Cours 3 : Programmation Orientée Objet Avancée en Python

Objectifs du Cours :

À l'issue de ce cours, vous serez capables de : - Comprendre et utiliser les **classes abstraites** pour structurer votre code. - Surcharger des **opérateurs** et des **méthodes** pour adapter le comportement des objets. - Créer et utiliser des **décorateurs** pour enrichir des fonctions ou méthodes. - Utiliser des **propriétés** pour contrôler l'accès aux attributs privés. - Gérer efficacement les **exceptions**, avec des captures groupées et des exceptions personnalisées. - Mettre en œuvre des **design patterns** classiques, tels que le **Pattern Factory** et le **Pattern Singleton**. - Manipuler des **context managers** avec le mot-clé `with`.

---

1. Classes Abstraites en Python

Les **classes abstraites** sont des classes qui ne peuvent pas être instanciées directement et qui contiennent des méthodes que les classes dérivées doivent implémenter. En Python, elles sont définies à l'aide du module `abc` (Abstract Base Classes).

Exemple :

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return f"{self.name} aboie."

class Cat(Animal):
    def sound(self):
        return f"{self.name} miaule."

# Utilisation
```

```python
dog = Dog("Rex")
cat = Cat("Whiskers")

print(dog.sound())  # Rex aboie.
print(cat.sound())  # Whiskers miaule.
```

- **Explication** : La classe `Animal` est abstraite et ne peut pas être instanciée directement. Elle oblige les classes dérivées (`Dog` et `Cat`) à implémenter la méthode `sound()`.

---

2. Surcharge d'Opérateurs

Python permet de **surcharger des opérateurs** en définissant des méthodes spéciales (aussi appelées méthodes magiques) comme `__add__`, `__sub__`, `__eq__`, etc., pour permettre aux objets d'interagir avec les opérateurs de manière personnalisée.

Exemple de surcharge de l'opérateur + :

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Surcharge de l'opérateur +
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Utilisation
v1 = Vector(1, 2)
v2 = Vector(3, 4)

v3 = v1 + v2  # Appelle __add__
print(v3)  # Vector(4, 6)
```

- **Explication** : La méthode `__add__` définit l'opérateur `+` pour la classe `Vector`.

---

3. Surcharge de Méthodes

La **surcharge de méthodes** permet de redéfinir une méthode d'une classe parent dans une classe dérivée. En Python, cela se fait en réécrivant la méthode dans la classe dérivée tout en conservant la possibilité d'appeler la méthode de la classe parent via `super()`.

Exemple :

```
class Animal:
    def speak(self):
        return "Un animal fait du bruit."

class Dog(Animal):
    def speak(self):
        return "Le chien aboie."

# Utilisation
animal = Animal()
dog = Dog()

print(animal.speak())  # Un animal fait du bruit.
print(dog.speak())  # Le chien aboie.
```

- **Explication** : La méthode `speak()` est surchargée dans la classe `Dog`.

---

4. Design Pattern - Factory

Le **Pattern Factory** permet de déléguer la création d'objets à une méthode dédiée, en masquant la logique de création derrière une interface commune. Cela permet de créer des objets dynamiquement en fonction des besoins, sans connaître les détails de leur instanciation.

Exemple :

```
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
```

```
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def get_animal(animal_type):
        match animal_type:
            case "dog":
                return Dog()
            case "cat":
                return Cat()
            case _:
                return None

# Utilisation
animal = AnimalFactory.get_animal("dog")
print(animal.speak())  # Woof!
```

- **Explication** : La factory `AnimalFactory` permet de créer un objet `Dog` ou `Cat` en fonction d'une chaîne de caractères. Le `match` permet de gérer dynamiquement la création des instances.

---

5. Décorateurs : Enrichissement des Méthodes

Les **décorateurs** sont utilisés pour enrichir ou modifier des fonctions ou méthodes de manière transparente, sans changer leur logique interne.

Exemple de décorateur :

```
def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Exécution de {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Fin de {func.__name__}")
        return result
    return wrapper
```

```python
@log_execution
def say_hello():
    print("Hello, World!")

say_hello()
```

- **Explication** : Le décorateur `log_execution` permet d'ajouter un message de log avant et après l'exécution de la fonction `say_hello()`.

---

6. Propriétés (`@property`)

Les **propriétés** permettent de définir des méthodes qui agissent comme des attributs. Cela permet de contrôler l'accès et la modification d'attributs privés.

Exemple d'utilisation des propriétés :

```python
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value < 0:
            raise ValueError("L'âge doit être positif")
        self._age = value

p = Person("Alice", 30)
p.age = -5  # Lève une erreur
```

- **Explication** : La propriété `age` permet de contrôler l'attribut `_age` et de valider que l'âge est positif.

---

7. Gestion Avancée des Exceptions

Les exceptions en Python peuvent être gérées de manière plus fine avec des captures groupées et des exceptions personnalisées.

Exemple de capture groupée d'exceptions :

```
try:
    x = int(input("Entrez un nombre : "))
    y = 10 / x
except (ValueError, ZeroDivisionError) as e:
    print(f"Erreur : {e}")
```

- **Explication** : Ici, plusieurs types d'exceptions (`ValueError` et `ZeroDivisionError`) sont capturés dans une seule clause `except`.

---

8. Méthodes Magiques Avancées

Outre la surcharge d'opérateurs, Python propose plusieurs **méthodes magiques** comme `__repr__`, `__str__`, ou encore `__call__` pour ajouter du comportement spécial aux objets.

Exemple de `__call__` :

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, x):
        return x * self.factor

double = Multiplier(2)
print(double(10))  # Renvoie 20
```

- **Explication** : La méthode `__call__` permet à un objet d'être utilisé comme une fonction.

---

9. Design Pattern - Singleton

Le **Pattern Singleton** garantit qu'une classe n'a qu'une seule instance. C'est un pattern utile pour gérer des ressources globales dans un programme.

Exemple de Singleton :

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

- **Explication** : Le pattern `Singleton` ici garantit que la classe n'aura qu'une seule instance.
