Nettoyer et structurer le code avec `Enum`, `dataclass` et `__str__`

refactorisation séance 1 sur 5

Table des matières

1. Contexte de la séance 1....1
2. Partie 1 — Rappels théoriques (on ne touche pas au code)....1
   a. Les méthodes spéciales (dunder methods)....1
   b. Les Enumération (Enum)....2
   c. Les dataclass....3
3. Partie 2 — Travail demandé....3
   a. Consigne 1 — Nettoyer TypeOperation....3
   b. Consigne 2 — Exploiter pleinement @dataclass pour Operation....4
   c. Consigne 3 — Remplacer info() par __str__....4
   d. Consigne 4 — Nettoyer le code de test....4
   e. Consigne 5 — Vérifier la non-régression....4
4. Conseil pratique pour la refactorisation....4
5. Ce que vous devez rendre....4
6. Questions de réflexion (à préparer pour la séance 2)....5

1. Contexte de la séance 1

Vous disposez du dossier PJT_banque. Il fonctionne, mais il contient plusieurs maladresses : une valeur parasite dans l'énumération, une `dataclass` définie mais jamais réellement exploitée, des `print` de debug disséminés, et une méthode `info()` là où Python attend une méthode spéciale.

🎯 **Objectif de la séance :** améliorer la qualité, la robustesse et la lisibilité du programme **sans ajouter de fonctionnalité**. À la fin de la séance, vous devez avoir une version plus propre du fichier, qui servira de base aux évolutions suivantes.

2. Partie 1 — Rappels théoriques (on ne touche pas au code)

a. Les méthodes spéciales (dunder methods)

Vous connaissez déjà les méthodes spéciales (ou *dunder methods*, pour *double underscore*), comme `__init__`, `__str__`, `__repr__`, `__eq__`. Pour rappel :

- `__str__` : appelée par `print(objet)`, `str(objet)`, et les f-strings (`f"{objet}"`). Elle doit renvoyer une représentation **lisible pour un humain**.
- `__repr__` : appelée par `repr(objet)`, dans la console interactive, et dans les conteneurs (`[objet1, objet2]`). Elle doit renvoyer une représentation **non ambiguë**, idéalement telle qu'on pourrait recréer l'objet.

Question à vous poser : dans `compte.py et banque.py`, la méthode `info()` de `Banque` et `Compte` joue le rôle de `__str__`. Quel est l'avantage de la remplacer par `__str__` ?

b. Les `Enumération (Enum)`

Vous avez vu en cours la notion de **classe** et d'**héritage**. Un `Enum` s'explique très bien avec ces notions. Observez la définition :

```
class TypeOperation(Enum):
    DEPOT = "DEPOT"
    RETRAIT = "RETRAIT"
    VIREMENT = "VIREMENT"
```

Lecture avec les notions de classe et d'héritage :

- `TypeOperation` est une **classe**.
- `(Enum)` signifie que `TypeOperation` **hérite** de la classe `Enum` fournie par Python (module `enum`).
- En héritant de `Enum`, `TypeOperation` bénéficie d'un **comportement spécial** : les attributs que vous déclarez (`DEPOT`, `RETRAIT`, `VIREMENT`) ne deviennent pas de simples attributs de classe, mais des **membres d'énumération**, c'est-à-dire des **instances uniques** de `TypeOperation`.
- Autrement dit, `TypeOperation.DEPOT` n'est pas la chaîne `"DEPOT"` : c'est un **objet** qui est une instance de `TypeOperation`.

Vérification en console :

```
>>> type(TypeOperation.DEPOT)
<enum 'TypeOperation'>
>>> isinstance(TypeOperation.DEPOT, TypeOperation)
True
>>> isinstance(TypeOperation.DEPOT, Enum)
True
```

On voit bien que `TypeOperation.DEPOT` est un **objet** dont le type est `TypeOperation`, et que ce type **hérite** de `Enum`. C'est l'héritage qui donne à `TypeOperation` tout son comportement (unicité des membres, comparaison par identité, accès à `.value`, `.name`, etc.).

Propriétés importantes :

- Les membres sont **uniques** et **non modifiables** après définition.
- On y accède par `TypeOperation.DEPOT`, jamais par la chaîne `"DEPOT"` directement.
- `TypeOperation.DEPOT is TypeOperation.DEPOT` → `True` (comparaison par identité).
- `TypeOperation.DEPOT.value` → `"DEPOT"` (la valeur associée).
- `TypeOperation.DEPOT.name` → `"DEPOT"` (le nom du membre).
- `TypeOperation("DEPOT")` → `TypeOperation.DEPOT` (retrouver un membre depuis sa valeur).
- Un `Enum` ne peut pas être instancié en dehors des membres définis : `TypeOperation("AUTRE")` lève une `ValueError`.

Pourquoi les Enumérations sont-elles utiles ? Cela empêche les fautes de frappe (comme `"STOC"` au lieu de `"STOCK"`) et rend le code auto-documenté.

Question à vous poser : dans `votre programme`, la classe `TypeOperation` contient un membre `STOCK = "STOC"`. Que remarquez-vous ?

c. Les `dataclass`

Le décorateur `@dataclass` (module `dataclasses`) génère automatiquement pour vous :

- `__init__` à partir des champs annotés,
- `__repr__` lisible,
- `__eq__` (comparaison champ par champ).

Exemple :

```
from dataclasses import dataclass

@dataclass
class Operation:
    id: int
    date: date
    type: TypeOperation
    montant: float
    compte_source: int | None = None
    compte_destination: int | None = None
```

Grâce à `@dataclass`, vous pouvez écrire :

```
op = Operation(1, date.today(), TypeOperation.DEPOT, 200.00,
compte_destination=1)
print(op)   # affichage lisible automatique
```

sans définir `__init__` ni `__repr__` à la main.

Points d'attention :

- Les champs **sans valeur par défaut** doivent précéder ceux qui en ont une.
- `int | None = None` signifie : le champ peut être un entier ou `None`, et vaut `None` par défaut.
- `@dataclass` est idéale pour des **structures de données** simples, sans logique métier lourde.

Question à vous poser : dans votre programme, `Operation` est bien une `@dataclass`, mais est-elle réellement exploitée comme telle? Comment le savez-vous ?

3. Partie 2 — Travail demandé

Copier les fichier de votre programme dans un dossier PJT_banque ; puis dans un nouveau dossier PJT_banque1 . Vous devez modifier les fichiers contenus dans PJT_banque1 pour produire une version qui respecte les consignes suivantes. **Aucune fonctionnalité nouvelle ne doit être ajoutée** : on ne fait que nettoyer et structurer.

a. Consigne 1 — Nettoyer `TypeOperation`

- Supprimez le membre `STOCK = "STOC"`.
- Vérifiez que l'énumération ne contient plus que les trois types réellement utilisés.
- **Bonus :** en console, vérifiez avec `isinstance` et `type` que `TypeOperation.DEPOT` est bien une instance de `TypeOperation` et que `TypeOperation` hérite bien de `Enum`.

b. Consigne 2 — Exploiter pleinement `@dataclass` pour `Operation`

- Conservez la `@dataclass` telle quelle (champs, annotations, valeurs par défaut).
- **Supprimez** les `print("opération deposer:", repr(operation))` et `print("opération retrait:", repr(operation))` dans `Compte.deposer` et `Compte.retirer`.
- À la place, laissez le soin à l'appelant d'afficher l'opération s'il le souhaite, via `print(operation)` (qui utilisera le `__repr__` généré).

c. Consigne 3 — Remplacer `info()` par `__str__`

- Dans `Compte` : remplacez la méthode `info()` par `__str__`.
- Dans `Banque` : remplacez la méthode `info()` par `__str__`.
- Adaptez le code de test en conséquence : `print(compte_a)` au lieu de `print(compte_a.info())`.

Format attendu pour `Compte.__str__` :

```
def __str__(self):
    return f"{self.banque.nom} - Compte {self.id} - Solde : {self.solde:.2f} €"
```

Format attendu pour `Banque.__str__` :

```
def __str__(self):
    return f"Banque {self.id} - {self.nom}"
```

d. Consigne 4 — Nettoyer le code de test

- Décommentez progressivement les blocs de test au fur et à mesure que vous validez.
- Supprimez les `print(repr(...))` de debug qui ne servent qu'à inspecter le comportement interne.
- Conservez uniquement les affichages qui montrent le **résultat métier** (soldes après opérations).

e. Consigne 5 — Vérifier la non-régression

Après modification, le programme doit produire **exactement les mêmes résultats métier** qu'avant :

- Dépôt de 1000,00 € sur le compte A → solde 1000,00 €.
- Retrait de 150,00 € → solde 850,00 €.
- Virement de 200,00 € vers le compte B → 650,00 € / 200,00 €.
- Virement de 300,00 € vers le compte C (autre banque) → 350,00 € / 300,00 €.

4. Conseil pratique pour la refactorisation

Travaillez par **petites étapes** : modifiez une chose, exécutez, vérifiez que le résultat est identique. Ne passez à l'étape suivante que lorsque la précédente est validée. C'est exactement la démarche d'un développeur qui **refactorise** du code sans en changer le comportement.

5. Ce que vous devez rendre

1. Les fichiers modifiés dans PJT_banque1.
2. Un court paragraphe (5 à 10 lignes) expliquant, pour chacune des 3 notions (`Enum`, `@dataclass`, `__str__`), **ce que vous avez changé et pourquoi**.

6. Questions de réflexion (à préparer pour la séance 2)

1. Le solde d'un compte est actuellement un `float`. Quels problèmes cela peut-il poser en finance ? (indice : essayez `0.1 + 0.2` dans une console Python)
2. La date d'une opération est actuellement une `date` (jour seulement). Est-ce suffisant pour auditer des opérations bancaires ?
3. La méthode `virer()` ne crée aucune `Operation`. Est-ce cohérent avec le fait que `deposer()` et `retirer()` en créent une ?

Ces questions seront le point de départ de la séance 2.