# Précision des montants et horodatage des opérations

Refactorisation séance 2 / 5

## Table des matières

1. Contexte de la séance....1
2. Partie 1 — Rappels théoriques (à lire avant de coder)....1
   a. Le problème des float en finance....1
   b. Le type Decimal....2
   c. Le module datetime et le type datetime....2
   d. Cohérence du typage....3
3. Partie 2 — Travail demandé....3
   a. Consigne 1 — Remplacer float par Decimal....3
   b. Consigne 2 — Remplacer date par datetime....3
   c. Consigne 3 — Vérifier la non-régression....4
   d. Consigne 4 — Test de précision....4
   e. Consigne 5 — Nettoyer les annotations....4
4. Ce que vous devez rendre....4
5. Questions de réflexion (à préparer pour la séance 3)....5
6. Conseil pratique....5

## 1. Contexte de la séance

À la fin de la séance 1, vous avez produit une version nettoyée du programme : `TypeOperation` ne contient plus que les trois types utiles, `Operation` exploite pleinement `@dataclass`, et `info()` a été remplacée par `__str__`.

Le programme fonctionne, mais il repose encore sur deux choix fragiles que nous allons corriger cette semaine :

1. Les montants sont des `float`. Or, en finance, les `float` introduisent des **erreurs d'arrondi** inacceptables.
2. Les dates sont des `date` (jour seulement). Or, pour auditer des opérations bancaires, il faut savoir **quand précisément** elles ont eu lieu.

**Objectif de la séance :** remplacer `float` par `Decimal` et `date` par `datetime`, **sans changer la logique métier**. Le programme doit continuer à produire les mêmes résultats, mais avec une précision et une traçabilité correctes.

## 2. Partie 1 — Rappels théoriques (à lire avant de coder)

### a. Le problème des `float` en finance

Ouvrez une console Python (cadre en bas à gauche dans Thonny par exemple) et tapez :

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

Ce comportement n'est pas un bug de Python : c'est une conséquence du **standard IEEE 754** utilisé par tous les langages pour représenter les nombres à virgule flottante. Un `float` est une

approximation binaire d'un nombre décimal, et certaines valeurs décimales (comme `0.1`, `0.2`, `0.3`) n'ont pas de représentation binaire exacte ( non décomposable exactement en puissance de 2).
**Conséquence en finance :** si vous additionnez 10 000 fois `0.01` à un solde, vous n'obtiendrez pas exactement `100.00`. Sur des millions d'opérations, l'écart peut devenir significatif. Un solde bancaire doit être **exact au centime**.

## b. Le type `Decimal`

Python fournit le module `decimal` et son type `Decimal`, qui représente les nombres décimaux **exactement**, comme un humain les écrit.

```
from decimal import Decimal

>>> Decimal("0.1") + Decimal("0.2")
Decimal('0.3')
>>> Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
True
```

**Règles d'usage importantes :**

- On construit un `Decimal` **à partir d'une chaîne** : `Decimal("1000.00")`. Si on écrit `Decimal(1000.00)`, on introduit dès le départ l'imprécision du `float`.
- On peut aussi utiliser `Decimal(0)` ou `Decimal(1)` pour des entiers, mais pour tout montant décimal, on passe par une chaîne.
- Les opérations `+`, `-`, `*`, `/` et les comparaisons fonctionnent comme avec les nombres.
- On ne mélange **jamais** `Decimal` et `float` dans une opération : `Decimal("1.0") + 0.5` lève une `TypeError`.

**Question à vous poser :** dans `PJT_banque1` , le solde initial est `0.00` (un `float`). Comment le remplacer par un `Decimal` qui représente exactement zéro ?

## c. Le module `datetime` et le type `datetime`

Vous connaissez déjà `date` du module `datetime` : `date.today()` renvoie le jour courant, sans heure. Le module `datetime` fournit aussi une classe `datetime` (même nom que le module, ce qui prête parfois à confusion) qui représente un **instant précis** : année, mois, jour, heure, minute, seconde, microseconde.

```
from datetime import datetime

>>> datetime.now()
datetime.datetime(2026, 10, 2, 14, 37, 12, 456789)
```

**Différences essentielles :**

| Aspect | `date` | `datetime` |
|---|---|---|
| Représente | Un jour | Un instant |
| Exemple | `2026-10-02` | `2026-10-02 14:37:12.456789` |
| Constructeur courant | `date.today()` | `datetime.now()` |
| Usage typique | Date de naissance, échéance | Horodatage, log, transaction |

**Pourquoi** `datetime` **en finance ?** Parce qu'une opération bancaire doit être **horodatée** : si deux virements ont lieu le même jour, il faut pouvoir les ordonner. Un audit réglementaire exige souvent l'heure et la minute.

**Question à vous poser :** dans `PJT_banque1`, la méthode `Banque.date_actuelle()` renvoie `date.today()`. Que faut-il changer pour obtenir un horodatage complet ?

### d. Cohérence du typage

Changer le type des montants et des dates ne se limite pas à une ligne. Il faut propager le changement **partout** :

- Le champ `solde` de `Compte`.
- Le champ `montant` de `Operation`.
- Les paramètres `montant` des méthodes `crediter`, `debiter`, `deposer`, `retirer`, `virer`.
- Le champ `date` de `Operation`.
- La méthode `date_actuelle()` de `Banque`.
- Le code de test qui appelle ces méthodes.

C'est l'occasion de vérifier que vos **annotations de type** (type hints), placés juste après les paramètres d'une fonction (:int par exemple) sont cohérentes avec la réalité du code.

## 3. Partie 2 — Travail demandé

Copier les fichier de votre programme PJT_banque1 dans un dossier PJT_banque2 . Vous devez modifier les fichiers contenus dans PJT_banque2 pour produire une version qui respecte les consignes suivantes. **La logique métier ne change pas** : seuls les types changent.

### a. Consigne 1 — Remplacer `float` par `Decimal`

- En haut du fichier, ajoutez `from decimal import Decimal`.
- Dans `Compte.__init__` :
  remplacez `self.solde = 0.00` par `self.solde = Decimal("0")`.
- Dans `Operation` : remplacez `montant: float` par `montant: Decimal`.
- Dans toutes les méthodes qui prennent un `montant` en paramètre (`crediter`, `debiter`, `deposer`, `retirer`, `virer`), remplacez l'annotation `float` par `Decimal`.
- Dans le code de test, remplacez tous les appels du type `compte_a.deposer(1000.00)` par `compte_a.deposer(Decimal("1000.00"))`.

**Attention :** ne laissez **aucun** `float` se glisser dans une opération avec un `Decimal`. Vérifiez en particulier le formatage dans `__str__` :
`return f"{self.banque.nom} - Compte {self.id} - Solde : {self.solde:.2f} €"`
Le format `:.2f` fonctionne aussi avec un `Decimal`, donc cette ligne peut rester telle quelle.

### b. Consigne 2 — Remplacer `date` par `datetime`

- En haut du fichier, remplacez `from datetime import date` par `from datetime import datetime`.
- Dans `Operation` : remplacez `date: date` par `date: datetime`.

- Dans `Banque.date_actuelle()` : remplacez `return date.today()` par `return datetime.now()`.
- Dans le code de test, remplacez `date.today()` par `datetime.now()` là où c'est utilisé.

### c. Consigne 3 — Vérifier la non-régression

Après modification, le programme doit produire **exactement les mêmes résultats métier** qu'à la version 1 dans PJT_banque1:

- 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 €.

**Vérification supplémentaire :** affichez une opération avec `print(operation)` et constatez que la date comporte désormais l'heure et les microsecondes.

### d. Consigne 4 — Test de précision

Ajoutez à la fin du fichier un petit bloc de test qui démontre l'intérêt du `Decimal` :

```
# Démonstration : précision du Decimal
compte_test = banque_a.creer_compte()
for _ in range(10):
    compte_test.deposer(Decimal("0.10"))
print(compte_test)   # doit afficher exactement 1.00 €
```

Comparez avec ce que donnerait le même code en `float` (vous pouvez tester à part, dans une console ou avec le programme contenu dans PJT_banque1) :

```
solde = 0.0
for i in range(10):   # boucle pour i de 0 à 9 (10 itérations)
    solde += 0.10
print(solde)   # affiche 0.9999999999999999
```

**Question à vous poser :** pourquoi le résultat avec `Decimal` est-il exact, alors que celui avec `float` ne l'est pas ?

### e. Consigne 5 — Nettoyer les annotations

Profitez-en pour vérifier que **toutes** vos annotations de type sont cohérentes :

- `montant: Decimal` partout,
- `date: datetime` dans `Operation`,
- `-> Decimal` si une méthode renvoie un montant (ce n'est pas le cas ici, mais vérifiez),
- `-> datetime` pour `date_actuelle()`.

## 4. Ce que vous devez rendre

1. Le dossier PJT_banque2 modifié.
2. Un court paragraphe (5 à 10 lignes) expliquant :
   - pourquoi `float` est inadapté aux montants financiers,
   - pourquoi `datetime` est préférable à `date` pour horodater une opération,
   - ce que vous avez dû modifier pour propager ces changements dans tout le code.

## 5. Questions de réflexion (à préparer pour la séance 3)

1. La méthode `virer()` ne crée toujours aucune `Operation`. Un virement devrait pourtant laisser une trace. Comment faire, sachant qu'un virement concerne **deux comptes** ?
2. Que se passe-t-il si `compte_source.debiter(montant)` réussit mais que `compte_destination.crediter(montant)` échoue ? Le solde total de la banque est-il cohérent ?
3. Dans `Operation`, les champs `compte_source` et `compte_destination` sont optionnels. Quelles combinaisons ont du sens selon le type d'opération ?

Ces questions seront le point de départ de la séance 3.

## 6. Conseil pratique

Procédez par **petites étapes** et exécutez le programme après chaque modification :

1. Changez d'abord `solde` en `Decimal`, adaptez `crediter`/`debiter`, testez un dépôt.
2. Changez ensuite `Operation.montant`, adaptez `deposer`/`retirer`, testez.
3. Changez `Operation.date` et `date_actuelle()`, testez.
4. Adaptez le code de test en dernier.

Si une `TypeError` apparaît du type `unsupported operand type(s) for +=: 'Decimal' and 'float'`, c'est que vous avez oublié de convertir un montant : cherchez l'endroit où un `float` se glisse dans une opération avec un `Decimal`.