.Résumons les concepts fondamentaux pour aborder un projet orienté objet (en GDP en Python)

1. Programmation orientée objet (POO)

La POO permet de modéliser un problème en décomposant les données et les comportements en entités appelées objets. Chaque objet est une instance d'une classe, qui définit :

- des **attributs** : les variables d'instance qui représentent l'état de l'objet,
- des **méthodes** : les fonctions associées à l'objet, qui modifient son état ou retournent des informations.

Encapsulation

Ce principe consiste à protéger les données internes d'un objet en les rendant inaccessibles directement depuis l'extérieur. On les rend privées (via le préfixe _) et on fournit des accesseurs (getters/setters) ou des propriétés Python pour y accéder de manière contrôlée. Svp utilisé le décorateur @property !

Héritage

Il s'agit de la capacité pour une classe de dériver d'une autre classe afin de réutiliser son code. On parle alors de classe parente (ou de base) et de classe enfant (ou dérivée). Cela favorise la factorisation du code et la création de hiérarchies logiques.

Abstraction

L'abstraction consiste à définir une interface (ou une classe abstraite) qui impose un contrat aux classes qui en héritent. Elle permet de travailler avec des objets de types différents, tant qu'ils respectent l'interface commune.

Polymorphisme

Ce principe permet d'utiliser de manière uniforme des objets différents ayant une même interface. Cela permet, par exemple, d'appeler une méthode afficher() sur des objets de classes différentes, à condition que cette méthode soit définie dans chacune de ces classes.

Autres

On pourrait encore parler de composition ou de la classe ABC pour faire les interfaces ou de l'UML qui vous permet de schématiser le process mais ceci est un résumé de votre cours de BA1 pas un cours.

2. Organisation modulaire du code

Un projet comme celui-ci est plus lisible et maintenable lorsqu'il est divisé en modules. Chaque classe ou ensemble logique est placé dans son propre fichier Python, appelé module. Les modules peuvent être importés les uns dans les autres pour organiser le code de façon claire et réutilisable. On utilise donc correctement son Git et on travaille en équipe en se partageant les modules ! Le premier qui dit je suis en sécu, je l'enferme avec Sir PTZ pour un moment de douceur !

3. Tests unitaires

Les tests unitaires permettent de vérifier automatiquement que chaque unité du programme (fonction, méthode, classe) fonctionne comme prévu. Ils sont essentiels pour garantir la fiabilité d'un code, surtout en cas de modification ultérieure.

En Python, le module unittest permet de créer des classes de test contenant des méthodes qui utilisent des assertions (telles que assertEqual, assertRaises, etc.).

Un test bien conçu doit :

- être indépendant des autres tests,
- tester un comportement précis,
- être exécuté automatiquement et régulièrement.

Mocks et tests isolés

Dans certains cas, les objets que nous testons dépendent d'autres objets ou services (comme une base de données, un fichier, une API externe ou tiens tiens un capteur). Pour éviter que le test ne devienne dépendant de ces éléments externes (ou que le code crame tout), on utilise des **mocks** (objets factices).

Un mock est un objet qui imite le comportement d'un objet réel, mais de manière contrôlée. Il permet de :

- simuler des réponses sans vraiment appeler le service externe,
- observer comment un objet est utilisé (combien de fois une méthode est appelée, avec quels arguments, etc.),
- forcer certaines erreurs ou comportements pour tester des cas limites.

Python fournit le module unittest.mock pour créer facilement ces objets simulés. Par exemple :

```
from unittest.mock import MagicMock
db = MagicMock()
db.insert.return_value = True
```

On peut maintenant utiliser db comme s'il s'agissait d'une vraie base

L'utilisation des mocks est indispensable pour tester des composants isolés, en évitant les effets de bord, tout en permettant de simuler des scénarios complexes de manière fiable.

4. Gestion des erreurs et des exceptions

Lorsqu'une situation anormale survient (division par zéro, accès interdit, données invalides), il est préférable de lever une exception plutôt que de laisser le programme planter silencieusement. En Python, on utilise les instructions raise, try, except, finally pour gérer proprement ces situations.

Lever des exceptions explicites permet de mieux contrôler le comportement de l'application et d'éviter des bugs silencieux. On peut donc créer notre propre comportent et ça j'achète !!!!

5. Notions de multithreading et accès concurrents

Le multithreading permet d'exécuter plusieurs tâches simultanément. En Python, cela peut poser des problèmes si plusieurs threads modifient les mêmes données sans coordination. Cela peut provoquer des erreurs dites "conditions de course".

Pour éviter cela, on utilise des **verrous (locks)** pour s'assurer qu'une seule tâche à la fois accède à une ressource partagée.

- threading.Lock() est un verrou simple : un seul thread peut l'acquérir.
- threading.RLock() est un verrou réentrant : le même thread peut l'acquérir plusieurs fois (utile dans les appels imbriqués).

Le mot-clé with est utilisé pour manipuler les verrous proprement :

```
with self._lock:
    # accès exclusif à la ressource partagée
```

6. SI je fais pas , je réussis pas

- Utiliser des noms clairs et significatifs pour les variables, méthodes et classes.
- Documenter les fonctions complexes ou les parties non évidentes.
- Respecter la convention d'encapsulation (_variable pour les attributs internes).
- Factoriser le code répété.
- Organiser les fichiers et les modules de manière logique.
- Tester fréquemment au fur et à mesure de l'avancement du code.