TP - Programmation Orientée Objet Avancée en Python

Ce TP est composé de 19 exercices indépendants.
Chaque exercice est conçu pour approfondir vos compétences en POO tout en introduisant des défis de plus en plus complexes.

Exercice 1 : Classe Abstraite Simple

Créez une classe abstraite `Shape` avec une méthode abstraite `area()`.
Implémentez deux classes dérivées : `Circle` et `Rectangle`.
Chaque classe devra implémenter sa propre version de la méthode `area()`.

Exemple de test :

```
circle = Circle(5)
assert round(circle.area(), 2) == 78.54  # Aire d'un cercle de rayon 5

rectangle = Rectangle(3, 4)
assert rectangle.area() == 12  # Aire d'un rectangle de 3x4
```

Exercice 2 : Surcharge d'Opérateurs

Créez une classe `BankAccount` avec un solde initial.
Surchargez les opérateurs `+` et `-` pour permettre d'ajouter ou retirer de l'argent du compte bancaire en utilisant ces opérateurs.

Exemple de test :

```
account = BankAccount(100)
account += 50  # Ajoute 50 euros
assert account.balance == 150

account -= 20  # Retire 20 euros
assert account.balance == 130
```

Exercice 3 : Décorateurs

Créez un décorateur `@check_positive` qui vérifie si le nombre passé en argument à une fonction est positif. Si le nombre est négatif, levez une exception `ValueError`.

Exemple de test :

```
@check_positive
def double(x):
    return x * 2


assert double(5) == 10
try:
    double(-1)
except ValueError:
    print("Exception levée correctement")
```

Exercice 4 : Propriétés (Property)

Créez une classe `Car` avec un attribut privé `speed`.
Utilisez le décorateur `@property` pour lire la vitesse et un `setter` pour la modifier. La vitesse ne peut pas dépasser 200 km/h (et doit être > 0), sinon une exception `ValueError` est levée.

Exemple de test :

```
car = Car()
car.speed = 120
assert car.speed == 120

try:
    car.speed = 250  # Doit lever une exception
except ValueError:
    print("Vitesse non valide")
```

Exercice 5 : Gestion des Exceptions

Créez une classe `Person` qui prend un nom et un âge en entrée.
Utilisez une exception personnalisée `InvalidAgeError` pour lever une erreur si l'âge est négatif ou supérieur à 150 ans.

Exemple de test :

```
try:
    person = Person("John", 200)
```

```
except InvalidAgeError:
    print("Age non valide")
```

Exercice 6 : Singleton et context manager

Implémentez un pattern **Singleton** pour une classe `DatabaseConnection` qui garantit qu'il n'existe qu'une seule instance de connexion à la base de données.

L'instance de cette classe doit permettre de créer un context (qui, lui, n'est pas unique), et qui permet d'ajouter une entrée (id, data), de la supprimer par id, ou de drop toutes les lignes.
Les opérations doivent être exécutées (flush) une fois le context fermé.

Exercice 7 : Factory Pattern

Créez une **factory** pour générer des objets `Circle` ou `Rectangle` (héritant d'une classe abstraite `Shape`) en fonction d'un paramètre passé à une méthode `create`. Vous devez pouvoir choisir quelle forme créer en fonction des paramètres fournis.

Exemple de test :

```
shape1 = ShapeFactory.create(shape_type="circle", radius=5)
assert isinstance(shape1, Circle)
assert round(shape1.area(), 2) == 78.54

shape2 = ShapeFactory.create(shape_type="rectangle", width=3, height=4)
assert isinstance(shape2, Rectangle)
assert shape2.area() == 12
```

Exercice 8 : Décorateurs avec Paramètres

Créez un décorateur `@timeout_limit` qui prend un paramètre indiquant une limite de temps en secondes (timeout).
Si une fonction prend plus de temps que cette limite pour s'exécuter, le décorateur devra lever une exception `TimeoutError`.

Exercice 8 Bonus : Avec un thread

Même chose, mais le décorateur prend un paramètre supplémentaire `raise_exception`, ayant la valeur par défaut à False. Si la valeur est True, vous devez lever une exception même si la fonction n'est pas terminée (vous devez interrompre l'action en cours).

Exercice 9 : Opérateurs Avancés

Créez une classe `Matrix` et surchargez les opérateurs `+` et `*` pour permettre l'addition et la multiplication de matrices.
Gérez les exceptions si les matrices ne sont pas de tailles compatibles.

Exemple de test :

```
matrix1 = Matrix([[1, 2], [3, 4]])
matrix2 = Matrix([[5, 6], [7, 8]])

result_add = matrix1 + matrix2
assert result_add.data == [[6, 8], [10, 12]]

result_mul = matrix1 * matrix2
assert result_mul.data == [[19, 22], [43, 50]]
```

Exercice 10 : Classes Abstraites et Factory

Créez une classe abstraite `Animal` avec une méthode abstraite `speak()`.
Implémentez des classes dérivées `Dog` et `Cat`.
Ensuite, implémentez une **factory** `AnimalFactory` qui génère une instance de `Dog` ou `Cat` en fonction des paramètres d'entrée.
Les paramètres sont :

- `animal_type` : une chaîne de caractères contenant soit "dog" soit "cat" (ou autre).
- `name` : une chaîne de caractères contenant le nom de l'animal.

Exemple de test :

```
dog = AnimalFactory.create(animal_type="dog", name="Buddy")
assert isinstance(dog, Dog)
assert dog.speak() == "Woof"

cat = AnimalFactory.create(animal_type="cat", name="Misty")
```

```python
assert isinstance(cat, Cat)
assert cat.speak() == "Meow"
```

Exercice 11 : Surcharge d'Opérateurs (Comparaison)

Créez une classe `Product` avec des attributs `name` et `price`.
Surchargez les opérateurs de comparaison (`==`, `<`, `>`, etc.) pour comparer des objets `Product` en fonction de leur prix.

Exemple de test :

```python
product1 = Product("Apple", 1.50)
product2 = Product("Banana", 1.20)

assert product1 > product2
```

Exercice 11 Bonus : Surcharge d'Opérateurs (Comparaison Avancée)

Créez une fonction `top_products(products, k)` qui prend en paramètre une liste de produits et un entier `k`, et qui retourne les `k` produits les plus chers.

Exemple de test :

```python
def top_products(products: list[Product], k: int):
    pass


# Exemples
product1 = Product("Laptop", 1200)
product2 = Product("Phone", 800)
product3 = Product("Tablet", 600)
product4 = Product("Monitor", 300)

# Liste de produits
products = [product1, product2, product3, product4]

# Récupérer les 2 produits les plus chers
```

```python
top_k = top_products(products, 2)
assert top_k == [product1, product2]
```

Exercice 12 : Propriétés et Gestion d'Exceptions

Créez une classe `Account` avec une propriété `balance`.
Si un dépôt est inférieur à 0 ou si un retrait rend le solde négatif, une exception `ValueError` doit être levée.
Utilisez le décorateur `@property` pour gérer les accès et modifications du solde.

Exemple de test :

```python
account = Account()
account.balance = 100

try:
    account.balance -= 150  # Doit lever une exception
except ValueError:
    print("Solde insuffisant")
```

Exercice 13 : Chainage de Décorateurs

Créez deux décorateurs : `@check_positive` et `@add_log`.
Le premier vérifie qu'un nombre est positif, le second logge l'exécution de la fonction.
Appliquez les deux décorateurs à une fonction et assurez-vous que le résultat soit correct.

Exercice 14 : Surcharge d'Opérateurs

Créez une classe `Vector` représentant un vecteur mathématique à deux dimensions.
Implémentez les opérations de `+`, `-`, et `*`.

Exemple de test :

```python
v1 = Vector(1, 2)
v2 = Vector(3, 4)

v3 = v1 + v2
assert v3.x == 4 and v3.y == 6
```

```python
v4 = v1 * 2
assert v4.x == 2 and v4.y == 4
```

Exercice 15 : Mock

Les frameworks de tests proposent des méthodes permettant de "mocker" des objets. Votre but est de créer une classe `UserService` qui récupère des informations d'un utilisateur via une API externe.
Implémentez une méthode `get_user_data(user_id)` qui simule une requête à une API pour obtenir les données d'un utilisateur.

Dans un test unitaire, utilisez un **mock** pour simuler l'API et assurez-vous que votre méthode renvoie les bonnes informations.

Exemple :

```python
class UserService:
    def get_user_data(self, user_id):
        # Simuler une requête API externe pour obtenir les données
        ↪ d'un utilisateur
        pass

    # Dans un test unitaire, utiliser


un
mock:


def test_get_user_data(mocker):
    user_service = UserService()
    mocker.patch.object(user_service, 'get_user_data',
↪   return_value={"id": 1, "name": "John Doe"})
    assert user_service.get_user_data(1) == {"id": 1, "name": "John
    ↪ Doe"}
```

Exercice 16 : Classes Génériques et Méthodes Statistiques

Créez une classe générique `Statistics` qui accepte une liste de nombres et fournit des méthodes pour calculer la moyenne, la médiane et la variance des données.

Utilisez le module `statistics` pour vous aider.

Exercice 17 : Vecteurs et Calculs

Créez une classe `Vector3D` représentant un vecteur en trois dimensions.
Implémentez la méthode pour calculer la norme, l'addition et le produit scalaire entre deux vecteurs 3D.
Utilisez la surcharge des opérateurs pour ces opérations.

Exemple de test :

```
v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)

norm_v1 = v1.norm()
assert round(norm_v1, 2) == 3.74

dot_product = v1 * v2
assert dot_product == 32  # Produit scalaire

v3 = v1 + v2
assert v3.x == 5 and v3.y == 7 and v3.z == 9
```

Consignes Générales :

- Chaque exercice est indépendant. Vous pouvez les faire dans l'ordre de votre choix.
- Testez bien vos programmes et assurez-vous que toutes les exceptions sont correctement gérées.
- L'ensemble du code doit être compatible avec le fichier de test fourni.