programmation, il est important d'avoir des structures de contrôle adéquates (structures itératives, structures conditionnelles, etc.)
- Par exemple, l'un des plus grands problèmes du premier BASIC est que sa seule instruction de contrôle était le « goto »

Types et structures de données :

- La présence de moyens appropriés pour définir des types et des structures de données dans un langage peut améliorer considérablement la lisibilité

Critères d'évaluation

Critères affectant la lisibilité

Critères affectant la facilité d'écriture

Critères affectant la fiabilité

Critères affectant le coût

Critères affectant la portabilité

La simplicité et l'orthogonalité:

- Si un langage a une grande variation de constructeurs syntaxiques, il est fort possible que certains programmeurs ne les connaissent pas
- De même, il se peut que le programmeur ne connaisse certains constructeurs que superficiellement et les utilise de manière erronée

L'abstraction:

- L'abstraction est la possibilité de définir des structures ou des opérations compliquées tout en cachant leurs détails (abstraction de processus et abstraction des données)

Abstraction de processus:

- quand un processus est abstrait dans un sous-programme il n'est pas Nécessaire de répéter son code à chaque fois qu'il est utilisé. Un simple appel de la procédure/fonction est suffisant

Abstraction des données :

- les données peuvent être abstraites par les langages de programmation de haut niveau dans des objets à interface simple. L'utilisateur n'a pas besoin de connaître les détails d'implémentation pour les utiliser (utilisation des arbres, tables de hachage...)

L'expressivité:

- Un langage est expressif s'il offre des outils simples, commodes et intuitifs pour permettre au programmeur d'exprimer les différents concepts de programmation

Exemple:

- Utiliser des boucles « for » et « while » au lieu de « goto »

Critères d'évaluation

Critères affectant la lisibilité

Critères affectant la facilité d'écriture

Critères affectant la fiabilité

Critères affectant le coût

Critères affectant la portabilité

La fiabilité désigne la **capacité d'un programme** écrit en langage de programmation particulier **d'assurer sa fonction dans des conditions données**

Vérification de types :

- La vérification de types signifie qu'un langage est capable de détecter les erreurs relatives aux types de données lors de la compilation et de l'exécution

Exemple :

- Des langages tels que Python, Ada, C++ et Java, Ruby, C# ont des capacités étendues de prise en charge des exceptions, mais de tels capacités sont absentes dans d'autres langages tels que le C ou le FORTRAN

Prise en charge des exceptions :

- La possibilité pour un programme d'intercepter les erreurs faites pendant l'exécution, de les corriger et de continuer l'exécution augmente de beaucoup la fiabilité du langage de programmation

Exemple :

- Des langages tels que Python, Ada, C++ et Java, Ruby, C# ont des capacités étendues de prise en charge des exceptions, mais de telles capacités sont absentes dans d'autres langages tels que le C ou le FORTRAN

Lisibilité et facilité d'écriture :

- La lisibilité et la facilité d'écriture influencent la fiabilité des langages de programmation.
- si il n'y a pas de moyens naturels d'exprimer un algorithme, des solutions complexes seront utilisées ce qui engendre un risque d'erreurs élevé

Critères d'évaluation

Critères affectant la
lisibilité

Critères affectant la
facilité d'écriture

Critères affectant la
fiabilité

Critères affectant
le coût

Critères affectant la
portabilité

- Coût lié à la **programmation** :
 - Si le langage n'est pas simple et orthogonal alors :
 - les coûts de formation de programmeurs seront plus élevés
 - l'écriture de programmes coûtera plus chère
- Coût lié à la **compilation et à l'exécution de programmes**
- Coût lié à la **maintenance de programmes**
- Coût lié à la **mise en marche du langage**

Critères d'évaluation

Critères affectant la
lisibilité

Critères affectant la
facilité d'écriture

Critères affectant la
fiabilité

Critères affectant
le coût

Critères affectant la
portabilité

- La portabilité désigne la **capacité d'un programme** écrit en langage de programmation particulier à **fonctionner dans différents environnements d'exécution**.
- Les différences peuvent porter sur l'environnement matériel (processeur) comme sur l'environnement logiciel (système d'exploitation)

01 – PYTHON

Critères de choix d'un langage de programmation



Langage Python

- Python est un langage de programmation développé **depuis 1989** par **Guido van Rossum** et de nombreux contributeurs bénévoles
- **En février 1991**, la première version publique, numérotée 0.9.0.
- En Octobre 2000, la version Python 2.0 a été publiée puis suivie par les versions 2.1, 2.2 jusqu'à la version 2.7
- Afin de réparer certains défauts du langage, **la version Python 3.0 a été publiée en décembre 2008**
- Cette version a été suivie par :
 - La version 3.1 en 2009
 - La version 3.3 en 2011
 - La version 3.4 en 2014
 - La version 3.6 en 2016
 - La version 3.7 en 2017



Symbole du langage PYTHON

Caractéristiques de Python

- Python est **orienté-objet** : Un langage qui s'articule autour des objets organisés en Classes plutôt que de simples fonctions et procédures.
- Python intègre un système d'exceptions : les exceptions permettent de simplifier considérablement la gestion des erreurs
- Python est **dynamiquement typé** : tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance
- Python est **orthogonal** : un petit nombre de concepts suffit à engendrer des constructions très riches
- Python est **introspectif** : un grand nombre d'outils de développement, comme le debugger sont implantés en Python lui-même
- Python est **extensible** : on peut facilement l'interfacer avec des bibliothèques existantes : La bibliothèque standard de Python, et les paquetages contribus, donnent accès à une grande variété de services: chaînes de caractères et expressions régulières, services UNIX standard (fichiers, pipes, signaux, sockets, threads, etc.), protocoles Internet (Web, News, FTP, CGI, HTML, etc.), persistance et bases de données, interfaces graphiques
- Python est **portable**, non seulement sur les différentes variantes d'Unix, mais aussi sur les OS propriétaires : MacOS, BeOS, NeXTStep, MS-DOS et les différentes variantes de Windows
- Python est **gratuit** mais on peut l'utiliser sans restriction dans des projets commerciaux
- La **syntaxe de Python est très simple** et, combinée à des **types de données évolués** (listes, dictionnaires, etc.), conduit à des programmes à la fois très compacts et très lisibles
- Python **gère ses ressources** (mémoire, descripteurs de fichiers, etc.) sans intervention du programmeur

Etapes d'installation de Python

- Pour installer Python, il faut se rendre sur le site officiel (<http://python.org/download/>) afin de télécharger un installateur

Il en existe deux versions de Python : les versions 3.x.x et la version 2.x.x. Toutes sont disponibles sur le site de Python, mais les possesseurs d'ordinateurs récents ont tout intérêt à opter pour la version 3.x.x

- Il faut veiller à prendre la version correspondant à l'architecture de la machine (x86 si l'on a une machine 32 bits ou x86_64 si l'on a une machine 64 bits)

Après le téléchargement, il faut lancer l'exécutable et suivre les étapes suivantes :

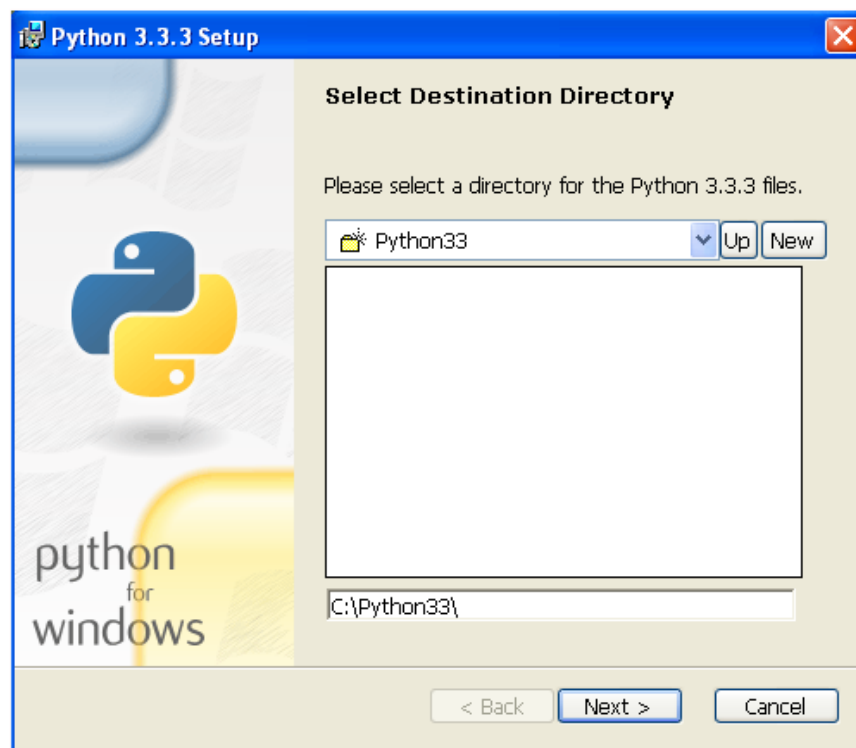
1. Choisir une installation pour tous les utilisateurs ou seulement pour son profil
(en général, il est conseillé de l'installer pour tous les profils).



Première interface d'installation de PYTHON

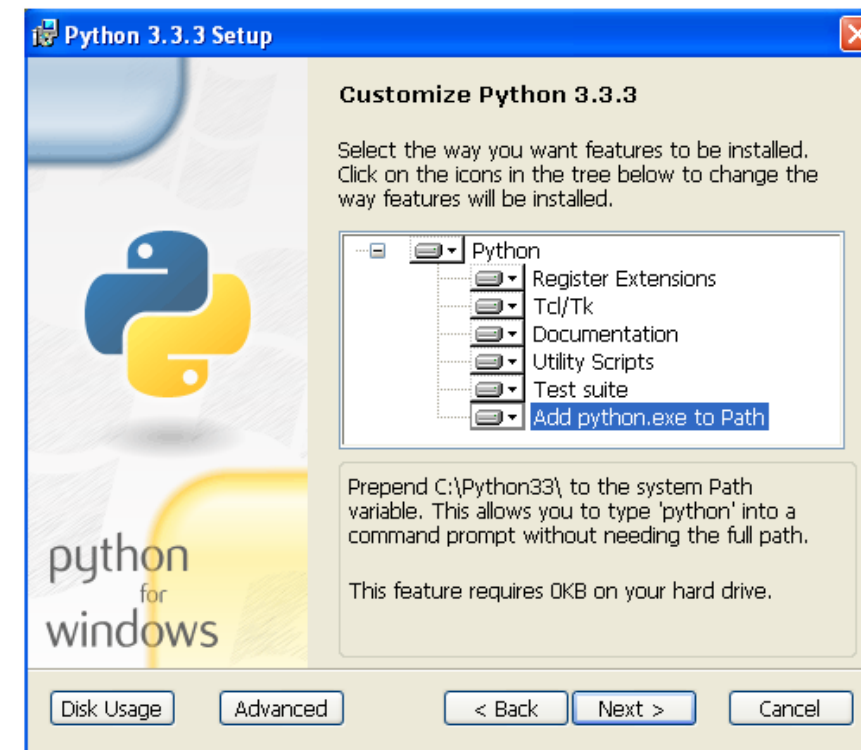
Etapes d'installation de Python

2. Choisir le répertoire d'installation (le choix par défaut est recommandé)



Interface de choix de répertoire d'installation

3. Choisir les modules secondaires à installer



Interface de choix de modules secondaires

01 – PYTHON

Critères de choix d'un langage de programmation

Etapes d'installation de Python

4. La fin de l'installation prend quelques secondes



Interface de fin d'installation

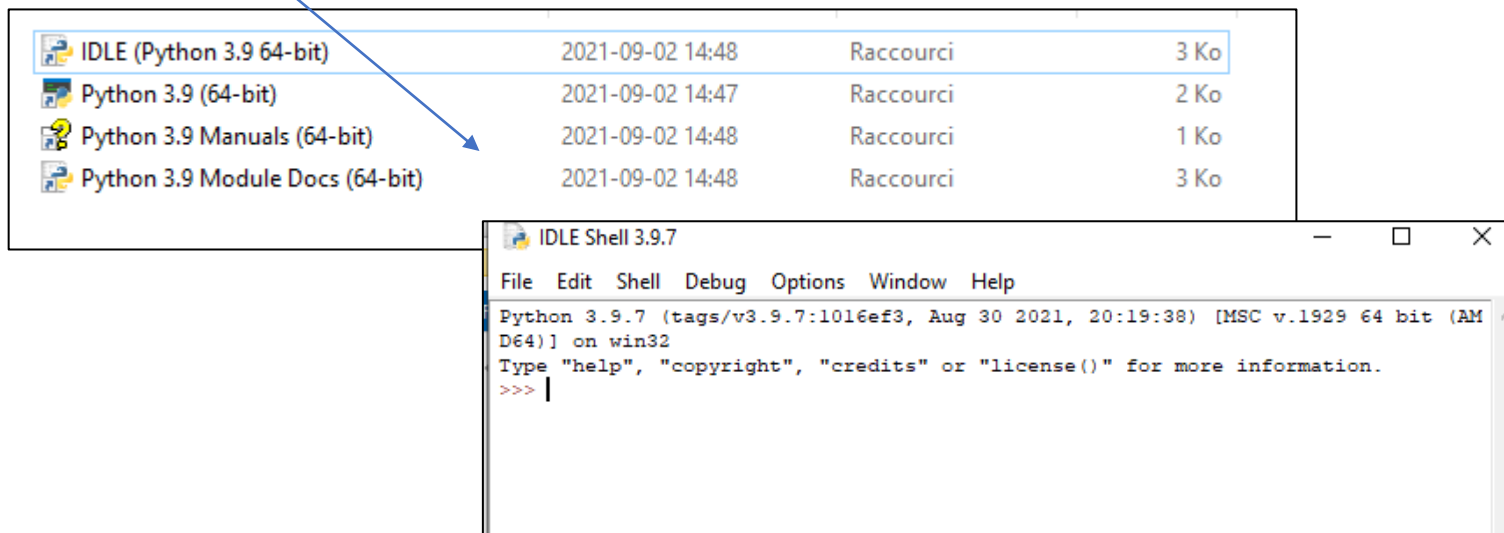
- L'installation de Python génère l'installation d'une interface, **appelée IDLE (Python GUI)**
- Cette interface vous permet de saisir des instructions en ligne de commande mais également d'exécuter des programmes Python enregistrés dans des fichiers

01 – PYTHON

Critères de choix d'un langage de programmation

Utilisation du IDLE Python

Il suffit de cliquer sur **IDLE (Python GUI)** pour ouvrir l'interface graphique relative à interpréteur de commandes en ligne.



Interface graphique relative à l'invite de commande

```
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Bonjour")
Bonjour
>>> x=1
>>> y=x+1
>>> print(y)
```

Exemple d'exécution d'un code Python en utilisant l'invite de commande

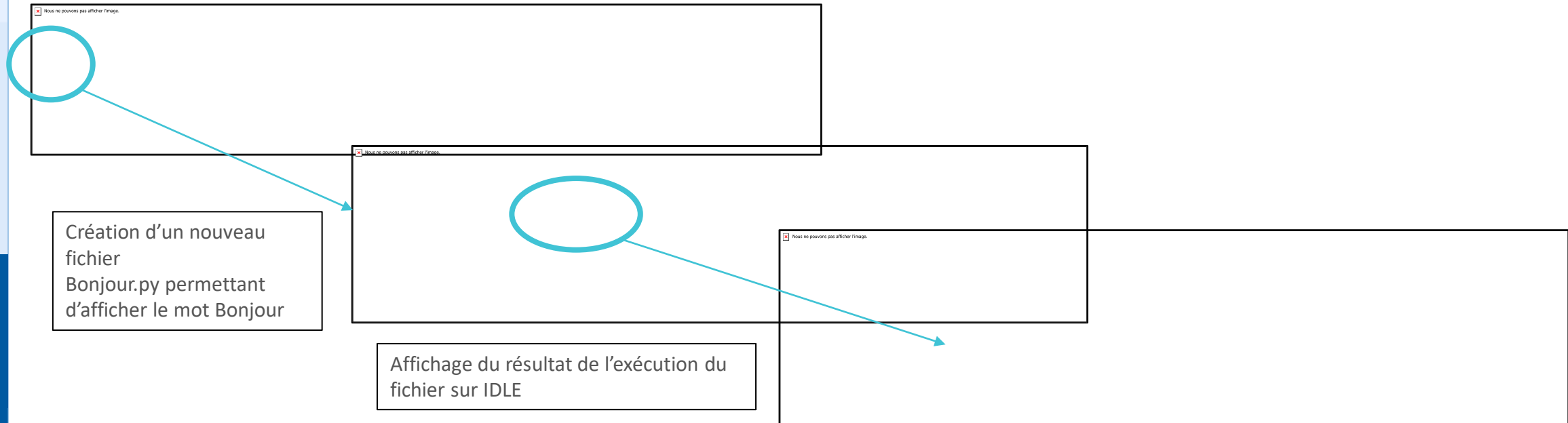
01 – PYTHON

Critères de choix d'un langage de programmation

Utilisation du IDLE Python

Pour écrire un programme dans un fichier, aller dans le menu File :

- Sélectionnez New File
- Une nouvelle fenêtre s'ouvre
- Tapez votre programme Python dans cette fenêtre
- Pour exécuter votre programme, allez dans le menu Run et faites Run Modules



Exemple d'exécution d'un fichier .py

CHAPITRE 1

TRANSFORMER UNE SUITE D'ÉTAPES ALGORITHMIQUES EN UNE SUITE D'INSTRUCTIONS PYTHON

1. Critères de choix d'un langage de programmation
2. **Blocs d'instructions**
3. Conversion de l'algorithme en Python
4. Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Structuration et notion de bloc

- En Python, chaque instruction s'écrit sur une ligne sans mettre d'espace au début

Exemple:

```
a = 10
b = 3
print(a, b)
```

- Ces instructions simples peuvent cependant être mises sur la même ligne en les séparant par des points virgules (;), les lignes étant exécutées dans l'ordre de gauche à droite

Exemple:

```
a = 10; b = 3; print(a, b)
```

- La séparation entre les en-têtes qui sont des lignes de définition de boucles, de fonction, de classe se terminent par les deux points (:)
- Le contenu ou « bloc » d'instructions correspondant se fait par indentation des lignes
- Une indentation s'obtient par le bouton tab (pour tabulation) ou bien par 4 espaces successifs
- L'ensemble des lignes indentées constitue un bloc d'instructions

```
instruction-1
  instruction-2
    instruction-2
      instruction-3
        instruction-3
          instruction-3
            instruction-3
              instruction-2
                instruction-2
instruction-1
instruction-1
```

Exemple de lignes indentées

CHAPITRE 1

TRANSFORMER UNE SUITE D'ÉTAPES ALGORITHMIQUES EN UNE SUITE D'INSTRUCTIONS PYTHON

1. Critères de choix d'un langage de programmation
2. Blocs d'instructions
- 3. Conversion de l'algorithme en Python**
4. Optimisation du code (bonnes pratiques de codage, commentaires, ...)

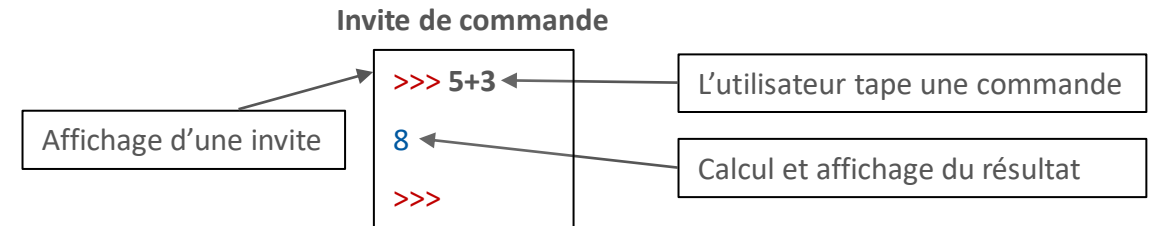


Script et langage Python

- Peu de ponctuation.
- Pas de point virgule ";"
- Tabulation ou 4 espaces significatifs
- Scripts avec exécution d'un fichier ayant l'extension .py

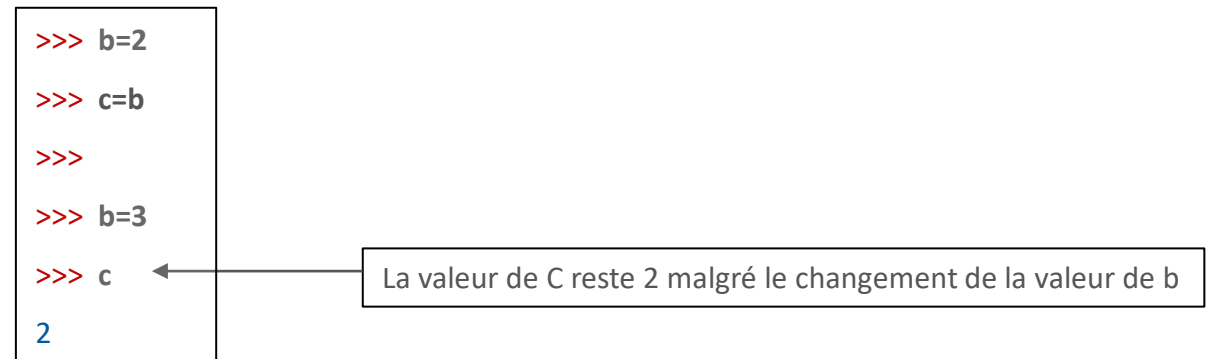
Exemple: script.py

- Python utilise un **identifiant** pour nommer chaque objet
- Python n'offre pas la notion de variable, mais plutôt celle de **référence** (adresse) d'objet



Fichier premierExemple.py

```
a= int (input('donner a:'))
b= int (input('donner b:'))
z= a+b
print('la somme est '+ str(z))
```



Types de données

Les types de données les plus utilisés sont :

- **Type entier (integer)**

```
a=1  
b=555  
c=6
```

- **Type réel (float)**

```
b=0.003  
b=2.
```

- **Type Boolean**

```
b=True  
C=False
```

- **Type caractère**

```
phrase1="les œufs durs"  
Phrase2="oui, répondit-il"  
Phrase3="j'aime bien"
```

Variables

Une variable est créée au moment où vous lui attribuez une valeur pour la première fois.

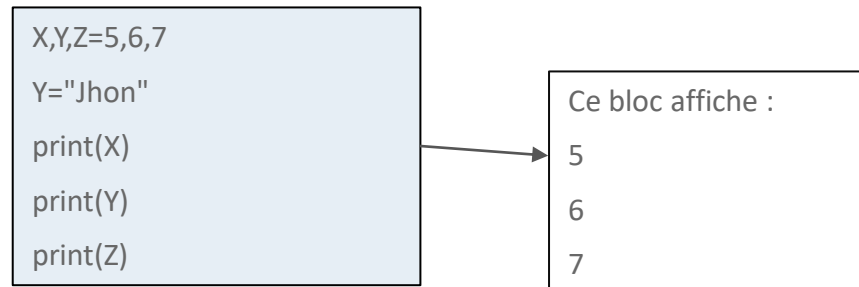


Les variables de chaîne peuvent être déclarées à l'aide de guillemets simples ou doubles :

Règles pour les variables Python :

- Un **nom** de variable doit commencer par une lettre ou le caractère de soulignement
- Un **nom** de variable ne peut pas commencer par un nombre
- Un **nom** de variable ne peut contenir que des caractères alphanumériques et des traits de soulignement (A-z, 0-9 et _)
- Les **noms** de variable sont sensibles à la casse (age, Age et AGE sont trois variables différentes)

Python permet d'affecter des valeurs à plusieurs variables sur une seule ligne :



Variables d'entrée

- La fonction **input()** retourne une valeur qui correspond à ce que l'utilisateur a entré. Cette valeur peut alors être assignée à une variable quelconque
- Input est une fonction qui renvoie toujours une chaîne de caractères

```
prénom=input("entrez votre nom: ")  
print ("bonjour: ", prénom)
```

- Pour changer le type d'une variable, on utilise :
 - **int()**: pour les entiers
 - **float()**: pour les nombres à virgule flottante

```
print ("veuillez saisir un nombre positif")  
nn= input()  
print("la carré de ", int(nn), "vaut", int(nn)*int(nn))
```

Conversion de la sortie en entier

Affichage:
veuillez saisir un nombre positif
5
la carré de 5 vaut 25

```
print ("veuillez saisir un nombre positif")  
nn= input()  
print("la carré de ", float(nn), "vaut", float(nn)*float(nn))
```

Conversion de la sortie en float

Affichage:
veuillez saisir un nombre positif
5
la carré de 5.0 vaut 25.0

Variables de sortie

- La fonction **print()** de Python est souvent utilisée pour afficher des variables et des chaînes de caractères
- Pour combiner à la fois du texte et une variable, Python utilise le caractère +:

```
x="Python est "  
y="cool"  
z=x+y  
print(z)
```

Ce bloc affiche:
Python est cool

- Pour tronquer une chaîne de caractères à afficher il est possible d'utiliser `\n`

```
print ("Bonjour \nPython")
```

Ce bloc affiche:
**Bonjour
Python**

- Pour imprimer des lignes vides, il est possible d'utiliser l'une des méthodes suivantes :
 - Utiliser un chiffre représentant le nombre de lignes vides suivi de « * » et `\n`:
 - Remplacer le nombre des lignes vides par des `\n`

```
x="Python est "  
y="cool"  
z=x+y  
print(z)  
print(3 * "\n")  
print ("Merci")
```

Ce bloc affiche:
Python est cool

Merci

Variables de sortie

- La fonction « **end** » permet d'ajouter n'importe quelle chaîne à la fin de la sortie de la fonction **print**

```
print("1--Hello", end="")
```

Pas d'espace à la sortie de **print**

```
print("Joe")
```

Ajouter un espace à la sortie de **print**

```
print("2--Hello", end=" ")
```

Ajouter @ à la sortie de **print**

```
print("Joe")
```

```
print("3--Hello", end="@")
```

```
print("Joe")
```

Ce bloc affiche:

```
1--HelloJoe
2--Hello Joe
3--Hello@Joe
```

- Le mot clé « **sep** » précise une séparation entre les variables chaînes

```
x = 32
```

```
nom = "John"
```

```
print(nom, "a", x, "ans", sep="")
```

Pas d'espace entre les variables

```
print(nom, "a", x, "ans", sep="-")
```

Le caractère – entre les variables

Ce bloc affiche:

```
Johna32ans
John-a-32-ans
```

Variables de sortie

- Pour afficher deux chaînes de caractères l'une à côté de l'autre, sans espace :
 - on peut soit les concaténer, soit utiliser l'argument par mot-clé `sep` avec une chaîne de caractères vide :

```
ani1 = "chat"  
ani2 = "souris"  
print(ani1, ani2)  
print(ani1 + ani2)  
print(ani1, ani2, sep="")
```

Ce bloc affiche:
chat souris
chatsouris
chatsouris

- La méthode **.format()** permet une meilleure organisation de l'affichage des variables dans une chaîne de caractères :

```
x=32  
nom="John"  
print("{} a {} ans.".format(nom,x))  
print("{0} a {1} ans.".format(nom,x))
```

Ce bloc affiche:
John a 32 ans.
John a 32 ans.

Manipulation des types numériques

Python reconnaît et accepte les opérateurs arithmétiques suivants :

Opérateur	nom
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
**	Puissance
//	Division entière

Opérateurs d'affectation composés Python

Python reconnaît également des opérateurs d'affectation qu'on appelle "composés" et qui vont permettre d'effectuer deux opérations à la suite : une première opération de calcul suivie immédiatement d'une opération d'affectation.

Opérateur	Utilisation	Explication
+=	X+=1	Ajoute 1 à la dernière valeur connue de x et affecte la nouvelle valeur (ancienne + 1) à x
-=	X-=1	Enlève 1 à la dernière valeur connue de x et affecte la nouvelle valeur à x
=	X=2	Multiplie par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
/=	x/=2	Divise par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
%=	X%=2	Calcule le reste de la division entière de x par 2 et affecte ce reste à x
//=	x//=2	Calcule le résultat entier de la division de x par 2 et affecte ce résultat à x
=	X=4	Elève x à la puissance 4 et affecte la nouvelle valeur dans x

Opérateurs d'affectation composés Python

Exemple :

```
i=10
i=i+1
print("1-",i)
i=i-1
print("2-",i)
i+=1
print("3-",i)
i-=2
print("4-",i)
i*=6
print("5-",i)
```

Ajouter 1 à i et affecter le résultat à i

soustraire 1 de i et affecter le résultat à i

Ajouter 1 à i et affecter le résultat à i

soustraire 2 de i et affecter le résultat à i

Multiplier i par 6 et affecter le résultat à i

Ce bloc affiche:

```
1- 11
2- 10
3- 11
4- 9
5- 54
```

```
x=15
print("6-",x)
x=x/2
print("7-",x)
x=x%2
print("8-",x)
x=x**2
print("9-",x)
```

Diviser x par 2 et affecter le résultat à x

Calculer le reste de la division entière de x par 2 et affecter ce reste à x

Calculer x à la puissance 2 et affecter la nouvelle valeur dans x

Ce bloc affiche:

```
6- 15
7- 7.5
8- 1.5
9- 2.25
```

Structure conditionnelle

Exemple1: structure conditionnelle avec if

```
a = 33
```

```
b = 200
```

```
if b > a:
```

```
    print("b est supérieur à a")
```

Affichage si b>a (c'est le cas)

Affichage:

b est supérieur à a

Exemple 2: structure conditionnelle avec if elif

```
a = 33
```

```
b = 33
```

```
if b > a:
```

```
    print("b est supérieur à a")
```

```
elif a == b:
```

```
    print("a and b sont égaux")
```

Affichage si b>a (ce n'est pas le cas)

Affichage si b=a (c'est le cas)

Exemple 3: structure conditionnelle avec if elif else

```
a = 200
```

```
b = 33
```

```
if b > a:
```

```
    print("b est supérieur à a")
```

```
elif a == b:
```

```
    print("a and b sont égaux")
```

```
else:
```

```
    print("a est supérieur à b")
```

Affichage si b>a (ce n'est pas le cas)

Affichage si b=a (ce n'est pas le cas)

Affichage si b<a (c'est le cas)

Affichage:

a and b sont égaux

Affichage:

a est supérieur à b

Structure conditionnelle

- Le mot clé **or** est un opérateur logique et est utilisé pour combiner des instructions conditionnelles :

```
a=200
```

```
b=33
```

```
c=500
```

```
if a>b or a<c:
```

```
    print("At least one of the conditions is true")
```

Affichage si a>b ou a<c (« Au moins l'une des conditions est vraie »)

Affichage:

At least one of the conditions is true

- Le mot clé **and** est un opérateur logique et est utilisé pour combiner des instructions conditionnelles :

```
a=200
```

```
b=33
```

```
c=500
```

```
if a>b and a<c:
```

```
    print("Both conditions are true")
```

Affichage si a>b et a<c (« les deux conditions sont vraies »)

Affichage :

Both conditions are true (les 2 conditions sont vraies)

Structures itératives

- Boucle While

Avec la boucle **while**, il est possible d'exécuter un ensemble d'instructions tant qu'une condition est vraie :

```
i=1
```

```
while i<6:
```

```
    print(i)
```

```
    i+=1
```

Condition d'entrée à la boucle

Instructions à exécuter tanque la condition est vérifiée

Affichage:

1

2

3

4

5

Structures itératives

- **Boucle For**

- Une boucle **for** est utilisée pour itérer sur une séquence
- Pour parcourir un ensemble de codes un nombre spécifié de fois, nous pouvons utiliser la fonction **range ()**
- La fonction **range ()** renvoie une séquence de nombres, commençant à **0 par défaut**, et incrémentant de **1 (par défaut)**, et se termine à un nombre spécifié

```
for i in range (6):  
    print(i)
```

Affichage d'une séquence de nombres de 0 à 6

Affichage:

0
1
2
3
4
5

- La fonction **range ()** par défaut a 0 comme valeur de départ, mais il est possible de spécifier la valeur de départ en ajoutant un paramètre: **range (2, 6)**, ce qui signifie des valeurs de 2 à 6 (mais pas 6) :

```
for i in range (2,6):  
    print(i)
```

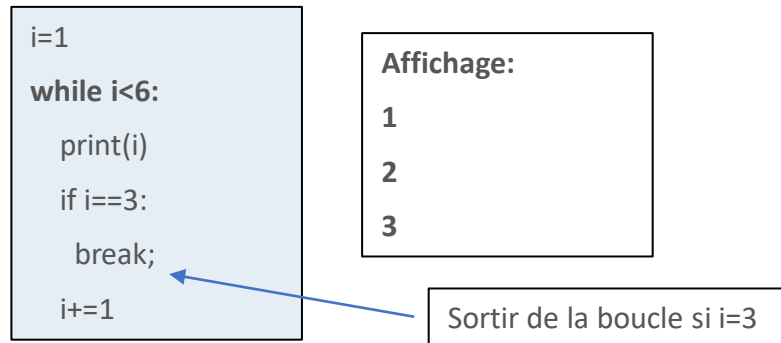
Affichage d'une séquence de nombres de 2 à 6

Affichage:

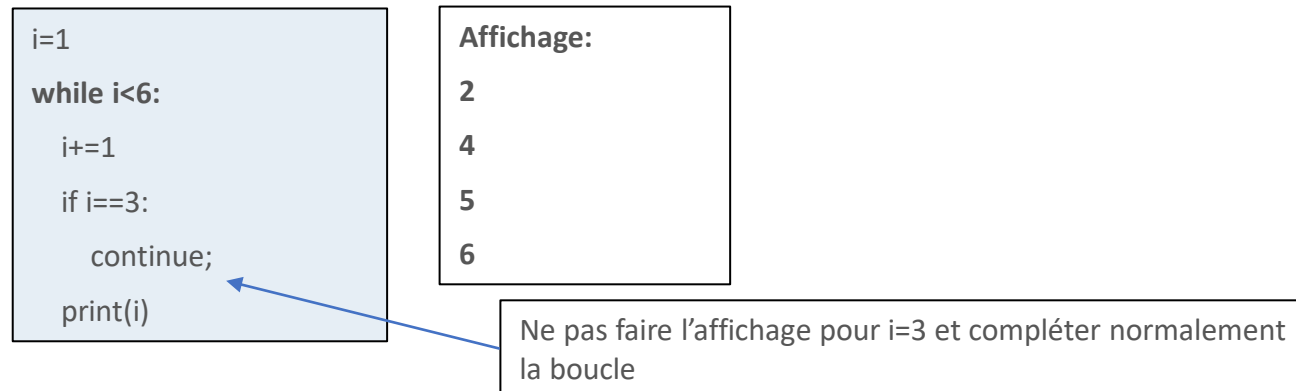
2
3
4
5

Structures itératives

- Avec l'instruction **break**, il est possible d'arrêter la boucle même si la condition de sortie n'est pas vérifiée :



- Avec l'instruction **continue**, nous pouvons arrêter l'itération en cours et continuer avec la suivante :



CHAPITRE 1

TRANSFORMER UNE SUITE D'ÉTAPES ALGORITHMIQUES EN UNE SUITE D'INSTRUCTIONS PYTHON

1. Critères de choix d'un langage de programmation

2. Blocs d'instructions

3. Conversion de l'algorithme en Python

4. Optimisation du code (bonnes pratiques de codage, commentaires, etc.)



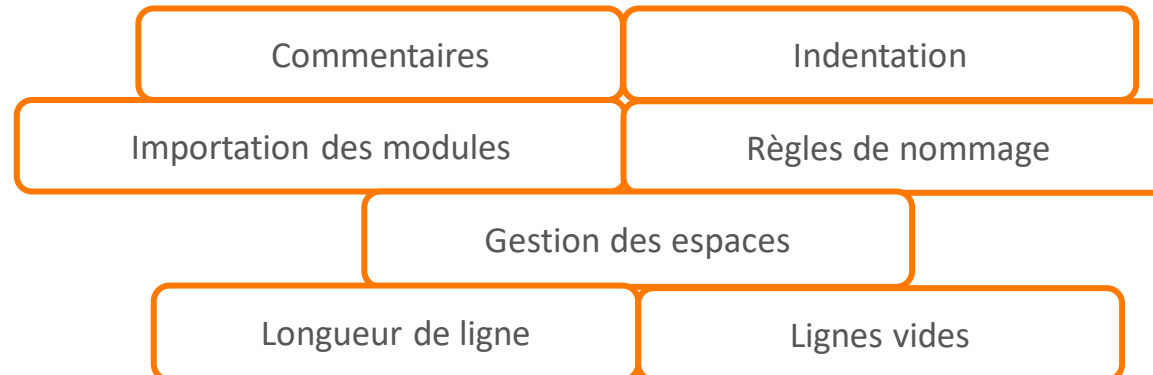
01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal

- Afin d'améliorer le langage Python, la communauté qui développe Python publie régulièrement des guides appelés **Python Enhancement Proposal (PEP)** suivis d'un numéro
- Il s'agit de propositions concrètes pour améliorer le code, ajouter de nouvelles fonctionnalités, mais aussi des recommandations sur la manière d'utiliser Python, bien écrire du code, etc.
- On parle de code **Pythonique** lorsque ce dernier respecte les règles d'écriture définies par la communauté Python mais aussi les règles d'usage du langage
- Le guide de style PEP 8 est une des plus anciennes versions de PEP (les numéros sont croissants avec le temps). Elle consiste en un nombre important de recommandations sur la syntaxe de Python.
- Quelques concepts de PEP :



01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal

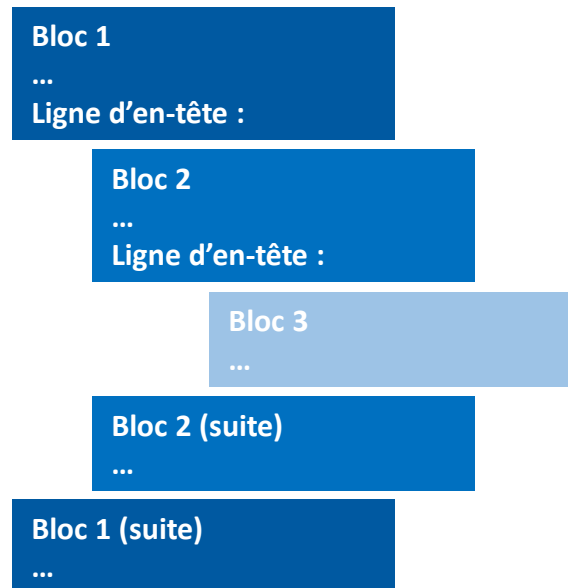
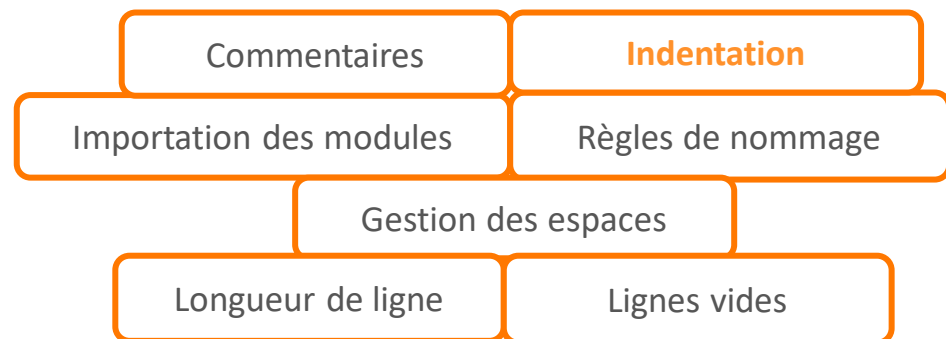


- Les commentaires débutent toujours par le **symbole #** suivi d'une espace
- Les commentaires donnent des explications claires sur l'utilité du code et doivent être synchronisés avec le code, c'est-à-dire que si le code est modifié, les commentaires doivent l'être aussi (le cas échéant)
- Les commentaires sont sur le même niveau d'indentation que le code qu'ils commentent
- Les commentaires sont constitués de phrases complètes, avec une majuscule au début (sauf si le premier mot est une variable qui s'écrit sans majuscule) et un point à la fin
- PEP 8 recommande la cohérence entre la langue utilisée pour les commentaires et la langue utilisée pour nommer les variables. Pour un programme scientifique, les commentaires et les noms de variables sont en anglais

```
print ("Suite de Fibonacci")
a,b,c=1,1,1      # a et b servent au calcul des termes successifs
                  # c est un simple compteur
print(b)         # affichage du premier terme
while c<15:      # nous affichons 15 termes au total
    a,b,c=a,a+b,c+1
    print(b)
```

Python Enhancement Proposal

- L'indentation est obligatoire en Python pour séparer les blocs d'instructions
- Cela vient d'un constat simple : l'indentation améliore la lisibilité d'un code
- Dans la PEP 8, la recommandation pour la syntaxe de chaque niveau d'indentation est très simple : 4 espaces



Indentation pour la séparation les blocs d'instructions.

01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal



- Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou librairies). Ce sont des « boîtes à outils » qui vont vous être très utiles
- L'utilisation de la syntaxe **import module** permet d'importer tout une série de fonctions organisées par « thèmes »

Exemple :

Les fonctions gérant les nombres aléatoires avec **random** et les fonctions mathématiques avec **math**. Python possède de nombreux autres modules internes (c'est-à-dire présents de base lorsqu'on installe Python).

```
import math  
print(math.cos(math.pi/2))  
print(math.sin(math.pi/2))
```

01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal



- Les noms de variables, de fonctions et de modules doivent être en minuscules avec un caractère « souligné » (« tiret du bas » ou « underscore » en anglais) pour séparer les différents « mots » dans le nom.

```
ma_variable  
fonction_test_27()  
mon_module
```

- Les constantes sont écrites en majuscules :

```
MA_CONSTANTE  
VITESSE_LUMIERE
```


01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal



La PEP 8 recommande d'entourer les opérateurs (+, -, /, *, ==, !=, >=, not, in, and, or. . .) d'un espace avant et d'un espace après

Exemple

```
# code recommandé :  
ma_variable = 3 + 7  
mon_texte = "souris "  
mon_texte == ma_variable
```

01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



WEBFORCE
BE THE CHANGE

Python Enhancement Proposal



- Une ligne de code ne doit pas dépasser 79 caractères
- Le caractère \ permet de couper des lignes trop longues :

```
print("ATGCGTACAGTATCGATAAC« \
...   "ATGACTGCTACGATCGGATA« \
...   "CGGGTAACGCCATGTACATT")
```

- À l'intérieur d'une parenthèse, on peut revenir à la ligne sans utiliser le caractère \. C'est particulièrement utile pour préciser les arguments d'une fonction ou d'une méthode lors de sa création ou lors de son utilisation :

```
ma_variable = 3
if (ma_variable > 1 and ma_variable < 10
... and ma_variable % 2 == 1 and ma_variable % 3 == 0):
...   print(f"ma variable vaut {ma_variable}")
```

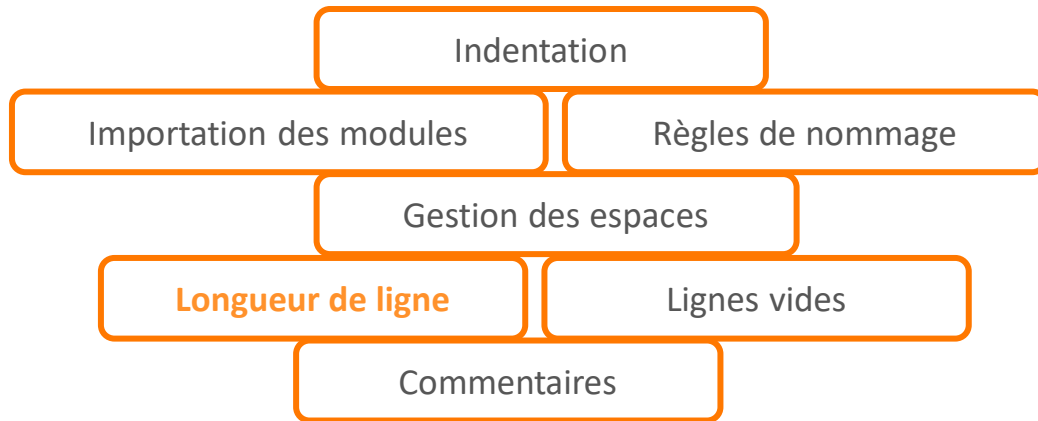
01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



WEBFORCE
BE THE CHANGE

Python Enhancement Proposal



- Les parenthèses sont également très pratiques pour répartir sur plusieurs lignes une chaîne de caractères qui sera affichée sur une seule ligne :

```
print("ATGCGTACAGTATCGATAAC"  
"ATGACTGCTACGATCGGATA"  
"CGGGTAACGCCATGTACATT")
```

- L'opérateur + est utilisé pour concaténer les trois chaînes de caractères et que celles-ci ne sont pas séparées par des virgules
- À partir du moment où elles sont entre parenthèses, Python les concatène automatiquement
- On peut aussi utiliser les parenthèses pour évaluer une expression trop longue :

```
ma_variable = 3  
if (ma_variable > 1 and ma_variable < 10  
and ma_variable % 2 == 1 and ma_variable % 3 == 0):  
    print(f"ma variable vaut {ma_variable}")
```

01 – PYTHON

Optimisation du code (bonnes pratiques de codage, commentaires, ...)



Python Enhancement Proposal



- Dans un script, les lignes vides sont utiles pour séparer visuellement les différentes parties du code
- Il est recommandé de laisser deux lignes vides avant la définition d'une fonction
- On peut aussi laisser une ligne vide dans le corps d'une fonction pour séparer les sections logiques de la fonction mais cela est à utiliser avec précaution

```
from math import *  
  
def maximum (a, b):  
    if a>b:  
        return a  
    else  
        return b
```

CHAPITRE 2

MANIPULER LES DONNÉES

Ce que vous allez apprendre dans ce chapitre :

- Manipuler les fonctions et les fonctions lambda en Python
- Maîtriser les structures de données Python et les différencier
- Maîtriser la manipulation des différents types de fichiers de données
- Connaître les principales bibliothèques standards de Python



28 heures



CHAPITRE 2

MANIPULER LES DONNÉES

1. Manipulation des fonctions lambda

2. Listes, tuples, dictionnaires, ensembles (set)

3. Fichiers de données

4. Bibliothèques standards



02 – MANIPULER LES DONNÉES

Manipulation des fonctions lambda



Manipulation des fonctions

- Une fonction est un bloc de code qui ne s'exécute que lorsqu'elle est appelée :
 - Vous pouvez transmettre des données, appelées paramètres, à une fonction
 - Une fonction peut renvoyer des données en conséquence.
 - Pour appeler une fonction, utilisez le nom de la fonction suivi de parenthèses
- Les informations peuvent être transmises aux fonctions comme paramètres/ arguments :
 - Les arguments sont spécifiés après le nom de la fonction, entre parenthèses
 - Vous pouvez ajouter autant d'arguments que vous le souhaitez, séparez-les simplement par une virgule
 - Lorsqu'on définit une fonction `def fct(x, y)`: les arguments `x` et `y` sont **appelés arguments positionnels**

```
def my_function():  
    print("Hello from a function")  
  
my_function()
```

Affichage:
Hello from a function

Déclaration de la fonction
`my_function()`

Appel de `my_function()`

```
def my_function(fname):  
    print(fname + " Refsnes")
```

Affichage:
Emil Refsnes
Tobias Refsnes

```
# appel avec fname=Emil  
my_function("Emil")  
  
# appel avec fname= Tobias  
my_function("Tobias")
```

02 – MANIPULER LES DONNÉES

Manipulation des fonctions lambda



Manipulation des fonctions

- Une fonction peut retourner une valeur

```
def my_function(x):  
    return 5 * x  
  
print(my_function(3))  # appel avec x =3  
print(my_function(5))  # appel avec x =5  
print(my_function(9))  # appel avec x =9
```

Affichage:

15
25
45

- Il est possible de donner une valeur par défaut à un paramètre d'une fonction
- Un argument défini avec une syntaxe `def fct(arg=val)`: est **appelé argument par mot-clé**

```
def my_function(country = "Norway"):  
    print("I am from " + country)  
  
my_function("Sweden")  # appel avec country =Sweden  
my_function("India")    # appel avec country =India  
my_function()           # appel avec country =Norway (valeur par défaut)
```

Affichage:

I am from Sweden
I am from India
I am from Norway

02 – MANIPULER LES DONNÉES

Manipulation des fonctions lambda



Manipulation des fonctions

- Il est bien sûr possible de passer plusieurs arguments par mot-clé :

```
def fct(x=0, y=0, z=0):  
    return x, y, z  
  
print (fct())           # appel de la fonction avec x=0, y=0,z=0  
print (fct(10))         # appel de la fonction avec x=10, y=0,z=0  
print (fct(10,8))       # appel de la fonction avec x=10, y=8,z=0  
print (fct(10,8,3))     # appel de la fonction avec x=10, y=8,z=3
```

Affichage:

```
(0, 0, 0)  
(10, 0, 0)  
(10, 8, 0)  
(10, 8, 3)
```

- Il est possible de préciser le nom de l'argument lors de l'appel :

```
fct(z=10)               # appel de la fonction avec x=0, y=0,z=10  
fct(z=10, x=3, y=80)    # appel de la fonction avec x=3, y=80,z=10  
fct(z=10, y=80)         # appel de la fonction avec x=0, y=80,z=10
```

Affichage:

```
(0, 0, 10)  
(3, 80, 10)  
(0, 80, 10)
```

02 – MANIPULER LES DONNÉES

Manipulation des fonctions lambda



Manipulation des fonctions

- Il est possible de faire un mélange d'arguments positionnels et par mot-clé. Ainsi **les arguments positionnels doivent toujours être placés avant les arguments par mot-clé** :

```
def fct(a, b, x=0, y=0, z=0):  
    return a, b, x, y, z  
  
print( fct(1, 1))           #appel de la fonction avec a=b=1 et x=y=z=0  
print( fct(1, 1, z=5))      #appel de la fonction avec a=b=1, x=y=0 et z=5  
print( fct(1, 1, z=5, y=32)) #appel de la fonction avec a=b=1, x=0, y=32 et z=5
```

Affichage:

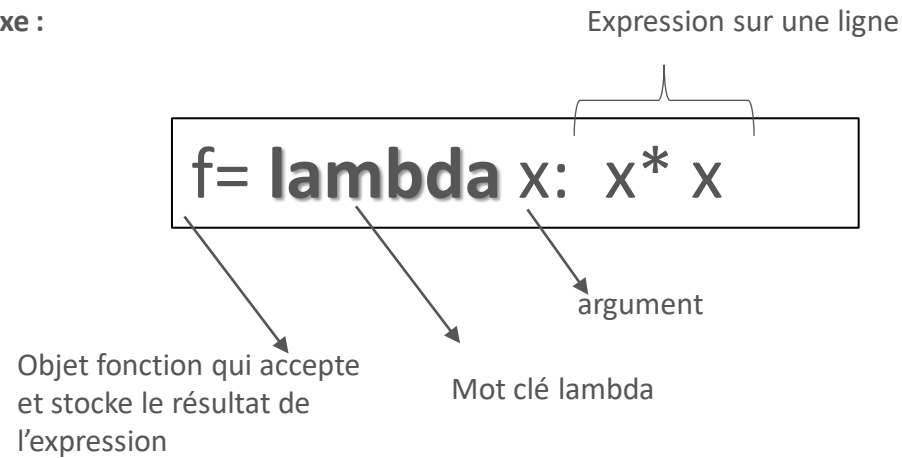
```
(1, 1, 0, 0, 0)  
(1, 1, 0, 0, 5)  
(1, 1, 0, 32, 5)
```

Fonction Lambda

En Python, le mot clé **lambda** est utilisé pour déclarer une fonction **anonyme** (sans nom), raison pour laquelle ces fonctions sont appelées « **fonction lambda** » :

- Une fonction lambda est comme n'importe quelle fonction Python normale, sauf **qu'elle n'a pas de nom lors de sa définition** et qu'elle est **contenue dans une ligne**
- Tout comme la définition d'une fonction normale par l'utilisateur à l'aide du mot clé « **def** », une fonction lambda est définie à l'aide du mot clé « **lambda** »
- Une fonction lambda peut avoir n nombre d'arguments mais une seule expression

Syntaxe :



02 – MANIPULER LES DONNÉES

Manipulation des fonctions lambda



Fonction Lambda

Exemple:

- Une fonction lambda qui ajoute 10 au nombre passé en argument et affiche le résultat :

```
x= lambda a: a+10  
print(x(5)) #affiche 15
```

- Une définition de fonction qui prend un argument et cet argument sera multiplié par un nombre inconnu :

```
def myfunc(n):  
    return lambda a: a*n  
  
mydouble= myfunc(2)  
print(mydouble(11)) #affiche 22
```

CHAPITRE 2

MANIPULER LES DONNÉES

1. Manipulation des fonctions lambda

2. Listes, tuples, dictionnaires, ensembles (set)

3. Fichiers de données

4. Bibliothèques standards



02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques (Liste)

- Une **liste** est une **collection** qui est **ordonnée, modifiable et** qui peuvent contenir plusieurs fois la même valeur
- Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets

Liste= [el1, elt2,...eln]

- **Liste[i]**: permet d'accéder à l'élément de la liste se trouvant à la i^{ème} position

```
thislist= ["apple","banana","cherry"]    #déclarer la liste
print(thislist)                          #afficher la liste
print(thislist[1])                       #afficher le premier élément de la liste
print(thislist[-1])                      #afficher la valeur de la position -1 (cycle)
thislist1= ["apple","banana","cherry","orange","kiwi","melon","mango"]
print(thislist1[2:5])                    #afficher les valeurs de la position 2 jusqu'à 5 (5 non inclus)
```

- **Liste[i]= val**: permet de changer la valeur de l'élément de la liste se trouvant à la position i

```
thislist= ["apple","banana","cherry"]
thislist[1] ="blackcurrant"             #modifier la valeur de l'élément à la 1ère position
print(thislist)
```

Affichage:
['apple', 'blackcurrant', 'cherry']

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques (Liste)

- Parcourir une liste est possible en utilisant l'instruction **for elem in liste** où **elem** est l'élément de la liste **liste**

```
thislist= ["apple", "banana", "cherry"]  
for x in thislist:  #parcourir des éléments de la liste  
    print(x)        #afficher la valeur de x qui correspond à un élément de la liste
```

- Vérifier si un élément **elem** est dans une liste **Liste** est possible en utilisant l'instruction : **if elem in Liste**

```
thislist= ["apple", "banana", "cherry"]  
if "apple" in thislist:  #vérifier si une chaîne est un élément de la liste  
    print("Yes, 'apple' is in the fruits list")
```

- **Fonction len(Liste)** est une fonction permettant de retourner la longueur de la liste **Liste**

```
thislist= ["apple", "banana", "cherry"]  
print(len(thislist))  # afficher la longueur d'une liste
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques (Liste)

- **Fonction append(élément)** est une fonction qui permet d'ajouter un élément **élément** à la fin de liste

```
thislist= ["apple", "banana", "cherry"]  
thislist.append("orange") # ajouter l'élément "orange" à la fin de la liste  
print(thislist)          # afficher ['apple', 'banana', 'cherry', 'orange']
```

- **Fonction insert(pos, élément)** est une fonction qui permet d'ajouter un élément **élément** à une position **pos** de la liste

```
thislist= ["apple", "banana", "cherry"]  
thislist.insert(1, "orange") # insérer l'élément "orange" à la deuxième position  
print(thislist)             # afficher les éléments de la liste ["apple", "orange", "banana", "cherry"]
```

- **Fonction pop()** est une fonction qui assure la suppression du dernier élément de la liste

```
thislist= ["apple", "banana", "cherry"]  
thislist.pop() #supprimer le dernier élément de la liste qui est cherry  
print(thislist) # afficher les éléments de la liste ["apple", "banana"]
```


02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques(Liste)

- **Fonction del(élément)** est une fonction qui permet de supprimer un élément particulier **élément** dans une liste

```
thislist= ["apple", "banana", "cherry"]  
delthislist[0]  #supprimer l'élément se trouvant à la première position  
print(thislist) # affiche ["banana", "cherry"]
```

- **Fonction extend(Liste)** est une fonction qui permet de fusionner deux listes

```
list1 = ["a", "b", "c"]  
list2 = [1, 2, 3]  
list1.extend(list2)    #fusionner le deux Listes Liste1 et Liste2  
print(list1)          #affiche ['a', 'b', 'c', 1, 2, 3]
```

- **Fonction copy()** est une fonction qui permet de copier le contenu d'une liste dans une autre

```
thislist= ["apple", "banana", "cherry"]  
mylist= thislist.copy()  
print(mylist) #affiche ["apple", "banana", "cherry"]
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques (Liste)

- **Fonction clear()** est une fonction qui permet de supprimer tous les éléments d'une liste

```
thislist= ["apple", "banana", "cherry"]  
thislist.clear()  #vider la liste  
print(thislist)  #afficher une liste vide []
```

- **Fonction reverse()** est une fonction qui permet d'inverser une liste

```
thislist= ["apple", "banana", "cherry"]  
thislist.reverse() #inverser la liste thislist  
print(thislist)    #afficher la liste inversée ['cherry', 'banana', 'apple']
```

- **Fonction sort()** est une fonction qui permet de trier une liste

```
thislist= ["banana", "cherry", "apple"]  
thislist.sort() #trier la liste thislist  
print(thislist) #afficher la liste triée ['apple', 'banana', 'cherry']
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Tableaux dynamiques (Liste)

Fonction **remove()** est une fonction qui permet de supprimer un élément spécifique de la liste

```
thislist= ["banana", "cherry","apple"]  
thislist.remove("banana") #supprimer l'élément banana  
print(thislist)    #afficher la liste ['cherry', 'apple']
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux dynamiques (Liste)

- Une chaîne de caractères est manipulée comme une liste de caractères
- Les opérateurs de concaténation (+) et de répétition (*)
 - L'opérateur (+) permet de concaténer une chaîne de caractères ou plus
 - L'opérateur (*) permet de définir le nombre de répétitions d'une chaîne

```
s1="welcome"
```

```
s2="Python"
```

```
s3= s1 + " to " + s2
```

```
print(s3)
```

```
s4 =3 * s1
```

```
print (s4)
```

Concaténation de s1, « to » et s2

Répétition de s1 3 fois

Affichage:
welcome to Python
welcomewelcomewelcome

- Les opérateurs in et not in

```
s1="welcome"
```

```
print("come" in s1)
```

```
print("come" not in s1)
```

Vérifier si la chaîne « come » appartient à s1

Vérifier si la chaîne « come » n'appartient pas à s1

Affichage:
True
False

- Création des chaînes de caractères en utilisant le mot clé str

```
s1=str()
```

```
s2=str("welcome")
```

Créer une chaîne de caractère vide

Créer une chaîne de caractère contenant « Welcome »

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Manipulation des chaînes de caractères

- Fonctions sur les chaînes de caractères:

- **Len()**: retourne la longueur d'une chaîne de caractères
- **Max()**: retourne le caractère inclus dans la chaîne ayant la plus grande valeur ASCII
- **Min()**: retourne le caractère inclus dans la chaîne ayant la plus petite valeur ASCII

```
s1="welcome"  
print(len(s1))  
print(max(s1))  
print(min(s1))
```

Affichage:

7
w
c

Longueur de s1=7

W a la plus grande valeur ASCII
C a la plus petite valeur ASCII

- L'opérateur indice []:

- Une chaîne de caractères est une séquence de caractères.
- Un caractère de la chaîne est accessible par l'opérateur indice []
- Le premier caractère a l'indice 0

```
s1="welcome"  
print(s1[3])
```

Affichage:
c

Affichage de la lettre à l'indice 3 donc à la 4^{ème} position

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Manipulation des chaînes de caractères

- Comparaison de chaînes de caractères :
 - La comparaison des caractères un par un selon leurs code ASCII

```
print("green" <= "glow")
```

Affichage:

False

Green est supérieur à « glow » car ASCII de la lettre r est supérieur à ASCII de la lettre l

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux statiques (tuple)

- Un **tuple** est une collection **ordonnée**, non **modifiable** (contrairement à la liste qui est modifiable) et qui peuvent contenir plusieurs fois la même valeur
 - En Python, les tuples ont écrits avec des **crochets ronds**

Tuple= (el1, elt2,...eln)

```
thistuple= ("apple","banana","cherry")
print(thistuple)    #affiche ('apple', 'banana', 'cherry')
print(thistuple[1]) #accéder au deuxième élément du tuple
                   #affiche banana
```

- La fonction **list(Tuple)** permet de convertir le tuple **Tuple** en liste pour pouvoir le modifier
- La fonction **tuple(Liste)** permet de convertir la liste **Liste** en tuple

```
x = ("apple","banana","cherry")
y =list(x)    #convertir le tuple en une liste
y[1] ="kiwi"  #changer la valeur d'un élément de la liste, c'est possible car y est une liste
x =tuple(y)   #convertir la liste en un tuple
print(x)      #affiche ('apple', 'kiwi', 'cherry')
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Tableaux statiques (tuple)

- Parcours d'un tuple :

```
thistuple= ("apple", "banana", "cherry")  
for x in thistuple:    # parcourir les éléments du tuple  
    print(x)           #afficher x l'élément du tuple  
  
                        #apple  
                        #banana  
                        #cherry
```

- Vérification si un élément est dans un tuple :

```
thistuple= ("apple", "banana", "cherry")  
if "apple" in thistuple:    #vérifier si "apple" est un élément du tuple  
    print("Yes, 'apple' is in the fruits tuple") #afficher le message si c'est le cas (oui)
```

- Fonction **del(tuple)** assure la suppression d'un tuple

```
thistuple= ("apple", "banana", "cherry")  
del(thistuple)    #supprimer le tuple  
print(thistuple)  #afficher une erreur car le tuple n'existe plus
```

Affichage:

Traceback (most recent call last):

File "C:\Users\DELL\Desktop\algo\ex1.py", line 214, in
<module> print(thistuple) #this will raise an error because
NameError: name 'thistuple' is not defined

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Tableaux statiques (tuple)

- Fusion de deux tuples avec « + »

```
tuple1 = ("a","b","c")  
tuple2 = (1,2,3)  
tuple3 = tuple1 + tuple2    #fusionner le 2 tuples tuple1 et tuple2  
print(tuple3)              #afficher ('a', 'b', 'c', 1, 2, 3) le résultat de la fusion des tuples
```

- La fonction **index(elem)** retourne la position de l'élément elem dans le tuple

```
tuple1 = ("a","b","c")  
n=tuple1.index("b")        #récupérer la position de b dans tuple1  
print(n)                   #affiche 1
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux statiques (set)

- Un set est une **collection** non **ordonnée**, non **indexée** et **non modifiables** qui n'acceptent pas de contenir plusieurs fois le même élément
- En Python, les **sets** sont écrits avec des **accolades**

`Set={elem1,elem2,.....,elemn}`

- Création d'un set :

```
thisset= {"apple", "banana", "cherry"} #déclarer un set thisset
print(thisset)                        #afficher thisset
# {'apple', 'banana', 'cherry'}
```

- Parcours d'un set :

```
thisset= {"apple", "banana", "cherry"}
for x in thisset:    #parcourir les éléments du set
print(x)            # affiche
                    #apple
                    #banana
                    #cherry
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux statiques (set)

- Vérification si **elem** est un élément dans un set **thisset** : **elem in thisset** (retourne true si oui false sinon)

```
thisset= {"apple", "banana", "cherry"}    #déclarer un set  
print("banana" in thisset)               #affiche True car banana est un élément de thisset
```

- Une fois qu'un ensemble est créé, vous ne pouvez pas modifier ses éléments (remplacer des éléments par d'autres) , mais vous pouvez ajouter de nouveaux éléments ou supprimer les éléments.
- La fonction **add()** permet d'ajoute un élément à la fin d'un set :

```
thisset= {"apple", "banana", "cherry"}  
thisset.add("orange")    #ajouter l'élément "orange" au set  
print(thisset)           #affiche {'cherry', 'banana', 'apple', 'orange'}
```

- La fonction **update()** permet l'ajout de plusieurs éléments à un set :

```
thisset= {"apple", "banana", "cherry"}  
thisset.update(["orange", "mango", "grapes"])    #ajouter des éléments au set  
print(thisset)                                   # affiche {'grapes', 'mango', 'banana', 'apple', 'cherry', 'orange'}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux statiques (set)

- Les fonctions **remove()** et **discard** permettent la suppression d'un élément d'un set :

```
thisset= {"apple", "banana", "cherry"}  
thisset.remove("banana") #supprimer banana  
print(thisset)           #affiche {'cherry', 'apple'}
```

```
thisset= {"apple", "banana", "cherry"}  
thisset.discard("banana") #supprimer " banana "  
print(thisset)           #affiche {'cherry', 'apple'}
```

- La fonction **pop()** permet la suppression d'un élément aléatoire d'un set et retourne l'élément supprimé

```
thisset= {"apple", "banana", "cherry"}  
x = thisset.pop()  
print(x)           #afficher "cherry " c'est l'élément supprimé  
print(thisset)     #affiche {"apple", "banana"}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Tableaux statiques (set)

- La fonction **clear()** assure la suppression de tous les éléments d'un set (rendre un set vide)

```
thisset= {"apple", "banana", "cherry"}  
thisset.clear()  #vider le set  
print(thisset)  #affiche set()
```

- La fonction **del()** assure la suppression d'un set

```
thisset= {"apple", "banana", "cherry"}  
del(thisset)  #supprime le set thisset  
print(thisset)  #affiche une erreur car le set thisset n'existe plus
```

Affichage:

Traceback (most recent call last):

```
File "C:\Users\DELL\Desktop\algo\ex1.py", line 213, in  
<module> print(thisset)  
NameError: name 'thisset' is not defined
```

- Les fonction **union()** et **update()** assurent la fusion de deux sets

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
set3 = set1.union(set2)  #fusionner set1 et set2  
print(set3)              # affiche {1, 2, 3, 'c', 'b', 'a'}
```

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
set1.update(set2)  #fusionner set1 et set2  
print(set1)        # affiche {1, 2, 3, 'c', 'b', 'a'}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



WEBFORCE
BE THE CHANGE

Tableaux statiques (set)

Autres méthodes :

Méthode	Description
<u>copy()</u>	Retourne une copie de set
<u>difference()</u>	Retourne un set contenant la différence entre 2 ou plusieurs sets
<u>intersection()</u>	Retourne un set qui est l'intersection de deux autres sets
<u>intersection_update()</u>	Supprime les éléments d'un set qui ne sont pas présents dans d'autres sets spécifiés
<u>isdisjoint()</u>	Retourne si deux sets ont une intersection ou non
<u>issuperset()</u>	Retourne si un set contient un autre set ou non

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- Un **dictionnaire** est une collection non **ordonnée**, **modifiable** et **indexée**.
- En Python, les dictionnaires sont écrits avec des **accolades**, et ils ont des clés et des valeurs

dictionnaire={clé1: val1, clé2:val2, clé3:valn}

- **Création d'un dictionnaire :**

```
thisdict={"brand":"Ford","model":"Mustang","year":1964}    #déclaration du dictionnaire thisdict
print(thisdict)    #affichage de thisdict
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

- **Parcours d'un dictionnaire** thisdict en utilisant l'instruction **for x in thisdict** où x est un élément de thisdict

```
thisdict={"brand":"Ford","model":"Mustang","year":1964}
for x in thisdict:    #parcourir les éléments du dictionnaire
    print(thisdict[x])    #afficher les éléments du dictionnaire
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- Modification d'une valeur d'une clé particulière :

```
thisdict={"brand":"Ford","model":"Mustang","year":1964} #déclaration du dictionnaire thisdict
thisdict["year"] =2018 #modifier la valeur de la clé "year"
print(thisdict) #afficher thisdict
# {'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

- La fonction **get()** permet de retourner la valeur d'une clé particulière :

```
thisdict={"brand":"Ford","model":"Mustang","year":1964} #déclaration du dictionnaire thisdict
x = thisdict.get("model") #retourner la valeur de la clé model
print(x) #afficher Mustang
```

- La fonction **items()** permet de retourner les clés et les valeurs d'un dictionnaire à la fois

```
thisdict={"brand":"Ford","model":"Mustang","year":1964} #declaration du dictionnaire thisdict
for x, y in thisdict.items() #retourner les clés et les valeurs
    print(x,y) #afficher:
                #brand Ford
                #model Mustang
                #year 1964
```


02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- Il est possible de vérifier si une clé existe dans un dictionnaire :

```
thisdict= {"brand": "Ford",  
"model": "Mustang",  
"year": 1964}  
if "model" in thisdict: #vérifier si model est une clé dans le dictionnaire  
    print("Yes, 'model' is one of the keys in the thisdictdictionary") #afficher le message si c'est le cas
```

- La fonction **len()** permet de retourner la longueur d'un dictionnaire :

```
thisdict={"brand":"Ford","model":"Mustang","year":1964} #déclaration du dictionnaire thisdict  
print(len(thisdict)) #afficher 3 la longueur du dictionnaire
```

- Il est possible d'ajouter un élément à un dictionnaire :

```
thisdict= {"brand": "Ford",  
"model": "Mustang",  
"year": 1964}  
thisdict["color"] = "red" #ajout d'un élément (" color " , "red " ) à un dictionnaire  
print(thisdict) #afficher {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- La fonction **popitem()** permet de supprimer le dernier élément du dictionnaire :

```
thisdict= {"brand": "Ford",  
"model": "Mustang",  
"year": 1964  
}  
thisdict.popitem ()  #supprimer le dernier élément du dictionnaire  
print(thisdict)      # afficher {'brand': 'Ford', 'model': 'Mustang'}
```

- La fonction **del()** permet de supprimer d'un dictionnaire un élément ayant une clé particulière :

```
thisdict= {  
"brand": "Ford",  
"model": "Mustang",  
"year": 1964  
}  
del (thisdict["model"])  #supprimer l'élément ayant la clé "model"  
print(thisdict)          #afficher {'brand': 'Ford', 'year': 1964}
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- La fonction **pop()** permet de supprimer un élément du dictionnaire ayant une clé particulière :

```
thisdict= {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("brand") #supprimer l'élément du dictionnaire ayant la clé "brand"  
print(thisdict)      # afficher {'model': 'Mustang', 'year': 1964}
```

- La fonction **keys()** permet de retourner toutes les clés d'un dictionnaire :

```
thisdict= {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
#retourner les clés du dictionnaire  
print(thisdict.keys() ) #afficher dict_keys(['brand', 'model', 'year'])
```

02 – MANIPULER LES DONNÉES

Listes, tuples, dictionnaires, ensembles (set)



Dictionnaires

- La fonction **values()** permet de retourner toutes les valeurs d'un dictionnaire :

```
thisdict= {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964}  
print(thisdict.values() )  #afficher dict_values(['Ford', 'Mustang', 1964])
```

- La fonction **copy()** permet de copier un dictionnaire dans un autre :

```
thisdict= {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict= thisdict.copy()  #copier le dictionnaire thisdict dans le dictionnaire mydict  
print(mydict)           #afficher: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

CHAPITRE 2

MANIPULER LES DONNÉES

1. Manipulation des fonctions lambda
2. Listes, tuples, dictionnaires, ensembles (set)
- 3. Fichiers de données**
4. Bibliothèques standards



Utilisation des fichiers

- Python a plusieurs fonctions pour **créer**, **lire**, **mettre à jour** et **supprimer** des fichiers texte.
- La fonction clé pour travailler avec des fichiers en Python est **open()**.
 - Elle prend deux paramètres; **nom de fichier** et **mode**
 - Il existe quatre méthodes (modes) différentes pour ouvrir un fichier :
 - **"r"** -Lecture -Par défaut. Ouvre un fichier en lecture, erreur si le fichier n'existe pas
 - **"a"** -Ajouter -Ouvre un fichier à ajouter du texte dedans, crée le fichier s'il n'existe pas
 - **"w"** -Écrire -Ouvre un fichier pour l'écriture, crée le fichier s'il n'existe pas
 - **"x"** -Créer -Crée le fichier spécifié, renvoie une erreur si le fichier existe
- **Ouvrir et lire un fichier :**
 - Pour ouvrir le fichier, utilisez la **fonction open()** intégrée
 - La fonction **open()** renvoie un objet fichier, qui a une méthode **read()** pour lire le contenu du fichier

```
f=open("demofile.txt","r")  
print(f.read())
```

- La fonction **read(n)** retourne les n premiers caractères du fichier :

```
f = open("demofile.txt", "r")  
print(f.read(5))
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Utilisation des fichiers

- La fonction **readline()** retourne une seule ligne du fichier :

```
f=open("demofile.txt","r")  
print(f.readline())
```

- Ecriture dans un fichier :
 - Pour écrire dans un fichier existant, vous devez ajouter un paramètre à la fonction **open()**:

```
f=open("demofile2.txt","a") #ouvrir le fichier en mode ajout c'est à dire il est possible d'ajouter du texte dedans  
f.write("Le fichier contient plus de texte") #ajouter la phrase "Le fichier contient plus de texte"  
f.close() #fermer le fichier  
f=open("demofile3.txt","w") #ouvrir le fichier en mode écriture en effaçant son ancien contenu  
f.write("domage!! l'ancien contenu est supprimé") #écrire la phrase domage!! l'ancien contenu est supprimé  
f.close() #fermer le fichier  
f=open("demofile3.txt","r")  
print(f.read()) # ouvrir et lire le fichier après l'ajout
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Utilisation des fichiers

- La fonction **close()** permet de fermer un fichier
 - Il est recommandé de toujours fermer le fichier lorsque vous en avez terminé

```
f=open("demofile.txt","r") #ouvrir le fichier en mode lecture  
print(f.readline()) #lire une ligne du fichier  
f.close() #fermer le fichier
```

- La fonction **os.remove()** permet de supprimer un fichier
 - Pour supprimer un fichier, vous devez importer le module OS et exécuter sa fonction **os.remove ()**

```
import os #importer la bibliothèque os  
os.remove("demofile.txt") #supprimer un fichier
```


Format CSV

- Le fichier **Comma-separated values (CSV)** est un format permettant de stocker des tableaux dans un fichier texte. Chaque ligne est représentée par une ligne de texte et chaque colonne est séparée par un **séparateur** (virgule, point-virgule, etc)
- Les champs texte peuvent également être délimités par des guillemets
- Lorsqu'un champ contient lui-même des guillemets, ils sont doublés afin de ne pas être considérés comme début ou fin du champ
- Si un champ contient un signe pouvant être utilisé comme séparateur de colonne (virgule, point-virgule, etc) ou comme séparateur de ligne, les guillemets sont donc obligatoires afin que ce signe ne soit pas confondu avec un séparateur

Données sous la forme d'un tableau:

Nom	Prénom	Age
Dubois	Marie	29
Duval	Julien "Paul"	47
Jacquet	Bernard	51
Martin	Lucie;Clara	14

Données sous la forme d'un fichier CSV:

```
Nom;Prénom;Age
"Dubois";"Marie";29
"D« ;"Julien ""Paul""";47
Jacquet;Bernard;51
Martin;"Lucie;Clara";14
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Format CSV

- Le module **CSV** de Python permet de simplifier l'utilisation des fichiers CSV
- **Lecture d'un fichier CSV :**
 - La fonction **reader()** permet de lire un fichier CSV.
 - Il faut ouvrir un flux de lecture de fichier et ouvrir à partir de ce flux un lecteur CSV.

```
import csv #importer la bibliothèque csv
fichier = open("exempleCSV.csv", "r")
lecteurCSV = csv.reader(fichier, delimiter=" , ") # Ouverture du lecteur CSV en lui fournissant le caractère séparateur (ici " , ")
for ligne in lecteurCSV:      # parcourir les lignes du fichier CSV
    print(ligne)              # afficher chaque ligne du fichier CSV
fichier.close() #fermer le fichier CSV
```

Affichage:

```
['Nom ', 'Prenom', 'Age']
['Dubois', 'Marie', '29']
['Duval', 'Julien ', '47']
['Jacquet', 'Bernard', '51']
['Martin', 'Lucie', '14']
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Format CSV

Il est également possible de lire les données et obtenir un dictionnaire par ligne contenant les données en utilisant **DictReader** au lieu de **reader**.

```
import csv #importer la bibliothèque csv
fichier = open("noms.csv", "r")
lecteurCSV = csv.DictReader(fichier, delimiter=";")
for ligne in lecteurCSV:
    print(ligne)
fichier.close()
```

Affichage:

```
{'Nom ': 'Dubois', 'Prenom': 'Marie', 'Age': '29'}
{'Nom ': 'Duval', 'Prenom': 'Julien', 'Age': '47'}
{'Nom ': 'Jacquet', 'Prenom': 'Bernard', 'Age': '51'}
{'Nom ': 'Martin', 'Prenom': 'Lucie', 'Age': '14'}
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Format CSV

- **Écriture dans un fichier CSV :**
 - À l'instar de la lecture, on ouvre un flux d'écriture (fonction **writer()**) et on ouvre un écrivain CSV à partir de ce flux (fonction **writerow()**) :

```
import csv #importer la bibliothèque csv
fichier = open("annuaire.csv", "w") #ouvrir un fichier en mode écriture
ecrivainCSV = csv.writer(fichier,delimiter=";") #ouvrir un flux d'écriture
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) #écrire une 1ère ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Dubois","Marie","0198546372"]) #écrire une 2ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Duval","Julien","0399741052"]) #écrire une 3ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"]) #écrire une 4ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Martin","Julie","0399731590"]) #écrire une 5ème ligne dans le fichier annuaire.csv
fichier.close() #Fermer le fichier
```

Génération du fichier annuaire.csv

Nom	Prénom	Téléphone
Dubois	Marie	198546372
Duval	Julien	399741052
Jacquet	Bernard	200749685
Martin	Julie	399731590

02 – MANIPULER LES DONNÉES

Fichiers de données



Format CSV

- Il est également possible d'écrire le fichier en fournissant un dictionnaire par ligne à condition que chaque dictionnaire possède les mêmes clés
- La fonction **DictWriter()** produit les lignes de sortie depuis des dictionnaires . Il faut également fournir la liste des clés des dictionnaires avec l'argument **fieldnames**

```
import csv #importer la bibliothèque csv
```

```
bonCommande = [
```

```
    {"produit": "cahier", "reference": "F452CP", "quantite": 41, "prixUnitaire": 1.6},
```

```
    {"produit": "stylo bleu", "reference": "D857BL", "quantite": 18, "prixUnitaire": 0.95},
```

```
    {"produit": "stylo noir", "reference": "D857NO", "quantite": 18, "prixUnitaire": 0.95},
```

```
    {"produit": "équerre", "reference": "GF955K", "quantite": 4, "prixUnitaire": 5.10},
```

```
    {"produit": "compas", "reference": "RT42AX", "quantite": 13, "prixUnitaire": 5.25}] #déclarer des dictionnaires
```

```
fichier = open("bon-commande.csv", "w") #ouvrir un fichier en écrire
```

```
ecrivainCSV = csv.DictWriter(fichier, delimiter=";", fieldnames=bonCommande[0].keys()) #produire des lignes de sortie depuis les dictionnaires,  
# fieldnames contient les clés des dictionnaires
```

```
ecrivainCSV.writeheader() # écrire la ligne d'en-tête avec le titre des colonnes
```

```
for ligne in bonCommande: # parcourir les dictionnaires
```

```
    ecrivainCSV.writerow(ligne) #Ecrire une ligne dans le fichier
```

```
fichier.close()
```

Génération du fichier bon-commande.csv

```
Produit;reference;quantite;prixUnitaire
```

```
cahier;F452CP;41;1.6
```

```
stylo bleu;D857BL;18;0.95
```

```
stylo noir;D857NO;18;0.95
```

```
équerre ;GF955K;4;5.1
```

```
compas ;RT42AX;13;5.25
```

Format JSON

- Le format **JavaScript Object Notation (JSON)** est issu de la notation des objets dans le langage JavaScript
- Les principaux avantages de **JSON**:
 - il est simple à mettre en œuvre par un développeur
 - Un format ouvert qui ne dépend d'aucun langage de programmation
 - Assure un stockage de données de différents types (booléens, entiers, chaînes de caractères, etc)
- Il ne comporte que des associations **clés → valeurs** (à l'instar des dictionnaires)
- Une valeur peut être une autre association clés → valeurs, une liste de valeurs, un entier, un nombre réel, une chaîne de caractères, un booléen ou une valeur nulle
- Sa syntaxe est similaire à celle des dictionnaires Python

Exemple de fichier JSON :

- Définition d'un menu: il s'agit d'un objet composé de membres qui sont un attribut (Fichier) et un tableau (commandes) qui contient d'autres objets: les lignes du menu. Une ligne de menu est identifiée par son titre et son action

```
{  
  "menu": "Fichier",  
  "commandes": [  
    {  
      "titre": "Nouveau",  
      "action": "CreateDoc"  
    },  
    {  
      "titre": "Ouvrir",  
      "action": "OpenDoc"  
    },  
    {  
      "titre": "Fermer",  
      "action": "CloseDoc"  
    }  
  ]  
}
```

02 – MANIPULER LES DONNÉES

Fichiers de données



Format JSON

- Lire un fichier JSON :

- La fonction **loads (texteJSON)** permet de décoder le texte JSON passé en argument et de le transformer en dictionnaire ou une liste :

```
import json                #importer la bibliothèque json
fichier = open ( "exemple.json", "r" ) #ouvrir le fichier exemple.json en lecture
x=json.loads(fichier.read( ))        #décoder le texte et le transformer en dictionnaire.
print(x)
```

Affichage:

```
{'menu': 'Fichier', 'commandes': [{'titre': 'Nouveau', 'action': 'CreateDoc'}, {'titre': 'Ouvrir', 'action': 'OpenDoc'}, {'titre': 'Fermer', 'action': 'CloseDoc'}]}
```

- Écrire un fichier JSON :

- la fonction **dumps(variable)** transforme un dictionnaire ou une liste en texte JSON en fournissant en argument la variable à transformer :

```
import json                #importer la bibliothèque json
quantiteFournitures= {" cahiers":134, "stylos":{"rouge":41,"bleu":74},"gommes":85} #déclarer un dictionnaire
fichier=open ("quantiteFournitures.json","w") #ouvrir un fichier en écriture
fichier.write(json.dumps(quantiteFournitures)) #transformer le dictionnaire en texte json
fichier.close() #fermer le fichier
```

CHAPITRE 2

MANIPULER LES DONNÉES

1. Manipulation des fonctions lambda
2. Listes, tuples, dictionnaires, ensembles (set)
3. Fichiers de données
- 4. Bibliothèques standards**



02 – MANIPULER LES DONNÉES

Bibliothèques standards



Bibliothèques standards

- Python dispose d'une très riche bibliothèque de modules (**classes (types d'objets), fonctions, constantes, etc**) étendant les capacités du langage dans de nombreux domaines : nouveaux types de données, interactions avec le système, gestion des fichiers et des processus, protocoles de communication (internet, mail, FTP, etc.), multimédia, etc.
- Certains des modules importants sont :
 - **math** pour les utilitaires mathématiques
 - **Statistics** pour le calcul de valeurs statistiques sur des données numériques
 - **re** pour les expressions régulières
 - **Json** pour travailler avec JSON
 - **Datetime** pour travailler avec des dates

Le module Math :

- Le module math fournit un accès à de nombreuses fonctions permettant de réaliser des opérations mathématiques comme le calcul d'un sinus, cosinus, d'une tangente, d'un logarithme ou d'une exponentielle
- Les fonctions les plus couramment utilisées sont les suivantes :
 - **ceil()** et **floor()** renvoient l'arrondi du nombre passé en argument en arrondissant respectivement à l'entier supérieur et inférieur ;
 - **fabs()** renvoie la valeur absolue d'un nombre passé en argument ;
 - **isnan()** renvoie True si le nombre passé en argument est NaN = Not a Number (pas un nombre en français) ou False sinon ;
 - **exp()** permet de calculer des exponentielles.

02 – MANIPULER LES DONNÉES

Bibliothèques standards



Bibliothèques standards

- **log()** permet de calculer des logarithmes
- La fonction **sqrt()** permet de calculer la racine carrée d'un nombre
- **cos()**, **sin()** et **tan()** permettent de calculer des cosinus, sinus et tangentes et renvoient des valeurs en radians
- les fonctions de ce module ne peuvent pas être utilisées avec des nombres complexes. Pour cela, il faudra plutôt utiliser les fonctions du module **cmath**
- Le module **math** définit également des constantes mathématiques utiles comme pi ou le nombre de Neper, accessibles via **math.pi** et **math.e**

```
import math #importer la bibliothèque math
#math.ceil() calcule l'arrondi entier supérieur
print(math.ceil(3.1)) # affiche 4
print(math.ceil(2.9)) # affiche 3
#math.floor() calcule l'arrondi entier inférieur
print(math.floor(3.1)) # affiche 3
print(math.floor(-4)) # affiche -4
print(math.pi)      #affiche 3.141592653589793
```

02 – MANIPULER LES DONNÉES

Bibliothèques standards



Bibliothèques standards

Le module **random** :

- **random** fournit des outils pour générer des nombres pseudo-aléatoires de différentes façons
- La fonction **random()** est la plus utilisée du module. Elle génère un nombre à virgule flottante aléatoire de façon uniforme dans la plage semi-ouverte [0.0, 1.0)
- La fonction **uniform()** génère un nombre à virgule flottante aléatoire compris dans un intervalle. Cette fonction a deux arguments : le premier nombre représente la borne basse de l'intervalle tandis que le second représente la borne supérieure

```
import random #importer la bibliothèque random  
print(random.random()) #générer un nombre aléatoire entre 0 et 1  
print(random.uniform(10,100)) #générer un nombre aléatoire entre 10 et 100
```

PARTIE 4

DÉPLOYER LA SOLUTION PYTHON

Dans ce module, vous allez :

- Déboguer une solution Python
- Déployer une solution Python



6 heures

CHAPITRE 1

DÉBOGUER LE CODE PYTHON

Ce que vous allez apprendre dans ce chapitre :

- Différencier entre les erreurs de syntaxe et celles de compilation
- Maîtriser la gestion des exceptions en Python
- Maîtriser la manipulation de l'outil de débogage de Python



3 heures



CHAPITRE 1

DÉBOGUER LE CODE PYTHON

1. Gestion des erreurs (compilation-syntaxe)

2. Débogage

3. Outils de suivi et de visualisation de l'exécution d'un code Python



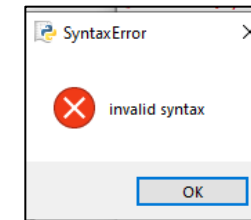
compilation-syntaxe

Erreurs de syntaxe :

- Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur
- Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme

Exemple :

```
while True print("hello word")
```



Interface signalant une erreur de syntaxe

- L'analyseur indique la ligne incriminée et affiche une petite « flèche » pointant vers le premier endroit de la ligne où l'erreur a été détectée
- L'erreur est causée par le symbole placé avant la flèche

Erreurs sémantiques ou logiques :

- Les **erreurs sémantiques** sont des erreurs relatives à la logique de la résolution du problème. Dans ce cas, le programme s'exécute sans produire des messages d'erreur, mais qui ne fait pas ce qu'il devrait

Exemple : une expression peut ne pas être évaluée dans l'ordre que vous attendiez, ce qui donne un résultat incorrect

Erreurs à l'exécution :

- Les **erreurs d'exécution** apparaissent seulement lorsque le programme fonctionne déjà, mais que des circonstances particulières se présentent
- Ces erreurs sont également appelées des exceptions,

Exemple : le programme essaie de lire un fichier qui n'existe plus

Exceptions

- Les erreurs détectées durant l'exécution sont appelées des **exceptions** et ne sont pas toujours fatales
- La plupart des exceptions toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :

```
print(10+(1/0))
```

Traceback (most recent call last):

File "C:\Users\DELL\Desktop\algo\ex1.py", line 232, in <module>

print(10+(1/0))

ZeroDivisionError: division by zero

- La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : le type indiqué dans l'exemple est **ZeroDivisionError**
- Python présente un **mécanisme d'exception** qui permet de gérer des exécutions exceptionnelles qui ne se produisent qu'en cas d'erreur à savoir l'instruction **try-except**

01 – DÉBOGUER LE CODE PYTHON

Gestion des erreurs (compilation-syntaxe)



Instruction try-except

- L'instruction **try-except** se compose de deux blocs de code.
- On place le code « risqué » dans le bloc **try** et le code à exécuter en cas d'erreur dans le bloc **except**.

Exemple :

On souhaite calculer l'âge saisi par l'utilisateur en soustrayant son année de naissance à 2016. Pour cela, il faut convertir la valeur de la variable `birthyear` en un `int`. Cette conversion peut échouer si la chaîne de caractères entrée par l'utilisateur n'est pas un nombre.

```
birthyear = input('Année de naissance ? ')
try:
    print('Tu as', 2016 - int(birthyear), 'ans.') #code risqué
except:
    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur
print('Fin du programme.')
```

01 – DÉBOGUER LE CODE PYTHON

Gestion des erreurs (compilation-syntaxe)



Instruction try-except

```
birthyear = input('Année de naissance ? ')\n\ntry:\n    print('Tu as', 2016 - int(birthyear), 'ans.') #code risqué\nexcept:\n    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur\nprint('Fin du programme.')
```

1^{er} cas:

- Si l'utilisateur entre un nombre entier, l'exécution se passe sans erreur et son âge est calculé et affiché

```
Année de naissance ? 1994\n\nTu as 22 ans.\n\nFin du programme.
```

2^{ème} cas:

- Si l'utilisateur entre une chaîne de caractères quelconque, qui ne représente pas un nombre entier, un message d'erreur est affiché

```
Année de naissance ? deux\n\nErreur, veuillez entrer un nombre.\n\nFin du programme.
```

- Dans le premier cas, la conversion s'est passée normalement, et le bloc try a donc pu s'exécuter intégralement sans erreur
- Dans le second cas, une erreur se produit dans le bloc try, lors de la conversion. L'exécution de ce bloc s'arrête donc immédiatement et passe au bloc except, avant de continuer également après l'instruction try-except

Type Exception

- Lorsqu'on utilise l'instruction **try-except**, le bloc **except** capture toutes les erreurs possibles qui peuvent survenir dans le bloc try correspondant.
- Une **exception** est en fait représentée par un objet, instance de la classe Exception.
- On peut récupérer cet objet en précisant un nom de variable après **except**

```
try:
    a = int(input('a ? '))    #code à risque
    b = int(input('b ? '))    #code à risque
    print(a, '/', b, '=', a / b) #code à risque
except Exception as e: # variable e de type Exception
    print(type(e))      #afficher le type de l'exception
    print(e)           #afficher l'exception
```

Exemple d'exécution

```
a ? 5
b ? 0
<class 'ZeroDivisionError'>
division by zero
```

message de
l'exception

Type de
l'exception

Type Exception

- On exécute le code précédent deux fois de suite. Voici deux exemples d'exécution qui révèlent deux types d'erreurs différents :
- Si on ne fournit pas un nombre entier, il ne pourra être converti en **int** et une erreur de type **ValueError** se produit :

```
a ? trois  
<class 'ValueError'>  
invalid literal for int() with base 10: 'trois'
```

- Si on fournit une valeur de 00 pour b, on aura une division par zéro qui produit une erreur de type **ZeroDivisionError** :

```
a ? 5  
b ? 0  
<class 'ZeroDivisionError'>  
division by zero
```

Capture d'erreur spécifique

- Chaque type d'erreur est donc défini par une classe spécifique
- Il est possible d'associer plusieurs blocs `except` à un même bloc `try`, pour exécuter un code différent en fonction de l'erreur capturée
- Lorsqu'une erreur se produit, les blocs `except` sont parcourus l'un après l'autre, du premier au dernier, jusqu'à en trouver un qui corresponde à l'erreur capturée

Exemple:

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except ValueError:
    print('Erreur de conversion.')
except ZeroDivisionError:
    print('Division par zéro.')
except:
    print('Autre erreur.')
```

- Lorsqu'une erreur se produit dans le bloc **try** l'un des blocs **except** seulement qui sera exécuté, selon le type de l'erreur qui s'est produite
- Le dernier bloc **except** est là pour prendre toutes les autres erreurs
- **L'ordre des blocs except est très important et il faut les classer du plus spécifique au plus général, celui par défaut devant venir en dernier**

Gestionnaire d'erreur partagé

- Si le code est le même pour différents types d'erreur, il est possible de lister ces erreurs dans un tuple après le mot réservé **except** au lieu de définir plusieurs blocs **except** (gestion d'erreur spécifique)
- Si on souhaite exécuter le même code pour une erreur de conversion et de division par zéro, il faudrait écrire :

```
try:
    a = int(input('a ? '))
    b = int(input('b ? '))
    print(a, '/', b, '=', a / b)
except (ValueError, ZeroDivisionError) as e:
    print('Erreur de calcul :', e) # exécuter le même code pour différents types d'exceptions
except:
    print('Autre erreur.')
```

Bloc finally

- Le bloc **finally** se lance lorsque le bloc try produit une exception ou non
- Ce bloc est généralement utilisé pour effectuer des nettoyages tels que la fermeture des fichiers, la libération des ressources, etc
- Le mot réservé **finally** permet d'introduire un bloc qui sera exécuté soit après que le bloc try se soit exécuté complètement sans erreur, soit après avoir exécuté le bloc except correspondant à l'erreur qui s'est produite lors de l'exécution du bloc try
- On obtient ainsi une instruction **try-except-finally**

```
print('Début du calcul.')  
  
try:  
    a = int(input('a ? '))  
    b = int(input('b ? '))  
    print('Résultat :', a / b)  
except:  
    print('Erreur.')  
finally:  
    print('Nettoyage de la mémoire.')  
print('Fin du calcul.')
```

- Si l'utilisateur fournit des valeurs correctes pour a et b l'affichage est le suivant :

```
Début du calcul.  
a ? 2  
b ? 8  
Résultat : 0.25  
Nettoyage de la mémoire.  
Fin du calcul.
```

- si une erreur se produit l'affichage est le suivant:

```
Début du calcul.  
a ? 2  
b ? 0  
Erreur.  
Nettoyage de la mémoire.  
Fin du calcul.
```

Dans les 2 cas le
bloc finally a été
exécuté

01 – DÉBOGUER LE CODE PYTHON

Gestion des erreurs (compilation-syntaxe)



Génération d'erreur

- Il est possible de générer une erreur ou de déclencher une exception pour indiquer qu'une erreur ou une condition exceptionnelle s'est produite dans un programme
- La génération d'erreur se fait grâce au mot-clé **raise**
- Pour générer une exception, Il suffit d'utiliser le mot réservé **raise** suivi d'une référence vers un objet représentant une exception

Exemple:

```
def fact(n):  
    if n < 0:  
        raise ArithmeticError() #signaler une erreur de type ArithmeticError si n<0  
    if n == 0:  
        return 1  
    return n * fact(n - 1)
```

- si n est strictement négatif, une exception de type **ArithmeticError** est générée

ArithmeticError

Le programme suivant permet de capturer spécifiquement l'exception de type **ArithmeticError** lors de l'appel de la fonction **fact**

```
try:
    n = int(input('Entrez un nombre : ')) #code à risque
    print(fact(n)) #code à risque
except ArithmeticError: #capturer de l'exception ArithmeticError
    print('Veuillez entrer un nombre positif.') #afficher le message si l'exception ArithmeticError est capturée
except: #capturer les autres types d'exception
    print('Veuillez entrer un nombre.') #afficher le message si d'autres types d'exceptions sont capturées
```

CHAPITRE 1

DÉBOGUER LE CODE PYTHON

1. Gestion des erreurs (compilation-syntaxe)

2. Débogage

3. Outils de suivi et de visualisation de l'exécution d'un code Python



Débogueur

- Un débogueur est un outil de développement qui s'attache à une application en cours d'exécution et qui permet d'inspecter le code
 - il permet d'exécuter le programme pas-à-pas
 - d'afficher la valeur des variables à tout moment
 - de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme
- De nombreux débogueurs permettent, en plus de l'observation de l'état des registres processeurs et de la mémoire, de les modifier avant de rendre la main au programme débogué

Exemple d'utilisation du débogueur :

Considérons le script test_debugger.py suivant qui affiche le carré des nombres entiers de 1 à 5. Nous allons tester son bon fonctionnement avec le débogueur de l'environnement IDLE.

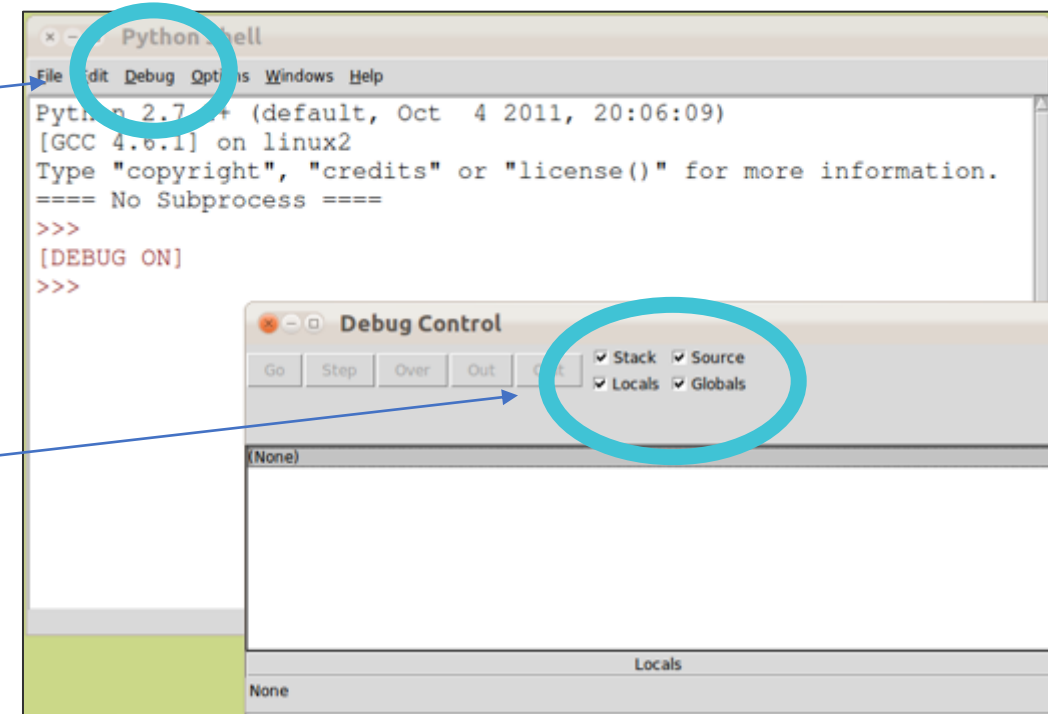
```
def carre(nb):  
    resultat=a*a  
    return resultat  
  
a=1  
print (a)  
while a<5:  
    a+=1  
    print(carre(a))  
print('fin')
```

Débogueur

- Ouvrir IDLE
- Lancer le débogueur
- 2- Debug → Debugger
- 3- Cocher les cases Source et Globals

Le débogueur possède 5 boutons de commande :

- **Go** : Exécution normale du programme jusqu'au prochain point d'arrêt
- **Step** : Exécution pas-à-pas (instruction par instruction)
- **Over** : Exécution pas-à-pas du programme principal (le débogueur ne rentre pas dans les fonctions)
- **Out** : Pour sortir de la fonction en cours
- **Quit** : Termine le programme



Interface de lancement de débogueur

01 – DÉBOGUER LE CODE PYTHON

Débogage



Débogueur

- Dans l'interpréteur interactif, ouvrir le script test_debugger.py :
- File → Open → test_debugger.py
- La fenêtre du code source s'ouvre.
 - Dans cette fenêtre : Run → Run Module

Pas-à-pas grossier:

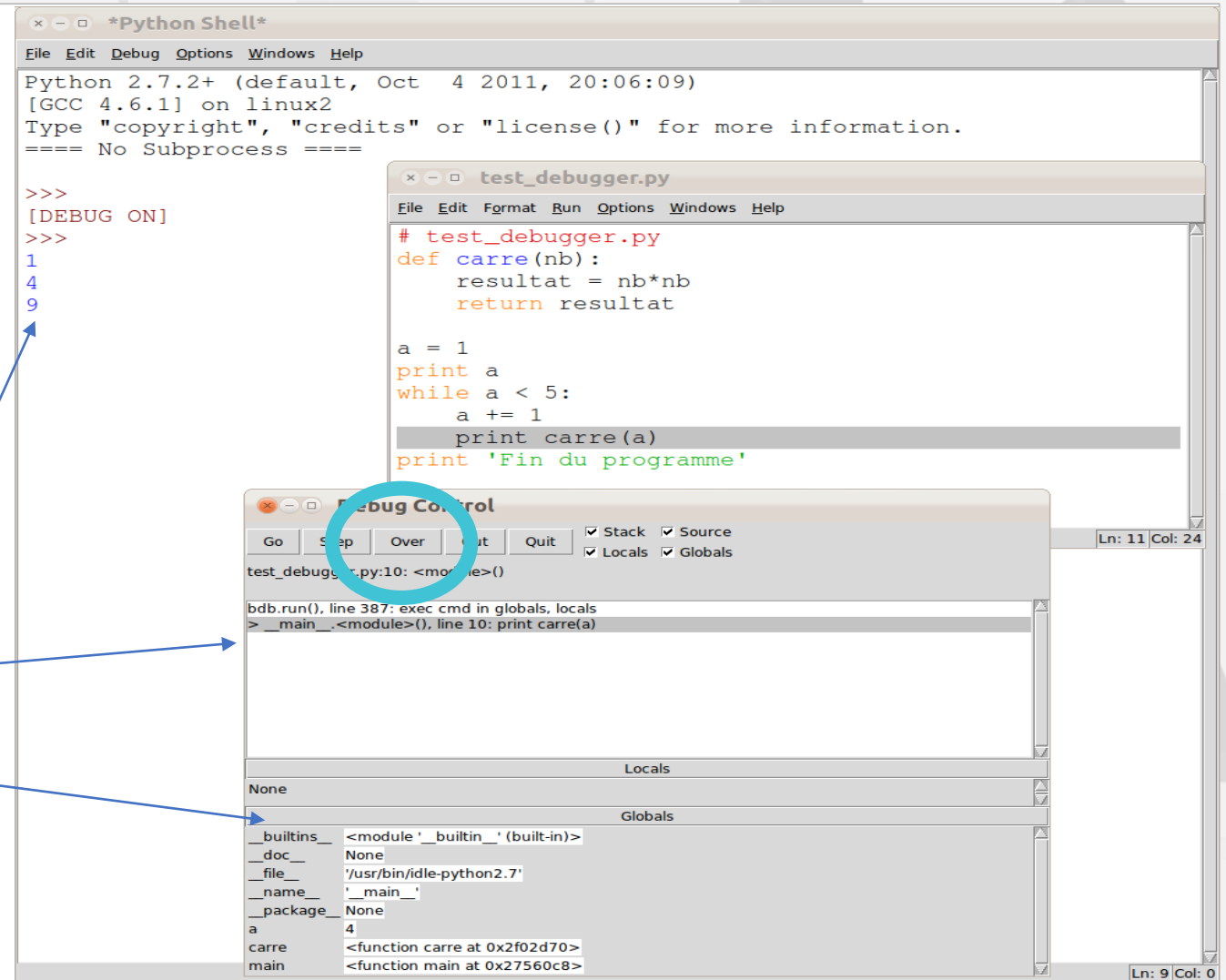
Pour faire du pas-à-pas grossier, cliquer sur le bouton **Over** du débogueur.

Résultats de l'exécution des instructions

Instruction en cours d'exécution

Observer le contenu des variables (actuellement a vaut 4)

Exemple de débogage pas-à-pas grossier



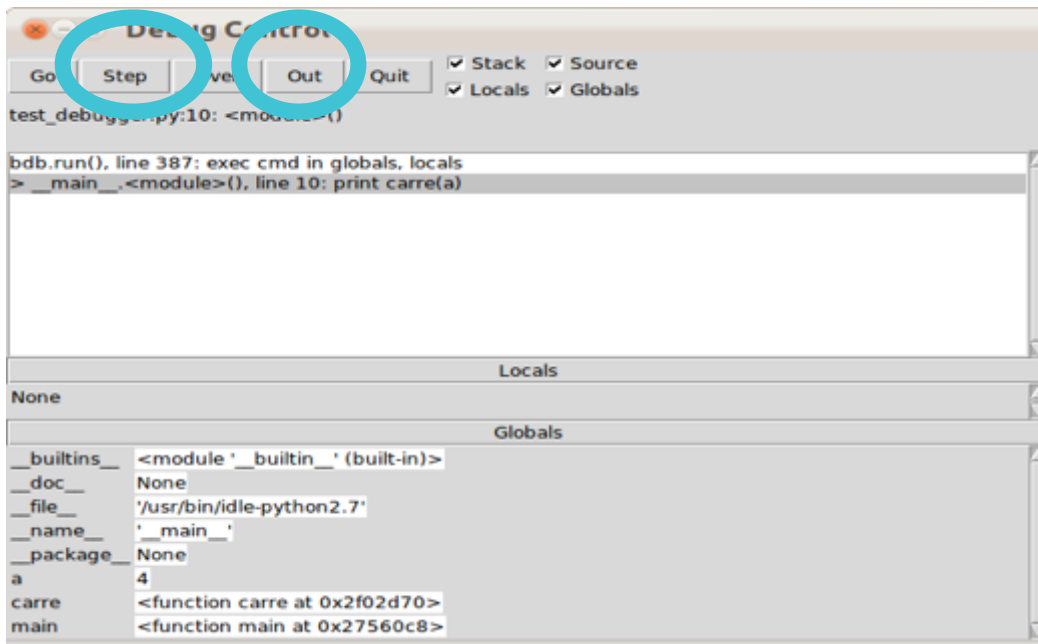
01 – DÉBOGUER LE CODE PYTHON

Débogage

Débogueur

Pas-à-pas détaillé:

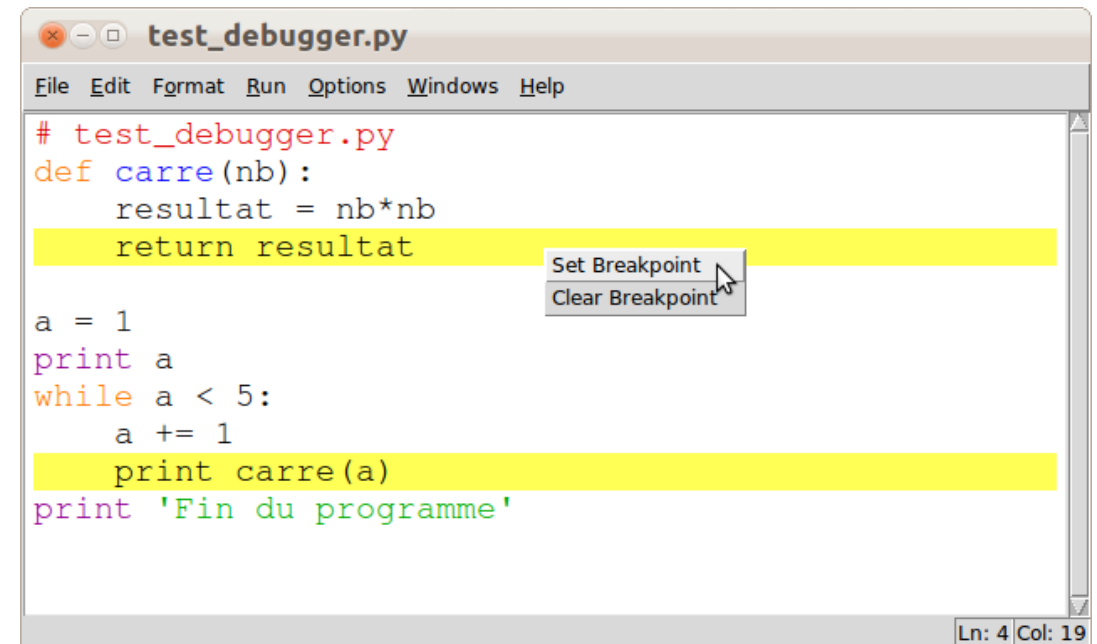
- Pour faire du pas-à-pas détaillé, cliquer sur le bouton **Step** du débogueur
- Pour sortir immédiatement d'une fonction, utiliser le bouton **Out**



Exemple de débogage pas-à-pas détaillé

Point d'arrêt (Breakpoint):

Dans la fenêtre du code source, sur la ligne d'instruction considérée, faire un clic droit et choisir Set **Breakpoint** (la ligne est alors surlignée en jaune)



Exemple de débogage avec point arrêt

CHAPITRE 1

DÉBOGUER LE CODE PYTHON

1. Gestion des erreurs (compilation-syntaxe)

2. Débogage

3. Outils de suivi et de visualisation de l'exécution d'un code Python



01 – DÉBOGUEUR LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débogueur Python

- Python est livré avec un débogueur intégré appelé **Python debugger** ou **pdb**
- **Pdb** est un environnement de débogage interactif pour les programmes Python il permet:
 - Une visualisation des valeurs des variables
 - Une visualisation de l'exécution du programme étape par étape
 - Une compréhension de ce que fait le programme et de trouver des bogues dans la logique
- **Exécution du débogueur à partir d'un interpréteur interactif :**
 - Utiliser **run ()** ou **runeval ()**
 - L'argument de **run ()** est une expression de chaîne qui peut être évaluée par l'interpréteur Python
 - Le débogueur l'analysera, puis interrompra l'exécution juste avant l'évaluation de la première expression
 - **set_trace ()** est une fonction Python qui peut être appelée à partir de n'importe quel point d'un programme

01 – DÉBOGUEUR LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débogueur Python

Exemple :

Soit le code Python suivant (debug.py), on souhaite exécuter le débogueur à partir d'un interpréteur interactif

```
def log(number): #Définition de la procédure log
    print('Processing', number)
    print('Adding 2 to number:', number+2)

def loopier(number): #Définition de la procédure loopier
    for i in range(number): #Appel de log(i) pour i de 1 à number
        log(i)

if __name__ == '__main__':
    import pdb #importation de la bibliothèque pdb
    pdb.run('loopier(5)') #lancement de run() pour exécuter le débogueur
```

```
===== RESTART: C:/Users/DELL/Desktop/algo/debug.py =====
> <string>(1) <module>()
(Pdb) |
```

La ligne suivante est préfixée par (Pdb).
Cela signifie que vous êtes maintenant
dans le débogueur. **Succès!**

Résultat du Lancement de déboguer avec run() sur IDLE

01 – DÉBOGUER LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débogueur Python

- Pour exécuter un code dans le débogueur, tapez **continue** ou **c**. Cela exécutera votre code jusqu'à ce que l'un des événements suivants se produise :
 - Le code lève une exception (une erreur).
 - Vous arrivez à un point d'arrêt
 - Le code se termine

Pour **debug.py**: le code est exécuté normalement jusqu'à la fin puisqu'il n'y a pas une erreur ou un point d'arrêt

```
===== RESTART: C:/Users/DELL/Desktop/algo/debug.py =====
> <string>(1)<module>()
(Pdb) c
Processing 0
Adding 2 to number: 2
Processing 1
Adding 2 to number: 3
Processing 2
Adding 2 to number: 4
Processing 3
Adding 2 to number: 5
Processing 4
Adding 2 to number: 6
```

Exemple d'exécution d'un code dans le débogueur

- Il est possible de démarrer le débogueur depuis l'intérieur d'un programme en utilisant la fonction **set_trace()**
 - Le débogueur Python permet d'importer le module **pdb** et d'ajouter directement un point d'arrêt (**breakpoint**) à votre code à l'aide de la fonction **set_trace()**

01 – DÉBOGUEUR LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débugueur Python

Exemple:

On souhaite ajouter un point d'arrêt au code précédent

```
def log(number):  
    print('Processing', number)  
    print('Adding 2 to number:', number+2)  
  
def loopier(number):  
    for i in range(number):  
  
        import pdb; #importer la bibliothèque pdb  
        pdb.set_trace() #ajouter un point d'arrêt  
  
        log(i)  
  
if __name__ == '__main__':  
    loopier(5)
```

```
===== RESTART: C:/Users/DELL/Desktop/algo/debug.py =====  
> c:\users\dell\desktop\algo\debug.py(10)loopier()  
-> log(i)  
(Pdb) c  
Processing 0  
Adding 2 to number: 2  
> c:\users\dell\desktop\algo\debug.py(9)loopier()  
-> pdb.set_trace()  
(Pdb) c  
Processing 1  
Adding 2 to number: 3  
> c:\users\dell\desktop\algo\debug.py(10)loopier()  
-> log(i)  
(Pdb) c  
Processing 2  
Adding 2 to number: 4  
> c:\users\dell\desktop\algo\debug.py(9)loopier()  
-> pdb.set_trace()  
(Pdb) c  
Processing 3  
Adding 2 to number: 5  
> c:\users\dell\desktop\algo\debug.py(10)loopier()  
-> log(i)  
(Pdb) c  
Processing 4  
Adding 2 to number: 6
```

Arrêt au niveau de chaque point d'arrêt

Compléter l'exécution en tapant c

Trace d'exécution du programme : **number=2,3,4→5**

Trace d'exécution du programme

01 – DÉBOGUEUR LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débugueur Python

Au niveau de chaque **breakpoint**, il est possible de vérifier le contenu des variables

Exemple:

```
def log(number):
    print('Processing', number)
    print('Adding 2 to number:', number+2)

def loopier(number):
    for i in range(number):

        import pdb; #importer la bibliothèque pdb
        pdb.set_trace() #ajouter un point d'arrêt

    log(i)

if __name__ == '__main__':
    loopier(5)
```

```
-> log(i)
(Pdb) c
Processing 0
Adding 2 to number: 2
> c:\users\dell\appdata\local\programs\python\python39\debug_code.py(9) loopier()
-> import pdb; pdb.set_trace()
(Pdb) i
1
(Pdb) c
Processing 1
Adding 2 to number: 3
> c:\users\dell\appdata\local\programs\python\python39\debug_code.py(10) loopier()
-> log(i)
(Pdb) i
2
(Pdb) |
```

Arrêt au niveau d'un point d'arrêt

Demande d'afficher la valeur de la variable i à ce niveau d'exécution (i=1)

Trace d'exécution du programme avec vérification du contenu des variables

01 – DÉBOGUEUR LE CODE PYTHON

Outils de suivi et visualisation d'exécution de code Python



Débugueur Python

Au niveau de chaque **point d'arrêt**, il est possible de changer le contenu des variables

Exemple :

```
def log(number):  
    print('Processing', number)  
    print('Adding 2 to number:', number+2)  
  
def loopier(number):  
    for i in range(number):  
  
        import pdb; #importer la bibliothèque pdb  
        pdb.set_trace() #ajouter un point d'arrêt  
  
        log(i)  
  
if __name__ == '__main__':  
    loopier(5)
```

Arrêt au niveau d'un point d'arrêt

```
===== RESTART: C:/Users/DELL/Desktop/algo/debug.py =====  
> c:\users\dell\desktop\algo\debug.py(10)loopier()  
-> log(i)  
(Pdb) i=20  
(Pdb) c  
Processing 20  
Adding 2 to number: 22  
> c:\users\dell\desktop\algo\debug.py(9)loopier()  
-> pdb.set_trace()
```

Changer valeur de i (i=20) et compléter le reste de l'exécution avec i initialisé à 20

Trace d'exécution du programme avec changement du contenu des variables

CHAPITRE 2

DÉPLOYER UNE SOLUTION PYTHON

Ce que vous allez apprendre dans ce chapitre :

- Connaître les outils de déploiement en Python
- Créer un fichier d'installation en Python
- Générer de la documentation en Python



3 heures



CHAPITRE 2

DÉPLOYER UNE SOLUTION PYTHON

1. Outils de déploiement de solution Python

2. Création de fichiers d'installation de solution Python

3. Documentation du programme



02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Outils de déploiement

Déploiement d'une application :

- Le déploiement d'une application se définit comme la promotion de composants d'une application depuis un environnement vers le suivant
- Le packaging est une étape importante pour préparer le déploiement d'une application

Packaging en Python :

- Le packaging est une étape importante lorsqu'on souhaite partager et réutiliser du code sans avoir à le dupliquer dans chacun de nos projets
- Le packaging consiste à regrouper dans un fichier unique des scripts nécessaires au déploiement automatisé d'un logiciel ou d'une application
- Il existe plusieurs outils permettant de packager une application tels que: distutils, setuptools, distribute ou distutils2

Setuptools :

- Setuptools est l'outil de packaging qui est aujourd'hui recommandé par la Python Packaging Authority (PyPA)
- Setuptools est une bibliothèque de processus de développement de packages conçue pour faciliter le packaging de projets Python
- Setuptools permet de créer des packages au format binaires (eggs).

PyPi (Python Package Index) :

- PyPi est le dépôt tiers officiel du langage de programmation Python. C'est une sorte de serveur centralisé de packages développés en Python
- L'objectif de PyPi est de centraliser les différents packages mis à disposition par la communauté et en gère les différentes versions.

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Outils de déploiement

Exemple de packaging un projet Python

- Pip (Pip Installs Packages) est un gestionnaire de paquets utilisé pour installer et gérer des paquets écrits en Python.
- Pour faire fonctionner cet exemple, certaines commandes de création de package nécessitent une version plus récente de pip, alors commencez par vous assurer que la dernière version est installée par la commande suivante exécutée dans l'invite de commande:

```
py -m pip install --upgrade pip
```

1- Créez localement la structure de fichiers suivante :

```
packaging_tutorial/  
└─ src/  
    └─ example_package/  
        ├── __init__.py  
        └─ example.py
```

Fichier example.py

```
def add_one(number)  
    return number+1
```

- **__init__.py** est un fichier requis pour importer le répertoire en tant que package et doit être vide
- **example.py** est un exemple de fichier python dans le package qui pourrait contenir la logique (fonctions, classes, constantes, etc.) de votre package

Outils de déploiement

Une fois la structure des fichiers est définie , toutes les commandes suivantes seront exécutées (dans l'invite de commande) dans le répertoire **packaging_tutorial**

2- Ajouter des fichiers qui seront utilisés pour préparer le projet pour la distribution :

- La structure du projet ressemblera à ceci :

```
packaging_tutorial/  
├── LICENSE  
├── pyproject.toml  
├── README.md  
├── setup.py  
├── src/  
│   ├── example_package/  
│   │   ├── __init__.py  
│   │   └── example.py  
└── tests/
```

pyproject.toml

Le fichier pyproject.toml indique aux outils de construction ce qui est nécessaire pour construire votre projet. Dans ce qui suit c'est l'outil **setuptools**

pyproject.toml

```
[build-system]  
requires = [  
    "setuptools>=42",  
    "wheel"  
]  
build-backend = "setuptools.build_meta"
```

- build-system.requires** donne une liste des packages nécessaires pour construire votre package
- build-system.build-backend** est le nom de l'objet Python qui sera utilisé pour effectuer la construction

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Outils de déploiement

```
packaging_tutorial/  
├── LICENSE  
├── pyproject.toml  
├── README.md  
├── setup.py  
├── src/  
│   └── example_package/  
│       ├── __init__.py  
│       └── example.py  
└── tests/
```

setup.py

setup.py est le script de construction pour setuptools.
Il indique à setuptools votre package (comme le nom et la version) ainsi que les fichiers de code à inclure.

La fonction setup() définie dans setup.py prend plusieurs paramètres :

Exemple :

- **Name:** est le nom de distribution de votre package.
- **Version:** est la version du paquet
- **author** et **author_email:** sont utilisés pour identifier l'auteur du package
- **Description:** est une brève description du package
- **url:** est l'URL de la page d'accueil du projet.

```
import setuptools
```

```
with open("README.md", "r", encoding="utf-8") as fh:  
    long_description = fh.read()
```

```
setuptools.setup(  
    name="example-pkg-YOUR-USERNAME-HERE",  
    version="0.0.1",  
    author="Example Author",  
    author_email="author@example.com",  
    description="A small example package",  
    long_description=long_description,  
    long_description_content_type="text/markdown",  
    url="https://github.com/pypa/sampleproject",  
    project_urls={  
        "Bug Tracker":  
        "https://github.com/pypa/sampleproject/issues",  
    },  
    classifiers=[  
        "Programming Language :: Python :: 3",  
        "License :: OSI Approved :: MIT License",  
        "Operating System :: OS Independent",  
    ],  
    package_dir={"": "src"},  
    packages=setuptools.find_packages(where="src"),  
    python_requires=">=3.6",  
)
```

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Outils de déploiement

```
packaging_tutorial/  
├── LICENSE  
├── pyproject.toml  
├── README.md  
├── setup.py  
├── src/  
│   ├── example_package/  
│   │   ├── __init__.py  
│   │   └── example.py  
└── tests/
```

README.md

Principalement adressé aux développeurs, le README peut contenir des informations pas très techniques servant à mieux comprendre le sujet de l'application.

Exemple: prendre connaissance des pré-requis et d'installer le projet afin de pouvoir développer et tester localement, liste des auteurs, avec leur titre et l'entreprise pour laquelle ils travaillent

README.md

```
# Example Package
```

This is a simple example package. You can use

[Github-flavored Markdown](https://guides.github.com/features/mastering-markdown/) to write your content.

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Outils de déploiement

```
packaging_tutorial/  
├── LICENSE  
├── pyproject.toml  
├── README.md  
├── setup.py  
├── src/  
│   ├── example_package/  
│   │   ├── __init__.py  
│   │   └── example.py  
└── tests/
```

LICENSE:

Il est important que chaque package téléchargé dans l'index des packages Python inclue une licence. Cela indique aux utilisateurs qui installent votre package les conditions dans lesquelles ils peuvent utiliser votre package.

LICENCE:

Copyright (c) 2018 The Python Packaging Authority

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

.....

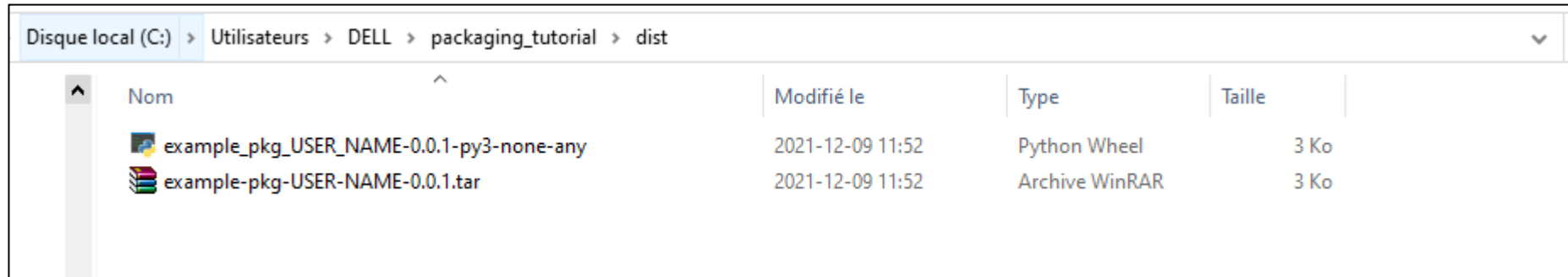
Génération d'archives de distribution



3- L'étape suivante consiste à générer des packages de distribution pour le package :

- Etape de génération d'archives de distribution : Exécutez maintenant la commande suivante à partir du même répertoire où se trouve **pyproject.toml**

```
py -m build
```

- Cette commande devrait générer beaucoup de texte et une fois terminée, elle devrait générer deux fichiers (.whl et .tar.gz) dans le répertoire **dist**



Disque local (C:) > Utilisateurs > DELL > packaging_tutorial > dist				
	Nom	Modifié le	Type	Taille
	 example_pkg_USER_NAME-0.0.1-py3-none-any	2021-12-09 11:52	Python Wheel	3 Ko
	 example-pkg-USER-NAME-0.0.1.tar	2021-12-09 11:52	Archive WinRAR	3 Ko

*Génération des fichiers .whl et .tar.gz dans le répertoire **dist***

- le fichier .whl est le package généré et sauvegardé au format **Wheel**, qui est le format de package intégré standard utilisé pour les distributions Python
- Le fichier .tar.gz est le package de code source compressé

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Publication des packages sur TestPYPI

- **TestPyPI** est une instance distincte de PyPI destinée aux tests et à l'expérimentation
- **Twine** est un outil assurant la publication des packages Python sur PyPI
- Pour publier un package sur **TestPyPI** commencez par créer un compte sur **TestPyPI**
- Pour accéder à un compte, accédez à <https://test.pypi.org/account/register/> et suivez les étapes sur cette page
- Vous devrez également vérifier votre adresse e-mail avant de pouvoir télécharger des packages

⚠ You are using TestPyPI – a separate instance of the Python Package Index that allows you to try distribution tools and processes without affecting the real

TESTING TESTING TESTING

Search projects 🔍

Help Sponsors Log in Register

Create an account on TestPyPI

Name

Email address (required)

Username (required)

Password (required) ☐ Show passwords

Choose a strong password that contains letters (uppercase and lowercase), numbers and special characters. Avoid common words or repetition.

Interface de création de compte sur TestPYPI

02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Publication des packages sur TestPYPI

Pour partager en toute sécurité votre projet, vous aurez besoin d'un Token (jeton) d'API PyPI. Créez-en un sur <https://test.pypi.org/manage/account/#api-tokens>

Add API token

Token name (required)

What is this token for?

Permissions

Upload packages

Scope (required)

Proceed with caution!

An API token scoped to your entire account will have upload permissions for all of your current and future projects.

Add token

Interface de création de Token

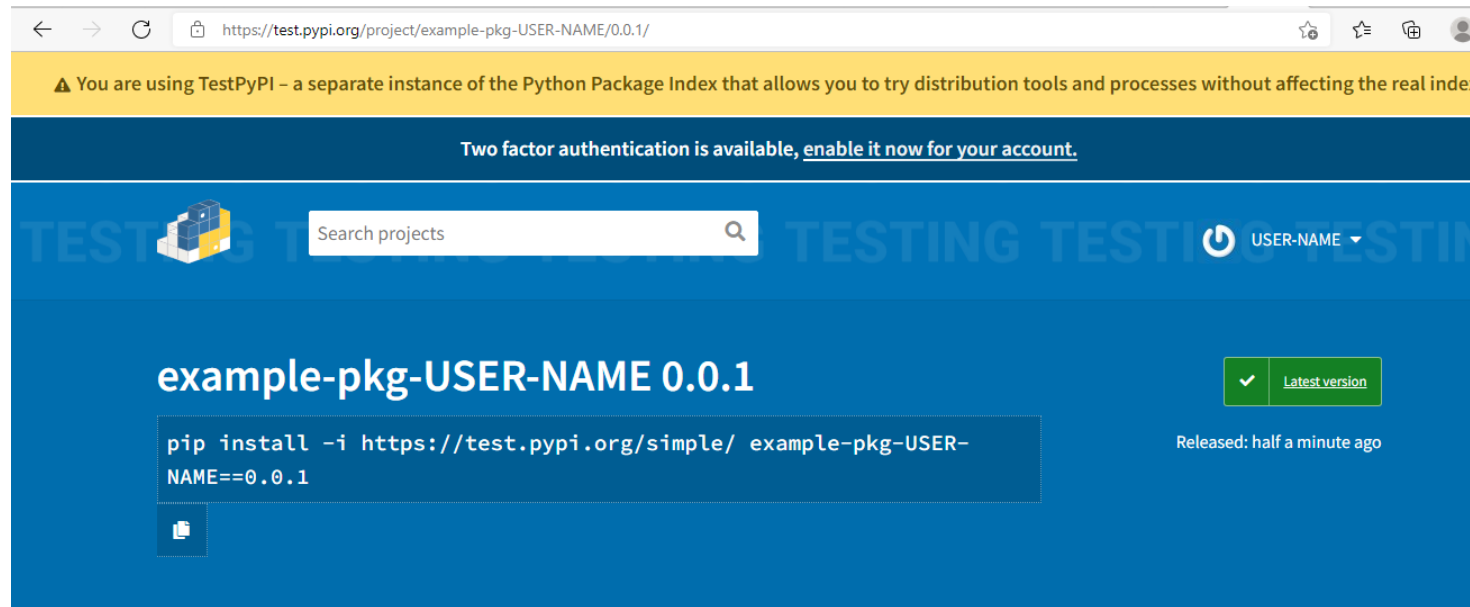
02 – DÉPLOYER UNE SOLUTION PYTHON

Outils de déploiement de solution Python



Publication des packages sur TestPYPI

- Une fois partagé, votre package doit être visible sur **TestPyPI**, par exemple, <https://test.pypi.org/project/example-pkg-YOUR-USERNAME-HERE>



Résultat de partage du package sur TestPYPI

CHAPITRE 2

DÉPLOYER UNE SOLUTION PYTHON

1. Outils de déploiement de solution Python

2. Création de fichiers d'installation de solution Python

3. Documentation du programme



02 – DÉPLOYER UNE SOLUTION PYTHON

Création de fichiers d'installation de solution Python



installation de solution Python

- **Un fichier d'installation** d'une solution développée permet d'utiliser votre script sur n'importe quel ordinateur, qu'il y ai ou non Python installé, sans problème de librairie non installée ni conflit de version. Le fichier .exe téléchargé pour installer une application est un fichier d'installation
- **PyInstaller** est un paquet de **PyPI** gestionnaire de librairies pour Python qui regroupe une application Python et toutes ses dépendances dans un seul package L'utilisateur ainsi peut exécuter l'application packagée sans installer d'interpréteur Python ou de modules
- **PyInstaller** est directement fournit avec les versions de **Python3.5** ou plus et avec **Python2**

Etapes de création de fichiers exécutable avec Pyinstaller :

1. Exécutez la commande suivante sur votre invite de commande pour télécharger **Pyinstaller** :

```
pip3 install pyinstaller
```

2. Placez-vous dans le dossier où se trouve le fichier et exécuter la commande suivante pour créer un fichier d'installation de votre script

```
pyinstaller votre_script_en_python.py
```

- → L'exécutable est alors placé dans le répertoire **Dist** à l'emplacement du script

02 – DÉPLOYER UNE SOLUTION PYTHON

Création de fichiers d'installation de solution Python



installation de solution Python

Exemple de création de fichier d'installation :

- Soit le script **fichierCSV.py** suivant permettant de créer un fichier **CSV**

```
import csv #importer la bibliothèque csv
fichier = open("annuaire.csv", "wt") #ouvrir un fichier en mode écriture
ecrivainCSV = csv.writer(fichier,delimiter=";") #ouvre un flux d'écriture
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) #écrire une 1ère ligne dans le fichier
annuaire.csv
ecrivainCSV.writerow(["Dubois","Marie","0198546372"])
ecrivainCSV.writerow(["Duval","Julien","0399741052"])
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"])
ecrivainCSV.writerow(["Martin","Julie","0399731590"])
fichier.close() #Fermer le fichier
```

- **fichierJson.py** est placé dans le répertoire: **E:\Executable**. La commande de création de fichier exécutable donne le résultat suivant

```
PS E:\Executable> pyinstaller fichierCSV.py
159 INFO: PyInstaller: 4.5.1
159 INFO: Python: 3.9.9
216 INFO: Platform: Windows-10-10.0.19043-SP0
217 INFO: wrote E:\Executable\fichierCSV.spec
222 INFO: UPX is not available.
270 INFO: Extending PYTHONPATH with paths
['E:\Executable', 'E:\Executable']
985 INFO: checking Analysis
986 INFO: Building Analysis because Analysis-00.toc is non existent
987 INFO: Initializing module dependency graph...
993 INFO: Caching module graph hooks...
1015 INFO: Analyzing base_library.zip ...
1021 INFO: Processing pre-build module with hooks: distutils (from C:\Python39\Lib\distutils)
```

Résultat d'exécution de la commande Pyinstaller

Disque local (E:) > Executable				
Nom	Modifié le	Type	Taille	
pycache	2021-11-26 11:28	Dossier de fichiers		
build	2021-11-26 11:28	Dossier de fichiers		
dist	2021-11-26 11:28	Dossier de fichiers		
fichierCSV	2021-11-26 11:27	Python File	1 Ko	
fichierCSV.spec	2021-11-26 11:28	Fichier SPEC	2 Ko	

Création du répertoire dist



02 – DÉPLOYER UNE SOLUTION PYTHON

Création de fichiers d'installation de solution Python

installation de solution Python

Le répertoire **dist\ichierJson** contient le fichier exécutable **fichierCSV.exe**

Disque local (E:) > Executable > dist > fichierCSV

Nom	Modifié le	Type	Taille
 annuaire	2021-11-26 11:30	Fichier CSV Micro...	1 Ko
 fichierCSV	2021-11-26 11:28	Application	1,960 Ko

Contenu du répertoire dist

L'exécution du fichier exécutable entraine la création du fichier annuaire.csv

Création du fichier exécutable fichierCSV.exe

CHAPITRE 2

DÉPLOYER UNE SOLUTION PYTHON

1. Outils de déploiement de solution Python
2. Création de fichiers d'installation de solution Python

3. Documentation du programme



Documentation du programme

- La documentation logicielle est un texte écrit qui accompagne le logiciel informatique
- La documentation explique comment le logiciel fonctionne, et/ou comment on doit l'employer
- **Sphinx** est un outil de génération de documentation automatique utilisant le format **ReStructured Text**. Après une phase de configuration, il permet de générer assez facilement la documentation d'un projet python en un format **pdf** ou **html**
- Installation de Sphinx se fait en utilisant la commande :

```
py -m pip install -U sphinx
```

- Etapes de création de documentation avec Sphinx:
 - Créez un répertoire docs dans votre projet pour contenir votre documentation
 - Exécutez **sphinx-quickstart** ou **python -m sphinx.cmd.quickstart** dans le répertoire docs

```
cd docs  
sphinx-quickstart
```

- Cela vous guide à travers certaines configurations de base à créer un fichier **index.rst** ainsi qu'un fichier **conf.py**
 - **index.rst** est un fichier qui présente la structure de la documentation
 - **Conf.py** est un fichier contenant toute la configuration du projet de documentation : **nom du projet, version, auteur, etc.**

02 – DÉPLOYER UNE SOLUTION PYTHON

Documentation du programme



Documentation du programme

Exemple d'exécution de la commande sphinx-quickstart :

```
PS E:\Executable\docs> sphinx-quickstart
Welcome to the Sphinx 4.2.0 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]: y

The project name will occur in several places in the built documentation.
> Project name: yes
> Author name(s): Meriam
> Project release []: 1

If the documents are to be written in a language other than English,
you can select a language here by its language code. Sphinx will then
translate text that it generates into that language.

For a list of supported codes, see
https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-language.
> Project language [en]: en

Creating file E:\Executable\docs\source\conf.py.
Creating file E:\Executable\docs\source\index.rst.
Creating file E:\Executable\docs\Makefile.
Creating file E:\Executable\docs\make.bat.

Finished: An initial directory structure has been created.

You should now populate your master file E:\Executable\docs\source\index.rst and create other documentation
source files. Use the Makefile to build the docs, like so:
    make builder
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.
```

Figure à gauche : Exemple d'exécution de la commande sphinx-quickstart

Disque local (E:) > Executable > docs > source

Nom	Modifié le	Type	Taille
_static	2021-11-26 11:39	Dossier de fichiers	
_templates	2021-11-26 11:39	Dossier de fichiers	
conf	2021-11-26 11:39	Python File	2 Ko
index	2021-11-26 11:39	Fichier RST	1 Ko

Création des fichiers conf.py et index.RST

Figure à droite : Création des fichiers conf.py et index.RST

02 – DÉPLOYER UNE SOLUTION PYTHON

Documentation du programme



Documentation du programme

Contenu du fichier: index.RST :

```
index - Bloc-notes
Fichier  Edition  Format  Affichage  Aide
[. yes documentation master file, created by
   sphinx-quickstart on Fri Nov 26 11:39:28 2021.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to yes's documentation!
=====

.. toctree::
   :maxdepth: 2
   :caption: Contents:

Indices and tables
=====

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Contenu du fichier: conf.py

Contenu du fichier: conf.py :

```
# Configuration file for the Sphinx documentation builder.
#
# This file only contains a selection of the most common options. For a full
# list see the documentation:
# https://www.sphinx-doc.org/en/master/usage/configuration.html

# -- Path setup -----

# If extensions (or modules to document with autodoc) are in another directory,
# add these directories to sys.path here. If the directory is relative to the
# documentation root, use os.path.abspath to make it absolute, like shown here.
#
# import os
# import sys
# sys.path.insert(0, os.path.abspath('.'))

# -- Project information -----

project = 'yes'
copyright = '2021, Meriam'
author = 'Meriam'

# The full version, including alpha/beta/rc tags
release = '1'

# -- General configuration -----

# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [
]
```

Contenu du fichier: index.RST

02 – DÉPLOYER UNE SOLUTION PYTHON

Documentation du programme



Documentation du programme

On considère que votre projet contient le module **Bonjour.py** suivant contenant la fonction `bonjour` et une description de cette fonction

```
def bonjour(nom):  
    """Cette fonction permet d'afficher le message bonjour  
    :paramètre: chaine de caractères le nom  
    :retour: un message de bonjour  
    :rtype: chaine de caractères  
    """  
    return('bonjour' + nom)
```

1. Apporter les modifications suivantes sur le fichier **conf.py**

Spécifier le chemin du fichier
Bonjour.py

Ajout de cette ligne pour assurer une
génération automatique de la doc

```
import os  
import sys  
# sys.path.insert(0, os.path.abspath('.'))  
sys.path.append('E:/Bonjour/src')  
  
# -- Project information -----  
project = 'scoreMP'  
copyright = '2021, meriam'  
author = 'meriam'  
  
# The full version, including alpha/beta/rc tags  
release = '1.0'  
# -- General configuration -----  
  
# Add any Sphinx extension module names here, as strings. They can be  
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom  
# ones.  
extensions = [  
    'sphinx.ext.autodoc'  
]
```

Contenu modifié du fichier: `conf.py`

02 – DÉPLOYER UNE SOLUTION PYTHON

Documentation du programme



Documentation du programme

2. Créer le fichier **bnj.rst** suivant dans le dossier source contenant un appel de la fonction `bonjour` du module `Bonjour`

```
.. autofunction:: Bonjour.bonjour
```

3. Dans le fichier **index.rst** ajouter un lien vers le fichier **bnj.rst**

Ajout d'un lien vers **bnj.rst** crée

```
.. Bonjour documentation master file, created by
sphinx-quickstart on Sun Jan 30 07:13:05 2022.
You can adapt this file completely to your liking, but it should at least
contain the root `toctree` directive.
```

Welcome to Bonjour's documentation!

=====

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:
```

[./bnj.rst](#)

Indices and tables

=====

```
* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Contenu modifié du fichier: index.rst

Documentation du programme

4. Dans le dossier docs, exécuter le commande suivante pour générer la documentation

```
.\make html
```

5

5. Le fichier de documentation **bnj.html** suivant sera généré dans **docs\build\html**

Fichier | E:/Bonjour/docs/build/html/bnj.html

Bonjour

Navigation

Quick search

Go

Bonjour . **bonjour** (*nom*)

Cette fonction permet d’afficher le message bonjour

Paramètre chaine de caractères:

le nom

Retour:

un message de bonjour

Return type:

chaine de caractères

©2022, meriam. | Powered by Sphinx 4.2.0 & Alabaster 0.7.12 | [Page source](#)

Contenu du fichier: bnj.html