Chapitre 5. Notions avancées : Design patterns

1. Objets fonctions en Python

En Python, **les fonctions sont des objets de première classe** *(*first-class objects*)*. Cela signifie qu'elles peuvent être :

- Affectées à des variables,
- Passées en arguments,
- Retournées par d'autres fonctions,
- Contenues dans des listes ou dictionnaires.

Exemple

```
def saluer(nom):
    print("Bonjour", nom)

# Affectation à une variable
f = saluer
f("Ahmed")

# Passage en argument
def executer_fonction(mafonction, valeur):
    mafonction(valeur)
executer_fonction(saluer, "yacine")
```

```
>>>
==============
Bonjour Ahmed
Bonjour yacine
```

Les fonctions se comportent comme des objets manipulables, ce qui ouvre la voie à des modèles de conception très flexibles.

Exercice 1

Créer une fonction operation(f, a, b) qui prend une fonction binaire f (par ex. add, mul, sub) et deux nombres, puis exécute l'opération correspondante.

Solution

```
#exercice 1
def operation(f, a, b):
    return f(a, b)

# Exemples de fonctions binaires :
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

def mul(x, y):
    return x * y

# Utilisation :
print(operation(add, 3, 4))  # 7
print(operation(sub, 10, 3)) # 7
print(operation(mul, 6, 5))  # 30
  >>>
  ===
  7
  7
  30
```

2. Introduction aux Design Patterns

2.1. Définition

Un **Design Pattern** *(*ou modèle de conception) est une **solution standard et réutilisable** à un problème de conception récurrent.

2.2. Catégories principales

Catégorie	**Objectif**
Création	Instanciation contrôlée d'objets.
Structure	Organisation des classes et objets.
Comportement	Interaction et communication entre objets.

2.3. Avantages

- Favorise la **réutilisabilité** du code,
- Améliore la **clarté** et la **maintenabilité**,
- Facilite la **collaboration** entre développeurs.

3. Patterns de création

3.1. Singleton

Un **Singleton** est un patron de conception (design pattern) dont le but est de garantir qu'une **classe n'a qu'une seule instance** dans tout le programme. Autrement dit : Peu importe combien de fois tu fais `MaClasse()`, tu obtiens **toujours le même objet**, créé une seule fois.

On utilise un Singleton quand on veut :

- **Un objet unique pour tout le programme**

Exemples :

- un gestionnaire de configuration

- Un journal de logs
- Une connexion à une base de données
- Un pilote matériel unique (ex : accès au GPU)

➢ **Un accès centralisé à une ressource**

Comme tout le programme utilise le même objet, cela évite incohérences, duplications et conflits.

Le principe est toujours le même :

1. **La classe possède une seule instance possible**, souvent stockée dans une variable interne.
2. **Quand on essaie de créer un nouvel objet**, la classe :
   - Soit **renvoie l'instance existante**,
   - Soit **créé l'instance** si elle n'existe pas encore.

Donc on "verrouille" la création pour qu'elle n'arrive *qu'une seule fois*.

Imagine un immeuble où il n'y a :

- Qu'un seul **gardien**
- Peu importe combien de personnes veulent parler au gardien
- Chacun est dirigé vers **le même gardien**

Le Singleton = ce gardien unique.

Exemple 1:

```
class Singleton:   #On définit une classe normalement.
    _instance = None  #C'est une variable de classe (partagée par toutes les instances).
                      #Elle servira à stocker l'unique instance du Singleton.

    def __new__(cls):   #__new__ est une méthode spéciale exécutée avant __init__.
                        #Elle sert à créer l'objet (alors que __init__ l'initiase)
                        #Idéal pour contrôler la création d'instances
        if cls._instance is None:  #Si aucune instance n'a encore été créée, on en crée une nouvelle.
            cls._instance = super().__new__(cls)  #On utilise la méthode normale de création d'objet.
        return cls._instance    #Peu importe combien de fois on appelle Singleton(),
                                #la même instance sera retournée.
# Test: vérifie si les deux variables pointent vers le même objet en mémoire.Et c'est le cas : True.
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
=====
True
>>>
```

Exemple 2: un gestionnaire de configuration (Singleton)

Ce Singleton charge une configuration une seule fois, puis toute l'application utilise la même instance.

```
#exemple 2
class ConfigManager:
    _instance = None  # stocke l'unique instance

    def __new__(cls):
        if cls._instance is None:
            # création de l'instance si elle n'existe pas encore
            cls._instance = super().__new__(cls)
            cls._instance.settings = {}  # un dictionnaire pour la config
        return cls._instance

    def set(self, key, value):
        self.settings[key] = value

    def get(self, key, default=None):
        return self.settings.get(key, default)


# --- Utilisation ----

c1 = ConfigManager()
c2 = ConfigManager()

c1.set("theme", "dark")
print(c2.get("theme"))  # affiche : dark

print(c1 is c2)  # True → les deux variables pointent vers le même objet

>>>dark
   True
```

☞ Très utile pour gérer les **ressources partagées** (base de données, logger, configuration globale…).

Le Singleton permet :

- **Une seule instance** dans tout le programme
- Un **accès global et partagé**
- De **centraliser les ressources** coûteuses ou sensibles

C'est un pattern simple mais très puissant pour structurer vos applications.

3.2. Factory Method (méthode fabrique)

Le pattern **Factory** Method permet de déléguer la création d'objets à une classe spécialisée (appelée « fabrique »). L'objectif : éviter d'instancier directement les objets dans le code et rendre le programme plus flexible et extensible. Le **Factory Method** est un design pattern qui sert à **centraliser la création d'objets** dans une classe dédiée — une fabrique

Au lieu d'écrire partout dans le code :

```
forme = Cercle()
# ou
forme = Carre()
```

on demande à une **factory** de fournir l'objet voulu. Pourquoi ? Parce que ça rend le code :

- **Plus flexible** (On peut changer les classes créées sans modifier le reste),

- **Plus propre** (Tout le code de création est regroupé),
- **Plus extensible** (Ajouter une nouvelle forme se fait sans casser le code existant).

Sans Factory, le code ferait ceci :

```
if type_forme == "cercle":
    forme = Cercle()
elif type_forme == "carre":
    forme = Carre()
```

- Ce code doit savoir comment créer chaque type d'objet = il n'est pas flexible.
- Le code dépend directement des classes `Cercle`, `Carre`, etc.
- Si on ajoute une nouvelle forme (`Triangle`), on doit modifier ce bloc partout où on le fait.
- Le code n'est pas modulable.

Avec Factory : création d'objets centralisée

Avec une factory, la création d'objets est isolée dans une seule classe, ce qui permet :

- De changer les classes sans toucher au reste du code
- D'ajouter de nouveaux types d'objets facilement
- D'éviter de dupliquer du code d'instanciation

```
class FormeFactory:
  def creer_forme(self, type_forme):
      if type_forme == "cercle":
          return Cercle()
      elif type_forme == "carre":
          return Carre()
```

Le code qui utilise les formes **n'a plus besoin de connaître leurs classes réelles**.

Exemple

```python
#1. Une classe abstraite
class Forme:
    def dessiner(self):
        pass
 #Toutes les formes doivent avoir une méthode dessiner().
#-------------------------------------------------------------------------
#2. Deux implémentations
class Cercle(Forme):
    def dessiner(self):
        print("Cercle dessiné")
class Carre(Forme):
    def dessiner(self):
        print("Carré dessiné")
#-------------------------------------------------------------------------
#3. La Factory
class FormeFactory:
    def creer_forme(self, type_forme):
        if type_forme == "cercle":
            return Cercle()
        elif type_forme == "carre":
            return Carre()

#C'est la seule classe qui connaît les types concrets (Cercle, Carre).
#-------------------------------------------------------------------------
#4. Utilisation
factory = FormeFactory()
forme = factory.creer_forme("cercle")
forme.dessiner()
#Le code utilisateur ne sait pas comment la forme est créée : il reçoit juste une forme.

#-------------------------------------------------------------------------
>>> Cercle dessiné
```

Pour ajouter une nouvelle forme, exemple Triangle, tu fais juste :

- Créer une classe Triangle
- Ajouter un cas dans FormeFactory

Aucun autre fichier du projet n'a besoin d'être modifié.

Exercice 2

Créer un pattern **Factory** pour produire des objets EtudiantLicence et EtudiantMaster, tous deux héritant d'une classe Etudiant.

La solution

Structure attendue

- Une classe **parent** : Etudiant
- Deux classes filles : EtudiantLicence, EtudiantMaster
- Une **Factory** : EtudiantFactory
- Un code d'utilisation

```python
class Etudiant:
    def __init__(self, nom):
        self.nom = nom
    def presentation(self):
        pass
class EtudiantLicence(Etudiant):
    def presentation(self):
        print(f"Étudiant Licence : {self.nom}")
class EtudiantMaster(Etudiant):
    def presentation(self):
        print(f"Étudiant Master : {self.nom}")
class EtudiantFactory:
    def creer_etudiant(self, type_etudiant, nom):
        if type_etudiant == "licence":
            return EtudiantLicence(nom)
        elif type_etudiant == "master":
            return EtudiantMaster(nom)
        else:
            raise ValueError("Type d'étudiant inconnu")

# --- Utilisation ---
factory = EtudiantFactory()

e1 = factory.creer_etudiant("licence", "ahmed")
e2 = factory.creer_etudiant("master", "yacine")
e1.presentation()
e2.presentation()

>>> Étudiant Licence : ahmed
    Étudiant Master : yacine
```

Explication rapide

- **Sans factory**

Le code devrait décider lui-même s'il doit créer un étudiant licence ou master → pas flexible.

- **Avec factory**

La méthode creer_etudiant() centralise toute la logique de création.

Le reste du programme ne connaît plus les classes EtudiantLicence ou EtudiantMaster.

4. Patterns structurels

Ces modèles organisent **les relations entre objets et classes** pour obtenir des architectures flexibles.

4.1. Le pattern Composite

Permet de traiter **de manière uniforme** des objets simples et des compositions d'objets.

Le **pattern Composite** sert à représenter une **hiérarchie d'objets en arbre** où :

- certains objets sont **simples** → *Feuille*
- d'autres sont des **compositions** d'objets → *Composite*

L'idée clé :

Traiter les objets simples et les objets composés de la même manière, via une interface commune.

C'est très utile pour manipuler des structures hiérarchiques : fichiers/dossiers, menus, arbres XML, scènes 3D, etc.

Exemple :

Classe Composant : C'est l'**interface commune**. Elle définit une méthode operation() que toutes les sous-classes devront implémenter.

Classe Feuille : Une feuille **n'a pas d'enfants**. La méthode operation() effectue directement l'action (ici : afficher "Feuille")

Classe Composite : C'est un **objet composé**, un **nœud** qui contient d'autres composants. enfants est une liste de composants (Feuille ou Composite). La méthode ajouter () ajoute un enfant à la liste.

La méthode operation() :

- affiche "Composite :"
- puis **appelle l'operation() de chaque enfant**. c'est là que la récursion naturelle apparaît et permet de parcourir tout l'arbre.

Construction de l'arbre :

racine (Composite)
├── Feuille
├── Feuille
└── sous_arbre (Composite)
  └── Feuille

Ce qui se passe quand on appelle racine.operation(). racine affiche :

Composite :

puis appelle operation() sur ses enfants :

- Feuille → affiche "Feuille"
- Feuille → affiche "Feuille"
- sous_arbre (Composite) → affiche "Composite :"
  puis appelle ses enfants → une Feuille

```
4.1. Le pattern Composite
Permet de traiter de manière uniforme des objets simples et des compositions d'objets.
Exemple :"""
class Composant:
    def operation(self):
        pass

class Feuille(Composant):
    def operation(self):
        print("Feuille")

class Composite(Composant):
    def __init__(self):
        self.enfants = []

    def ajouter(self, c):
        self.enfants.append(c)

    def operation(self):
        print("Composite:")
        for c in self.enfants:
            c.operation()

# Exemple
racine = Composite()
racine.ajouter(Feuille())
racine.ajouter(Feuille())

sous_arbre = Composite()
sous_arbre.ajouter(Feuille())
racine.ajouter(sous_arbre)
```

```
>>> Composite:
    Feuille
    Feuille
    Composite:
    Feuille
```

Très utilisé pour **représenter des arborescences** (fichiers, dossiers, hiérarchies d'objets).

4.2. Le pattern Décorateur

Permet d'**ajouter dynamiquement des fonctionnalités** à un objet sans modifier sa structure.

Exercice 3

Créer un décorateur Reduction qui réduit le prix d'un produit selon un pourcentage donné.

La solution

Objectif

Créer un décorateur Reduction qui :

- prend un pourcentage (ex : 20 = -20 %)
- modifie le résultat d'une fonction qui renvoie un prix

```
def Reduction(pourcentage):
    def decorateur(fonction_prix):
        def wrapper(*args, **kwargs):
            prix_initial = fonction_prix(*args, **kwargs)
            reduction = prix_initial * (pourcentage / 100)
            prix_final = prix_initial - reduction
            return prix_final
        return wrapper
    return decorateur
# Exemple d'utilisation
class Produit:
    def __init__(self, nom, prix):
        self.nom = nom
        self.prix = prix
    @Reduction(20)   # applique une réduction de 20 %
    def get_prix(self):
        return self.prix
p = Produit("Chaussures", 100)
print(p.get_prix())   # → 80

>>> 80.0
```

Explication

◆ **Le décorateur Reduction(pourcentage)**

- prend un pourcentage en paramètre
- retourne un décorateur
- qui **intercepte le prix**, calcule la réduction et renvoie le prix réduit

◆ **L'utilisation**

```
@Reduction(20)
def get_prix():
```

signifie : "utilise cette fonction, mais avec -20 % appliqués au résultat".

5. Décorateurs de fonctions

Les **décorateurs** sont une fonctionnalité puissante de Python permettant de **modifier le comportement d'une fonction** sans en changer le code.

◆ **Syntaxe de base**

```
def mon_decorateur(fonction):
    def wrapper(*args, **kwargs):
        print("Avant l'exécution")
        resultat = fonction(*args, **kwargs)
        print("Après l'exécution")
        return resultat
    return wrapper

@mon_decorateur
def dire_bonjour():
    print("Bonjour !")

dire_bonjour()
```

```
Avant l'exécution
Bonjour !
Après l'exécution
```

◆ **Application pratique : mesure du temps d'exécution**

```
import time

def mesurer_temps(f):
    def wrapper(*args, **kwargs):
        debut = time.time()
        result = f(*args, **kwargs)
        fin = time.time()
        print(f"Durée : {fin - debut:.4f}s")
        return result
    return wrapper

@mesurer_temps
def somme(n):
    return sum(range(n))

somme(10_000_000)
```

```
>>> Durée : 0.7140s
```

Exercice 4

Créer un décorateur verifier_type qui affiche un avertissement si les arguments d'une fonction ne sont pas du bon type.

6. Décorateurs de classes

Un **décorateur de classe** permet de modifier ou d'enrichir une classe entière, au moment de sa création.

◆ **Exemple**

```python
def ajouter_id(cls):
    cls.id = 0
    def incrementer_id(self):
        cls.id += 1
        return cls.id
    cls.incrementer_id = incrementer_id
    return cls

@ajouter_id
class Personne:
    def __init__(self, nom):
        self.nom = nom

p1 = Personne("Alice")
print(p1.incrementer_id())  # 1
```

```
>>> 1
```

Très utile pour **ajouter des comportements communs** à plusieurs classes (ex. suivi d'instances, validation automatique, journalisation).