

Introduction à Python

Cours 2 - Types, listes, dictionnaires, logique

Etienne Dagorn

Université de Lille - LEM

18 septembre 2025

Objectifs de la séance

- ➊ Comprendre les **types de base** (int, float, str, bool) et les conversions.
- ➋ Savoir manipuler **listes** (séquences) et **dictionnaires** (clé → valeur).
- ➌ Écrire des **boucles** et une **fonction** simple.

RÉVISIONS

Ce qu'il faut retenir de la séance 1

Révision #1 - La programmation, c'est quoi ?

- ⇒ Écrire une **suite d'instructions précises** que la machine exécute.
- ⇒ Cycle : **Problème** → **Modélisation** → **Algorithme** → **Programme** → **Résultat**.
- ⇒ **Algorithme** : procédure finie, non ambiguë, reproductible.
- ⇒ **Programme** : implémentation concrète dans un langage (ex. Python).

Révision #1 - La programmation, c'est quoi ?

- ⇒ Écrire une **suite d'instructions précises** que la machine exécute.
- ⇒ Cycle : **Problème** → **Modélisation** → **Algorithme** → **Programme** → **Résultat**.
- ⇒ **Algorithme** : procédure finie, non ambiguë, reproductible.
- ⇒ **Programme** : implémentation concrète dans un langage (ex. Python).

Idée clé : on passe d'un monde flou à des **étapes vérifiables**.

Révision #2 - Penser comme une machine

- ❶ **Décomposition** : découper un problème.
- ❷ **Reconnaissance de motifs** : repérer les schémas répétitifs.
- ❸ **Abstraction** : garder seulement l'essentiel.
- ❹ **Conception d'algorithmes** : écrire des étapes non ambiguës.

Révision #2 - Penser comme une machine

- ❶ **Décomposition** : découper un problème.
- ❷ **Reconnaissance de motifs** : repérer les schémas répétitifs.
- ❸ **Abstraction** : garder seulement l'essentiel.
- ❹ **Conception d'algorithmes** : écrire des étapes non ambiguës.

Quelles étapes retrouverais-tu dans le *choix d'un stand à la braderie* ?

Révision #3 - Tracer un algorithme

- ⇒ **Suivre** le programme **ligne par ligne**.
- ⇒ **Observer** après chaque ligne : état des variables et sorties.
- ⇒ **Utiliser** cette trace pour **comprendre, déboguer, expliquer**.

Révision #3 - Tracer un algorithme

- ⇒ **Suivre** le programme **ligne par ligne**.
- ⇒ **Observer** après chaque ligne : état des variables et sorties.
- ⇒ **Utiliser** cette trace pour **comprendre, déboguer, expliquer**.

Ligne	Variables	Sortie / Commentaire
1	$x = 2$	init
2	$x = 3$	après incrément

Révision #4 - Packages et écosystème

- ⇒ **Langage** = règles pour écrire du code.
- ⇒ **Librairie standard** = outils de base (fichiers, maths ...).
- ⇒ **Packages externes** = briques spécialisées (*pandas*, *matplotlib* ...).
- ⇒ **Gestionnaire de paquets** = installation centralisée (**pip**).

Révision #4 - Packages et écosystème

- ⇒ **Langage** = règles pour écrire du code.
- ⇒ **Librairie standard** = outils de base (fichiers, maths ...).
- ⇒ **Packages externes** = briques spécialisées (*pandas*, *matplotlib* ...).
- ⇒ **Gestionnaire de paquets** = installation centralisée (*pip*).

Principe : **stdlib** pour le simple, **packages** pour le complexe.

Quiz éclair

- ❶ Quelle est la différence entre **algorithme** et **programme** ?
- ❷ Un **algorithme** est une suite d'étapes logiques abstraites ; un **programme** est la *traduction concrète en code* d'un algorithme.
- ❸ Que garantit un **invariant** dans une simulation ?

Quiz éclair

- ❶ Quelle est la différence entre **algorithme** et **programme** ?
- ❷ Un **algorithme** est une suite d'étapes logiques abstraites ; un **programme** est la *traduction concrète en code* d'un algorithme.
- ❸ Que garantit un **invariant** dans une simulation ?
Un invariant est une **propriété qui reste vraie à chaque étape** de la simulation, ce qui permet de vérifier la cohérence du modèle.

Rappel Cours 1 $\Rightarrow$ Cours 2

- $\Rightarrow$ **Pseudocode** $\Rightarrow$ aujourd'hui on le **traduit en Python** (types, listes, dicos, fonctions).
- $\Rightarrow$ **Organisation de projet** (proj/ data/ notebooks/ src/ outputs/) $\Rightarrow$ on crée un **mini-module** dans src/.
- $\Rightarrow$ **Packages** (stdlib vs externes) $\Rightarrow$ on **importe** ce dont on a besoin, proprement.

Fil rouge : à partir de listes/dictionnaires, calculer un **indice de prix** et le **tester**, **puis** le rejouer dans un notebook.

PARTIE I

Variables

Les briques de base

Du pseudocode au Python (ex. total moules-frites)

Pseudocode (Cours 1) :

```
1. total <- 0
2. total <- nb * (prix_m + prix_f)
3. SI budget < total ALORS STOP
4. SINON afficher total
```

Python (Cours 2) :

```
nb, prix_m, prix_f, budget = 3, 12, 4, 50
total = nb * (prix_m + prix_f)
if budget < total:
    print("Budget insuffisant")
else:
    print("Total =", total)
```

Lien : on remplace les étapes par des **affectations** (types int/float) et des **tests logiques** (if).



```
1. nb <- 3 # int
2. produit <- "moules" # str
3. promo <- vrai # bool
4. prix <- {"moules": 12.0, #
            "frites": 4.0}
5. panier <- ["moules", # list
             "frites"]
6. total <- nb * prix[produit]
7. SI promo ALORS total <- tot
8. AFFICHER produit, total
```

Qu'est-ce qu'un "type" en Python ?

Test

- ⇒ Que représentent ces données? "42", 42, True, [42], "Alice"
- ⇒ Peut-on les additionner? les trier? les comparer?

Donnée	Type Python	Signification	Opérations possibles
"42"	str (chaîne)	Texte contenant " 42 "	"42"+"42" → "4242"
42	int (entier)	Nombre entier	42+42 → 84
True	bool	Valeur logique (1 ou 0)	True+True → 2
[42]	list	Liste contenant un entier	[42]+[42] → [42,42]
"Alice"	str (chaîne)	Texte	"Alice" < "Bob" → True

Qu'est-ce qu'un "type" en Python ?

Test

- ⇒ Que représentent ces données ? "42", 42, True, [42], "Alice"
- ⇒ Peut-on les additionner ? les trier ? les comparer ?

Donnée	Type Python	Signification	Opérations possibles
"42"	str (chaîne)	Texte contenant " 42 "	"42"+"42" → "4242"
42	int (entier)	Nombre entier	42+42 → 84
True	bool	Valeur logique (1 ou 0)	True+True → 2
[42]	list	Liste contenant un entier	[42]+[42] → [42,42]
"Alice"	str (chaîne)	Texte	"Alice" < "Bob" → True

- ⇒ Chaque donnée appartient à un **type** : nombre entier, texte, booléen, etc.
- ⇒ Le **type** indique comment Python les stocke et quelles opérations sont permises.
- ⇒ Comprendre les types est **indispensable** pour manipuler et transformer des données.

Récapitulatif

Type Python	À quoi ça sert ?	Exemple (code)	Mu- table ?
int	Entiers (comptages, identifiants)	age = 22	Non
float	Réels binaires (prix, mesures)	prix = 3.14	Non
decimal.Decimal	Réels décimaux exacts (finances)	from decimal import Decimal tva = Decimal("0.20")	Non
str	Texte Unicode (libellés)	nom = "Alice"	Non
bool	Logique Vrai/Faux (tests)	majeur = True	Non
NoneType	Absence de valeur	val = None	N/A
list	Séquence ordonnée, modifiable	notes = [12, 14, 9]	Oui
tuple	Séquence ordonnée, immuable	coord = (50.6, 3.1)	Non
dict	Assoc. <i>clé</i> → <i>valeur</i>	prix = {"pain": 1.2, "lait": 0.95}	Oui
set	Ensemble (sans doublons)	U = {"A", "B", "B"} ⇒ {"A", "B"}	Oui
range	Suite arithmétique paresseuse	r = range(0, 10, 2)	Non
bytes	Octets bruts (I/O, binaires)	sig = b"PDF\x25"	Non

Astuce : `type(x)` donne le type ; conversions usuelles `int("42")`, `float("3.5")`, `str(2025)`. *Mutable* ? = modifiable en place (sinon, toute "modif" crée un nouvel objet).

Types & conversions - Q1

Question : Que renvoient `type(3)`, `type(3.0)`, `type("3")` ?

Réponse :

⇒ `type(3) ⇒ int`

⇒ `type(3.0) ⇒ float`

⇒ `type("3") ⇒ str`

Types & conversions - Q2

Question : Pourquoi `42 + "3"` plante-t-il ? Donne 2 corrections valides.

Types & conversions - Q2

Question : Pourquoi `42 + "3"` plante-t-il ? Donne 2 corrections valides.

Réponse :

⇒ Erreur : `TypeError` (on mélange `int` et `str`).

Types & conversions - Q2

Question : Pourquoi `42 + "3"` plante-t-il ? Donne 2 corrections valides.

Réponse :

- ⇒ Erreur : `TypeError` (on mélange `int` et `str`).
- ⇒ Correction 1 (numérique) : `42 + int("3") ⇒ 45`

Types & conversions - Q2

Question : Pourquoi `42 + "3"` plante-t-il ? Donne 2 corrections valides.

Réponse :

- ⇒ Erreur : `TypeError` (on mélange `int` et `str`).
- ⇒ Correction 1 (numérique) : `42 + int("3") ⇒ 45`
- ⇒ Correction 2 (chaîne) : `str(42) + "3" ⇒ "423"`

Checkpoint - Types & conversions

Objectif : corriger le script pour afficher un total valide.

```
nb = "3" # <- devrait être un entier
prix_unitaire = "12,5" # <- virgule locale
promo = True

# 1) Convertir proprement nb, prix_unitaire
# 2) Si promo, appliquer -10%
# 3) Afficher: "Total TTC = XX.YY"

# Votre code ici...
```

Pistes : `int()`, `float(... .replace(",", ".", "."))`, `round(x, 2)`.

Erreurs fréquentes - Types & conversions

- ⇒ **Mélange texte et nombres** `"2" + 2` ⇒ `TypeError` *Solution : convertir l'un des deux, par ex. `int("2")+2` ou `"2"+str(2)`.*
- ⇒ **Virgule vs point décimal (localisation)** `float("12,5")` ⇒ `ValueError` *Solution : corriger la chaîne : `float("12,5".replace(",",".", "."))`.*
- ⇒ **Division entière non voulue** `3 // 2 == 1` (division entière, arrondi vers le bas) *Solution : utiliser `/` pour obtenir 1.5.*
- ⇒ **Erreurs d'arrondis binaires** `0.1 + 0.2 == 0.30000000000000004` *Car les nombres décimaux ne sont pas représentables exactement en binaire. Solution : arrondir (`round(..., 2)`) ou utiliser `Decimal`.*
- ⇒ **Valeurs manquantes (None)** `None + 3` ⇒ `TypeError` *Solution : tester avant de calculer (`if x is not None: ...`).*

Astuce : utiliser `type(x)` et `print()` régulièrement pour vérifier ce que contient vraiment une variable.

PARTIE II

Base de données

Les maisons

Exemple bdd

interview_id	age	age5	gender	uda5	csp	taille d'agglomération	Q1_1_0	Q1_1_1	Q1_1_2	Q1_1_3	Q1_1_4	Q2_1	Q3_1_1er enfant
159253	47	35-49	female	Nord-Ouest	CSP+	Agglo <20K hab	3					3	14
164468	34	18-34	male	Réunion Parisienne	CSP-	Rural	1	2	3			4	12
159222	55	50-64	male	Nord-Est	CSP-	Agglo 20k-100K hab	3	5				2	20
159223	57	50-64	male	Nord-Est	CSP+	Agglo <20K hab	3	4				2	15
159220	41	35-49	female	Réunion Parisienne	CSP-	Agglo <20K hab	1	3				2	11
159221	31	18-34	female	Sud-Est	Inactif	Rural	2	3				2	7
159228	34	18-34	female	Nord-Est	CSP-	Agglo 20k-100K hab	1	3				2	12
159229	51	50-64	male	Nord-Ouest	CSP-	Rural	3	4				2	16
159233	42	35-49	female	Sud-Est	CSP-	Agglo <20K hab	3	4				3	16
159239	43	35-49	female	Nord-Est	CSP-	Rural	2	3				3	11
159324	40	35-49	female	Nord-Est	CSP-	Agglo >100K hab	2	3	4			3	17
159247	34	18-34	female	Nord-Est	Inactif	Agglo >100K hab	2	3				3	0
159261	45	35-49	female	Nord-Est	CSP-	Agglo 20k-100K hab	3					1	13
159262	39	35-49	female	Sud-Est	Inactif	Agglo >100K hab	2	3				3	14
159289	46	35-49	female	Sud-Est	CSP+	Agglo >100K hab	3	4				2	14
159296	40	35-49	male	Nord-Est	CSP-	Agglo >100K hab	1	2	3	4		4	4
159302	50	50-64	male	Sud-Est	CSP-	Agglo <20K hab	3	4				3	17
159304	43	35-49	male	Nord-Ouest	CSP-	Rural	3					1	11
159225	66	65	male	Nord-Est	Inactif	Agglo >100K hab	3					2	13
159226	41	35-49	female	Nord-Ouest	Inactif	Agglo >100K hab	2	3				3	13
159227	41	35-49	female	Sud-Est	CSP+	Agglo 20k-100K hab	3	4	5			3	19
159230	34	18-34	female	Nord-Est	Inactif	Agglo >100K hab	1	3				2	11
159245	49	35-49	male	Nord-Ouest	CSP-	Agglo <20K hab	1	2	3	4		4	15
159252	73	65	female	Sud-Est	Inactif	Agglo <20K hab	3					1	13

Panorama des types de bases de données

⇒ Fichiers plats : CSV, JSON, Parquet

- Simples, portables, bons pour le partage et la reproduction (Git).
- Parquet : format *colonnaire* compressé (analyse rapide).
- Limites : pas de contraintes d'intégrité, requêtes limitées.

⇒ Relationnelles (SQL) : *PostgreSQL*, *SQLite*, *MySQL*

- Tables reliées par **clés**, schéma explicite, **ACID**, jointures.
- Idéal pour données structurées, intégrité, historiques “ propres ”.

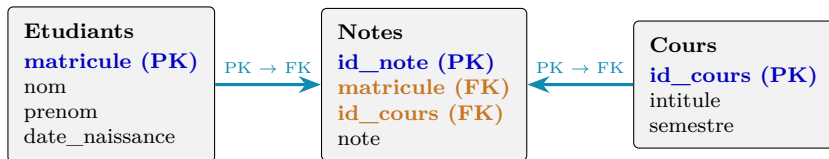
⇒ NoSQL (selon usage)

- *Documents* (MongoDB), *Clé-valeur* (Redis), *Colonnes* (Cassandra), *Graphes* (Neo4j).
- Schéma souple, scalabilité horizontale ; cohérence souvent *éventuelle*.

⇒ Spécialisées : séries temporelles (*TimescaleDB*, *InfluxDB*), spatial (PostGIS).

⇒ Entrepôts / Data lakes : *BigQuery*, *Snowflake*, *Redshift* ; fichiers Parquet/ORC sur objet (*S3*).

Relationnel : tables, PK et FK

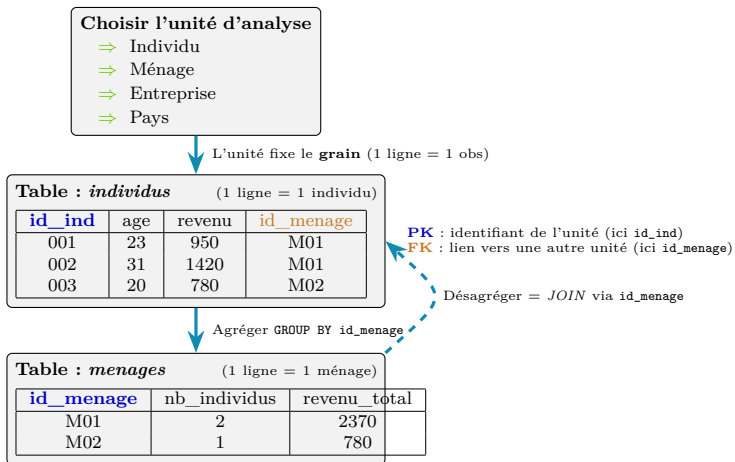


Unité d'analyse :

- *Etudiants* : 1 ligne = 1 étudiant ;
- *Cours* : 1 ligne = 1 cours ;
- *Notes* : 1 ligne = 1 *observation* (étudiant, cours, note).

Jointure (ex.) : moyenne par étudiant $\Rightarrow$ *GROUP BY matricule*.

Unité d'analyse → structure de la base (compact)



Règle : l'unité d'analyse détermine la **PK**, le **grain** de la table et les **jointures/agrégations** possibles.

Exemple - Instagram et enjeux RGPD

Comment Instagram structure ses données :

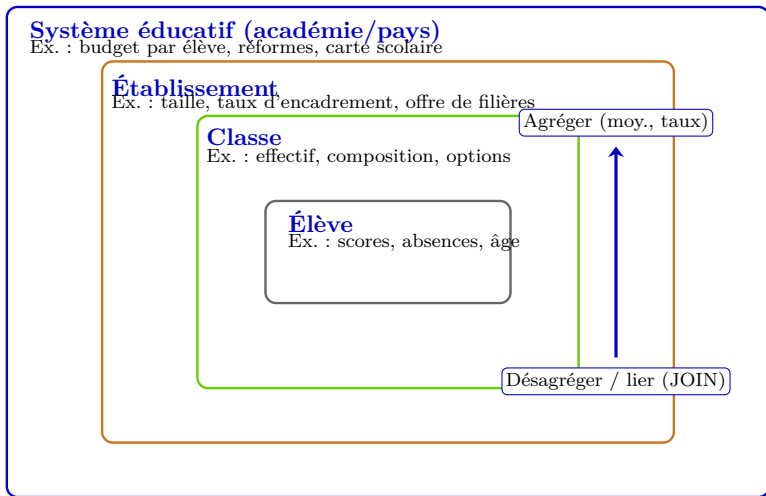
- ⇒ Utilisateurs : `user_id` (PK), nom, âge, e-mail
- ⇒ Posts : `post_id` (PK), texte, image, `user_id` (FK)
- ⇒ Likes : `like_id` (PK), `user_id` (FK), `post_id` (FK)
- ⇒ Campagnes marketing : clics, hashtags, préférences (index sur hashtags)

Enjeux RGPD :

- ⇒ **Minimisation** : ne collecter que les données utiles
- ⇒ **Consentement** et droit à l'oubli (suppression à la demande)
- ⇒ **Anonymisation / pseudonymisation** pour les analyses marketing

Structurer les données = relier efficacement tout en protégeant les individus.

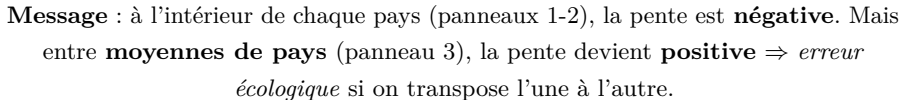
Unités d'analyse imbriquées - *éducation*



Choisir l'**unité** = choisir le **grain de la table** (*1 ligne = 1 élève / 1 classe / 1 établissement*).
Les **PK/FK** suivent ce grain (ex. PK = id_élève; FK = id_classe).

Pourquoi c'est important ?

- ⇒ **Un mauvais choix d'unité peut fausser les conclusions.**
 - Exemple : calculer une moyenne par pays alors que l'effet réel est individuel.
 - Les corrélations au niveau agrégé peuvent disparaître (ou s'inverser) au niveau individuel.
- ⇒ **Erreur écologique :**
 - On observe une relation au niveau des groupes (pays, régions, classes...),
 - mais on en déduit à tort qu'elle vaut pour chaque individu.
 - *Corrélation agrégée $\neq$ corrélation individuelle.*



INTERMÈDE CULTURE DATA

Comprendre le cycle de vie d'une donnée

Avant de manipuler, il faut savoir ce qu'on manipule

Du concept à la donnée : un exemple concret

Étape 1 - Question de recherche

- ⇒ Est-ce que l'usage des réseaux sociaux réduit le sommeil des étudiant·e·s ?

Étape 2 - Hypothèse testable

- ⇒ Les étudiant·es qui passent plus de 3h/jour sur les réseaux sociaux dorment en moyenne moins de 7h/nuit.

Étape 3 - Opérationnalisation (mesures)

- ⇒ **Variable dépendante (VD)** : durée moyenne de sommeil (en heures/nuits) - *quantitative*.
- ⇒ **Variable indépendante (VI)** : temps passé (en heures/jour) - *quantitative*.
- ⇒ **Variables de contrôle** : âge, sexe, niveau d'études, activité sportive.

Du concept à la donnée : un exemple concret

Étape 1 - Question de recherche

- ⇒ Est-ce que l'usage des réseaux sociaux réduit le sommeil des étudiant·es ?

Étape 2 - Hypothèse testable

- ⇒ Les étudiant·es qui passent plus de 3h/jour sur les réseaux sociaux dorment en moyenne moins de 7h/nuit.

Étape 3 - Opérationnalisation (mesures)

- ⇒ **Variable dépendante (VD)** : durée moyenne de sommeil (en heures/nuits) - *quantitative*.
- ⇒ **Variable indépendante (VI)** : temps passé (en heures/jour) - *quantitative*.
- ⇒ **Variables de contrôle** : âge, sexe, niveau d'études, activité sportive.

Chaque donnée collectée doit avoir un **rôle clair** dans le **raisonnement scientifique**.

Origine des données

Trois grandes sources possibles :

- ⇒ **Collecte active** : enquêtes, formulaires, expériences, capteurs.
 - Avantages : données ciblées, adaptées à la question de recherche.
 - Limites : coûteuses, lentes à produire, risque de biais de déclaration.
- ⇒ **Collecte passive** : traces numériques (clics, géolocalisation, achats ...).
 - Avantages : données riches, en continu et à grande échelle.
 - Limites : non prévues pour la recherche, bruit élevé, contraintes légales (vie privée).
- ⇒ **Production automatique** : logs serveurs, objets connectés, bases transactionnelles.
 - Avantages : volume, exhaustivité, enregistrement en temps réel.
 - Limites : très hétérogènes, souvent non documentées, nettoyage complexe.

Ces données sont souvent **massives, variées et non structurées** :
il faut les **documenter, nettoyer et structurer** avant toute analyse.

Collecter des données : respecter la vie privée

Risques principaux :

- ⇒ **Atteinte à la vie privée** : données personnelles mal anonymisées (noms, emails, localisation ...)
- ⇒ **Détournement d'usage** : réutilisation non prévue (marketing, surveillance, profilage ...)
- ⇒ **Fuites / piratages** : stockage non sécurisé, accès non autorisés
- ⇒ **Biais et discrimination** : collecte partielle ou non représentative

Collecter des données : respecter la vie privée

Risques principaux :

- ⇒ **Atteinte à la vie privée** : données personnelles mal anonymisées (noms, emails, localisation ...)
- ⇒ **Détournement d'usage** : réutilisation non prévue (marketing, surveillance, profilage ...)
- ⇒ **Fuites / piratages** : stockage non sécurisé, accès non autorisés
- ⇒ **Biais et discrimination** : collecte partielle ou non représentative

Cadre légal en Europe (CNIL - RGPD) :

- ⇒ Informer clairement les personnes concernées
- ⇒ Collecter seulement les données **strictement nécessaires** (*minimisation*)
- ⇒ Stocker les données de façon **sécurisée et temporaire**

Règle d'or : toute donnée permettant d'identifier une personne = protégée par la loi.

PARTIE III

Les listes en python

Les constructrices

Les listes en Python

- ⇒ **Définition** : structure **ordonnée**, **modifiable** et **indexée** à partir de 0.
- ⇒ **Création** : `ma_liste = [val1, val2, val3]`
- ⇒ **Accès à un élément** : `ma_liste[0]` (1^{er} élément)
- ⇒ **Fonctions intégrées utiles** :
 - `len()` (taille), `sum()`, `max()`, `min()`
 - `append()` (ajouter à la fin), `insert()` (insérer à une position), `pop()` (retirer un élément)

Exemple pratique : dépenses = [23.5, 12, 18.9, 30, 15]

- ⇒ `sum(depenses)` ⇒ total hebdo
- ⇒ `sum(depenses)/len(depenses)` ⇒ moyenne/jour
- ⇒ `max(depenses)` ⇒ jour le plus cher

Correction - Exercice flash (listes)

Énoncé : Créez une liste de 7 dépenses quotidiennes et affichez seulement celles du week-end (les 2 derniers éléments).

```
# Création d'une liste de 7 dépenses (1 par jour)
```

```
depenses = [23.5, 12.0, 18.9, 30.0, 15.0, 28.5, 22.0]
```

```
# Afficher seulement les 2 derniers éléments (index négatifs)
```

```
weekend = depenses[-2:]
```

```
print("Dépenses du week-end :", weekend)
```

À retenir : `ma_liste[-2:]` retourne une **sous-liste** contenant les deux derniers éléments.

Mutabilité & copies : bonnes pratiques

```
L1 = [1, 2, 3]
```

```
# Copier pour éviter les effets de bord
```

```
L2 = L1.copy()
```

```
L2.append(4)
```

```
print(L1) # [1, 2, 3]
```

```
print(L2) # [1, 2, 3, 4]
```

```
# Utiliser un tuple si aucune modification n'est attendue
```

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

Mutabilité & copies : bonnes pratiques

```
L1 = [1, 2, 3]
```

```
# Copier pour éviter les effets de bord
```

```
L2 = L1.copy()
```

```
L2.append(4)
```

```
print(L1) # [1, 2, 3]
```

```
print(L2) # [1, 2, 3, 4]
```

```
# Utiliser un tuple si aucune modification n'est attendue
```

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

Listes vs Tuples :

⇒ **Listes** : ordonnées, **mutables** (modifiables après création)

⇒ **Tuples** : ordonnés, **immuables** (figés une fois créés)

Utiliser `.copy()` ou `list(L1)` pour dupliquer une liste.

Listes vs Tuples : comparaison rapide

	Listes (list)	Tuples (tuple)
Définition	Séquences ordonnées et modifiables	Séquences ordonnées et fixes
Syntaxe	[1, 2, 3]	(1, 2, 3)
Mutabilité	<i>Mutable</i> : on peut ajouter/supprimer/modifier	<i>Immutable</i> : aucune modification possible
Utilisations typiques	Listes de valeurs évolutives (notes, achats, logs...)	Coordonnées, clés de dictionnaire, données fixes
Méthodes disponibles	<code>append()</code> , <code>insert()</code> , <code>pop()</code> , ...	Très limitées (<code>count()</code> , <code>index()</code>)
Performance	Légèrement plus lent et plus lourd	Plus rapide et léger en mémoire

Règle pratique : liste quand les données peuvent changer, un tuple quand elles doivent rester constantes.

Correction - Checkpoint Listes

```
jours = ["lun", "mar", "mer", "jeu", "ven", "sam", "dim"]  
depenses = [12, 9, 0, 6, 13, 18]
```

```
# 1) Ajouter la dépense du dimanche  
depenses.append(7)
```

```
# 2) Afficher la moyenne des 7 jours  
moyenne = sum(depenses) / len(depenses)  
print("Dépense moyenne :", round(moyenne, 2))
```

```
# 3) Afficher les 2 jours les plus chers (valeurs)  
top2 = sorted(depenses, reverse=True)[:2]  
print("Top 2 dépenses :", top2)
```

```
# 4) Créer la liste "we" = week-end  
we = depenses[-2:]  
print("Dépenses week-end :", we)
```

Erreurs fréquentes - Listes

⇒ **append vs extend**

- `L.append([4,5])` ⇒ ajoute *une sous-liste* à la fin → `[1,2,[4,5]]`
- `L.extend([4,5])` ⇒ ajoute *chaque élément* → `[1,2,4,5]`

Erreurs fréquentes - Listes

⇒ **append vs extend**

- `L.append([4,5])` ⇒ ajoute *une sous-liste* à la fin → `[1,2,[4,5]]`
- `L.extend([4,5])` ⇒ ajoute *chaque élément* → `[1,2,4,5]`

⇒ **Bornes de slicing**

- `L[1:4]` prend les éléments d'indices 1,2,3 (la borne 4 est exclue)
- Toujours vérifier les indices de début/fin pour éviter d'en “ perdre ” un

Erreurs fréquentes - Listes

⇒ append vs extend

- `L.append([4,5])` ⇒ ajoute *une sous-liste* à la fin → `[1,2,[4,5]]`
- `L.extend([4,5])` ⇒ ajoute *chaque élément* → `[1,2,4,5]`

⇒ Bornes de slicing

- `L[1:4]` prend les éléments d'indices 1,2,3 (la borne 4 est exclue)
- Toujours vérifier les indices de début/fin pour éviter d'en “ perdre ” un

⇒ Alias involontaires

- `L2 = L1` lie le *même objet en mémoire*
- Toute modification de L2 modifie L1 → utiliser `L1.copy()`

Erreurs fréquentes - Listes

⇒ **append vs extend**

- `L.append([4,5])` ⇒ ajoute *une sous-liste* à la fin → `[1,2,[4,5]]`
- `L.extend([4,5])` ⇒ ajoute *chaque élément* → `[1,2,4,5]`

⇒ **Bornes de slicing**

- `L[1:4]` prend les éléments d'indices 1,2,3 (la borne 4 est exclue)
- Toujours vérifier les indices de début/fin pour éviter d'en “ perdre ” un

⇒ **Alias involontaires**

- `L2 = L1` lie le *même objet en mémoire*
- Toute modification de `L2` modifie `L1` → utiliser `L1.copy()`

⇒ **Tri en place vs copie triée**

- `L.sort()` trie *en place* et retourne **None**
- `sorted(L)` retourne *une nouvelle liste triée* sans modifier l'originale

Erreurs fréquentes - Listes

⇒ append vs extend

- `L.append([4,5])` ⇒ ajoute *une sous-liste* à la fin → `[1,2,[4,5]]`
- `L.extend([4,5])` ⇒ ajoute *chaque élément* → `[1,2,4,5]`

⇒ Bornes de slicing

- `L[1:4]` prend les éléments d'indices 1,2,3 (la borne 4 est exclue)
- Toujours vérifier les indices de début/fin pour éviter d'en “perdre” un

⇒ Alias involontaires

- `L2 = L1` lie le *même objet en mémoire*
- Toute modification de L2 modifie L1 → utiliser `L1.copy()`

⇒ Tri en place vs copie triée

- `L.sort()` trie *en place* et retourne `None`
- `sorted(L)` retourne *une nouvelle liste triée* sans modifier l'originale

⇒ Dépassement d'index

- `L[10]` lève un `IndexError` si `len(L)<11`
- Solutions : tester `len(L)`, ou utiliser `L[-1]` pour accéder au dernier élément

Les dictionnaires en Python

- ⇒ **Structure non ordonnée (mais indexable par clé) de paires clé**
→ valeur
- ⇒ **Clés uniques et immuables** (chaînes, entiers, tuples...)
- ⇒ **Syntaxe :**
 - `prix = {"pain": 1.2, "lait": 0.9}`

Les dictionnaires en Python

- ⇒ **Structure non ordonnée (mais indexable par clé) de paires clé**
→ **valeur**
- ⇒ **Clés uniques et immuables (chaînes, entiers, tuples...)**
- ⇒ **Syntaxe :**
 - `prix = {"pain": 1.2, "lait": 0.9}`
- ⇒ **Opérations de base :**
 - Accès : `prix["pain"] ⇒ 1.2`
 - Ajout / modification : `prix["pain"] = 1.3`
 - Suppression : `del prix["lait"]`
 - Tester l'existence : `"pain" in prix ⇒ True`

⇒ Méthodes utiles :

- `prix.keys()` ⇒ toutes les clés
- `prix.values()` ⇒ toutes les valeurs
- `prix.items()` ⇒ couples (clé, valeur)

⇒ Méthodes utiles :

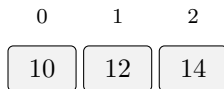
- `prix.keys()` ⇒ toutes les clés
- `prix.values()` ⇒ toutes les valeurs
- `prix.items()` ⇒ couples (clé, valeur)

Exemple réel :

`panier = {"pain": (2,1.2), "lait": (1,0.9)}` ⇒ quantité
et prix de chaque article → calculer total.

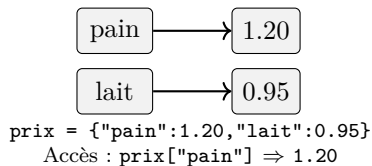
Visuel : liste vs dictionnaire

Liste (index → valeur)



`L = [10,12,14]`
Accès : `L[1] ⇒ 12`

Dictionnaire (clé → valeur)



`prix = {"pain":1.20,"lait":0.95}`
Accès : `prix["pain"] ⇒ 1.20`

Comparer : listes vs dictionnaires

Listes (séquences)

Ordonnées, indexées par position

Ex : [10,12,14]

Accès : L[0]

Outils : `append`, `sort`, `slicing`

Bon pour des séries/collections

Dictionnaires (associatif)

Non ordonnés (en pratique insertion-ordered), accès par *clé*

Ex : "Alice":14, "Bob":10

Accès : D["Alice"]

Outils : `keys`, `values`, `items`, `get`

Bon pour des enregistrements nommés

Checkpoint - Dictionnaires I

Solution possible :

```
prix = {"pain": 1.20, "lait": 0.95, "pates": 1.80}
maj = {"pain": 1.25, "lait": 0.99, "beurre": 2.20}
```

```
# 1) Mettre à jour "prix" avec "maj"
```

```
prix.update(maj)
```

```
# -> ajoute beurre, et remplace pain/lait
```

```
# 2) Récupérer le prix du beurre sans erreur
```

```
p_beurre = prix.get("beurre", 0)
```

```
print("Prix du beurre :", p_beurre)
```

```
# 3) Lister les produits triés par prix croissant
```

```
tri = sorted(prix.items(), key=lambda kv: kv[1])
```

```
print(tri)
```

Sortie attendue :

Checkpoint - Dictionnaires II

⇒ Prix du beurre : 2.2

⇒ [('lait', 0.99), ('pain', 1.25), ('pates', 1.8), ('beurre', 2.2)]

Erreurs fréquentes - Dictionnaires

⇒ Clé absente

- `d["x"]` ⇒ `KeyError` si "x" n'existe pas.
- Utiliser `d.get("x", 0)` pour un *valeur par défaut* (ici 0).

⇒ Mutation pendant itération

- Modifier un dictionnaire pendant `for k,v in d.items()` peut causer des incohérences.
- Itérer sur une copie : `for k in list(d.keys()): ...`

⇒ Clé non hachable

- Les clés doivent être *immuables* (str, int, tuple...).
- `d[[1,2]] = "x"` ⇒ `TypeError`.
- `d[(1,2)] = "x"`

⇒ Shadowing du nom dict

- `dict = {"a":1}` rend ensuite `dict()` inutilisable.
- utiliser un nom neutre (`d`, `mon_dico`...)

Erreurs fréquentes (et correctifs)

⇒ Mélanger int et str

`42 + "3" ⇒ TypeError` (*on additionne un entier et une chaîne*)

Correctifs : `42 + int("3") ⇒ 45` ou `str(42) + "3" ⇒ "423"`

⇒ Index hors borne (liste)

`L = [10,12,14] L[3] ⇒ IndexError` (*indices valides : 0..2*)

Bon réflexe : vérifier `len(L)` ou utiliser `-1` pour le dernier élément : `L[-1] ⇒ 14`

⇒ Clé absente (dictionnaire)

`prix = {"pain":1.2} prix["lait"] ⇒ KeyError`

Correctifs : `"lait" in prix` ou `prix.get("lait", 0) ⇒ 0`

Pont : de listes/dicos à un tableau (pandas)

```
import pandas as pd

df = pd.DataFrame({
    "produit": ["pain", "lait", "pates"],
    "p_base": [1.20, 0.95, 1.80],
    "p_cour": [1.25, 0.99, 1.95],
    "q": [300, 250, 200]
})

df["poste_base"] = df["p_base"] * df["q"]
df["poste_cour"] = df["p_cour"] * df["q"]
indice = df["poste_cour"].sum() / df["poste_base"].sum()
print("Indice (pandas) =", indice)
```

Lien Cours 1 : même résultat qu'avec nos dicos + **moins de code** pour l'agrégation.

set et Counter pour dédupliquer/compter

```
from collections import Counter
produits = ["pain","lait","pain","pates","lait","pain"]
uniques = set(produits) # {'pain','lait','pates'}
freq = Counter(produits) # {'pain':3, 'lait':2, 'pates':1}
top = freq.most_common(1)[0] # ('pain', 3)
```

PARTIE IV

Les fonctions

Travailler avec de bons matériaux

Tests logiques - rappels essentiels

- ⇒ **Opérateurs de comparaison** : `==`, `!=`, `<`, `<=`, `>`, `>=`
- ⇒ **Opérateurs logiques** : `and`, `or`, `not` ⇒ permettent de combiner plusieurs conditions
- ⇒ **Valeurs “truthy/falsy”** :
 - Évaluées comme **False** : `0`, `0.0`, `""`, `[]`, `{}`, `None`, `False`
 - Tout le reste est considéré **True**
- ⇒ **Syntaxe utile** : comparaisons chaînées `a < b <= c`

Tests logiques - rappels essentiels

- ⇒ **Opérateurs de comparaison** : `==`, `!=`, `<`, `<=`, `>`, `>=`
- ⇒ **Opérateurs logiques** : `and`, `or`, `not` ⇒ permettent de combiner plusieurs conditions
- ⇒ **Valeurs “truthy/falsy”** :
 - Évaluées comme `False` : `0`, `0.0`, `""`, `[]`, `{}`, `None`, `False`
 - Tout le reste est considéré `True`
- ⇒ **Syntaxe utile** : comparaisons chaînées `a < b <= c`

```
prix, budget = 12.5, 30
```

```
# Condition simple
```

```
if prix <= budget:  
    print("Achat possible")
```

```
# Combinaison avec plusieurs conditions
```

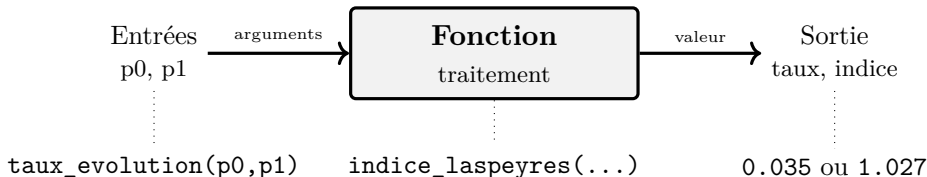
```
if 0 < prix <= budget and prix is not None:  
    print("Condition remplie")
```

Boucles : for, range, enumerate

```
depenses = [12, 9, 0, 6, 13, 18, 7]
total = 0
for i, x in enumerate(depenses): # i = index, x = valeur
    total += x
    if x == 0:
        print("Jour", i, ": dépense manquante")
print("Total =", total)
# Compréhension équivalente :
total2 = sum(x for x in depenses)
```

Idee : `enumerate` pour lier **position** et **valeur** ; compréhension pour les cas “ transformer/filtrer ”.

Fonction : une boîte noire (inputs → sortie)



Idée clé

Une fonction prend des **arguments**, effectue un **traitement**, et renvoie une **valeur**.

Les fonctions en Python

- ⇒ Une **fonction** est un bloc de code nommé qui permet de :
- éviter les répétitions (**réutilisation**)
 - rendre le code plus **lisible et structuré**

Les fonctions en Python

- ⇒ Une **fonction** est un bloc de code nommé qui permet de :
 - éviter les répétitions (**réutilisation**)
 - rendre le code plus **lisible et structuré**
- ⇒ **Déclaration** : `def nom_fonction(param1, param2,...):`
- ⇒ **Utilisation (appel)** : `nom_fonction(arg1, arg2,...)`
- ⇒ **Retourner une valeur** avec `return` (sinon la fonction renvoie `None`)

Les fonctions en Python

- ⇒ Une **fonction** est un bloc de code nommé qui permet de :
 - éviter les répétitions (**réutilisation**)
 - rendre le code plus **lisible et structuré**
- ⇒ **Déclaration** : `def nom_fonction(param1, param2,...):`
- ⇒ **Utilisation (appel)** : `nom_fonction(arg1, arg2,...)`
- ⇒ **Retourner une valeur** avec `return` (sinon la fonction renvoie `None`)

Exemple :

```
def taux_evolution(p0, p1):  
    """Calcule le taux d'évolution en %"""  
    return (p1 - p0) / p0 * 100  
  
print(taux_evolution(100, 120)) # 20.0
```

Les fonctions en Python

- ⇒ Une **fonction** est un bloc de code nommé qui permet de :
 - éviter les répétitions (**réutilisation**)
 - rendre le code plus **lisible et structuré**
- ⇒ **Déclaration** : `def nom_fonction(param1, param2,...):`
- ⇒ **Utilisation (appel)** : `nom_fonction(arg1, arg2,...)`
- ⇒ **Retourner une valeur** avec `return` (sinon la fonction renvoie `None`)

Exemple :

```
def taux_evolution(p0, p1):  
    """Calcule le taux d'évolution en %"""  
    return (p1 - p0) / p0 * 100  
  
print(taux_evolution(100, 120)) # 20.0
```

Valider & expliquer les erreurs (try/except)

```
def taux_evolution(p0, p1):  
    if p0 == 0:  
        raise ValueError("p0 ne peut pas être 0 (division par zéro).")  
    return (p1 - p0) / p0  
  
def parse_prix(s):  
    try:  
        return float(s.replace(",", "."))  
    except ValueError:  
        print("Prix invalide :", s); return None
```

Lien Cours 1 : lire le **traceback** ⇒ ici on **intercepte** et on **explique** proprement.

Rendre les fonctions *auto-explicites*

```
def indice_laspeyres(p0: dict[str,float], p1: dict[str,float], q: dict[str,float])  
    """  
    Calcule l'indice de Laspeyres (p1/p0 pondéré par q).  
    >>> indice_laspeyres({"pain":1.2}, {"pain":1.26}, {"pain":300})  
    1.05  
    """  
    num = sum(p1[k]*q[k] for k in q)  
    den = sum(p0[k]*q[k] for k in q)  
    assert den > 0, "Somme des postes de base nulle."  
    return num/den
```

doctest = vérif. rapide en doc ; **assert** = contrat minimal. Ajoute un test **pytest** pour le devoir.

Checkpoint - Fonctions

Objectif : écrire une fonction robuste & démontrer les arguments nommés.

```
# 1) Ecrire une fonction "taux_evol(p0, p1, pct=True)"  
# - si pct=True, retourner en pourcentage arrondi à 1 déc.  
# - sinon, retourner le ratio (p1/p0 - 1) en décimal  
# 2) Gérer p0=0 par ValueError  
# 3) Appeler avec arguments *nommés* (ex: p1=105, p0=100)  
  
# 4) Ecrire une fonction "stats(L)" qui retourne (moy, maxi, mini)  
# Puis: moy, maxi, mini = stats([12, 14, 10])  
.
```

Checkpoint - Fonctions (corrigé) I

Solution attendue :

```
def taux_evol(p0: float, p1: float, pct: bool = True) -> float:
    """
    Taux d'évolution entre p0 et p1.
    - Si pct=True : renvoie un pourcentage arrondi à 1 déc. (ex: 5.3)
    - Sinon : renvoie le ratio en décimal (ex: 0.053)
    """
    if p0 == 0:
        raise ValueError("p0 ne peut pas être 0 (division par zéro).")
    ratio = p1 / p0 - 1
    return round(ratio * 100, 1) if pct else ratio

# Appels avec arguments nommés :
print(taux_evol(p1=105, p0=100)) # 5.0
print(taux_evol(p0=100, p1=105, pct=False)) # 0.05
```

Checkpoint - Fonctions (corrigé) II

```
def stats(L: list[float]) -> tuple[float, float, float]:  
    """  
    Retourne (moyenne, maximum, minimum) pour une liste non vide.  
    """  
    if not L:  
        raise ValueError("Liste vide : impossible de calculer des st  
    return (sum(L) / len(L), max(L), min(L))  
  
moy, maxi, mini = stats([12, 14, 10])  
print(moy, maxi, mini) # 12.0 14 10
```

Points clés : **raise** pour `p0=0`, argument par défaut **pct=True**, **arguments nommés** à l'appel, **type hints**, contrôle liste vide.

Erreurs fréquentes - Fonctions (détaillé)

⇒ **Oublier return** ⇒ la fonction renvoie None.

```
def f(x): x + 1 # -> None
def f(x): return x + 1
```

⇒ **Ordre des paramètres** (positionnels vs nommés) : utilisez des arguments nommés quand l'ordre prête à confusion.

```
def resize(w, h, keep_ratio=True): ...
resize(h=600, w=800) # explicite
```

⇒ **Valeur mutable en défaut** : *jamais* = [] ou {}. Utiliser None + initialisation interne.

```
def add(x, acc=None):
    if acc is None: acc = []
    acc.append(x); return acc
```

⇒ **Shadowing (masquer un builtin)** : éviter `sum = 0`, `list = ...`, `dict = ....`

```
total = 0 # ok, ne masque pas sum()
```

⇒ **Exceptions trop larges** : `except` : masque les bugs. Ciblez le type.

```
try:
    return float(s)
except ValueError:
    return None
```

⇒ **Chemins de retour incohérents** : toujours retourner le même type.

```
def parse_int(s):
    try: return int(s)
    except ValueError: return None # pas "0" ni False
```

⇒ **Effets de bord non voulus** : ne modifiez pas les arguments mutables si la fonction est censée être *pure*.

```
def tri(L):
    return sorted(L) # préfère créer une copie au lieu de L.sort
```

⇒ **Absence de contrat** : ajoutez **docstring**, **préconditions** (tests) et **type hints**.

```
def ratio(a: float, b: float) -> float:
```

Fonctions : aller plus loin

- ⇒ **Arguments par défaut** : permettent d'éviter de répéter toujours la même valeur.
`def taux(p0, p1=100): return (p1 - p0)/p0`
- ⇒ **Variables locales** : une variable définie dans une fonction n'existe pas en dehors.
- ⇒ **Docstring** : description incluse dans la fonction (bonne pratique).
- ⇒ **Retour multiple** : une fonction peut renvoyer plusieurs valeurs sous forme de tuple.

Fonctions : aller plus loin

- ⇒ **Arguments par défaut** : permettent d'éviter de répéter toujours la même valeur.
`def taux(p0, p1=100): return (p1 - p0)/p0`
- ⇒ **Variables locales** : une variable définie dans une fonction n'existe pas en dehors.
- ⇒ **Docstring** : description incluse dans la fonction (bonne pratique).
- ⇒ **Retour multiple** : une fonction peut renvoyer plusieurs valeurs sous forme de tuple.

Exemple :

```
def stats(L): return sum(L)/len(L), max(L), min(L)
moy, maxi, mini = stats([12,14,10])
```

À retenir

- ⇒ **Types de base** : int, float, str, bool ⇒ les briques pour représenter l'information.
- ⇒ **Listes** : séquences ordonnées et modifiables ⇒ parfait pour des séries (prix, notes).
- ⇒ **Dictionnaires** : paires clé → valeur ⇒ accès direct par nom (produit, id, date).
- ⇒ **Fonctions** : réutiliser, nommer une idée, isoler un calcul ⇒ lisibilité et tests faciles.
- ⇒ **Unité d'analyse** : ligne = unité, colonnes = variables ⇒ structure de base cohérente.
- ⇒ **Réflexe éco** : relier chaque concept à un cas concret (panier, indice, classement de notes).

PARTIE V

Efficacité

Bien travailler, c'est de ne pas se fatiguer

Coder efficacement : quelques principes clés

⇒ Comprendre le fonctionnement d'un ordinateur :

- L'ordinateur ne "*comprend*" pas : il suit des instructions exactes.
- Il exécute uniquement ce qu'on lui demande, même si cela n'a aucun sens logique.
- Un code ambigu ou mal structuré produira des erreurs ... ou de mauvais résultats.

Coder efficacement : quelques principes clés

⇒ Comprendre le fonctionnement d'un ordinateur :

- L'ordinateur ne "*comprend*" pas : il suit des instructions exactes.
- Il exécute uniquement ce qu'on lui demande, même si cela n'a aucun sens logique.
- Un code ambigu ou mal structuré produira des erreurs ... ou de mauvais résultats.

⇒ Bonnes pratiques en programmation :

- Écrire un code clair, lisible et **réutilisable**.
- Ajouter des commentaires explicatifs **à chaque étape importante**.
- Limiter le nombre de packages : privilégier ceux bien documentés et maintenus.
- Automatiser les tâches répétitives dès que possible.

Règle d'or : Codez pour vous, mais surtout pour la personne qui relira votre code dans 6 mois (même si cette personne, c'est vous).

Sauvegarder les différentes versions de son code

Pourquoi ?

- ⇒ Éviter de perdre son travail en cas d'erreur ou de bug.
- ⇒ Pouvoir revenir à une version qui fonctionnait.
- ⇒ Garder une trace de l'évolution du projet (utile pour les comptes rendus et mémoires).

Sauvegarder les différentes versions de son code

Pourquoi ?

- ⇒ Éviter de perdre son travail en cas d'erreur ou de bug.
- ⇒ Pouvoir revenir à une version qui fonctionnait.
- ⇒ Garder une trace de l'évolution du projet (utile pour les comptes rendus et mémoires).

Comment faire simplement ?

- ⇒ Créer un dossier `/versions` dans votre projet.
- ⇒ Sauvegarder régulièrement une copie avec un nom explicite : `script_analyse_v1.py`, `v2_ajout_graphiques.py`, etc.
- ⇒ Ajouter un petit fichier `journal.txt` avec :
 - la date,
 - ce qui a été modifié,
 - les éventuels bugs connus.

Astuce : Sauvegarder souvent, mais pas tout : seulement les étapes importantes ou stables.

Lire un message d'erreur (Traceback)

Traceback (most recent call last):

```
File "scripts/analyse.py", line 4, in <module>  
    import pandas as pd
```

ModuleNotFoundError: No module named 'pandas'

- ⇒ **Type d'erreur** : `ModuleNotFoundError` ⇒ package manquant.
- ⇒ **Ligne** : où ça casse (`analyse.py`, ligne 4).
- ⇒ **Solution** : activer l'environnement et `pip install pandas`.

Méthode : lire *du bas vers le haut*, identifier *type + ligne*, corriger, relancer.

Débogage sans paniquer (5 pas)

- ❶ Lire le **type d'erreur** et la **ligne** indiquée
- ❷ **Reproduire** l'erreur avec un *exemple minimal*
- ❸ Inspecter les **valeurs** (afficher variables intermédiaires)
- ❹ Vérifier **chemins/encodage/types** (les 3 suspects courants)
- ❺ Modifier une chose à la fois & **relancer**

Mantra : "*Qu'attendais-je ? Qu'est-ce qui s'est réellement passé ?
Où divergent-ils ?*"

Exercice transversal - Fil rouge : Indice de prix

Données (prix base & courant, quantités de base) :

```
p0 = {"pain": 1.20, "lait": 0.95, "pates": 1.80}
```

```
p1 = {"pain": 1.25, "lait": 0.99, "pates": 1.95}
```

```
q = {"pain": 300, "lait": 250, "pates": 200}
```

Tâches

- ❶ Écrire `indice_laspeyres(p0, p1, q)` → renvoie un float.
- ❷ Afficher l'indice **en %** (arrondi à 1 décimale).
- ❸ Bonus : gérer un produit manquant dans `p1` via `get(..., p0[k])`.
- ❹ Bonus 2 : écrire `variation_par_poste(p0,p1,q)` → top 2 postes qui contribuent le plus.

Indice de prix - Soutien (architecture)

Squelette suggéré

```
def indice_laspeyres(p0: dict[str,float], p1: dict[str,float], q: dict[str,float])> float:
    num = 0.0
    den = 0.0
    for k in q:
        p0k = p0[k]
        p1k = p1.get(k, p0k) # bonus: fallback si manquant
        num += p1k * q[k]
        den += p0k * q[k]
    assert den > 0, "Somme des postes de base nulle."
    return num / den
```

Affichage : `print(f"Indice = {(indice-1)*100:.1f}%")`

