Encapsulation et attributs privés en Python

Documents/LMG/STS_A_2025-2027/PYTHON_STS2/C_classe_(heritage)/07_private_public_attributes

Table des matières

1. Contexte....1
2. Partie 1 — Mise en évidence du problème....1
a. Reproduire une fraude....1
b. Réflexion....2
3. Partie 2 — Correction du code....2
a. Rendre l'attribut privé....2
b. Que se passe-t-il dans demonstration_fraude.py ?....2
c. Vérifier le programme principal....2
4. Partie 3 — Pour aller plus loin....3
a. Le *name mangling (self.__xxx)*....3
5. Questions de synthèse....3
6. Livrables attendus....3
II. Pour aller encore plus loin : le décorateur @property en Python....4
1. Le problème qu'il résout....4
2. Principe de base....4
3. Ajouter un setter....4
4. Les trois accesseurs possibles....5

1. Contexte

Vous travaillez sur une petite application de gestion de comptes bancaires écrite en Python. Le code source se compose de quatre fichiers : `operation.py`, `compte.py`, `banque.py` et `main.py`.

Une particularité de ce programme est que **l'accès au solde d'un compte est totalement libre** : n'importe quelle partie du programme peut lire ou modifier directement l'attribut `solde` d'un objet `Compte`, sans passer par les méthodes prévues à cet effet.

Votre objectif est de :

1. **Mettre en évidence** ce problème de sécurité à l'aide d'un exemple concret.
2. **Corriger** le code pour empêcher cet accès direct, en utilisant la convention Python des attributs privés.
3. **Vérifier** que le comportement du programme reste correct après la correction.

2. Partie 1 — Mise en évidence du problème

a. Reproduire une fraude

Créez un nouveau fichier `demonstration_fraude.py` qui :

1. Importe les classes `Banque` et `Compte`, ainsi que `Decimal`.
2. Crée une banque et un compte.
3. Dépose `100.00 €` sur ce compte via la méthode `deposer`.

4. Affiche le solde du compte.
5. **Modifie directement** l'attribut `solde` du compte pour y mettre `1_000_000.00 €`, sans passer par `deposer` ni `crediter`.
6. Affiche à nouveau le solde.

Exécutez ce fichier. Que constatez-vous ?

b. Réflexion

Répondez aux questions suivantes :

1. Pourquoi est-il problématique qu'un utilisateur puisse modifier directement `compte.solde` ?
2. Quelles méthodes de la classe `Compte` devraient être les **seules** à pouvoir modifier le solde ?
   En C++ il suffit de placer le mot-clé `private` devant l'attribut d'une classe pour s'assurer que seuls les objets de la classe auront accès à cet attribut. En Python, nous utiliserons une autre syntaxe.

3. Partie 2 — Correction du code

a. Rendre l'attribut privé

En Python, un attribut précédé de **deux underscores** (par exemple `__solde`) devient **privé** : il n'est plus accessible directement depuis l'extérieur de la classe.

Modifiez le fichier `compte.py` pour :

1. Renommer l'attribut `self.solde` en `self.__solde` dans le constructeur.
2. Renommer **toutes** les utilisations de `self.solde` en `self.__solde` dans les méthodes `crediter`, `debiter` et `__str__`.
   Remarque : utilisez de préférence Edit>Find & Replace et validez une à une les occurrences trouvées ; bonne pratique pour n'en oublier aucune tout en s'assurant de ne pas modifier par erreur une variable ou un texte *self.solde_précédent* en *self.__solde_précédent* .

b. Que se passe-t-il dans `demonstration_fraude.py` ?

Relancez votre fichier `demonstration_fraude.py`.

1. Le programme fonctionne-t-il encore ? Quelle erreur obtenez-vous ?
2. Cette erreur est-elle **souhaitable** du point de vue de la sécurité ? Expliquez.

c. Vérifier le programme principal

Relancez `main.py`. Le programme doit fonctionner **exactement comme avant** :

- les dépôts, retraits et virements doivent toujours fonctionner ;
- les soldes affichés doivent être identiques à ceux obtenus avant la modification.

4. Partie 3 — Pour aller plus loin

a. Le *name mangling (self.__xxx)*

Un attribut privé n'est pas **complètement** inaccessible en Python : il est simplement renommé automatiquement en `self._Compte__solde` par l'interpréteur (on parle de *name mangling*). Cela signifie que :

- depuis l'extérieur de la classe, `compte.__solde` lève une `AttributeError` ;
- mais un utilisateur déterminé pourrait tout de même y accéder via `compte._Compte__solde`.

L'objectif du __ n'est donc **pas** d'empêcher totalement l'accès, mais de :

- signaler clairement que l'attribut est **interne** à la classe ;
- éviter les collisions de noms dans l'héritage ;
- décourager fortement toute manipulation directe.

En python, on ne peut pas sécuriser à 100 % l'accès à une variable ou un attribut. En Python, on dit souvent : « We are all consenting adults here. » (« Nous sommes tous des adultes consentants. »)
C'est un principe culturel du langage : le code dit **ce qu'il faut faire**, mais ne **force** pas l'utilisateur à le respecter. Le _ et le ___ sont des **panneaux indicateurs**, pas des **murs**.

À comparer avec :

- **Java / C#** : `private` est vérifié par le compilateur (mais contournable par réflexion).
- **C++** : `private` est vérifié à la compilation (mais contournable par pointeurs).
- **Python** : rien n'est vérifié à l'exécution ; tout repose sur la discipline du développeur.

Même en Java/C++, `private` n'est **pas** une frontière de sécurité contre du code hostile : c'est une frontière **architecturale** entre composants qui se font confiance.

5. Questions de synthèse

1. Quelle est la différence entre un attribut **privé** (`__solde`) et un attribut **public** (`solde`) en Python ?
2. Pourquoi la convention __ n'offre-t-elle pas une sécurité absolue en Python ? Quel est son véritable objectif ?

6. Livrables attendus

Vous déposerez :

- le fichier `compte.py` modifié ;
- le fichier `demonstration_fraude.py` ;
- le fichier `reponses.txt` contenant vos réponses aux questions des Parties 1, 2 et 3.

II. Pour aller encore plus loin : le décorateur `@property` en Python

1. Le problème qu'il résout

En Python, un attribut public est accessible en lecture **et** en écriture :

```
compte.solde          # lecture
compte.solde = 1000   # écriture
```

Si vous voulez contrôler l'accès (par exemple interdire l'écriture, ou valider la valeur avant de l'affecter), il faudrait normalement écrire des méthodes `get_solde()` et `set_solde()`, ce qui casse la syntaxe naturelle `compte.solde`.

`@property` permet de **garder la syntaxe d'un attribut** tout en exécutant du code (getter / setter / deleter) lors de l'accès.

2. Principe de base

`@property` transforme une **méthode** en **attribut calculé** :

```
class Compte:
    def __init__(self):
        self.__solde = Decimal("0")

    @property
    def solde(self) -> Decimal:
        return self.__solde
```

Désormais :

```
compte.solde        # ✅ appelle la méthode solde() → renvoie __solde
compte.solde = 100  # ❌ AttributeError : can't set attribute
```

La méthode `solde` n'est **plus appelée avec des parenthèses** : on l'utilise comme un attribut.

3. Ajouter un setter

Pour autoriser l'écriture tout en la contrôlant, on définit un *setter* avec `@<nom>.setter` :

```
class Compte:
    def __init__(self):
        self.__solde = Decimal("0")

    @property
    def solde(self) -> Decimal:
        return self.__solde

    @solde.setter
    def solde(self, valeur: Decimal):
        if valeur < 0:
            raise ValueError("Le solde ne peut pas être négatif.")
        self.__solde = valeur
```

Maintenant :

```
compte.solde = Decimal("500")   # ✅ passe par le setter
compte.solde = Decimal("-10")   # ❌ ValueError
```

4. Les trois accesseurs possibles

Décorateur	Rôle	Syntaxe déclenchée
`@property`	Getter (lecture)	`x = obj.attr`
`@attr.setter`	Setter (écriture)	`obj.attr = valeur`
`@attr.deleter`	Deleter (suppression)	`del obj.attr`

Exemple complet :

```
class Compte:
    def __init__(self):
        self.__solde = Decimal("0")

    @property
    def solde(self):
        return self.__solde

    @solde.setter
    def solde(self, valeur):
        if valeur < 0:
            raise ValueError("Solde négatif interdit.")
        self.__solde = valeur

    @solde.deleter
    def solde(self):
        raise AttributeError("Impossible de supprimer le solde.")
```