Table of Contents

TD_01 _Les opérateurs en Python

🔢 Opérateurs arithmétiques (nombres)

Opérateur	Signification	Marche quand…	Ne marche pas quand…	Exemple
`+`	addition	nombres	chaîne + nombre	`3 + 4`
`-`	soustraction	nombres	chaînes	`10 - 3`
`*`	multiplication	nombres	chaîne * chaîne	`5 * 6`
`**`	puissance	nombres	chaînes	`2 ** 3`
`/`	division	nombres (≠0)	division par 0	`10 / 3`
`//`	division entière	nombres	chaînes	`10 // 3`
`%`	reste	nombres	chaînes	`23 % 5`

❌ Exemple d'erreur :

```
4 / 0        # division par zéro
"abc" - "a" # opération invalide
```

📝 Opérateurs sur les chaînes de caractères

Opérateur	Effet	Marche quand…	Exemple
`+`	concaténation	chaîne + chaîne	`"bon" + "jour"`
`*`	répétition	chaîne * entier	`"ha" * 3`

❌ Ne marche pas :

```
"bonjour" * "3"
```

📦 Opérateurs sur les listes

Opérateur	Effet	Exemple
`+`	concaténation	`[1,2] + [3]`
`*`	répétition	`[1,2] * 3`
`in`	présence	`3 in [1,2,3]`

🔍 Opérateurs de comparaison

Opérateur	Signification	Exemple
`==`	égal	`10 == 5`
`!=`	différent	`4 != 3`

Opérateur	Signification	Exemple
<	plus petit	`3 < 5`
>	plus grand	`7 > 2`

⚠️ Attention aux types :

```
'1' == 1   # False
```

2️⃣ Les fonctions importantes

🔹 `len()`

👉 Rôle

Donne **le nombre d'éléments** :

- d'une chaîne
- d'une liste

✔️ Exemple

```
len("Bonjour")      # 7
len([1, 2, 3, 4])   # 4
```

❌ Ne marche pas

```
len(10)  # erreur
```

🔹 `max()`

👉 Rôle

Retourne **le plus grand élément**.

✔️ Exemple

```
max([3, 7, 2])      # 7
max("abc")         # 'c'
```

❌ Ne marche pas

```
max(10)  # pas une collection
```

🔹 `sorted()`

👉 Rôle

Retourne une **nouvelle liste triée** (ne modifie pas l'originale).

✔️ Exemple

```
sorted([3, 1, 2])   # [1, 2, 3]
```

⚠️ Important

```
l = [3, 1, 2]
sorted(l)
print(l)  # toujours [3, 1, 2]
```

🔹 `append()` (méthode de liste)

👉 Rôle

Ajoute **un élément à la fin** de la liste.

✔️ Exemple

```
l = [1, 2]
l.append(3)
print(l)  # [1, 2, 3]
```

⚠️ Très important

```
x = l.append(4)
print(x)  # None
```

👉 `append` **ne retourne rien** (retourne `None`).

3️⃣ Le type et la fonction `type()`

🔹 Qu'est-ce qu'un type ?

Le **type** indique :

- ce que représente une valeur
- quelles opérations sont autorisées

Exemples de types :

- `int` → entier
- `float` → nombre à virgule
- `str` → chaîne
- `list` → liste

🔹 La fonction `type()`

👉 Rôle

Donne le **type d'une valeur**.

✔️ Exemples

```
type(23)        # int
type(1.5)       # float
type("23")      # str
type([1,2,3])   # list
```

⚠️ Important

```
23      # nombre
"23"    # texte
```

➡️ même écriture, **type différent**, comportement différent.

4️⃣ La fonction `help()`

🔹 À quoi sert `help()` ?

Afficher **l'aide intégrée de Python** :

- fonctions
- types
- méthodes
- modules

🔹 Utilisation simple

```
help(len)
help(list)
help(str)
```

➡️ Affiche :

- description
- paramètres
- exemples

🔹 Mode aide interactif

```
help()
```

Puis taper :

```
len
LISTS
INTEGER
```

Quitter avec :

```
q
```

🔹 Pourquoi `help()` est important ?

- comprendre une fonction sans Internet
- éviter les erreurs
- devenir autonome en programmation

🧠 Résumé ultra-clair

- Les **opérateurs** dépendent du **type**
- `append()` modifie la liste mais retourne `None`
- `sorted()` crée une nouvelle liste
- `len()` et `max()` fonctionnent sur des collections
- `type()` permet de comprendre les erreurs
- `help()` est ton **meilleur ami** pour apprendre Python

❌ Tableau des expressions erronées (TD01)

Expression	Type d'erreur	Pourquoi l'expression est erronée
`4 / 0`	Erreur mathématique	La division par zéro est interdite en mathématiques → Python lève une erreur
`Bonjour * 5`	Erreur de type	`Bonjour` n'est pas une chaîne (pas entre guillemets), Python le prend pour une variable inexistante
`"10" / 3`	Erreur de type	`/` est un opérateur numérique, `"10"` est une chaîne
`"abc" - "a"`	Erreur de type	`-` n'est défini que pour les nombres
`"5" * "2"`	Erreur de type	`*` ne fonctionne pas entre deux chaînes
`"10.54" ** 5`	Erreur de type	`**` (puissance) fonctionne uniquement sur des nombres
`"1" / 2`	Erreur de type	Impossible de diviser une chaîne par un nombre
`10,54**5 - 234`	Erreur de syntaxe	En Python, les nombres décimaux utilisent un **point**, pas une virgule
`523 <`	Erreur de syntaxe	L'opérateur `<` attend une valeur à droite
`"Bonjour" - 3`	Erreur de type	On ne peut pas soustraire un nombre à une chaîne

Expression	Type d’erreur	Pourquoi l’expression est erronée
`1 // 2 + 1 // 3 + 1 // 4 + 1 // 5`	Erreur logique	`//` est une division entière → tous les termes valent `0`, le calcul est faux même s’il n’y a pas d’erreur Python

TD02 _ Les variables

🔹 Qu’est-ce qu’une variable ?

Une **variable** est un **nom** qui référence une **valeur** en mémoire.

```
age = 20
```

- `age` → nom de la variable
- `20` → valeur stockée

👉 On peut ensuite réutiliser cette valeur :

```
age + 1
```

🔹 À quoi servent les variables ?

- stocker un résultat
- éviter de répéter une valeur
- rendre le code lisible

Exemple :

```
rayon = 6.5
aire = 3.14 * rayon**2
```

🔹 Comment bien nommer une variable ?

✔️ Règles obligatoires

- commence par une **lettre** ou _
- pas d’espace
- pas de caractères spéciaux
- pas de mot réservé (`if`, `for`, `while`, …)

✔️ Bonnes pratiques

- nom **clair** et **explicite**
- en minuscules
- utiliser _ pour séparer les mots

✔️ Bons exemples :

```
age
rayon_cercle
total_minutes
```

❌ Mauvais exemples :

```
2age        # commence par un chiffre
mon age     # espace interdit
if          # mot réservé
```

Résumé à retenir (très important)

- ✔️ **Correct ≠ logiquement correct**
- Les opérateurs mathématiques :
    - fonctionnent **uniquement sur des nombres**
- `/` → division réelle
- `//` → division entière
- `%` → reste
- `**` → puissance
- Python **refuse** toute opération non définie par le type

❌ Tableau des expressions erronées (TD02)

Rappels de variables du TD02

```
largeur = 2.8
longueur = 34
texte = "le langage Python"
ma_liste = ["A", "B", "C"]
```

Expression	Type d'erreur	Pourquoi c'est erroné
`longueur + texte`	Erreur de type	Addition entre nombre et chaîne impossible
`ma_liste + 2`	Erreur de type	Une liste ne peut pas être additionnée à un entier
`texte // 10`	Erreur de type	`//` est réservé aux nombres
`5 = longueur`	Erreur de syntaxe	Le côté gauche de = doit être une variable
`longueur + largeur = texte`	Erreur de syntaxe	Une expression ne peut pas être assignée
`longueur = texte`	Erreur de type (logique)	`longueur` est utilisé comme nombre, `texte` est une chaîne
`largeur = largeur / 0`	Erreur mathématique	Division par zéro
`texte - "Python"`	Erreur de type	Soustraction impossible sur des chaînes

Expression	Type d’erreur	Pourquoi c’est erroné
`ma_liste * ma_liste`	Erreur de type	`*` nécessite un entier comme multiplicateur
`texte[100]`	Erreur d’indice	Indice hors limites de la chaîne

🧠 Synthèse à retenir absolument

Types d’erreurs rencontrées

- ❌ **Syntaxe** : Python ne comprend pas l’écriture
  → ex : `523 <`
- ❌ **Type** : opération interdite entre types
  → ex : `"10" / 3`
- ❌ **Mathématique** : règle mathématique violée
  → ex : `4 / 0`
- ⚠️ **Logique** : Python accepte mais le résultat est faux
  → ex : `1 // 2` au lieu de `1 / 2`

TD03 _ Programme et fonction

🔹 Définition

Une **fonction** est un **outil réutilisable** qui :

- reçoit des **paramètres** (entrées),
- effectue un **traitement**,
- retourne (ou affiche) un **résultat**.

Schéma mental :

`entrée → fonction → sortie`

Exemple simple :

`max(3, 7)`

- entrée : `3` et `7`
- fonction : `max`
- sortie : `7`

2️⃣ Fonctions intégrées (built-in) de Python

Python fournit déjà **beaucoup de fonctions prêtes à l’emploi**.

🔹 `min()` et `max()`

Rôle

- `min()` → plus petite valeur
- `max()` → plus grande valeur

Exemples

```
min(3, 7, 2)     # 2
max(3, 7, 2)     # 7
```

Avec une liste :

```
notes = [12, 8, 15]
max(notes)       # 15
```

🧠 **Pourquoi c'est utile ?**
Pas besoin d'écrire une boucle → Python le fait pour nous.

🔹 `int()` et `str()`

👉 Ce sont des fonctions de conversion

Fonction	Rôle
`int()`	convertir en entier
`str()`	convertir en chaîne

Exemples

```
int("25")     # 25
str(25)       # "25"
```

⚠️ Attention :

```
int("bonjour")  # erreur
```

👉 On ne peut convertir que ce qui a un **sens numérique**.

3️⃣ Comprendre `int(input("..."))` (très important)

Prenons cette ligne :

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

Étape par étape (ce que fait Python)

1️⃣ `input("Entrez votre âge : ")`
➡️ affiche le message
➡️ attend une saisie clavier
➡️ retourne **toujours une chaîne**

Exemple :

`"18"`

2️⃣ `int("18")`
➡️ convertit la chaîne `"18"` en entier `18`

3️⃣ `age = 18`
➡️ la valeur entière est stockée dans la variable `age`

❌ Sans `int()`

```
age = input("Age : ")
age + 1
```

➡️ **Erreur**, car :

- `age` est une chaîne
- `+ 1` est une opération numérique

4️⃣ La fonction `print()`

🔹 Rôle

Afficher quelque chose à l'écran.

```
print("Bonjour")
print(3 + 4)
```

🧠 Dans un **script**, sans `print()`, **rien ne s'affiche**.

5️⃣ Qu'est-ce que `import` ?

🔹 Définition

`import` permet de **charger un module** (une boîte à outils).

```
import math
```

➡️ donne accès à toutes les fonctions du module `math`.

🔹 Exemple avec `math`

```
import math

math.sqrt(16)    # 4.0
math.factorial(5) # 120
```

🧠 Pourquoi `math.sqrt` ?

- `sqrt` appartient au module `math`
- on précise le module pour éviter les confusions

6️⃣ Différence entre fonctions Python et fonctions importées

Type de fonction	Exemple	Import nécessaire ?
Fonction intégrée	`print`, `min`, `max`, `int`, `str`	❌ Non
Fonction de module	`math.sqrt`, `random.randint`	✅ Oui

7️⃣ Exemple complet (script du TD03)

```
import math

print("**** Bienvenue ****")
x = int(input("Entrez un nombre positif : "))
print("La racine carrée vaut :", math.sqrt(x))
```

🧠 Ce script :

- utilise `print`
- lit une valeur avec `input`
- convertit avec `int`
- utilise une fonction d'un module (`math.sqrt`)

8️⃣ Erreurs classiques à éviter

❌ Oublier la conversion :

```
x = input("Nombre : ")
print(x * 2)   # répète la chaîne
```

✔️ Correct :

```
x = int(input("Nombre : "))
print(x * 2)
```

🧠 Résumé final (à apprendre par cœur)

- une **fonction** = outil réutilisable
- `min`, `max`, `int`, `str` → fonctions intégrées
- `input()` retourne **toujours une chaîne**
- `int(input())` = lire → convertir → utiliser
- `print()` affiche
- `import` permet d'utiliser des modules
- `math.sqrt` ≠ `sqrt` tout seul

TD04 _ Les alternatives en Python

`if`

Exécute un bloc **seulement si** la condition est vraie.

```
age = int(input("Âge : "))
if age >= 18:
    print("Majeur")
```

- si `age >= 18` est `True` → on affiche
- sinon → on ne fait rien

`if ... else`

Deux chemins possibles (vrai / faux).

```
n = int(input("Entier : "))
if n % 2 == 0:
    print("pair")
else:
    print("impair")
```

`if ... elif ... else`

Plusieurs cas (plus lisible que plusieurs `if` séparés).

```
x = int(input("Nombre : "))
if x > 0:
    print("positif")
elif x < 0:
    print("négatif")
else:
    print("nul")
```

⚠️ Important : `elif` et `else` sont **optionnels**, mais l'**indentation** est obligatoire (4 espaces recommandés).

td04-alternatives

2) Expressions booléennes (ta liste) : valeur + sémantique

Tableau : résultats et explications

Expression	Valeur	Sémantique (en français)
`10 < 20`	`True`	10 est plus petit que 20
`10 > 20`	`False`	10 n'est pas plus grand que 20
`1 == 2`	`False`	1 n'est pas égal à 2
`21/2 != 10`	`True`	21/2 vaut 10.5, donc ce n'est pas 10
`21//2 != 10`	`False`	21//2 vaut 10 (division entière), donc c'est égal à 10 → "!= 10" est faux

Expression	Valeur	Sémantique (en français)
`'ABC' == 'ABC'`	`True`	les deux chaînes sont identiques
`'abc' == 'ABC'`	`False`	majuscules ≠ minuscules : chaînes différentes
`'1' == 1`	`False`	`'1'` est une chaîne, `1` est un entier → types différents, donc pas égal
`[1,2,3] == [1,2,3]`	`True`	mêmes éléments dans le même ordre
`[1,2,3] == [1,3,2]`	`False`	ordre différent → listes différentes
`1 in [7,8,9,0]`	`False`	1 n'est pas dans la liste
`'A' not in ['ABC','def','G']`	`True`	`'A'` n'est pas un élément exact de la liste (il y a `'ABC'`, pas `'A'`)
`'ABC' not in 'ABCDEFGH'`	`False`	`'ABC'` est bien contenu dans la chaîne `"ABCDEFGH"`
`'c' in 'ABCDEFGH'`	`False`	`'c'` (minuscule) n'est pas dans `"ABCDEFGH"` (majuscules)

---

3) Mini-rappel sur les opérateurs utilisés

- `<` et `>` : comparaison de nombres
- `==` : égalité
- `!=` : différence
- `in` : "est contenu dans"
- `not in` : "n'est pas contenu dans"
- `/` : division réelle (donne souvent un float)
- `//` : division entière (on garde la partie entière)

TD05 _ Fonction

🔹 Principe

Une **fonction** peut **renvoyer une valeur** grâce à `return`.
Cette valeur peut être **stockée dans une variable**.

Exemple simple

```
def carre(x):
    return x * x

resultat = carre(5)
print(resultat)
```

🧠 Ce qui se passe

1. `carre(5)` est appelé
2. la fonction calcule `5 * 5`
3. `return 25` renvoie la valeur
4. `resultat` reçoit `25`

👉 **Sans `return`, rien n'est stocké.**

2️⃣ Le mot-clé `return`

🔹 Rôle de `return`

- renvoyer une valeur à l'endroit où la fonction est appelée
- **arrêter immédiatement la fonction**

Exemple

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

x = somme(3, 4)
```

➡️ `x` vaut 7.

❌ Fonction sans `return`

```
def somme(a, b):
    print(a + b)

x = somme(3, 4)
print(x)
```

Résultat :

```
7
None
```

🧠 Pourquoi ?

- la fonction affiche
- mais ne renvoie rien → Python renvoie `None`

3️⃣ Code mort (très important)

🔹 Définition

Le **code mort** est du code **jamais exécuté**, car placé **après un `return`**.

Exemple (TD05)

```
def afficher(x):
    print(x)
    return 0
    print("Les fonctions en Python")  # code mort
```

🧠 Explication

- dès que `return 0` est exécuté
- la fonction **s'arrête**
- tout ce qui suit est **ignoré**

👉 La ligne `"Les fonctions en Python"` **ne s'affichera jamais**.

4️⃣ Importer des fonctions depuis un autre fichier

🔹 Pourquoi découper en fichiers ?

- code plus propre
- fonctions réutilisables
- programme principal plus lisible

Exemple simple (2 fichiers)

📁 `calcul.py`

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

📁 `main.py`

```
import calcul

resultat = calcul.addition(3, 4)
print(resultat)
```

🧠 Explication

- `import calcul` charge le fichier `calcul.py`
- `calcul.addition()` appelle la fonction
- le résultat est récupéré dans `resultat`

⚠️ Les deux fichiers doivent être **dans le même dossier**.

Variante (import d’une seule fonction)

```
from calcul import addition

print(addition(3, 4))
```

5️⃣ Fonction qui ne renvoie rien (affichage seulement)

```
def afficher_etoiles(n):
    print("*" * n)

afficher_etoiles(5)
```

🧠 Cette fonction :

- **fait une action**
- **ne renvoie aucune valeur**

  retourne implicitement `None`

🧠 Résumé à retenir absolument

- `def` → définir une fonction
- `return` :
  - renvoie une valeur
  - arrête la fonction
- résultat d’une fonction peut être **stocké dans une variable**
- code après `return` = **code mort**
- `import fichier` permet d’utiliser des fonctions d’un autre fichier
-

TD06 _ Calendrier

Le TD06 est un **TD de synthèse** : il réutilise **toutes les notions vues avant** pour construire un programme complet.

1️⃣ Découpage du programme en fichiers (modules)

🔹 Notion

Le programme est découpé en :

- **un module** `calendrier.py` → contient les fonctions

- **un programme principal** `main.py` → utilise les fonctions

🧠 Pourquoi ?

- code plus lisible
- fonctions réutilisables
- séparation logique (calcul / affichage)

👉 C'est une **bonne pratique fondamentale** en programmation.

2️⃣ Fonctions qui retournent une valeur

Beaucoup de fonctions du TD **renvoient une valeur** grâce à `return`.

Exemples

- `nom_du_mois(mois)` → renvoie une chaîne
- `est_bissextile(annee)` → renvoie `True` ou `False`
- `suite_numeros_jours(mois, annee)` → renvoie une chaîne
- `numero_jour(jour, mois, annee)` → renvoie un entier

🧠 Le `return` :

- envoie le résultat au programme appelant
- arrête immédiatement la fonction

3️⃣ Fonctions qui affichent (sans return)

Certaines fonctions **ne renvoient rien**, elles **affichent seulement**.

Exemples

- `afficher_titre(mois, annee)`
- `afficher_entete()`

🧠 Elles effectuent une action visuelle, mais ne produisent pas de valeur exploitable.

4️⃣ Listes pour stocker des données fixes

🔹 Exemple : noms des mois

```
mois = [
 "Janvier", "Février", "Mars", "Avril", "Mai", "Juin",
 "Juillet", "Août", "Septembre", "Octobre", "Novembre", "Décembre"
]
```

🧠 Pourquoi une liste ?

- accès direct par index

- structure simple et lisible
- évite les `if` répétitifs

5️⃣ Conditions (`if / elif / else`)

Utilisées pour :

- déterminer le nombre de jours d'un mois
- savoir si une année est bissextile

Exemple conceptuel

```
if mois == 2:
    if est_bissextile(annee):
        jours = 29
    else:
        jours = 28
```

🧠 Les alternatives permettent de **choisir un comportement** selon les valeurs.

6️⃣ Années bissextiles (logique conditionnelle)

🔹 Règle

Une année est bissextile si :

- divisible par 4 et pas par 100
  ou
- divisible par 400

Fonction associée

```
def est_bissextile(annee):
    return (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0)
```

🧠 Cette fonction :

- renvoie un booléen
- est réutilisée ailleurs dans le programme

7️⃣ Construction de chaînes de caractères

Le calendrier est construit sous forme de **chaîne**.

Exemple

```
"01 02 03 04 05 06"
```

🧠 On concatène :

- des nombres formatés (`01`, `02`, …)

- des espaces
- des retours à la ligne (plus tard)

8️⃣ Boucles (`for`)

Utilisées pour :

- parcourir les jours du mois
- construire la suite `"01 02 03 ..."`

Exemple conceptuel

```
for jour in range(1, nb_jours + 1):
    # ajouter le jour à la chaîne
```

🧠 La boucle évite d'écrire 30 fois la même instruction.

9️⃣ Calcul du premier jour du mois (abstraction)

🔹 Notion clé

Le TD **ne te demande pas de comprendre la formule**, mais de :

- l'implémenter dans une fonction
- utiliser le résultat

```
numero_jour(1, mois, annee)
```

🧠 C'est un exemple de :

- **fonction "boîte noire"**
- on sait ce qu'elle fait, pas forcément comment

🔟 Mise en forme du texte (`textwrap`)

🔹 Module `textwrap`

Permet d'ajouter des retours à la ligne automatiquement.

```
import textwrap
textwrap.fill(chaine, width=20)
```

🧠 Très utile pour :

- aligner les jours
- rendre l'affichage propre
- éviter les calculs manuels de sauts de ligne

🧠 Résumé global à retenir

Le TD06 te fait pratiquer :

- ✔️ **fonctions avec `return`**
- ✔️ **fonctions sans `return`**
- ✔️ **modules et `import`**
- ✔️ **listes**
- ✔️ **boucles**
- ✔️ **conditions**
- ✔️ **chaînes de caractères**
- ✔️ **réutilisation de fonctions**
- ✔️ **construction d'un programme complet**

👉 C'est un **TD très important**, car il montre comment **assembler toutes les notions** vues depuis le début.

TD07 _ Boucles `while`

1️⃣ La boucle `while`

🔹 Définition

La boucle `while` permet de **répéter un bloc d'instructions tant qu'une condition est vraie** (`True`).

```
while condition:
    instructions
```

👉 La condition est testée **avant chaque tour** de boucle.

🔹 Exemple simple

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

🧠 Pourquoi ça s'arrête ?

- `compteur` est modifié dans la boucle
- la condition devient fausse → la boucle s'arrête

2️⃣ Initialisation + modification (très important)

Une boucle `while` doit toujours avoir :

1. une **variable initialisée avant**
2. une **modification dans la boucle**

❌ Sans modification → **boucle infinie**

```
while compteur <= 5:
    print(compteur)   # compteur ne change jamais
```

3️⃣ Boucle `while` vs boucle `for`

`while`	`for`
nombre de tours **inconnu**	nombre de tours **connu**
dépend d'une condition	dépend d'une séquence
très flexible	plus simple

👉 Le TD07 insiste sur `while` car elle est **plus générale**.

4️⃣ Tester les entrées de l'utilisateur

🔹 Principe

On **redemande une valeur** tant qu'elle n'est pas correcte.

```
n = int(input("Entrez un nombre positif : "))
while n < 0:
    n = int(input("Erreur. Entrez un nombre positif : "))
```

🧠 Pourquoi `while` ?

- on ne sait pas combien de fois l'utilisateur va se tromper

5️⃣ Boucle `while` dans une fonction

Le TD insiste sur l'écriture de **fonctions**.

```
def afficher_1_a_n(n):
    i = 1
    while i <= n:
        print(i)
        i += 1
```

🧠 Avantage :

- code réutilisable
- plus lisible
- plus facile à tester

6️⃣ Valeur sentinelle

🔹 Définition

Une **valeur sentinelle** est une valeur spéciale qui **indique l'arrêt de la boucle**.

Exemple classique : `-1`

```
valeur = int(input("Entrez un nombre (-1 pour terminer) : "))
while valeur != -1:
    print(valeur)
    valeur = int(input("Entrez un nombre (-1 pour terminer) : "))
```

🧠 Ici :

- la boucle continue tant que la valeur **n'est pas** `-1`

7️⃣ Accumulation (somme, moyenne, compteur)

On utilise souvent des **variables accumulatrices**.

Exemple : somme

```
somme = 0
valeur = int(input("Entrez un nombre (-1 pour terminer) : "))

while valeur != -1:
    somme += valeur
    valeur = int(input("Autre nombre : "))
```

🧠 Même principe pour :

- compteur (`nb += 1`)
- moyenne (`somme / nb`)
- maximum / minimum

8️⃣ Petit jeu (nombre secret)

🔹 Principe

On répète tant que la réponse est fausse.

```
while proposition != secret:
    proposition = int(input("Essayez encore : "))
```

🧠 La condition dépend de la **comparaison** (`!=`, `<`, `>`).

9️⃣ Aider l'utilisateur (conditions dans la boucle)

```
if proposition < secret:
    print("Trop petit")
else:
    print("Trop grand")
```

👉 Combinaison de :

- `while`
- `if / else`

🔟 Erreurs classiques à éviter

❌ oublier de modifier la variable de condition
❌ condition toujours vraie
❌ ne pas gérer le cas “aucune valeur entrée”
❌ confondre `while` et `if`

🧠 Résumé final (à apprendre)

- `while` = répéter **tant que** une condition est vraie
- toujours :
  - initialiser
  - modifier
- idéal pour :
  - validation d’entrées
  - jeux
  - lectures avec sentinelle
- souvent combinée avec :
  - `if / else`
  - fonctions
  - accumulateurs

TD08 _ boucle `for`

🔹 Définition

Une boucle `for` permet de **répéter un bloc d’instructions un nombre de fois déterminé à l’avance**, en parcourant une **séquence**.

```
for variable in sequence:
    instructions
```

🧠 À chaque itération :

- `variable` prend une nouvelle valeur de `sequence`
- le bloc est exécuté une fois

2️⃣ La variable de boucle

```
for i in [1, 2, 3, 4, 5]:
    print(i)
```

- `i` est la **variable de boucle**
- elle prend successivement les valeurs `1`, `2`, `3`, `4`, `5`

👉 Une itération = un passage dans la boucle.

3️⃣ Boucle `for` avec une liste

🔹 Principe

La boucle parcourt **chaque élément de la liste**.

```
for mot in ["une", "boucle", "for"]:
    print(mot)
```

🧠 Peu importe le type des éléments :

- entiers
- réels
- chaînes
- listes

4️⃣ La fonction `range()`

🔹 À quoi sert `range()` ?

Créer automatiquement une **suite de nombres**.

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

➡️ Affiche `0 1 2 3 4`

🔹 Formes de `range()`

Écriture	Signification
`range(n)`	de `0` à `n-1`
`range(a, b)`	de `a` à `b-1`
`range(a, b, pas)`	avec un pas

Exemples :

```
range(1, 6)      # 1 2 3 4 5
range(0, 10, 2)  # 0 2 4 6 8
```

🧠 Le **deuxième nombre n'est jamais inclus**.

5️⃣ Boucle `for` avec du texte

Une chaîne est une **séquence de caractères**.

```
for lettre in "esi":
    print(lettre)
```

🧠 La variable de boucle prend chaque caractère :

- `'e'`, puis `'s'`, puis `'i'`

6️⃣ Compter les itérations (sans `enumerate`)

```
mot = "for"
i = 1

for lettre in mot:
    print("La lettre", i, "est un", lettre)
    i += 1
```

🧠 On gère le compteur **manuellement**.

7️⃣ La fonction `enumerate()`

🔹 À quoi sert `enumerate()` ?

Fournir :

- la **position**
- la **valeur**

en même temps.

```
for i, lettre in enumerate("boucle"):
    print(i, lettre)
```

🧠

- `i` = indice (commence à 0)
- `lettre` = caractère

⚠️ Pour commencer à 1 :

```
enumerate("boucle", start=1)
```

8️⃣ Boucle `for` et calculs

La boucle `for` est idéale pour :

- accumuler une somme
- tester chaque élément
- faire des calculs répétés

Exemple : somme des carrés

```
somme = 0
for i in range(1, 101):
    somme += i*i
```

9️⃣ `for` + conditions (`if`)

```
for lettre in "python":
    if lettre in "aeiouy":
        print(lettre, "voyelle")
    else:
        print(lettre, "consonne")
```

🧠 Combinaison très fréquente :

- boucle = parcourir
- condition = décider

🔟 Erreurs classiques à éviter

❌ oublier les deux points `:`
❌ mauvaise indentation
❌ confondre `range(n)` et `range(1, n)`
❌ penser que `range(10)` va jusqu'à 10
❌ modifier la séquence pendant la boucle

🧠 Résumé final (à retenir)

- `for` = parcourir une séquence
- la variable de boucle change automatiquement
- `range()` génère des nombres
- une chaîne est parcourable caractère par caractère
- `enumerate()` donne indice + valeur
- souvent combinée avec `if`

TD09 : Boucles `for` imbriquées

1️⃣ Qu'est-ce qu'une boucle imbriquée ?

🔹 Définition

Une **boucle imbriquée** est une **boucle placée à l'intérieur d'une autre boucle**.

```
for i in range(...):        # boucle extérieure
```

```
    for j in range(...):    # boucle intérieure
        instruction
```

🧠 **Règle essentielle**
👉 Pour **chaque itération** de la boucle extérieure, la boucle intérieure s'exécute **complètement**.

2️⃣ Rôle des deux boucles

- **Boucle extérieure** → gère les *lignes*, les *groupes*, les *grands ensembles*
- **Boucle intérieure** → gère les *colonnes*, les *éléments d'une ligne*, les *détails*

Exemple conceptuel

```
for ligne in range(3):
    for colonne in range(4):
        print("*", end="")
    print()
```

🧠

- boucle extérieure → 3 lignes
- boucle intérieure → 4 caractères par ligne

3️⃣ Boucles imbriquées et tables de multiplication

Principe

- table de 2, puis table de 3, … → **boucle extérieure**
- multiplier de 1 à 10 → **boucle intérieure**

```
for table in range(2, 11):
    for i in range(1, 11):
        print(i, "x", table, "=", i * table)
```

🧠 On parcourt **toutes les combinaisons possibles**.

4️⃣ Tester des nombres (ex : nombres premiers)

Principe

- boucle extérieure → chaque nombre à tester
- boucle intérieure → tous ses diviseurs possibles

```
for i in range(2, nombre):
    if nombre % i == 0:
        # pas premier
```

🧠 Une boucle sert à **tester un nombre**, l'autre à **vérifier une condition répétée**.

5️⃣ Dessins dans la console (cas très important)

Les dessins utilisent **presque toujours** des boucles imbriquées.

Exemple : carré

```
for i in range(n):        # lignes
    for j in range(n):    # colonnes
        print("*", end="")
    print()
```

🧠

- `print(end="")` → reste sur la même ligne
- `print()` seul → retour à la ligne

6️⃣ Formes géométriques (triangles, pyramides)

Principe général

- la boucle extérieure contrôle la **hauteur**
- la boucle intérieure contrôle la **largeur de chaque ligne**

Exemple : triangle

```
for i in range(1, n+1):
    for j in range(i):
        print("*", end="")
    print()
```

🧠 Le nombre d'itérations de la boucle intérieure **dépend de la boucle extérieure**.

7️⃣ Pyramide numérique (exercice difficile)

Idée clé

Une seule ligne peut nécessiter **plusieurs boucles intérieures**.

```
for i in range(1, n+1):
    for j in range(i, 0, -1):
        print(j, end="")
    for j in range(2, i+1):
        print(j, end="")
    print()
```

🧠

- 1re boucle : nombres décroissants
- 2e boucle : nombres croissants
- ensemble → forme symétrique

8️⃣ Indentation (cruciale)

Les boucles imbriquées dépendent **entièrement de l'indentation**.

❌ Mauvaise indentation → logique fausse
✔️ Bonne indentation → structure correcte

```
for i in range(3):
    for j in range(3):
        print("*")
```

9️⃣ Quand utiliser des boucles imbriquées ?

Utilise-les quand :

- tu as des **tableaux / grilles**
- tu fais des **dessins**
- tu testes **toutes les combinaisons**
- un traitement dépend de **plusieurs niveaux**

🧠 Résumé final (à retenir absolument)

- une boucle imbriquée = boucle dans une boucle
- la boucle intérieure s'exécute entièrement à chaque tour
- extérieur = structure générale
- intérieur = détail
- très utilisée pour :
    - dessins
    - tableaux
    - mathématiques
- `print(end="")` est indispensable pour dessiner
- indentation = logique du programme

TD10 _ Tuples et listes

Le TD10 a pour objectif principal de **comprendre les structures de données séquentielles** en Python et surtout **la différence fondamentale entre tuples et listes**.

1️⃣ Séquences et indexation

🔹 Notion

Les **tuples** et les **listes** sont des **séquences** :

- elles contiennent plusieurs valeurs
- accessibles par un **indice** qui commence à `0`

```
t = (10, 20, 30)
l = [10, 20, 30]

t[0]  # 10
l[1]  # 20
```

🧠 À retenir

👉 L'indice commence toujours à **0**, jamais à 1.

2️⃣ Les tuples ( )

🔹 Définition

Un **tuple** est une séquence :

- ordonnée
- indexée
- **immuable**

```
t = (1, 2, 3)
```

3️⃣ Immuabilité des tuples (notion clé)

🔹 Qu'est-ce que l'immuabilité ?

Une fois créé, un tuple :

- **ne peut pas être modifié**
- **ne peut pas changer de taille**

❌ Interdit :

```
t[0] = 5      # erreur
t.append(4)  # erreur
```

✔️ Autorisé :

```
t2 = t + (4,)   # nouveau tuple
```

🧠 Pourquoi utiliser un tuple ?

- données fixes
- sécurité (pas de modification accidentelle)

- plus rapide en lecture

4️⃣ Temps d’accès

🔹 Notion

Le **temps d’accès** mesure le temps nécessaire pour :

- lire
- parcourir
  une séquence.

Le TD montre que :

- parcourir **directement** un tuple est plus rapide
- utiliser un indice (`t[i]`) est plus lent

```
for v in t:      # plus rapide
    pass

for i in range(len(t)):
    v = t[i]     # plus lent
```

🧠 **Idée clé**
👉 Préférer `for v in sequence` quand c’est possible.

5️⃣ Découpage (slicing)

🔹 Définition

Le **slicing** permet d’extraire une **tranche** d’une séquence.

```
t = ('a', 'b', 'c', 'd', 'e')
t2 = t[2:4]   # ('c', 'd')
```

🧠 Règles importantes :

- début inclus
- fin exclue
- le résultat est une **nouvelle séquence**
- le type est conservé (tuple → tuple, liste → liste)

6️⃣ Les listes `[]`

🔹 Définition

Une **liste** est une séquence :

- ordonnée
- indexée

- **mutable**

```
l = [1, 2, 3]
l[0] = 10
```

🧠 **Différence majeure avec les tuples**
👉 Les listes **peuvent être modifiées**.

7️⃣ Mutabilité des listes (notion clé)

🔹 Conséquence

- on peut changer les valeurs
- on peut ajouter ou supprimer des éléments

```
l.append(4)
l.remove(2)
```

🧠 C’est ce qui rend les listes très flexibles.

8️⃣ Accès indexé vs accès direct (listes)

Le TD montre une différence importante :

```
for i in range(len(l)):
    l[i] = l[i] + 1   # modifie la liste

for v in l:
    v = v + 1        # ne modifie PAS la liste
```

🧠 **Pourquoi ?**

- `l[i]` accède à la vraie case mémoire
- `v` est juste une copie de la valeur

9️⃣ append, `extend` et +

🔹 append

Ajoute **un seul élément** à la liste.

```
l.append(5)
```

🔹 `extend`

Ajoute **plusieurs éléments** (contenu d’une autre liste).

```
l.extend([6, 7])
```

🔹 +

Crée une **nouvelle liste**.

```
l2 = l + [8, 9]
```

🧠 **Différence essentielle**

- `append` / `extend` → modifient la liste
- `+` → crée une nouvelle liste

🔟 Comparaison tuple / liste (essentiel)

Tuple	Liste
`()`	`[]`
immuable	mutable
plus sûr	plus flexible
données fixes	données évolutives

🧠 Résumé final à retenir absolument

- tuple = **immuable**
- liste = **mutable**
- indice commence à 0
- slicing : début inclus, fin exclue
- `for v in seq` est plus rapide que l'accès indexé
- `append` ≠ `extend` ≠ `+`
- attention à la modification réelle vs copie

TD11 _ Listes (suite)

1️⃣ Rappel : les listes sont mutables

🔹 Notion fondamentale

Une **liste** peut être **modifiée après sa création** :

- changer une valeur
- ajouter / supprimer des éléments
- changer sa taille

```
l = [1, 2, 3]
l[0] = 10
```

```
l.append(4)
```

🧠 C’est la grande différence avec les tuples.

2️⃣ Méthodes, fonctions et opérateurs sur les listes

Le TD insiste sur le fait de **bien distinguer** :

🔹 Ce qui modifie la liste

- `append`
- `extend`
- `remove`
- `reverse`
- `insert`
- `pop`
- `del`

👉 La liste originale change.

🔹 Ce qui crée une nouvelle liste

- `copy`
- opérateurs `+` et `*`
- slicing `l[i:j]`

👉 La liste de départ reste inchangée.

🔹 Ce qui retourne une valeur

- `index`
- `pop`
- `count`

🔹 Ce qui ne retourne rien (`None`)

- `append`
- `extend`
- `remove`
- `reverse`

⚠️ **Piège classique** :

```
l = l.append(5)   # l devient None
```

3️⃣ Différence entre modifier et insérer

```
l = [1, 2, 3]

l[1] = 9      # modifie → [1, 9, 3]
l.insert(1,9) # insère → [1, 9, 2, 3]
```

🧠 Modifier = remplacer
🧠 Insérer = décaler les éléments

4️⃣ Copie vs affectation (très important)

```
l1 = [1, 2, 3]
l2 = l1.copy()
l3 = l1
```

- `l2` → **vraie copie**
- `l3` → **même liste en mémoire**

🧠 Modifier `l3` modifie aussi `l1`.

5️⃣ Parcourir une liste en la modifiant (gros piège)

❌ Mauvais cas (boucle infinie)

```
for x in l:
    l.append(x)
```

✔️ Bon cas (copie)

```
for x in l[:]:
    l.append(x)
```

🧠 Pourquoi ?

- `l[:]` = copie figée
- `l` change pendant la boucle → comportement imprévisible

6️⃣ Création de listes aléatoires

🔹 Principe

Utiliser :

- `import random`
- `random.randint`
- `random.choice`

Exemple :

```
import random
l = [random.randint(0,5) for _ in range(10)]
```

🧠 Très utile pour tester des fonctions.

7️⃣ Listes en compréhension (notion clé)

🔹 Définition

Une **liste en compréhension** permet de créer une liste en **une seule ligne**.

```
[x for x in range(10) if x % 2 == 0]
```

Équivalent à :

```
l = []
for x in range(10):
    if x % 2 == 0:
        l.append(x)
```

🧠 Avantages :

- plus lisible
- plus court
- plus rapide

8️⃣ Traitements sur les listes (fonctions demandées)

Le TD te fait écrire des fonctions qui :

- retournent une **nouvelle liste**
- ou modifient la liste **sur place**

Exemples d'objectifs

- moyenne
- éléments communs
- suppression d'occurrences
- suppression de doublons consécutifs

🧠 Il faut toujours savoir :

- **si la liste de départ doit changer ou non**
- **ce que la fonction doit retourner**

9️⃣ Suppression d'éléments : pièges

`l.remove(x)`

- supprime la **première occurrence**
- erreur si `x` n'existe pas

`l.pop(i)`

- supprime et **retourne** l'élément

🔟 Ce que le TD veut absolument que tu maîtrises

✔️ méthodes de la classe `list`
✔️ différence **copie / référence**
✔️ modification vs création
✔️ listes en compréhension
✔️ pièges des boucles
✔️ fonctions qui retournent une liste
✔️ fonctions qui modifient sur place

🧠 Résumé final (à apprendre)

- une liste est **mutable**
- certaines méthodes retournent `None`
- `copy()` ≠ `=`
- ne jamais modifier une liste pendant son parcours
- listes en compréhension = outil clé
- toujours savoir **ce qui est modifié** et **ce qui est retourné**

TD12 _ Dictionnaires

1️⃣ Qu'est-ce qu'un dictionnaire ?

🔹 Définition

Un **dictionnaire** est une structure de données qui associe :

- une **clef**
- à une **valeur**

On parle de **paires clef → valeur**.

```
personne = {
    "nom": "Dupont",
    "age": 20
}
```

🧠 **Idée clé**
👉 En connaissant la **clef**, on retrouve **directement** la valeur.

2️⃣ Pourquoi on parle de “dictionnaire” ?

Comme dans un vrai dictionnaire :

- le **mot** = la clef
- la **définition** = la valeur

👉 On ne cherche pas par position (contrairement aux listes), mais **par identifiant**.

3️⃣ Contraintes importantes sur les clefs

🔹 Règles à retenir

- une clef est **unique**
- une clef doit être **hachable**
- types autorisés comme clefs :
    - `int`
    - `float`
    - `str`
    - `tuple`

❌ Interdit :

```
d = {[1, 2]: "erreur"}   # liste non hachable
```

✔️ Autorisé :

```
d = {(1, 2): "ok"}
```

🧠 **Pourquoi ?**
Les dictionnaires utilisent un mécanisme de **hachage** pour accéder rapidement aux valeurs.

4️⃣ Les valeurs (beaucoup plus libres)

Les **valeurs** peuvent être :

- des nombres
- des chaînes
- des listes
- des tuples
- même d’autres dictionnaires

```
etudiant = {
    52104: {
        "nom": "Apollinaire",
        "groupe": "A321"
    }
}
```

5️⃣ Création et modification d’un dictionnaire

🔹 Création

```
d = {"a": 1, "b": 2}
```

🔹 Ajouter ou modifier

```
d["c"] = 3      # ajout
d["a"] = 10     # modification
```

🧠 Donner deux valeurs à une même clef → **la dernière écrase l’ancienne**.

6️⃣ Accès aux valeurs

🔹 Accès direct

```
d["a"]
```

❌ Problème si la clef n’existe pas → **KeyError**

🔹 Accès sécurisé avec `get()`

```
d.get("x", 0)
```

🧠 Si `"x"` n’existe pas → retourne `0` au lieu d’une erreur.

7️⃣ Tester l’existence d’une clef

```
"age" in personne
```

👉 Teste **les clefs**, pas les valeurs.

8️⃣ Parcourir un dictionnaire

🔹 Parcourir les clefs

```
for clef in d:
    print(clef)
```

🔹 Parcourir clefs + valeurs

```
for clef, valeur in d.items():
    print(clef, valeur)
```

🧠 `.items()` est **fondamental**.

9️⃣ Méthodes essentielles à connaître

Méthode	Rôle
`len(d)`	nombre de clefs
`d.keys()`	liste des clefs

Méthode	Rôle
`d.values()`	liste des valeurs
`d.items()`	paires (clef, valeur)
`d.pop(clef)`	supprime et retourne
`del d[clef]`	supprime
`d.clear()`	vide le dictionnaire
`d.copy()`	copie indépendante

🔟 Usages typiques des dictionnaires (très important)

✔️ Associations

```
groupe_prof = {"A311": "bis"}
```

✔️ Comptage (fréquences)

```
{"bej": 2, "cuv": 2, "hal": 1}
```

✔️ Objets simples

```
{"nom": "Jean", "age": 20}
```

✔️ Mini base de données

```
{
  52104: {"nom": "Dupont", "groupe": "A321"}
}
```

1️⃣1️⃣ Erreurs et pièges classiques

❌ accéder à une clef inexistante
❌ utiliser une liste comme clef
❌ confondre clefs et valeurs
❌ penser qu'un dictionnaire est ordonné par indice

🧠 Résumé final (à apprendre)

- dictionnaire = **clef → valeur**
- accès **rapide**, sans indice
- clefs :
  - uniques
  - hachables
- valeurs :
  - de n'importe quel type

- `.get()` évite les erreurs
- `.items()` est essentiel pour parcourir
- idéal pour :
    - comptage
    - associations
    - structures complexes

TD14 _ Erreurs et Exceptions

1️⃣ Qu'est-ce qu'une erreur ?

🔹 Idée clé

Une **erreur** signifie que le programme **ne peut pas continuer normalement** ou qu'il **donne un mauvais résultat**.

Le TD distingue **3 grands types d'erreurs**.

2️⃣ Les trois types d'erreurs

1️⃣ Erreur de syntaxe

➡️ Python **ne comprend pas** le code.

Exemples :

```
if x > 0
print("bonjour")
```

Causes fréquentes :

- oubli de `:`
- mauvaise indentation
- mot-clé mal écrit
- parenthèse oubliée

👉 Le programme **ne démarre pas**.

2️⃣ Erreur d'exécution (exception)

➡️ Le programme démarre, mais **plante pendant l'exécution**.

Exemples :

```
1 / 0              # ZeroDivisionError
int("abc")         # ValueError
```

```
liste[10]          # IndexError
```

🧠 Ces erreurs lèvent des **exceptions**.

3️⃣ Erreur sémantique (logique)

➡️ Le programme s'exécute **sans erreur**, mais le résultat est **faux**.

Exemple :

```
1 // 2   # vaut 0, mais ce n'est pas le résultat attendu
```

⚠️ **La plus dangereuse**, car Python ne signale rien.

3️⃣ Qu'est-ce qu'une exception ?

🔹 Définition

Une **exception** est un **signal d'erreur levé par Python** lorsqu'un problème survient à l'exécution.

Quand une exception est levée :

- le programme **s'arrête**
- un message d'erreur est affiché
  ➡️ sauf si on l'intercepte.

4️⃣ Exceptions prédéfinies importantes

Le TD insiste sur celles-ci :

Exception	Quand elle apparaît
`ValueError`	valeur invalide (`int("abc")`)
`TypeError`	types incompatibles (`"a" + 1`)
`ZeroDivisionError`	division par zéro
`IndexError`	indice hors limites
`ModuleNotFoundError`	module inexistant
`NameError`	variable non définie

5️⃣ Intercepter une exception : `try / except`

🔹 Pourquoi ?

Pour éviter que le programme **plante** et permettre de **réagir intelligemment**.

Syntaxe de base

```
try:
    # code risqué
except NomDeLException:
    # code exécuté en cas d'erreur
```

Exemple simple

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Division par zéro interdite")
```

🧠 Le programme **continue** au lieu de s'arrêter.

6️⃣ Rendre une entrée utilisateur robuste

Problème

```
b = int(input("Entrez un entier : "))
```

➡️ plante si l'utilisateur entre du texte.

Solution du TD (très importante)

```
b = None
while b is None:
    try:
        b = int(input("Entrez un entier : "))
    except ValueError:
        print("Ce n'était pas un entier. Essayez encore.")
```

🧠 Combinaison essentielle :

- `while`
- `try / except`

7️⃣ Lever une exception avec `raise`

🔹 Pourquoi lever soi-même une exception ?

Parce qu'une **fonction est responsable de la validité de ses paramètres**.

Exemple

```
def aire_cercle(rayon):
    if rayon < 0:
        raise ValueError("Le rayon doit être positif")
    return 3.14 * rayon**2
```

🧠

- si le paramètre est invalide → la fonction **refuse de travailler**
- l'erreur est claire et contrôlée

8️⃣ Code mort après `raise`

🔹 Notion clé

Dès qu'une exception est levée :

- la fonction **s’arrête immédiatement**

```
def f():
    raise Exception
    print("Ceci ne s'affichera jamais")  # code mort
```

👉 Tout code après `raise` est **code mort**.

9️⃣ Propagation des exceptions (pile d’appels)

🔹 Principe

Si une fonction lève une exception :

- Python remonte les appels
- jusqu’à trouver un `try / except`
- sinon → le programme plante

```
def f3():
    raise Exception

def f2():
    f3()

def f1():
    f2()

f1()
```

🧠 Sans `try/except`, l’erreur remonte jusqu’au script principal.

🔟 Exception interceptée = programme vivant

```
try:
    f1()
except Exception:
    print("Erreur interceptée")

print("Le programme continue")
```

🧠 Le TD montre bien la différence entre :

- **code mort**
- **code vivant après interception**

1️⃣1️⃣ Objectif final du TD

Le TD veut t’apprendre à :

- ✔️ identifier les types d’erreurs
- ✔️ reconnaître les exceptions courantes
- ✔️ intercepter les erreurs d’exécution

- ✔️ lever des exceptions pertinentes
- ✔️ écrire du **code robuste**
- ✔️ éviter les plantages brutaux

🧠 Résumé final à retenir absolument

- erreurs :
  - syntaxe
  - exécution (exceptions)
  - sémantique
- une exception **interrompt le programme**
- `try / except` permet de **continuer**
- `raise` permet de **refuser une situation invalide**
- code après `raise` = **code mort**
- intercepter = programme robuste