

HEC-KIN
Section informatique
L1 informatique de gestion
Année académique 2024-2025



◆ **Semaine 1 – Introduction à la programmation orientée objet**

- **Motivation : pourquoi la POO ?**

La **programmation orientée objet (POO)** a été développée pour mieux modéliser des systèmes complexes, en s'inspirant de la manière dont nous percevons le monde réel : comme un ensemble d'objets interagissant entre eux. Voici les principales **raisons (motivations)** pour lesquelles la POO est utilisée :

1. Modélisation naturelle du monde réel

- Dans la POO, on représente des entités sous forme **d'objets** (ex : Voiture, Utilisateur, Facture), ce qui correspond à notre façon naturelle de penser.
- Chaque objet possède des **propriétés (attributs)** et des **comportements (méthodes)**.

Exemple :

class Voiture:

```
def __init__(self, marque, vitesse):  
    self.marque = marque  
    self.vitesse = vitesse  
  
def accelerer(self):  
    self.vitesse += 10
```

2. Réutilisabilité du code

- Grâce à l'**héritage**, une classe peut hériter des propriétés et comportements d'une autre, évitant la duplication de code.
- Les classes peuvent être **étendues** ou modifiées sans changer le code existant.

3. Encapsulation

- L'encapsulation consiste à **cacher les détails internes** d'un objet et à exposer une interface publique propre.
- Cela protège l'intégrité des données et **réduit les erreurs**.

Exemple :

```
class CompteBancaire:
    def __init__(self, solde):
        self.__solde = solde # Attribut privé
    def déposer(self, montant):
        if montant > 0:
            self.__solde += montant
```

4. Facilité de maintenance et d'évolution

- La POO permet de **modifier une partie du système sans impacter le reste** si le code est bien structuré.
- Cela facilite la correction de bugs et l'ajout de fonctionnalités.

5. Polymorphisme

- Le polymorphisme permet d'utiliser la **même interface pour des objets différents**, ce qui favorise la flexibilité et l'extensibilité du code.

Exemple :

```
class Animal:
    def parler(self):
        pass
class Chien(Animal):
    def parler(self):
        print("Wouf!")
class Chat(Animal):
    def parler(self):
        print("Miaou!")
def faire_parler(animal):
    animal.parler()
```

6. Organisation et structure

- La POO encourage une **architecture modulaire** claire et bien définie (ex : MVC, design patterns).
- Elle permet de **diviser un gros problème en petits objets collaborant entre eux**.

En résumé :

La POO est une **réponse à la complexité croissante des logiciels**. Elle rend le code plus **intuitif, modulaire, réutilisable et maintenable**, en s'appuyant sur des concepts inspirés du monde réel.

- **Comparaison procédural vs objet**

Voici un **tableau comparatif clair et synthétique** entre la **programmation procédurale** et la **programmation orientée objet (POO)** :

| Critère | Programmation procédurale | Programmation orientée objet (POO) |
|----------------------------------|--|---|
| Principe de base | Basée sur les fonctions et l' exécution séquentielle | Basée sur les objets (données + comportements) |
| Structure du code | Organisé en procédures (fonctions) | Organisé en classes et objets |
| Modélisation | Modèle centré sur les actions | Modèle centré sur les entités (objets) |
| Réutilisabilité | Moins flexible, nécessite souvent du copier-coller | Réutilisation facilitée via héritage et composition |
| Encapsulation | Données et fonctions souvent séparées | Données et méthodes regroupées dans les objets |
| Maintenance | Peut devenir difficile avec des projets complexes | Plus facile à maintenir grâce à une structure modulaire |
| Extensibilité | Plus difficile à faire évoluer sans risquer de casser le code | Plus facile à faire évoluer sans casser le reste du code |
| Exemples de langages | C, Pascal, Fortran | Java, C++, Python, C#, Ruby |
| Polymorphisme / Héritage | Non natifs (doit être simulé manuellement) | Intégrés nativement à la plupart des langages OO |
| Lisibilité et abstraction | Moins adaptée aux grands systèmes | Mieux adaptée aux systèmes complexes et à l'abstraction |

 **Exemple simple :**
 **Procédural (en Python) :**

```
def aire_rectangle(longueur, largeur):
    return longueur * largeur

print(aire_rectangle(5, 3))
```

 **Orienté objet :**

```
class Rectangle:
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur

    def aire(self):
        return self.longueur * self.largeur

r = Rectangle(5, 3)
print(r.aire())
```

- **Concepts fondamentaux : objets, classes, attributs, méthodes**

Voici une **explication claire et structurée** des **concept fondamentaux de la programmation orientée objet (POO)** : objets, classes, attributs et méthodes.

 **1. Classe**

Une **classe** est un **modèle** ou un **plan** qui définit la structure et le comportement d'un objet.

- Elle décrit **ce que l'objet sait** (attributs) et **ce qu'il peut faire** (méthodes).
- C'est une **abstraction**, une sorte de "plan de fabrication".

 **Exemple :**

```
class Voiture:
```

```
    pass
```

Ici, Voiture est une classe. Elle ne fait rien pour l'instant, mais elle pourrait contenir des attributs (comme la couleur) et des méthodes (comme démarrer).

 **2. Objet**

Un **objet** est une **instance concrète d'une classe**. C'est une entité créée à partir d'une classe.

- Chaque objet possède ses **propres valeurs d'attributs**.
- On peut créer plusieurs objets à partir d'une même classe.

👉 Exemple :

```
ma_voiture = Voiture()
```

ma_voiture est un objet (ou une instance) de la classe Voiture.

◆ 3. Attribut

Un **attribut** est une **variable appartenant à un objet ou à une classe**. Il représente un **état ou une propriété**.

- Il est défini dans la méthode `__init__()` lors de l'instanciation de l'objet.

👉 Exemple :

```
class Voiture:
```

```
    def __init__(self, marque, couleur):
        self.marque = marque # attribut
        self.couleur = couleur # attribut
```

marque et couleur sont des attributs de la classe Voiture.

◆ 4. Méthode

Une **méthode** est une **fonction définie dans une classe**. Elle décrit un **comportement** que l'objet peut effectuer.

- Les méthodes agissent souvent sur les attributs de l'objet.
- Elles prennent `self` en premier paramètre (référence à l'objet lui-même).

👉 Exemple :

```
class Voiture:
    def __init__(self, marque, couleur):
        self.marque = marque
        self.couleur = couleur

    def demarrer(self): # méthode
        print(f"La {self.marque} démarre.")
```

⌚ Récap visuel :

```
class Voiture:      # Classe
    def __init__(self, marque, couleur): # Constructeur
        self.marque = marque # Attribut
        self.couleur = couleur # Attribut

    def demarrer(self):    # Méthode
```

```

print(f"La {self.marque} démarre.")

v1 = Voiture("Toyota", "rouge") # Objet
v1.demarrer()                 # Appel de méthode

```

Résumé rapide :

| Concept | Définition |
|----------|--|
| Classe | Modèle d'un objet (plan de construction) |
| Objet | Instance concrète d'une classe |
| Attribut | Donnée ou propriété de l'objet |
| Méthode | Action que l'objet peut effectuer |

- **Premier exemple en Python : création d'une classe simple**

Voici un **premier exemple simple en Python** pour créer une classe, instancier un objet, utiliser des **attributs et méthodes**.

Exemple : Classe Voiture

1. Définition de la classe

```
class Voiture:
    def __init__(self, marque, couleur):      # Constructeur
        self.marque = marque                  # Attribut
        self.couleur = couleur                # Attribut

    def demarrer(self):                      # Méthode
        print(f"La voiture {self.marque} de couleur {self.couleur} démarre.")
```

2. Crédit d'un objet (instance)

```
# Crédit d'une voiture
ma_voiture = Voiture("Toyota", "rouge")
```

3. Utilisation des attributs et méthode

```
# Accès aux attributs
print(ma_voiture.marque)      # Affiche : Toyota
print(ma_voiture.couleur)     # Affiche : rouge

# Appel de la méthode
ma_voiture.demarrer()         # Affiche : La voiture Toyota de couleur rouge démarre.
```

Résumé :

| Élément | Ce que ça fait |
|---------------|--|
| class Voiture | Déclare une nouvelle classe |
| __init__() | Initialise les attributs lors de la création |
| self | Représente l'objet lui-même |
| ma_voiture | Instance de la classe Voiture |
| .demarrer() | Appel d'une méthode de l'objet |

- **TP : Créer une classe Voiture avec quelques attributs et méthodes simples**

Voici un **TP (travail pratique)** guidé pour créer une classe Voiture en Python, avec des **attributs** et **méthodes simples**.

Objectif du TP :

Créer une classe Voiture qui peut :

1. Stocker des informations sur une voiture (marque, modèle, couleur, niveau d'essence)
2. Démarrer la voiture (si elle a de l'essence)
3. Afficher les informations de la voiture
4. Faire le plein

1. Structure de base attendue

Étape 1 : Crée la classe avec un constructeur

class Voiture:

```
def __init__(self, marque, modele, couleur):
    self.marque = marque
    self.modele = modele
    self.couleur = couleur
    self.essence = 0 # En litres
```

Étape 2 : Méthode pour afficher les infos

```
def afficher_infos(self):
    print(f'{self.marque} {self.modele}, couleur {self.couleur}, essence: {self.essence}L')
```

Étape 3 : Méthode pour démarrer

```
def demarrer(self):
    if self.essence > 0:
        print("La voiture démarre 🚗")
    else:
        print("Impossible de démarrer : réservoir vide 🔍")
```

Étape 4 : Méthode pour faire le plein

```
def faire_le_plein(self, litres):
    if litres > 0:
        self.essence += litres
        print(f"You avez ajouté {litres}L d'essence. Total: {self.essence}L")
    else:
        print("Quantité invalide.")
```

 **Exemple d'utilisation complète :**

```
# Création d'un objet Voiture
ma_voiture = Voiture("Peugeot", "208", "bleue")

# Affichage des infos
ma_voiture.afficher_infos()

# Essai de démarrage sans essence
ma_voiture.demarrer()

# Faire le plein
ma_voiture.faire_le_plein(20)

# Nouveau démarrage
ma_voiture.demarrer()

# Affichage final
ma_voiture.afficher_infos()
```

Ce que vous avez appris/pratiqué :

- Création de classe et constructeur `__init__`
- Définition et utilisation d'attributs
- Appel de méthodes d'objets
- Utilisation de conditions

◆ Semaine 2 – Attributs et méthodes d’instance

• `__init__`, `self` et instanciation

Bien sûr ! Voici une **explication claire** des éléments liés à l’initialisation d’un objet en POO avec Python, en mettant l’accent sur les notions de :

`__init__` (le constructeur)

`self`

Attributs d’instance

Méthodes d’instance

Instanciation

1. Méthode spéciale `__init__`

C’est le **constructeur** de la classe. Il est automatiquement appelé **lorsqu’on crée un nouvel objet**.

Exemple :

```
class Voiture:
```

```
    def __init__(self, marque, couleur): # Constructeur
        self.marque = marque      # Attribut d'instance
        self.couleur = couleur
```

- `__init__` initialise les **attributs** de l’objet.

- Il prend toujours `self` en **premier paramètre**, puis les autres valeurs que vous voulez passer à l’objet.

2. Mot-clé `self`

`self` est une **référence à l’objet en cours de création ou de manipulation**.

- Il permet d'**accéder aux attributs et méthodes de l’objet**.
- Ce n’est **pas un mot réservé**, mais c’est une convention fortement respectée en Python.

3. Attributs d’instance

Les **attributs d’instance** sont des **variables propres à chaque objet**.

- Ils sont généralement définis dans `__init__` avec la syntaxe `self.nom_attribut = valeur`.

Exemple :

```
self.marque = marque
self.couleur = couleur
```

Chaque objet aura **ses propres valeurs** pour ces attributs.

4. Méthodes d’instance

Ce sont des **fonctions définies dans la classe**, qui agissent sur l’objet (ses attributs).

- Elles prennent toujours `self` comme premier paramètre.

Exemple :

```
def afficher_infos(self): # Méthode d'instance
    print(f"Marque : {self.marque}, Couleur : {self.couleur}")
```

💡 5. Instanciation d'un objet

Créer une instance d'une classe = **instancier un objet**

Exemple :

```
ma_voiture = Voiture("Renault", "noire")
```

Cela appelle automatiquement `__init__()` et crée un **objet avec ses attributs personnalisés**.

✳️ Exemple complet avec tout ce qui précède :

```
class Voiture:
    def __init__(self, marque, couleur): # __init__ + self + attributs d'instance
        self.marque = marque
        self.couleur = couleur

    def afficher_infos(self):          # méthode d'instance
        print(f"Marque : {self.marque}, Couleur : {self.couleur}")

# Instanciation
v1 = Voiture("Peugeot", "bleue")

# Appel de méthode
v1.afficher_infos() # => Marque : Peugeot, Couleur : bleue
```

✓ Résumé :

| Élément | Définition |
|-----------------------|---|
| <code>__init__</code> | Méthode spéciale appelée à la création de l'objet |
| <code>self</code> | Référence à l'objet courant (permet d'accéder à ses attributs) |
| Attribut | Variable propre à chaque objet (définie avec <code>self.nom</code>) |
| Méthode d'instance | Fonction qui agit sur un objet (prend <code>self</code> comme 1er argument) |
| Instanciation | Création d'un objet à partir d'une classe |

• Portée des variables et encapsulation basique

Voici une explication complète et claire sur les **attributs et méthodes d'instance**, en lien avec la **portée des variables** et l'**encapsulation basique** en Python (POO).

1. Portée des variables (scope)

| Type | Définition |
|----------------|---|
| Locale | Définie à l'intérieur d'une méthode/fonction. Visible uniquement là. |
| D'instance | Définie avec self. Accessible dans toute la classe pour chaque objet. |
| Globale (rare) | Définie en dehors des classes/fonctions. Accessible partout (à éviter souvent). |

Exemple :

```
class Exemple:
    def __init__(self, valeur):
        self.valeur = valeur # Attribut d'instance

    def afficher(self):
        message = "Bonjour" # Variable locale
        print(message, self.valeur)
    • message est local à la méthode afficher()
    • self.valeur est un attribut d'instance accessible partout dans l'objet
```

2. Encapsulation basique

L'**encapsulation** consiste à **cacher les détails internes** d'un objet pour **protéger ses données**.

- En Python, on simule l'**encapsulation** avec une **convention de nommage** :
 - `_nom` : attribut "protégé" (on évite d'y accéder directement)
 - `__nom` : attribut "privé" (non accessible directement de l'extérieur)

Exemple de classe avec encapsulation :

```
class CompteBancaire:
    def __init__(self, titulaire, solde):
        self.titulaire = titulaire # attribut public
        self.__solde = solde # attribut privé (encapsulé)

    def deposer(self, montant):
        if montant > 0:
            self.__solde += montant
```

```

def retirer(self, montant):
    if 0 < montant <= self.__solde:
        self.__solde -= montant
    else:
        print("Fonds insuffisants.")

def afficher_solde(self):
    print(f"Solde de {self.titulaire} : {self.__solde}€")

```

 **Utilisation :**

```

compte = CompteBancaire("Alice", 100)
compte.afficher_solde()    # OK
compte.deposer(50)        # OK
compte.retirer(30)         # OK

# Tentative d'accès direct (non recommandé / échoue si __)
# print(compte.__solde)   # Erreur !

```

 **Résumé des termes :**

| Terme | Définition |
|---------------------|--|
| self.attribut | Attribut d'instance (accessible partout dans l'objet) |
| Variable locale | Définie dans une méthode, visible uniquement à l'intérieur |
| __attribut | Attribut privé (encapsulé), inaccessible directement |
| Méthodes d'instance | Fonctions définies dans la classe, qui utilisent self |

• TP : Classe CompteBancaire, opérations de base (dépôt, retrait)

Voici un **TP complet** pour créer une classe CompteBancaire en Python, avec des **attributs** et **méthodes d'instance** pour effectuer des opérations de base comme **dépôt**, **retrait**, et **affichage du solde**.

Objectif du TP

Créer une classe CompteBancaire qui gère :

- Le titulaire du compte
- Le solde (encapsulé)
- Les opérations : dépôt, retrait, affichage du solde

Structure de la classe

Étape 1 : Définition de la classe et du constructeur

```
class CompteBancaire:
    def __init__(self, titulaire, solde_initial=0):
        self.titulaire = titulaire      # Attribut public
        self.__solde = solde_initial    # Attribut privé (encapsulation)
```

Étape 2 : Méthode pour afficher le solde

```
def afficher_solde(self):
    print(f"Solde de {self.titulaire} : {self.__solde} €")
```

Étape 3 : Méthode pour faire un dépôt

```
def deposer(self, montant):
    if montant > 0:
        self.__solde += montant
        print(f"{montant} € déposés.")
    else:
        print("Montant invalide.")
```

Étape 4 : Méthode pour faire un retrait

```
def retirer(self, montant):
    if montant <= 0:
        print("Montant invalide.")
    elif montant > self.__solde:
        print("Fonds insuffisants.")
    else:
        self.__solde -= montant
        print(f"{montant} € retirés.")
```

 **Exemple d'utilisation :**

```
# Création d'un compte
compte1 = CompteBancaire("Alice", 100)
```

Opérations

```
compte1.afficher_solde() # Affiche : Solde de Alice : 100 €
compte1.deposer(50)      # Affiche : 50 € déposés.
compte1.retirer(30)      # Affiche : 30 € retirés.
compte1.afficher_solde() # Affiche : Solde de Alice : 120 €
compte1.retirer(200)      # Affiche : Fonds insuffisants.
```

 Résumé des éléments utilisés

| Élément | Exemple | Description |
|---------------------|----------------------------------|---------------------------------------|
| Attribut d'instance | self.titulaire, self.__solde | Données propres à chaque compte |
| Méthode d'instance | deposer, retirer, afficher_solde | Actions réalisables par l'objet |
| Encapsulation (__) | __solde | Protège les données sensibles |
| self | Référence à l'objet lui-même | Pour accéder à ses attributs/méthodes |

◆ Semaine 3 – Encapsulation et méthodes spéciales

• Visibilité (publique, protégée, privée)

En Python, la **visibilité** des attributs (publique, protégée, privée) est **basée sur des conventions de nommage**, car Python ne restreint pas strictement l'accès comme certains autres langages (Java, C++). Voici une explication claire avec la **notion d'encapsulation** et les **méthodes spéciales** liées.

1. Encapsulation et visibilité

| Type | Convention Python | Accès |
|----------|-------------------|---|
| Publique | nom | Accessible partout |
| Protégée | _nom | Accessible mais à éviter depuis l'extérieur |
| Privée | __nom | Nom mangling : pas directement accessible |

Exemple complet :

```
class Exemple:
    def __init__(self):
        self.publique = "accessible"
        self._protegee = "à utiliser avec précaution"
        self.__privée = "cachée"
    • self.publique → tout le monde peut y accéder librement.
    • self._protegee → convention : réservé à l'usage interne ou à l'héritage.
    • self.__privée → Python renomme en _NomClasse__privée → protège contre l'accès direct.
```

Accès :

```
obj = Exemple()
```

```
print(obj.publique)    # OK
print(obj._protegee)   # OK mais déconseillé
# print(obj.__privée)  # ❌ Erreur : Attribut inaccessible
```

```
# Mais contournable :
```

```
print(obj._Exemple__privée) # OK (name mangling)
```

2. Méthodes spéciales liées à l'encapsulation

Pour un accès contrôlé aux attributs privés, on utilise des **méthodes spéciales** appelées **accesseurs** (getters) et **mutateurs** (setters).

Exemple avec get et set :

```
class Compte:
    def __init__(self, titulaire, solde):
        self.__titulaire = titulaire
        self.__solde = solde

    # Getter
    def get_solde(self):
        return self.__solde

    # Setter
    def set_solde(self, montant):
        if montant >= 0:
            self.__solde = montant
        else:
            print("Montant invalide")
```

Utilisation :

```
compte = Compte("Alice", 200)

print(compte.get_solde()) # Accès sécurisé
compte.set_solde(300)    # Modification avec contrôle
compte.set_solde(-100)   # Rejetée
```

Bonus : @property et @nom.setter

Python offre une syntaxe élégante pour cela via des **propriétés** :

```
class Produit:
    def __init__(self):
        self.__prix = 0

    @property
    def prix(self):      # Getter
        return self.__prix

    @prix.setter
    def prix(self, valeur): # Setter
        if valeur >= 0:
            self.__prix = valeur
        else:
            print("Prix invalide")
```

Utilisation :

```
p = Produit()
p.prix = 50    # Appelle le setter
print(p.prix)  # Appelle le getter
```

Résumé

| Type | Syntaxe | Accès direct | Contrôle possible | Usage recommandé |
|----------|------------|--------------|---------------------------|---------------------------|
| Publique | self.nom | Oui | Non | Attributs ouverts |
| Protégée | self._nom | Oui | Non | Usage interne ou héritage |
| Privée | self.__nom | Non | Oui (via getters/setters) | Pour données sensibles |

- **Méthodes spéciales : `__str__`, `__repr__`, `__eq__`**

Voici une **explication claire et pratique** des principales **méthodes spéciales** en Python : `__str__`, `__repr__`, et `__eq__`. Ces méthodes permettent de personnaliser le **comportement des objets** dans certains contextes (affichage, comparaison, etc.).

1. `__str__` → Affichage lisible pour l'utilisateur

- Sert à **afficher un objet** de manière **humaine et conviviale**.
- Appelée automatiquement par `print(obj)` ou `str(obj)`.

Exemple :

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def __str__(self):
        return f"Voiture : {self.marque} {self.modele}"
```

Utilisation :

```
v = Voiture("Peugeot", "208")
print(v) # => Voiture : Peugeot 208
```

2. `__repr__` → Représentation technique pour les développeurs

- Sert à fournir une **représentation non ambiguë** de l'objet.
- C'est ce que Python affiche dans l'interpréteur.
- Appelée par `repr(obj)` ou en cas de debug.

Exemple :

```
def __repr__(self):
    return f"Voiture('{self.marque}', '{self.modele}')"
```

💡 Utilisation :

```
repr(v) # => Voiture('Peugeot', '208')
```

⚠ Si `__str__` est absent, `print(v)` utilisera `__repr__` par défaut.

💼 3. `__eq__` → Comparaison d'égalité entre objets

- Permet de définir l'égalité entre deux objets avec `==`.
- Par défaut, `==` compare les identités mémoire (pas les contenus).

✓ Exemple :

```
def __eq__(self, other):
    if isinstance(other, Voiture):
        return self.marque == other.marque and self.modele == other.modele
    return False
```

💡 Utilisation :

```
v1 = Voiture("Peugeot", "208")
v2 = Voiture("Peugeot", "208")
v3 = Voiture("Renault", "Clio")

print(v1 == v2) # True
print(v1 == v3) # False
```

🎯 Résumé des trois méthodes :

| Méthode | But | Appelée par | Utilité |
|-----------------------|---|--|--------------------------|
| <code>__str__</code> | Affichage lisible de l'objet | <code>print(obj)</code> ou <code>str(obj)</code> | Présentation utilisateur |
| <code>__repr__</code> | Représentation "développeur/debug" | <code>repr(obj)</code> | Reconstitution exacte |
| <code>__eq__</code> | Comparer deux objets avec <code>==</code> | <code>obj1 == obj2</code> | Égalité personnalisée |

⌚ Exemple combiné :

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def __str__(self):
        return f"{self.marque} {self.modele}"
```

```

def __repr__(self):
    return f"Voiture('{self.marque}', '{self.modele}')"

def __eq__(self, other):
    return isinstance(other, Voiture) and \
        self.marque == other.marque and \
        self.modele == other.modele

```

- **Propriétés avec @property et setters**

Voici une explication complète et pratique sur l'utilisation des **propriétés avec @property** et les **setters** en Python, pour un **encapsulation propre et contrôlée**.

 **Objectif :**

Utiliser `@property` pour **accéder à des attributs privés** tout en **contrôlant la lecture et l'écriture** via des méthodes (getters et setters).

 **Pourquoi utiliser @property ?**

- Masquer l'accès direct aux **attributs privés** (`__attribut`)
- Ajouter de la **logique de validation** sans changer la syntaxe d'accès
- Faciliter la **maintenabilité** du code

 **Syntaxe de base :**

```

class Produit:
    def __init__(self, nom, prix):
        self.__nom = nom
        self.__prix = prix # Attribut privé

    @property
    def prix(self):      # Getter
        return self.__prix

    @prix.setter
    def prix(self, valeur): # Setter
        if valeur >= 0:
            self.__prix = valeur
        else:
            print("Erreur : le prix doit être positif.")

```

 **Utilisation :**

```
p = Produit("Chaise", 50)
```

```
print(p.prix) # Appelle automatiquement le getter → 50
```

```
p.prix = 70    # Appelle automatiquement le setter → modifie __prix
p.prix = -10   # Affiche erreur → prix reste inchangé
```

★ **Avantage** : vous utilisez p.prix comme si c'était un attribut public, mais **en coulisse** c'est une méthode avec vérification.

⚙ Définir aussi un getter pour un autre attribut :

```
@property
def nom(self):
    return self.__nom
```

Vous pouvez également rendre une propriété en lecture seule :

```
class Article:
    def __init__(self, reference):
        self.__reference = reference

    @property
    def reference(self):
        return self.__reference # Pas de setter ⇒ lecture seule
```

✓ Résumé tableau :

| Élément | Syntaxe | But |
|---------------|--------------------------------|-------------------------------|
| Getter | @property | Lire un attribut privé |
| Setter | @attribut.setter | Modifier avec contrôle |
| Lecture seule | Juste @property, pas de setter | Empêcher modification externe |

✳ Exemple complet :

```
class Compte:
    def __init__(self, titulaire, solde):
        self.__titulaire = titulaire
        self.__solde = solde

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

    @solde.setter
    def solde(self, valeur):
        if valeur >= 0:
            self.__solde = valeur
```

```

else:
    print("Solde invalide")

@property
def titulaire(self):
    return self.__titulaire # lecture seule

```

Utilisation :

```

c = Compte("Alice", 100)

print(c.solde)  # 100

c.solde = 200  # OK

c.solde = -50  # Erreur : Solde invalide

print(c.titulaire) # Alice

# c.titulaire = "Bob" ✗ Pas autorisé (pas de setter)

```

- **TP : Gestion de classe Personne avec validation des attributs**

Voici un **TP complet** pour créer une **classe Personne** en Python, avec des **attributs privés**, des **propriétés (@property)**, des **setters** et de la **validation**.

🎯 Objectif du TP

Créer une classe Personne avec les caractéristiques suivantes :

- Attributs privés : nom, age
- Validations :
 - nom ne doit pas être vide
 - age doit être un entier positif
- Accès sécurisé via propriétés (@property, @setter)
- Affichage lisible de l'objet

🚧 Étapes de construction

Étape 1 : Définir la classe avec les attributs privés

```

class Personne:
    def __init__(self, nom, age):
        self.nom = nom # Utilise le setter
        self.age = age # Utilise le setter

```

Étape 2 : Ajouter les propriétés avec validation

► nom (obligatoire, non vide)

@property

```

def nom(self):
    return self.__nom

@nom.setter
def nom(self, valeur):
    if isinstance(valeur, str) and valeur.strip():
        self.__nom = valeur.strip()
    else:
        raise ValueError("Le nom ne peut pas être vide.")

```

► age (entier ≥ 0)

```

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

@age.setter
def age(self, valeur):
    if isinstance(valeur, int) and valeur >= 0:
        self.__age = valeur
    else:
        raise ValueError("L'âge doit être un entier positif.")

```

Étape 3 : Méthode spéciale `__str__` pour affichage

```

def __str__(self):
    return f"{self.nom}, {self.age} ans"

```

Exemple d'utilisation

```

# Création d'une personne valide
p1 = Personne("Alice", 30)
print(p1) # => Alice, 30 ans

# Mise à jour des attributs
p1.age = 35
p1.nom = "Alice Dupont"
print(p1)

# Cas avec erreur
try:
    p2 = Personne("", -5)
except ValueError as e:
    print("Erreur :", e)

```

Résumé technique

| Élément | Fonction |
|---------------------------|---|
| <code>__nom, __age</code> | Attributs privés (encapsulation) |
| <code>@property</code> | Permet d'accéder à nom et age comme attributs |

| Élément | Fonction |
|------------------------|---|
| <code>@setter</code> | Permet de valider et modifier les valeurs |
| <code>__str__()</code> | Permet un affichage lisible de l'objet |

💡 **Bonus possible :**

- Ajouter un **attribut email** avec validation de format.
- Ajouter une **méthode est_majeur()** qui retourne True si age ≥ 18 .

◆ Semaine 4 – Méthodes et attributs de classe

- Différence entre méthode d'instance, méthode de classe (`@classmethod`) et méthode statique (`@staticmethod`)

Voici une **explication claire et complète** sur les **méthodes et attributs de classe** en Python, avec la différence entre :

- Méthode d'instance
- Méthode de classe (`@classmethod`)
- Méthode statique (`@staticmethod`)

1. Méthode d'instance

Définition :

- Elle agit sur **une instance spécifique** de la classe.
- Elle utilise `self` pour accéder aux **attributs de l'objet**.

Exemple :

```
class Voiture:
    def __init__(self, marque):
        self.marque = marque # Attribut d'instance

    def afficher(self):    # Méthode d'instance
        print(f"Marque : {self.marque}")
```

Utilisation :

```
v = Voiture("Toyota")
v.afficher() # Affiche : Marque : Toyota
```

2. Méthode de classe (`@classmethod`)

Définition :

- Elle agit sur la **classe elle-même**, pas une instance.
- Elle utilise `cls` pour accéder aux **attributs de classe**.
- Utile pour des **constructeurs alternatifs** ou des **paramètres globaux partagés**.

Exemple :

```
class Voiture:
    total_voitures = 0 # Attribut de classe
```

```

def __init__(self, marque):
    self.marque = marque
    Voiture.total_voitures += 1

@classmethod
def afficher_total(cls): # Méthode de classe
    print(f"Nombre de voitures : {cls.total_voitures}")

```

 Utilisation :

```
Voiture.afficher_total()
```

3. Méthode statique (@staticmethod)

 Définition :

- Indépendante de la classe et de l'instance
- N'utilise ni **self** ni **cls**
- C'est simplement une **fonction utilitaire** liée logiquement à la classe

 Exemple :

```

class Voiture:
    @staticmethod
    def convertir_km_miles(km):
        return km * 0.621371

```

 Utilisation :

```

miles = Voiture.convertir_km_miles(100)
print(miles) # 62.1371

```

 Récapitulatif tableau

| Type | Décorateur | Premier paramètre | Accès à | Quand l'utiliser ? |
|--------------------|---------------|-------------------|-----------------------|--|
| Méthode d'instance | (aucun) | self | Attributs de l'objet | Comportement propre à chaque objet |
| Méthode de classe | @classmethod | cls | Attributs de classe | Suivi global, création d'objets, statistiques... |
| Méthode statique | @staticmethod | (aucun) | Aucun (fonction pure) | Outils auxiliaires liés au domaine de la classe |

 Exemple combiné :

```

class Personne:
    nb_personnes = 0 # Attribut de classe

    def __init__(self, nom):
        self.nom = nom

```

```

Personne.nb_personnes += 1

def se_presenter(self):      # Méthode d'instance
    print(f"Bonjour, je m'appelle {self.nom}.") 

@classmethod
def combien(cls):           # Méthode de classe
    print(f"{cls.nb_personnes} personnes créées.")

@staticmethod
def est_majeur(age):        # Méthode statique
    return age >= 18

```

+ Utilisation :

```

p1 = Personne("Alice")
p2 = Personne("Bob")

p1.se_presenter()      # Bonjour, je m'appelle Alice.
Personne.combien()     # 2 personnes créées.
print(Personne.est_majeur(20)) # True

```

- **Attributs de classe vs d'instance**

Voici une **explication claire** de la **différence entre les attributs de classe et les attributs d'instance**, avec exemples et tableau récapitulatif.

1. Attribut d'instance

Définition :

- Propre à **chaque objet créé** à partir de la classe.
- Déclaré généralement dans la méthode `__init__`.
- Préfixé par `self`.

Exemple :

```

class Voiture:
    def __init__(self, marque):
        self.marque = marque # Attribut d'instance

```

+ Utilisation :

```

v1 = Voiture("Renault")
v2 = Voiture("Peugeot")

print(v1.marque) # Renault
print(v2.marque) # Peugeot

```

Chaque objet a **sa propre copie** de l'attribut marque.

💡 2. Attribut de classe

🎯 Définition :

- Partagé par **toutes les instances** de la classe.
- Déclaré **en dehors des méthodes**, directement dans la classe.
- Préfixé par le **nom de la classe** ou par `cls` dans une méthode de classe.

✅ Exemple :

```
class Voiture:
    total_voitures = 0 # Attribut de classe

    def __init__(self, marque):
        self.marque = marque
        Voiture.total_voitures += 1 # On modifie l'attribut de classe
```

➕ Utilisation :

```
v1 = Voiture("Renault")
v2 = Voiture("Peugeot")

print(Voiture.total_voitures) # 2 (partagé)
```

Tous les objets **partagent la même valeur** de `total_voitures`.

📊 Tableau récapitulatif

| Caractéristique | Attribut d'instance | Attribut de classe |
|----------------------------|--|---|
| Déclaration | Dans <code>__init__</code> avec <code>self.attribut</code> | Directement dans la classe |
| Portée | Propre à chaque objet | Partagé par toutes les instances |
| Modification depuis objet | Affecte uniquement cet objet | Peut créer un attribut d'instance masqué |
| Modification depuis classe | N'a aucun effet | Change pour toutes les instances |
| Exemple d'usage | <code>self.nom = "Alice"</code> | <code>Personne.nb_personnes = 5</code> |

⚠️ Attention : masquage

Si on modifie un **attribut de classe** via une **instance**, cela crée un **attribut d'instance du même nom**, sans toucher à l'attribut de classe.

```
class Exemple:
    compteur = 0

    e = Exemple()
    e.compteur = 10 # crée un attribut d'instance "compteur"
    print(e.compteur) # 10 (attribut d'instance)
```

```
print(Exemple.compteur) # 0 (toujours l'attribut de classe)
```

Exemple combiné

```
class Employe:  
    nb_employes = 0 # Attribut de classe  
  
    def __init__(self, nom):  
        self.nom = nom # Attribut d'instance  
        Employe.nb_employes += 1  
    e1 = Employe("Alice")  
    e2 = Employe("Bob")  
  
    print(e1.nom)      # Alice  
    print(Employe.nb_employes) # 2
```

- **TP : Gestion d'un compteur d'instances avec un attribut de classe**

Voici un **TP guidé** pour apprendre à gérer un **compteur d'instances** à l'aide d'un **attribut de classe**. Ce TP est très utile pour bien distinguer les **attributs de classe** des **attributs d'instance**.

TP : Compter les objets créés

Objectif :

Créer une classe Personne qui :

- Possède un **attribut de classe nb_personnes** pour compter le nombre d'instances créées.
- Incrémente ce compteur à chaque fois qu'une **nouvelle instance est créée**.
- Permet d'afficher ce compteur via une **méthode de classe**.

Étapes à suivre

Étape 1 : Déclaration de la classe et des attributs

```
class Personne:  
    nb_personnes = 0 #  Attribut de classe (compteur)  
  
    def __init__(self, nom):  
        self.nom = nom      #  Attribut d'instance  
        Personne.nb_personnes += 1 #  Incrémentation du compteur
```

Étape 2 : Ajouter une méthode de classe pour afficher le compteur

`@classmethod`

```
def afficher_nombre(cls):
    print(f"Nombre total de personnes : {cls.nb_personnes}")
```

 **Étape 3 : Test dans un script**

```
p1 = Personne("Alice")
p2 = Personne("Bob")
p3 = Personne("Charlie")
```

```
Personne.afficher_nombre() # Résultat attendu : 3
```

 **Bonus (optionnel) : Décrémenter le compteur à la suppression d'un objet**

Ajouter une méthode spéciale `__del__` :

```
def __del__(self):
    Personne.nb_personnes -= 1
```

 Note : `__del__` est **appelée automatiquement** lors de la **destruction de l'objet** par le ramasse-miettes (non garanti immédiatement).

 **Résumé des éléments utilisés**

| Élément | Utilité |
|---------------------------|---|
| <code>nb_personnes</code> | Attribut de classe → partagé par tous |
| <code>self.nom</code> | Attribut d'instance → propre à chaque objet |
| <code>@classmethod</code> | Accès/modification à un attribut de classe |
| <code>cls</code> | Référence à la classe dans une méthode |

 **Test complet :**

```
p1 = Personne("Alice")
p2 = Personne("Bob")
```

```
print(p1.nom)
print(p2.nom)
```

```
Personne.afficher_nombre() # => Nombre total de personnes : 2
```

Ajout à ce TP :

- un **identifiant unique auto-incrémentation** par personne ?

Parfait ! Voici comment modifier la classe `Personne` pour ajouter un **identifiant unique auto-incrémentation** à chaque instance.

 **Objectif :**

Chaque objet `Personne` reçoit un **ID unique**, automatiquement attribué à sa création, en plus du compteur total des instances.

Étapes :

Étape 1 : Ajout d'un attribut de classe pour gérer l'ID unique

```
class Personne:
    nb_personnes = 0           # Compteur total d'instances
    dernier_id = 0             # Attribut de classe pour générer des IDs uniques

    def __init__(self, nom):
        self.nom = nom
        Personne.nb_personnes += 1

        Personne.dernier_id += 1      # Incrémentation de l'ID global
        self.id = Personne.dernier_id # Affectation de l'ID unique à l'instance
```

Étape 2 : Méthode de classe pour afficher le nombre total

```
@classmethod
def afficher_nombre(cls):
    print(f"Nombre total de personnes : {cls.nb_personnes}")
```

Étape 3 : Méthode d'instance pour se présenter

```
def se_presenter(self):
    print(f"[ID: {self.id}] Bonjour, je m'appelle {self.nom}.")
```

Exemple d'utilisation :

```
p1 = Personne("Alice")
p2 = Personne("Bob")
p3 = Personne("Charlie")

p1.se_presenter()    # [ID: 1] Bonjour, je m'appelle Alice.
p2.se_presenter()    # [ID: 2] Bonjour, je m'appelle Bob.
p3.se_presenter()    # [ID: 3] Bonjour, je m'appelle Charlie.

Personne.afficher_nombre() # Nombre total de personnes : 3
```

Résumé

| Nom | Type | But |
|--------------|--------------------|------------------------------|
| nb_personnes | Attribut de classe | Nombre total d'objets créés |
| dernier_id | Attribut de classe | Dernier identifiant attribué |

| Nom | Type | But |
|----------|---------------------|---------------------------------------|
| id | Attribut d'instance | Identifiant unique de chaque personne |
| self.nom | Attribut d'instance | Nom de la personne |

- un **suivi des noms de toutes les personnes créées** (liste partagée) ?

Très bonne idée ! Voici comment modifier la classe Personne pour maintenir un **suivi de tous les noms créés**, via une **liste partagée** entre les instances (grâce à un **attribut de classe**).

🎯 Objectif :

Ajouter une **liste partagée** qui enregistre **le nom de chaque personne créée**.

🚧 Mise à jour de la classe Personne :

```
class Personne:
    nb_personnes = 0          # Compteur total
    dernier_id = 0            # ID auto-incrémenté
    liste_noms = []           # 📋 Liste partagée des noms

    def __init__(self, nom):
        self.nom = nom

        # ID unique auto-incrémenté
        Personne.dernier_id += 1
        self.id = Personne.dernier_id

        # Compteur d'instances
        Personne.nb_personnes += 1

        # Ajout du nom dans la liste partagée
        Personne.liste_noms.append(nom)

    def se_presenter(self):
        print(f"[ID: {self.id}] Bonjour, je m'appelle {self.nom}.")

    @classmethod
    def afficher_nombre(cls):
        print(f"Nombre total de personnes : {cls.nb_personnes}")

    @classmethod
    def afficher_noms(cls):
        print("Liste des personnes créées :")
        for nom in cls.liste_noms:
            print(f"- {nom}")
```

✓ Exemple d'utilisation :

```
p1 = Personne("Alice")
p2 = Personne("Bob")
p3 = Personne("Charlie")

p1.se_presenter()
p2.se_presenter()
p3.se_presenter()

Personne.afficher_nombre()
Personne.afficher_noms()
```

Résultat attendu :

[ID: 1] Bonjour, je m'appelle Alice.
 [ID: 2] Bonjour, je m'appelle Bob.
 [ID: 3] Bonjour, je m'appelle Charlie.
Nombre total de personnes : 3
Liste des personnes créées :
 - Alice
 - Bob
 - Charlie

 Résumé des nouveaux éléments :

| Élément | Type | Rôle |
|-----------------|--------------------|-------------------------------------|
| liste_noms | Attribut de classe | Liste partagée des noms enregistrés |
| append(nom) | Méthode | Ajoute chaque nom à la création |
| afficher_noms() | Méthode de classe | Affiche la liste complète des noms |

◆ Semaine 5 – Héritage

• Héritage simple

Voici une introduction simple à l'**héritage en Python**, un concept clé de la programmation orientée objet (POO).

Héritage simple : qu'est-ce que c'est ?

- Permet de créer une **classe enfant** (ou sous-classe) à partir d'une **classe parent** (ou super-classe).
- La sous-classe **hérite des attributs et méthodes** de la super-classe.
- Permet de **réutiliser le code** et de **spécialiser** des comportements.

Syntaxe de base

```
class Animal:
    def parler(self):
        print("L'animal fait un bruit.")

class Chien(Animal): # Chien hérite d'Animal
    def parler(self):
        print("Le chien aboie.")
```

Exemple complet

```
# Classe parent
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def se_presenter(self):
        print(f"Je suis un animal nommé {self.nom}.")

    def parler(self):
        print("L'animal fait un bruit.")

# Classe enfant
class Chien(Animal):
    def parler(self):
        print("Le chien aboie.")

# Utilisation
a = Animal("Inconnu")
a.se_presenter() # Je suis un animal nommé Inconnu.
a.parler()       # L'animal fait un bruit.

c = Chien("Rex")
c.se_presenter() # Hérité de Animal
c.parler()       # Méthode redéfinie dans Chien
```

Explications clés

| Concept | Description |
|-------------------------|---|
| Classe parent | Classe dont hérite une autre classe |
| Classe enfant | Sous-classe qui hérite et peut spécialiser |
| Redéfinition (override) | La sous-classe peut modifier des méthodes |
| super() | Permet d'appeler une méthode de la super-classe |

Bonus : utiliser super() pour étendre le comportement

```
class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(nom) # Appelle le __init__ de Animal
        self.race = race

    def parler(self):
        super().parler() # Appelle la méthode parler de Animal
        print("Le chien aboie.")
```

Exemple d'utilisation du bonus

```
c = Chien("Rex", "Berger Allemand")

c.se_presenter() # Hérité de Animal

c.parler() # Affiche d'abord "L'animal fait un bruit." puis "Le chien aboie."

print(c.race) # Berger Allemand
```

• Redéfinition de méthodes (overriding)

Voici une explication claire sur la **redéfinition de méthodes (overriding)** en programmation orientée objet, avec exemples en Python.

Redéfinition de méthodes (Overriding)

Qu'est-ce que c'est ?

- C'est lorsque **une classe enfant (sous-classe)** redéfinit **une méthode** héritée de la classe parent.
- La méthode dans la sous-classe **remplace** celle de la classe parent pour les objets de cette sous-classe.
- Cela permet de **spécialiser ou modifier** le comportement hérité.

Exemple simple

```

class Animal:
    def parler(self):
        print("L'animal fait un bruit.")

class Chien(Animal):
    def parler(self): # Redéfinition (override)
        print("Le chien aboie.")
a = Animal()
a.parler() # Affiche : L'animal fait un bruit.

c = Chien()
c.parler() # Affiche : Le chien aboie.

```

Utiliser super() pour étendre la méthode parent

Souvent on veut **appeler la méthode parent** puis **ajouter un comportement** spécifique dans la sous-classe.

```

class Chien(Animal):
    def parler(self):
        super().parler() # Appelle parler() de la classe parent
        print("Le chien aboie.")
c = Chien()
c.parler()
# Résultat :
# L'animal fait un bruit.
# Le chien aboie.

```

Résumé

| Aspect | Détails |
|---------------|--|
| Override | Redéfinition d'une méthode héritée |
| But | Adapter/modifier le comportement |
| super() | Permet d'appeler la méthode de la super-classe |
| Sans override | Méthode de la classe parent est utilisée |

• Appel à super()

Voici une explication claire sur l'**appel à super()** en Python, un concept essentiel pour gérer l'héritage et la redéfinition de méthodes.

💡 super() : qu'est-ce que c'est ?

- super() permet d'**appeler une méthode (ou constructeur) de la classe parente depuis une sous-classe**.
- Utile pour **réutiliser** le comportement de la super-classe tout en l'**étendant** ou le modifiant.
- Évite d'avoir à nommer explicitement la super-classe, ce qui rend le code plus flexible (utile notamment dans l'héritage multiple).

🛠️ Syntaxe et usage

Exemple classique avec constructeur (`__init__`)

```
class Animal:
    def __init__(self, nom):
        self.nom = nom
        print(f"Animal créé : {self.nom}")

class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(nom) # Appel du __init__ de Animal
        self.race = race
        print(f"Chien créé : {self.nom}, race : {self.race}")
```

Utilisation

```
c = Chien("Rex", "Berger Allemand")
```

Sortie :

```
Animal créé : Rex
Chien créé : Rex, race : Berger Allemand
```

✳️ Exemple avec méthode redéfinie

```
class Animal:
    def parler(self):
        print("L'animal fait un bruit.")

class Chien(Animal):
    def parler(self):
        super().parler() # Appel de parler() de la super-classe
        print("Le chien aboie.")
c = Chien()
c.parler()
```

Sortie :

```
L'animal fait un bruit.
```

```
Le chien aboie.
```


Pourquoi utiliser super() ?

- Permet d'éviter de réécrire du code déjà défini dans la super-classe.
- Assure que la chaîne d'héritage est respectée, utile surtout dans l'héritage multiple.
- Facilite la maintenance et l'évolution du code.

Résumé rapide

| Cas d'usage | Description |
|-------------------|--|
| Constructeur | Appeler le <code>__init__</code> de la classe parente |
| Méthode redéfinie | Appeler la version parent avant d'ajouter du code |
| Héritage multiple | Gérer proprement l'appel aux méthodes des classes parentes |

• TP : Hiérarchie de classes Employé, Manager, Développeur

Super, voici un TP complet pour travailler sur une **hiérarchie de classes** avec Employé, Manager et Developpeur en Python, en utilisant l'héritage simple, la redéfinition de méthodes et `super()`.

💡 TP : Hiérarchie de classes Employé, Manager, Developpeur

Objectifs

- Créer une classe de base Employé.
- Créer deux classes filles Manager et Developpeur qui héritent de Employé.
- Redéfinir des méthodes dans les classes filles.
- Utiliser `super()` pour réutiliser le code parent.
- Ajouter des attributs et méthodes spécifiques à chaque classe.

Étape 1 : Classe Employé

```
class Employe:
    def __init__(self, nom, salaire):
        self.nom = nom
        self.salaire = salaire

    def afficher_info(self):
        print(f"Employé : {self.nom}, salaire : {self.salaire}€")
```

```
def travailler(self):
    print(f"{self.nom} travaille.")
```

Étape 2 : Classe Manager (hérite d'Employe)

```
class Manager(Employe):
    def __init__(self, nom, salaire, equipe=None):
        super().__init__(nom, salaire)
        if equipe is None:
            equipe = []
        self.equipe = equipe # liste des employés sous sa responsabilité

    def afficher_info(self):
        super().afficher_info()
        print(f"Manager de l'équipe : {[emp.nom for emp in self.equipe]}")

    def travailler(self):
        print(f"{self.nom} gère son équipe.")

    def ajouter_employe(self, employe):
        self.equipe.append(employe)
        print(f"{employe.nom} a été ajouté à l'équipe de {self.nom}.")
```

Étape 3 : Classe Developpeur (hérite d'Employe)

```
class Developpeur(Employe):
    def __init__(self, nom, salaire, langage):
        super().__init__(nom, salaire)
        self.langage = langage # langage de programmation préféré

    def afficher_info(self):
        super().afficher_info()
        print(f"Developpeur spécialisé en {self.langage}.")"

    def travailler(self):
        print(f"{self.nom} écrit du code en {self.langage}.")
```

Étape 4 : Test complet

```
# Création des employés
dev1 = Developpeur("Alice", 3500, "Python")
dev2 = Developpeur("Bob", 3200, "JavaScript")
mgr = Manager("Claire", 5000)

# Ajouter des développeurs à l'équipe du manager
mgr.ajouter_employe(dev1)
mgr.ajouter_employe(dev2)

# Afficher infos
```

```
dev1.afficher_info()  
dev2.afficher_info()  
mgr.afficher_info()  
  
# Travailler  
dev1.travailler()  
dev2.travailler()  
mgr.travailler()
```

Résultat attendu

*Alice travaille en Python.
Bob travaille en JavaScript.
Claire gère son équipe.*

*Employé : Alice, salaire : 3500€
Développeur spécialisé en Python.
Employé : Bob, salaire : 3200€
Développeur spécialisé en JavaScript.
Employé : Claire, salaire : 5000€
Manager de l'équipe : ['Alice', 'Bob']
Alice écrit du code en Python.
Bob écrit du code en JavaScript.
Claire gère son équipe.*

◆ Semaine 6 – Polymorphisme

Le **polymorphisme** est un concept fondamental en **programmation orientée objet** (POO), qui désigne la capacité d'un **même objet** ou **d'une même méthode** à se comporter de différentes manières selon le **contexte**.

Définition simple :

Le polymorphisme permet d'utiliser **une même interface** pour des **types différents**.

Types de polymorphisme :

1. Polymorphisme de sous-typage (ou inclusion)

- Permet de traiter des objets de **classes dérivées** comme s'ils étaient des objets de leur **classe de base**.
- Exemple : une méthode qui accepte un paramètre de type Animal pourra accepter un Chien, un Chat, etc., si ces classes héritent de Animal.

2. Polymorphisme paramétrique (ou générique)

- Permet d'écrire du code indépendant du **type** des données.
- *Exemple en Java :*
- ```
public class Boîte<T> {
```
- ```
    private T contenu;
```
- ```
 public void setContenu(T contenu) { this.contenu = contenu; }
```
- ```
    public T getContenu() { return contenu; }
```
- ```
}
```

### 3. Polymorphisme ad hoc (surcharge et redéfinition)

- **Surcharge** : plusieurs méthodes dans une même classe avec **le même nom** mais **des paramètres différents**.
- **Redéfinition (override)** : une sous-classe redéfinit une méthode de la classe parente avec **le même nom et la même signature**.

**Exemple en Python :**

```
class Animal:
 def parler(self):
 pass
```

```
class Chien(Animal):
 def parler(self):
 return "Wouf"
```

```
class Chat(Animal):
 def parler(self):
 return "Miaou"
```

```
def faire_parler(animal):
 print(animal.parler())

faire_parler(Chien()) # Wouf
faire_parler(Chat()) # Miaou
```

💡 Conclusion : Le polymorphisme améliore la **flexibilité** et la **réutilisabilité** du code, en permettant à des objets différents de répondre à une **même interface** de façon **adaptée à leur nature**.

- **Principe du polymorphisme**

- ◆ **Principe du polymorphisme**

Le **principe du polymorphisme** en programmation orientée objet repose sur l'**unification de comportements** via des **interfaces communes**, tout en permettant à chaque objet de **réagir différemment** selon son **type réel**.

- ✓ **Principe fondamental :**

“Une même action peut être effectuée de différentes manières selon le type de l’objet qui l’effectue.”

- 🎯 **Objectif :**

Permettre d’écrire du code **plus générique, flexible et extensible**, sans se soucier des types exacts des objets manipulés.

- 📌 **Fonctionnement général :**

1. **Définir une interface ou une classe de base** avec des méthodes génériques.
2. **Créer des classes dérivées** qui redéfinissent ces méthodes selon leur comportement spécifique.
3. **Utiliser un objet de la classe de base** (ou interface) pour manipuler tous les objets dérivés **de manière uniforme**.

⌚ **Exemple illustratif (en pseudo-code) :**

*Classe Animal:  
méthode parler()*

*Classe Chien hérite de Animal:  
méthode parler() → "Wouf"*

*Classe Chat hérite de Animal:  
méthode parler() → "Miaou"*

*Procédure faireParler(Animal a):  
afficher a.parler()*

*faireParler(Chien) → "Wouf"  
faireParler(Chat) → "Miaou"*

 Ce que cela implique :

- On programme sur le **type parent**, mais l'exécution dépend du **type enfant réel**.
- C'est le mécanisme de **liaison dynamique** ou **liaison tardive** (résolution de la méthode à l'exécution).

 Avantage :

- Réduction des **dépendances** au type spécifique.
- Facilite la **maintenabilité** et l'**évolution** du code.

## Un exemple dans le langage précis comme Python ?

Bien sûr ! Voici un **exemple simple et clair de polymorphisme en Python**.

 Exemple en Python :

```

class Animal:
 def parler(self):
 raise NotImplementedError("Cette méthode doit être redéfinie dans la sous-classe")

class Chien(Animal):
 def parler(self):
 return "Wouf"

class Chat(Animal):
 def parler(self):
 return "Miaou"

class Vache(Animal):
 def parler(self):
 return "Meuh"

def faire_parler(animal):
 print(animal.parler())

Utilisation
liste_animaux = [Chien(), Chat(), Vache()]

for animal in liste_animaux:
 faire_parler(animal)

```

 **Sortie du programme :**

```
Wouf
Miaou
Meuh
```

 **Explication :**

- La fonction faire\_parler prend un objet de **type Animal** (ou d'une classe qui en hérite).
- Chaque objet (Chien, Chat, Vache) **redéfinit** la méthode parler() selon son propre comportement.
- Lorsqu'on appelle animal.parler(), Python utilise **la méthode de la classe réelle** de l'objet, **même si on le manipule via une variable de type Animal** → c'est ça, le **polymorphisme**.

• **Surcharge des méthodes avec comportements différents**

Très bien ! Parlons maintenant de la **surcharge de méthodes** (overloading) avec **comportements différents**.

 **Définition :**

La **surcharge** signifie définir **plusieurs méthodes avec le même nom**, mais avec **des paramètres différents** (en nombre, type ou ordre).

Chaque méthode exécutera un **comportement différent** selon les paramètres.

 **En Python :**

Contrairement à Java ou C++, **Python ne supporte pas la surcharge de manière native** via la signature des fonctions.

Mais on peut l'**imiter** en utilisant :

- Des **valeurs par défaut**
- Le module `functools.singledispatch`
- Des conditions dans la méthode

 **Exemple 1 : Avec des paramètres optionnels**

```
class Calculatrice:
 def addition(self, a, b=0, c=0):
 if c != 0:
 return a + b + c
 else:
 return a + b

 # Utilisation
 calc = Calculatrice()
 print(calc.addition(2, 3)) # 5
```

```
print(calc.addition(2, 3, 4)) # 9
```

- 💡 Ici, même nom de méthode (**addition**), mais comportement différent selon le nombre d'arguments.

✓ Exemple 2 : Avec `functools.singledispatch`

```
from functools import singledispatchmethod
```

```
class Afficheur:
 @singledispatchmethod
 def afficher(self, arg):
 print("Type non pris en charge")

 @afficher.register
 def _(self, arg: int):
 print(f"Nombre entier : {arg}")

 @afficher.register
 def _(self, arg: str):
 print(f"Chaîne de caractères : {arg}")

Utilisation
a = Afficheur()
a.afficher(42) # Nombre entier : 42
a.afficher("Salut") # Chaîne de caractères : Salut
a.afficher(3.14) # Type non pris en charge
```

🔍 Résumé :

| Langage    | Surcharge native ? | Exemple                                         |
|------------|--------------------|-------------------------------------------------|
| Python     | ✗ Pas directement  | Contournement avec conditions ou singledispatch |
| Java / C++ | ✓ Oui              | Par type/nb d'arguments                         |

- Utilisation de fonctions génériques

💡 Utilisation de fonctions génériques en Python

Les **fonctions génériques** permettent d'écrire du **code réutilisable** qui fonctionne avec **differents types de données**, tout en **conservant la sécurité des types** grâce à l'utilisation de **types génériques**.

📦 Module utilisé : `typing`

Python permet d'écrire des **fonctions génériques** à l'aide du module `typing`, notamment avec :

- TypeVar : pour déclarer un **type générique**

- Generic : pour créer des **classes génériques**
- 

 **Exemple simple : fonction générique d'inversion**

```
from typing import TypeVar

T = TypeVar('T') # Déclaration d'un type générique

def inverser(a: T, b: T) -> tuple[T, T]:
 return b, a
```

# Utilisation

```
print(inverser(1, 2)) # (2, 1)
print(inverser("chat", "chien")) # ('chien', 'chat')
```

 Ici, inverser accepte **n'importe quel type**, tant que les deux arguments sont du **même type** (int, str, etc.).

 **Exemple avec une classe générique**

```
from typing import Generic, TypeVar
```

```
T = TypeVar('T')
```

```
class Boîte(Generic[T]):
 def __init__(self, contenu: T):
 self.contenu = contenu

 def afficher(self) -> None:
 print(f"Contenu : {self.contenu}")

Utilisation
boîte_int = Boîte(42)
boîte_str = Boîte("Bonjour")

boîte_int.afficher() # Contenu : 42
boîte_str.afficher() # Contenu : Bonjour
```

 **Pourquoi utiliser des fonctions génériques ?**

| Avantage                       | Détail                                         |
|--------------------------------|------------------------------------------------|
| Réutilisabilité                | Éviter de dupliquer du code pour chaque type   |
| Lisibilité                     | Le comportement est défini une seule fois      |
| Sécurité des types (avec mypy) | Détection d'erreurs lors de l'analyse statique |

Un exemple **plus avancé**, comme une fonction générique avec plusieurs types (`TypeVar('T', int, float)`), ou bien l'utilisation avec des **collections génériques** comme `List[T]`

Parfait ! Voici des **exemples plus avancés de fonctions génériques** en Python utilisant :

1.  TypeVar avec **types contraints** (`int, float`)
2.  **Collections génériques** comme `List[T]`

### 💡 1. Fonction générique avec `TypeVar('T', int, float)`

```
from typing import TypeVar
```

```
T = TypeVar('T', int, float) # Limité à int et float
```

```
def moyenne(a: T, b: T) -> float:
```

```
 return (a + b) / 2
```

```
Utilisation
```

```
print(moyenne(10, 20)) # 15.0
```

```
print(moyenne(10.5, 20.5)) # 15.5
```

Ici, la fonction moyenne accepte **soit deux int, soit deux float**, et retourne un float.

Elle n'acceptera **pas une chaîne de caractères** ou tout autre type, ce qui améliore la **sécurité des types** (utile avec des outils comme mypy).

### 💡 2. Fonction générique avec `List[T]`

```
from typing import TypeVar, List
```

```
T = TypeVar('T')
```

```
def premier_element(liste: List[T]) -> T | None:
 if liste:
```

```

 return liste[0]
 return None

Utilisation
print(premier_element([1, 2, 3])) # 1
print(premier_element(["a", "b", "c"])) # 'a'
print(premier_element([])) # None

```

- Ici, la fonction `premier_element` est **générique** : elle retourne le **premier élément de n'importe quel type de liste** (int, str, float, objets, etc.).

### 💡 Astuce : Fonction avec plusieurs TypeVar

On peut même aller plus loin :

```

from typing import TypeVar

T = TypeVar('T')
U = TypeVar('U')

def pair(val1: T, val2: U) -> tuple[T, U]:
 return (val1, val2)

Utilisation
print(pair(1, "chat")) # (1, 'chat')
print(pair("clé", 3.14)) # ('clé', 3.14)

```

### 🔍 Résumé

| Élément           | But                                    |
|-------------------|----------------------------------------|
| TypeVar('T')      | Déclarer un type générique             |
| TypeVar('T', ...) | Constrainer le type (ex : int, float)  |
| List[T]           | Liste contenant des éléments de type T |
| Generic[T]        | Classe générique basée sur le type T   |

- **TP : Interface Forme, classes Cercle, Rectangle, méthode afficher\_surface()**

Parfait ! Voici un **exercice pratique (TP)** en Python orienté objet, basé sur :

- Une **interface Forme**
- Deux classes : **Cercle** et **Rectangle**
- Une méthode : **afficher\_surface()**, qui affiche la **surface** de la forme

### 🚧 Objectif du TP :

1. Définir une **interface Forme** (classe abstraite avec méthode afficher\_surface)
2. Créer les classes Cercle et Rectangle qui **implémentent cette interface**
3. Utiliser le **polymorphisme** pour appeler afficher\_surface() de façon générique

### Solution Python avec abc (abstract base class)

```
from abc import ABC, abstractmethod
from math import pi

Interface Forme
class Forme(ABC):
 @abstractmethod
 def afficher_surface(self):
 pass

Classe Cercle
class Cercle(Forme):
 def __init__(self, rayon: float):
 self.rayon = rayon

 def afficher_surface(self):
 surface = pi * self.rayon ** 2
 print(f"Surface du cercle : {surface:.2f}")

Classe Rectangle
class Rectangle(Forme):
 def __init__(self, largeur: float, hauteur: float):
 self.largeur = largeur
 self.hauteur = hauteur

 def afficher_surface(self):
 surface = self.largeur * self.hauteur
 print(f"Surface du rectangle : {surface:.2f}")
```

### Utilisation et démonstration polymorphe

```
Liste de formes
formes: list[Forme] = [
 Cercle(5),
 Rectangle(4, 6),
 Cercle(3),
 Rectangle(2.5, 8)
]

Affichage des surfaces de manière polymorphe
for forme in formes:
 forme.afficher_surface()
```

 **Résultat attendu :**

Surface du cercle : 78.54  
 Surface du rectangle : 24.00  
 Surface du cercle : 28.27  
 Surface du rectangle : 20.00

 **Concepts illustrés :**

| Élément             | Description                                                          |
|---------------------|----------------------------------------------------------------------|
| Forme(ABC)          | Interface abstraite                                                  |
| @abstractmethod     | Méthode obligatoire dans les sous-classes                            |
| Cercle et Rectangle | Implémentations concrètes                                            |
| Polymorphisme       | forme.afficher_surface() appelle la bonne méthode selon le type réel |

## ◆ Semaine 7 – Classes abstraites et interfaces

Voici une explication claire des **classes abstraites** et des **interfaces**, deux concepts fondamentaux en **programmation orientée objet**.

### 1. Classe Abstraite

#### ◆ Définition :

Une **classe abstraite** est une classe **incomplète** qui ne peut pas être instanciée directement. Elle sert de **modèle** pour les classes dérivées, en définissant des **méthodes communes** à implémenter.

#### Caractéristiques :

- Peut contenir des **méthodes abstraites** (non implémentées) **et** des méthodes normales (implémentées).
- Sert à **factoriser du code commun** à plusieurs classes.
- En Python, on utilise ABC et @abstractmethod du module abc.

#### Exemple en Python :

```
from abc import ABC, abstractmethod

class Forme(ABC): # Classe abstraite
 @abstractmethod
 def afficher_surface(self):
 pass # Pas d'implémentation ici
```

### 2. Interface

#### ◆ Définition :

Une **interface** est une **spécification pure** : elle ne contient que des **méthodes abstraites** (sans code). Elle définit ce qu'une classe **doit faire**, sans dire **comment**.

#### Caractéristiques :

- 100 % **abstraite** (dans les langages comme Java).
- En Python, on utilise aussi ABC pour représenter ce concept (car Python n'a pas d'interface "pure" comme Java).

 **Différence en résumé :**

| Élément            | Classe Abstraite                                                       | Interface                                                            |
|--------------------|------------------------------------------------------------------------|----------------------------------------------------------------------|
| Peut avoir du code | <input checked="" type="checkbox"/> Oui (méthodes concrètes possibles) | <input type="checkbox"/> Non (100 % abstraite en général)            |
| Héritage multiple  | <input type="checkbox"/> Souvent limité à une classe                   | <input checked="" type="checkbox"/> Possible dans plusieurs langages |
| But principal      | Factoriser du code                                                     | Imposer un contrat                                                   |

 **Exemple : interface style Python**

```
from abc import ABC, abstractmethod
```

```
class Affichable(ABC):
 @abstractmethod
 def afficher(self):
 pass
```

Toute classe qui "implémente" cette interface doit définir afficher().

 **En résumé :**

- Utilise une **classe abstraite** quand tu veux partager du **code commun** à plusieurs classes (base avec du comportement).
- Utilise une **interface** quand tu veux forcer les classes à **respecter un contrat** (structure), sans fournir d'implémentation.

Voici une **comparaison concrète entre Python et Java**, suivie d'un **diagramme UML simple** pour illustrer les relations entre **classe abstraite**, **interface**, et **classes concrètes**.

 **Comparaison Python / Java**

 **1. Interface**

 **Java**

```
public interface Affichable {
 void afficher();
}
```

-  Toutes les méthodes sont **abstraites et publiques** par défaut.

 **Python (via ABC)**

```
from abc import ABC, abstractmethod
```

```
class Affichable(ABC):
 @abstractmethod
 def afficher(self):
 pass
```

- Simule une interface avec ABC, mais peut techniquement contenir du code (Python ne distingue pas formellement interface et classe abstraite).

## 2. Classe Abstraite

### Java

```
public abstract class Forme {
 abstract void afficherSurface(); // Méthode abstraite

 void info() { // Méthode concrète
 System.out.println("Je suis une forme");
 }
}
```

### Python

```
from abc import ABC, abstractmethod

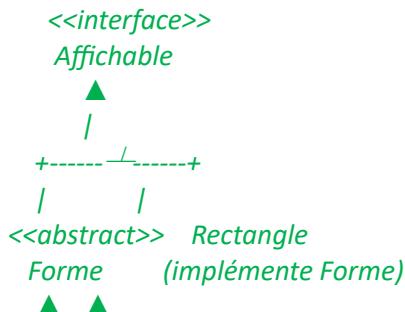
class Forme(ABC):
 @abstractmethod
 def afficher_surface(self):
 pass

 def info(self):
 print("Je suis une forme")
```

- Les deux permettent de combiner **méthodes abstraites et concrètes**.

## Diagramme UML

Voici un **diagramme UML simplifié** qui montre les relations entre une interface, une classe abstraite, et des classes concrètes :





### 📌 Signification UML :

- **Flèche en trait plein** : héritage (classe abstraite → classe concrète)
- **Flèche en pointillés** : implémentation (interface → classe ou classe abstraite)
- <>interface>> : représente une interface
- <>abstract>> : représente une classe abstraite

### 🎯 Interprétation :

- Affichable est une interface (définit une méthode afficher() par exemple).
- Forme est une classe abstraite qui **implémente l'interface Affichable** et ajoute des méthodes communes comme afficher\_surface().
- Cercle, Rectangle, Triangle héritent de Forme et doivent **implémenter les méthodes abstraites**.

## • Introduction au module abc (ABC, abstractmethod)

Voici une **introduction claire et pratique au module abc** en Python, qui permet de créer des **classes abstraites** et des **interfaces**.

### Qu'est-ce que le module abc ?

Le module abc (Abstract Base Classes) est un module standard de Python qui permet de :

- Créer des **classes abstraites**
- Définir des **méthodes abstraites** qui **doivent** être implémentées dans les sous-classes
- Simuler le comportement des **interfaces** comme en Java ou C#

### 1. Éléments principaux du module abc

| Élément         | Description                                                    |
|-----------------|----------------------------------------------------------------|
| ABC             | Classe de base pour toutes les classes abstraites              |
| @abstractmethod | Déclare une méthode abstraite (sans implémentation par défaut) |

### 2. Exemple simple : classe abstraite Forme

```
from abc import ABC, abstractmethod

class Forme(ABC): # Déclaration d'une classe abstraite
 @abstractmethod
 def afficher_surface(self):
```

*pass # Pas d'implémentation ici*

- Forme est une **classe abstraite** (ne peut pas être instanciée)
- afficher\_surface() est une **méthode abstraite** (obligatoire dans les sous-classes)

3. Implémentation concrète

```
class Cercle(Forme):
 def __init__(self, rayon):
 self.rayon = rayon

 def afficher_surface(self):
 surface = 3.14 * self.rayon ** 2
 print(f"Surface du cercle : {surface}")
```

Ici, Cercle **hérite** de Forme et **implémente la méthode abstraite**.

**⚠ 4. Si on n'implémente pas la méthode abstraite :**

```
class Rectangle(Forme):
 pass

Cette ligne provoque une erreur :
TypeError: Can't instantiate abstract class Rectangle with abstract method afficher_surface
```

Python interdit d'instancier une sous-classe **tant que toutes les méthodes abstraites ne sont pas redéfinies.**

 Pourquoi utiliser abc ?

| Avantage                                      | Explication                                          |
|-----------------------------------------------|------------------------------------------------------|
| Garantit une structure commune                | Oblige les sous-classes à définir certaines méthodes |
| Encourage la programmation orientée interface | On code par comportement, pas par type concret       |
| Facilite la maintenance                       | Les erreurs sont détectées tôt                       |

Voici un **exemple complet** d'une **classe abstraite** avec **plusieurs méthodes abstraites**, utilisant le module abc en Python.

 Objectif :

Créer une classe abstraite Forme avec plusieurs méthodes abstraites que **toutes les sous-classes devront implémenter**.

Exemple complet :

```
from abc import ABC, abstractmethod

from math import pi
```

- # ◆ Classe abstraite avec plusieurs méthodes abstraites

```
class Forme(ABC):
 @abstractmethod
 def afficher_nom(self):
 pass

 @abstractmethod
 def afficher_surface(self):
 pass

 @abstractmethod
 def calculer_surface(self) -> float:
 pass
```

◆ Sous-classe : Cercle

```
class Cercle(Forme):
 def __init__(self, rayon):
 self.rayon = rayon

 def afficher_nom(self):
 print("Cercle")

 def afficher_surface(self):
 print(f"Surface : {self.calculer_surface():.2f}")

 def calculer_surface(self):
 return pi * self.rayon ** 2
```

◆ Sous-classe : Rectangle

```
class Rectangle(Forme):
 def __init__(self, largeur, hauteur):
 self.largeur = largeur
 self.hauteur = hauteur

 def afficher_nom(self):
 print("Rectangle")

 def afficher_surface(self):
 print(f"Surface : {self.calculer_surface():.2f}")

 def calculer_surface(self):
 return self.largeur * self.hauteur
```

 Utilisation polymorphe

```
formes = [
 Cercle(5),
 Rectangle(4, 6)
]
```

```
for f in formes:
 f.afficher_nom()
 f.afficher_surface()
```

 **Résultat attendu :**

```
Cercle
Surface : 78.54
Rectangle
Surface : 24.00
```

 **Ce que ça montre :**

- Forme définit **3 comportements abstraits** : afficher\_nom(), afficher\_surface() et calculer\_surface().
- Cercle et Rectangle **doivent obligatoirement** les implémenter.
- Cela permet d'utiliser les objets via un **interface commune**, peu importe leur classe réelle.

• **Différence entre interface et classe concrète**

Voici une **explication claire** de la **différence entre une interface et une classe concrète**, dans le contexte de la **programmation orientée objet** (en particulier en Python et Java).

## Comparaison : Interface vs Classe concrète

| Critère                | Interface                                                                                                                       | Classe concrète                                                                                                              |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| Définition             | Un contrat qui <b>déclare</b> ce qu'une classe doit faire, <b>sans l'implémenter</b>                                            | Une classe avec des <b>méthodes complètement définies</b>                                                                    |
| Peut être instanciée ? |  Non                                         |  Oui                                    |
| Contient du code ?     |  Non (en général, uniquement des signatures) |  Oui (attributs, méthodes implémentées) |
| Utilité principale     | Garantir un <b>comportement commun</b> entre plusieurs classes sans hériter de code                                             | Représente une <b>implémentation complète</b> d'un objet                                                                     |
| Héritage               | Une classe peut <b>implémenter plusieurs interfaces</b>                                                                         | Une classe peut <b>hériter d'une seule classe</b> (en général)                                                               |
| Exemple en Python      | Classe avec ABC et uniquement des @abstractmethod                                                                               | Classe normale avec constructeur et méthodes définies                                                                        |
| Exemple en Java        | interface Affichable { void afficher(); }                                                                                       | class Cercle { double rayon; double surface(); }                                                                             |



## Exemple Python

◆ Interface simulée :

```
from abc import ABC, abstractmethod

class Affichable(ABC):
 @abstractmethod
 def afficher(self):
 pass
```

→ Ne contient **aucune logique**. Toute classe qui hérite de `Affichable` doit définir `afficher()`.

◆ Classe concrète :

```
class Cercle:
 def __init__(self, rayon):
 self.rayon = rayon

 def afficher(self):
 print(f"Cercle de rayon {self.rayon}")
```

→ Contient une **implémentation complète**. Elle peut être utilisée **directement** (instanciée).

🎯 En résumé :

| Interface                      | Classe concrète                         |
|--------------------------------|-----------------------------------------|
| Spécifie quoi faire            | Fait réellement quelque chose           |
| Pas d'implémentation           | Contient de l'implémentation            |
| Utilisée pour le polymorphisme | Utilisée pour représenter un objet réel |

• TP : **Implémentation de classes abstraites pour un système de paiement**

Voici un **TP (travail pratique)** complet en Python sur l'**implémentation de classes abstraites pour un système de paiement**, basé sur la programmation orientée objet.



🎯 Objectif :

Créer une structure où différents **moyens de paiement** (Carte Bancaire, PayPal, etc.) partagent une **interface commune**, tout en ayant chacun leur **comportement spécifique**.



✓ Étapes du TP :

1. Créer une **classe abstraite MoyenPaiement** avec les méthodes :

- authentifier()
  - payer(montant)
2. Implémenter des **classes concrètes** :
- CarteBancaire
  - PayPal
3. Tester le système de paiement de manière **polymorphe**.

### 1. Définition de la classe abstraite

```
from abc import ABC, abstractmethod

class MoyenPaiement(ABC):

 @abstractmethod
 def authentifier(self):
 pass

 @abstractmethod
 def payer(self, montant: float):
 pass
```

### 2. Classe concrète : CarteBancaire

```
class CarteBancaire(MoyenPaiement):
 def __init__(self, numero, titulaire):
 self.numero = numero
 self.titulaire = titulaire
 self.authentifie = False

 def authentifier(self):
 print(f"Carte de {self.titulaire} authentifiée.")
 self.authentifie = True

 def payer(self, montant: float):
 if self.authentifie:
 print(f"Paiement de {montant} € effectué avec la carte.")
 else:
 print("Erreur : la carte n'est pas authentifiée.")
```

### 3. Classe concrète : PayPal

```
class PayPal(MoyenPaiement):
 def __init__(self, email):
 self.email = email
 self.connecte = False
```

```

def authentifier(self):
 print(f"Utilisateur PayPal {self.email} connecté.")
 self.connecte = True

def payer(self, montant: float):
 if self.connecte:
 print(f"Paiement de {montant} € effectué via PayPal.")
 else:
 print("Erreur : utilisateur PayPal non connecté.")

```

#### 4. Utilisation de polymorphe

```

def traitement_paiement(moyen: MoyenPaiement, montant: float):
 moyen.authentifier()
 moyen.payer(montant)

Liste des moyens de paiement
paiements = [
 CarteBancaire("1234 5678 9876 5432", "Alice"),
 PayPal("bob@example.com")
]

for paiement in paiements:
 traitement_paiement(paiement, 150.00)

```

#### Résultat attendu :

Carte de Alice authentifiée.  
 Paiement de 150.0 € effectué avec la carte.  
 Utilisateur PayPal bob@example.com connecté.  
 Paiement de 150.0 € effectué via PayPal.

#### Concepts illustrés :

| Concept                        | Application ici                          |
|--------------------------------|------------------------------------------|
| Classe abstraite               | MoyenPaiement avec méthodes non définies |
| Implémentation concrète        | CarteBancaire, PayPal                    |
| Polymorphisme                  | Appel uniforme à payer()                 |
| Séparation des responsabilités | Chaque classe gère sa logique métier     |

Ajoutons maintenant une **classe GestionnairePaiement** qui centralise la **gestion des paiements**, quelle que soit la méthode utilisée.

### Objectif :

Créer une classe GestionnairePaiement qui :

- Reçoit un **objet de type MoyenPaiement**.
- Appelle authentifier() puis payer(montant).
- Gère plusieurs paiements avec **polymorphisme**.

### Mise à jour du TP

#### 1. Classe GestionnairePaiement

```
class GestionnairePaiement:
 def __init__(self):
 self.historique = []

 def traiter(self, moyen: MoyenPaiement, montant: float):
 print("----- Traitement du paiement -----")
 moyen.authentifier()
 moyen.payer(montant)
 self.historique.append((type(moyen).__name__, montant))
 print("-----\n")

 def afficher_historique(self):
 print("Historique des paiements :")
 for methode, montant in self.historique:
 print(f" - {methode} : {montant:.2f} €")
```

#### 2. Utilisation

```
Création du gestionnaire
gestionnaire = GestionnairePaiement()

Moyens de paiement
paiements = [
 CarteBancaire("1234 5678 9876 5432", "Alice"),
 PayPal("bob@example.com"),
 CarteBancaire("4444 3333 2222 1111", "Charlie")
]

Traitement des paiements
for p in paiements:
 gestionnaire.traiter(p, 75.00)

Affichage de l'historique
```

**Résultat attendu :**

----- Traitement du paiement -----  
 Carte de Alice authentifiée.  
 Paiement de 75.0 € effectué avec la carte.

---

----- Traitement du paiement -----  
 Utilisateur PayPal bob@example.com connecté.  
 Paiement de 75.0 € effectué via PayPal.

---

----- Traitement du paiement -----  
 Carte de Charlie authentifiée.  
 Paiement de 75.0 € effectué avec la carte.

---

Historique des paiements :

- CarteBancaire : 75.00 €
- PayPal : 75.00 €
- CarteBancaire : 75.00 €

**Concepts supplémentaires ajoutés :**

| Élément                     | Description                                                    |
|-----------------------------|----------------------------------------------------------------|
| Classe GestionnairePaiement | Applique le principe de <b>séparation des responsabilités</b>  |
| Historique                  | Utilise une <b>liste interne</b> pour stocker les transactions |
| Polymorphisme               | Appelle authentifier() et payer() sans connaître le type réel  |

Ajoutons maintenant une méthode **annuler\_paiement()** au système, pour permettre à chaque moyen de paiement de définir sa **propre manière d'annuler** une transaction.

### Objectif :

1. Ajouter une méthode abstraite annuler\_paiement() dans la classe MoyenPaiement.
2. Implémenter cette méthode dans les classes CarteBancaire et PayPal.
3. Mettre à jour le GestionnairePaiement pour permettre l'annulation.

### 1. Mise à jour de la classe abstraite :

```
from abc import ABC, abstractmethod

class MoyenPaiement(ABC):

 @abstractmethod
 def authentifier(self):
 pass

 @abstractmethod
 def payer(self, montant: float):
 pass

 @abstractmethod
 def annuler_paiement(self, montant: float):
 pass
```

### 2. Mise à jour de CarteBancaire :

```
class CarteBancaire(MoyenPaiement):
 def __init__(self, numero, titulaire):
 self.numero = numero
 self.titulaire = titulaire
 self.authentifie = False

 def authentifier(self):
 print(f"Carte de {self.titulaire} authentifiée.")
 self.authentifie = True

 def payer(self, montant: float):
 if self.authentifie:
 print(f"Paiement de {montant} € effectué avec la carte.")
 else:
 print("Erreur : carte non authentifiée.")

 def annuler_paiement(self, montant: float):
 print(f"Annulation de {montant} € sur la carte de {self.titulaire}.")
```

---

 **3. Mise à jour de PayPal :**

```
class PayPal(MoyenPaiement):
 def __init__(self, email):
 self.email = email
 self.connecte = False

 def authentifier(self):
 print(f"Utilisateur PayPal {self.email} connecté.")
 self.connecte = True

 def payer(self, montant: float):
 if self.connecte:
 print(f"Paiement de {montant} € effectué via PayPal.")
 else:
 print("Erreur : utilisateur non connecté.")

 def annuler_paiement(self, montant: float):
 print(f"Annulation de {montant} € via le compte PayPal {self.email}.")
```

 **4. Mise à jour de GestionnairePaiement :**

```
class GestionnairePaiement:
 def __init__(self):
 self.historique = []

 def traiter(self, moyen: MoyenPaiement, montant: float):
 print("----- Traitement du paiement -----")
 moyen.authentifier()
 moyen.payer(montant)
 self.historique.append((moyen, montant))
 print("-----\n")

 def annuler_dernier(self):
 if not self.historique:
 print("Aucun paiement à annuler.")
 return
 moyen, montant = self.historique.pop()
 print("----- Annulation du paiement -----")
 moyen.annuler_paiement(montant)
 print("-----\n")

 def afficher_historique(self):
 print("Historique des paiements :")
 for moyen, montant in self.historique:
 print(f" - {type(moyen).__name__} : {montant:.2f} €")
```

 **5. Exemple d'utilisation :**

```

gestionnaire = GestionnairePaiement()

c1 = CarteBancaire("1234 5678", "Alice")
p1 = PayPal("alice@example.com")

gestionnaire.traiter(c1, 100)
gestionnaire.traiter(p1, 50)

gestionnaire.afficher_historique()

Annuler le dernier paiement (PayPal)
gestionnaire.annuler_dernier()
gestionnaire.afficher_historique()

```

 **Résultat attendu :**

----- Traitement du paiement -----  
 Carte de Alice authentifiée.  
 Paiement de 100 € effectué avec la carte.

---

----- Traitement du paiement -----  
 Utilisateur PayPal alice@example.com connecté.  
 Paiement de 50 € effectué via PayPal.

---

Historique des paiements :  
 - CarteBancaire : 100.00 €  
 - PayPal : 50.00 €

----- Annulation du paiement -----  
 Annulation de 50 € via le compte PayPal alice@example.com.

---

Historique des paiements :  
 - CarteBancaire : 100.00 €

 **Concepts abordés :**

| Élément                 | Description                                            |
|-------------------------|--------------------------------------------------------|
| Méthode abstraite       | annuler_paiement() définie dans l'interface            |
| Polymorphisme inversé   | GestionnairePaiement appelle une méthode selon l'objet |
| Historique géré en pile | L'annulation retire le dernier paiement effectué       |

Nous allons maintenant ajouter une **interface de journalisation**, pour enregistrer les actions importantes (authentification, paiements, annulations).

### Objectif :

1. Créer une **interface Journalisable** avec une méthode abstraite `journaliser(message: str)`.
2. Permettre à des classes (comme `GestionnairePaiement`) d'utiliser cette interface pour enregistrer des actions.
3. Implémenter un **journal simple dans la console** (ou plus tard dans un fichier).

### 1. Interface Journalisable

```
from abc import ABC, abstractmethod
```

```
class Journalisable(ABC):
 @abstractmethod
 def journaliser(self, message: str):
 pass
```

### 2. Implémentation : ConsoleJournal

```
class ConsoleJournal(Journalisable):
 def journaliser(self, message: str):
 print(f"[JOURNAL] {message}")
```

→ Cette classe écrit les logs simplement dans la console.  
Plus tard, on pourrait en faire une version **fichier** ou **base de données**.

### 3. Mise à jour de `GestionnairePaiement` pour utiliser un `Journalisable`

```
class GestionnairePaiement:
 def __init__(self, journal: Journalisable = None):
 self.historique = []
 self.journal = journal

 def traiter(self, moyen: MoyenPaiement, montant: float):
 self.journaliser("DÉBUT du paiement.")
 moyen.authentifier()
 moyen.payer(montant)
```

```

self.historique.append((moyen, montant))
self.journaliser(f"Paiement de {montant:.2f} € effectué avec {type(moyen).__name__}.")
self.journaliser("FIN du paiement.\n")

def annuler_dernier(self):
 if not self.historique:
 self.journaliser("Aucun paiement à annuler.")
 return
 moyen, montant = self.historique.pop()
 moyen.annuler_paiement(montant)
 self.journaliser(f"Annulation de {montant:.2f} € avec {type(moyen).__name__}.\n")

def afficher_historique(self):
 print("Historique des paiements :")
 for moyen, montant in self.historique:
 print(f"- {type(moyen).__name__} : {montant:.2f} €")

def journaliser(self, message: str):
 if self.journal:
 self.journal.journaliser(message)

```

#### 4. Exemple d'utilisation

```

Création du journal
journal_console = ConsoleJournal()

Création du gestionnaire avec journalisation
gestionnaire = GestionnairePaiement(journal=journal_console)

Paiements
cb = CarteBancaire("9999 8888", "David")
paypal = PayPal("david@mail.com")

gestionnaire.traiter(cb, 200)
gestionnaire.traiter(paypal, 75)

Annulation
gestionnaire.annuler_dernier()

```

#### Résultat attendu (extrait console) :

[JOURNAL] DÉBUT du paiement.  
Carte de David authentifiée.  
Paiement de 200 € effectué avec la carte.  
[JOURNAL] Paiement de 200.00 € effectué avec CarteBancaire.  
[JOURNAL] FIN du paiement.

[JOURNAL] DÉBUT du paiement.  
Utilisateur PayPal david@mail.com connecté.  
Paiement de 75 € effectué via PayPal.

[JOURNAL] Paiement de 75.00 € effectué avec PayPal.

[JOURNAL] FIN du paiement.

Annulation de 75 € via le compte PayPal david@mail.com.

[JOURNAL] Annulation de 75.00 € avec PayPal.



### Concepts renforcés :

| Élément                         | Rôle                                                                           |
|---------------------------------|--------------------------------------------------------------------------------|
| Journalisable                   | Interface abstraite = contrat pour la journalisation                           |
| ConsoleJournal                  | Implémentation concrète (écriture sur la console)                              |
| Injection de dépendance         | GestionnairePaiement reçoit un journal externe                                 |
| SOLID : D (Dépendance inversée) | Le gestionnaire dépend d'une <b>abstraction</b> (et non d'une classe concrète) |

## ◆ Semaine 8 – Composition et agrégation

Voici une définition claire et concise de la **composition** et de l'**agrégation** en programmation orientée objet, avec leurs différences principales.

### Définitions

#### 1. Agrégation

- C'est une relation "**a un**" (has-a) entre deux objets, où un objet **contient** un ou plusieurs objets, mais ces objets **peuvent exister indépendamment**.
- L'objet contenu **n'est pas responsable** de la durée de vie de l'objet qu'il contient.
- Exemple : Une Classe contient plusieurs Étudiants, mais les étudiants peuvent exister sans cette classe (ils peuvent appartenir à une autre classe).

#### 2. Composition

- C'est une forme plus forte d'agrégation où l'objet **possède** un ou plusieurs objets, et **est responsable** de leur cycle de vie.
- Si l'objet parent est détruit, **les objets contenus le sont aussi**.
- Exemple : Un Livre est composé de plusieurs Pages. Si le livre est détruit, ses pages ne peuvent pas exister sans lui.

### Différences clés

| Caractéristique | Agrégation                             | Composition                            |
|-----------------|----------------------------------------|----------------------------------------|
| Relation        | "a un" faible                          | "a un" fort                            |
| Cycle de vie    | Objets liés peuvent exister séparément | Objets contenus dépendent du contenant |
| Dépendance      | Faible                                 | Forte                                  |
| Exemple         | Classe – Étudiants                     | Livre – Pages                          |

#### Exemple en UML

- **Agrégation** : une flèche vide (ligne avec un losange vide du côté du conteneur).
- **Composition** : une flèche avec un losange rempli du côté du conteneur.

### Exemple simple en Python

```
Agrégation : Une équipe contient des joueurs, mais les joueurs existent seuls
```

```
class Joueur:
 def __init__(self, nom):
 self.nom = nom

class Equipe:
 def __init__(self, nom, joueurs):
 self.nom = nom
```

```
self.joueurs = joueurs # liste de joueurs (existe indépendamment)
```

# Composition : Une maison contient des pièces qui n'existent pas sans la maison

```
class Piece:
 def __init__(self, nom):
 self.nom = nom

class Maison:
 def __init__(self):
 self.pieces = [Piece("Salon"), Piece("Cuisine")] # pièces créées par la maison
```

- **Composition vs héritage**

Bien sûr ! Voici une comparaison claire entre **composition** et **héritage**, deux concepts fondamentaux en programmation orientée objet (POO) :

### ❖ Composition vs Héritage

| Critère                            | Composition                                                                             | Héritage                                                                 |
|------------------------------------|-----------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| <b>Définition</b>                  | Un objet contient un ou plusieurs objets (relation "a un")                              | Une classe dérive d'une autre classe (relation "est un")                 |
| <b>Relation entre classes</b>      | « a un » (has-a)                                                                        | « est un » (is-a)                                                        |
| <b>Utilisation</b>                 | Pour assembler des objets et réutiliser des fonctionnalités sans modifier la hiérarchie | Pour étendre ou spécialiser une classe existante                         |
| <b>Flexibilité</b>                 | Plus flexible, car on peut changer les composants à l'exécution                         | Moins flexible, car les relations sont fixées à la compilation           |
| <b>Couplage</b>                    | Faible couplage (les objets sont indépendants)                                          | Couplage fort (sous-classe dépend de la super-classe)                    |
| <b>Réutilisation du code</b>       | Par délégation (l'objet "compose" des comportements)                                    | Par héritage direct (la sous-classe hérite du code)                      |
| <b>Exemple</b>                     | Une voiture a un moteur                                                                 | Une voiture est une voiture électrique (hérite d'une voiture)            |
| <b>Changement d'implémentation</b> | Plus facile : on peut remplacer les objets composés                                     | Plus compliqué, modifier la super-classe affecte toutes les sous-classes |

 Exemple en Python

## Héritage

```
class Animal:

 def parler(self):
 print("Son générique")

class Chien(Animal):

 def parler(self):
 print("Ouaf !")

chien = Chien()
chien.parler() # Ouaf !
```

## Composition

```
class Moteur:

 def demarrer(self):
 print("Moteur démarré")

class Voiture:

 def __init__(self):
 self.moteur = Moteur() # Voiture a un moteur

 def demarrer(self):
 self.moteur.demarrer()

voiture = Voiture()
voiture.demarrer() # Moteur démarré
```

### ⚠️ Quand utiliser quoi ?

- **Héritage** : lorsque la relation “est un” est vraie et que la spécialisation est naturelle.
- **Composition** : lorsque vous voulez construire des objets complexes à partir de composants, ou quand vous voulez plus de flexibilité et éviter les rigidités de l’héritage.



## • Encapsulation de classes dans d'autres classes

L'**encapsulation de classes dans d'autres classes** est une pratique en programmation orientée objet (POO) qui consiste à **définir une classe à l'intérieur d'une autre**, ou à **masquer certaines classes aux utilisateurs externes**.

Ce concept peut se décliner de **deux manières principales** :

### 1. Encapsulation logique (via composition ou classe interne)

#### Exemple : Classe interne (nested class)

```
class Ordinateur:
 def __init__(self, marque):
 self.marque = marque
 self.processeur = self.Processeur("Intel i7") # Création d'un objet de la classe interne

 def afficher_info(self):
 print(f"Ordinateur {self.marque}")
 self.processeur.afficher_info()

class Processeur:
 def __init__(self, modele):
 self.modele = modele

 def afficher_info(self):
 print(f"Processeur : {self.modele}")
```

#### Utilisation :

```
pc = Ordinateur("HP")
pc.afficher_info()
➤ Résultat :
Ordinateur HP
Processeur : Intel i7
```

### 2. Encapsulation par composition

C'est la forme la plus courante : une classe **utilise** une autre classe comme attribut, sans forcément la définir à l'intérieur.

#### ◆ Exemple :

```
class Moteur:
 def demarrer(self):
 print("Moteur démarré")

class Voiture:
 def __init__(self):
 self.moteur = Moteur() # Encapsulation d'objet moteur dans voiture

 def demarrer(self):
```

```
self.moteur.demarrer()
```

## ❖ Pourquoi encapsuler une classe dans une autre ?

| Avantage                           | Détail                                                     |
|------------------------------------|------------------------------------------------------------|
| Regroupe des composants liés       | Ex : Ordinateur et Processeur partagent un contexte commun |
| Limite la visibilité / le couplage | La classe interne est cachée à l'extérieur                 |
| Respecte l'encapsulation           | L'utilisateur n'a accès qu'à l'interface publique          |
| Facilite l'organisation du code    | Permet de structurer des modules complexes                 |

### ⚠ À éviter si...

- La classe interne est trop complexe → mieux vaut la sortir.
- La classe doit être réutilisée ailleurs.

## • TP : Modéliser un Livre qui contient des Chapitre

Voici un **TP complet** pour modéliser un système orienté objet où un **Livre** est composé de **Chapitres**, en illustrant le concept de **composition**.

### 🎯 Objectif du TP

- Créer une classe **Chapitre** représentant un chapitre de livre.
- Créer une classe **Livre** qui contient plusieurs chapitres.
- Mettre en place des méthodes pour **ajouter**, **afficher**, et **compter** les chapitres.
- Illustrer une **relation de composition** (un chapitre n'existe pas sans un livre).

### ✓ 1. Classe Chapitre

```
class Chapitre:
 def __init__(self, titre, numero, contenu):
 self.titre = titre
 self.numero = numero
 self.contenu = contenu

 def afficher(self):
 print(f"Chapitre {self.numero} : {self.titre}")
```

### ✓ 2. Classe Livre

```
class Livre:
 def __init__(self, titre, auteur):
 self.titre = titre
 self.auteur = auteur
```

```
self.chapitres = [] # Composition : le livre contient les chapitres
```

```
def ajouter_chapitre(self, titre, contenu):
 numero = len(self.chapitres) + 1
 chapitre = Chapitre(titre, numero, contenu)
 self.chapitres.append(chapitre)

def afficher_table_matiere(self):
 print(f'Livre : {self.titre} - {self.auteur}')
 print("Table des matières :")
 for chapitre in self.chapitres:
 chapitre.afficher()

def nombre_chapitres(self):
 return len(self.chapitres)
```

### 3. Exemple d'utilisation

```
livre = Livre("Le voyage intérieur", "Claire Martin")

livre.ajouter_chapitre("Introduction", "Bienvenue dans le voyage intérieur.")
livre.ajouter_chapitre("Chapitre 1 : Origines", "Dans ce chapitre, nous explorons les origines.")
livre.ajouter_chapitre("Chapitre 2 : Destinées", "Vers où allons-nous ?")

livre.afficher_table_matiere()

print(f"\nNombre de chapitres : {livre.nombre_chapitres()}")
```

### Résultat attendu :

```
Livre : Le voyage intérieur - Claire Martin
Table des matières :
Chapitre 1 : Introduction
Chapitre 2 : Chapitre 1 : Origines
Chapitre 3 : Chapitre 2 : Destinées
```

```
Nombre de chapitres : 3
```

### Concepts illustrés

| Concept                      | Description                                                                      |
|------------------------------|----------------------------------------------------------------------------------|
| <b>Composition</b>           | Les chapitres sont créés <b>dans</b> le livre. Ils <b>n'existent pas seuls</b> . |
| <b>Encapsulation</b>         | Les données sont protégées à l'intérieur des classes.                            |
| <b>Modularité</b>            | Le code est divisé en <b>unités claires</b> : Livre et Chapitre.                 |
| <b>Responsabilité unique</b> | Chaque classe gère uniquement <b>ses propres données et comportements</b> .      |



## ◆ Semaine 9 – Gestion des erreurs et exceptions personnalisées

La **gestion des erreurs** et l'utilisation d'**exceptions personnalisées** sont des éléments essentiels pour écrire du **code Python robuste, lisible et maintenable**.

### 1. Gestion des erreurs en Python

Python gère les erreurs via des **exceptions** avec le bloc `try/except`.

#### ◆ Exemple de base :

```
try:
 x = int(input("Entrez un entier : "))
 print(10 / x)
except ValueError:
 print("Erreur : vous devez entrer un entier.")
except ZeroDivisionError:
 print("Erreur : division par zéro.")
finally:
 print("Opération terminée.")
```

### 2. Créer des exceptions personnalisées

Pour créer une exception personnalisée, on hérite de la classe `Exception`.

#### ◆ Exemple :

```
class ChapitreVideError(Exception):
 """Exception levée si un chapitre est vide."""
 def __init__(self, message="Le chapitre ne peut pas être vide."):
 super().__init__(message)
```

### 3. Utilisation dans un contexte réel

Voici comment intégrer une exception personnalisée dans une classe :

#### ◆ Exemple avec un livre

```
class Livre:
 def __init__(self, titre):
 self.titre = titre
 self.chapitres = []

 def ajouter_chapitre(self, titre, contenu):
 if not contenu.strip():
 raise ChapitreVideError("Contenu du chapitre vide.")
 self.chapitres.append((titre, contenu))
```

#### ◆ Utilisation avec gestion des erreurs

```
livre = Livre("Apprendre Python")
```

try:

```
livre.ajouter_chapitre("Introduction", " ") # contenu vide
except ChapitreVideError as e:
 print(f"Erreur personnalisée : {e}")
```

→ Résultat :

Erreur personnalisée : Contenu du chapitre vide.

#### 4. Pourquoi utiliser des exceptions personnalisées ?

| Avantage                                                                                                | Description                                                                    |
|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
|  Plus de clarté        | Le type d'erreur est explicite dans le code.                                   |
|  Meilleure maintenance | On peut gérer chaque type d'erreur de manière distincte.                       |
|  Bonnes pratiques      | Respecte le principe de <b>séparation des préoccupations</b> .                 |
|  Réutilisable          | Une exception personnalisée peut être utilisée dans plusieurs classes/modules. |

#### 5. Bonnes pratiques

- Ne pas **abuser** des exceptions personnalisées : créez-en uniquement si l'exception **ajoute de la valeur** au traitement.
- Toujours hériter de Exception ou d'une sous-classe.
- Ajoutez un **message explicite**.
- Pensez à utiliser des blocs try/except/finally pour garantir un comportement même en cas d'erreur.

Voici un **TP complet en Python** illustrant l'usage de **plusieurs exceptions personnalisées**, intégré dans un mini-projet pratique : **gestion d'une bibliothèque de livres**.

#### Objectif pédagogique

Mettre en place un système :

- Qui gère une collection de livres et de chapitres.
- Qui valide les entrées.
- Qui lève et capture plusieurs **exceptions personnalisées**.

#### Structure du projet

- Livre : contient des Chapitres.
- Bibliotheque : contient plusieurs livres.

- Exceptions personnalisées :

- LivreExisteDejaError
- LivreInexistantError
- ChapitreVideError
- ChapitreExisteDejaError

 **1. Déclaration des exceptions personnalisées**

```
class LivreExisteDejaError(Exception):
 pass

class LivreInexistantError(Exception):
 pass

class ChapitreVideError(Exception):
 pass

class ChapitreExisteDejaError(Exception):
 pass
```

 **2. Classe Chapitre**

```
class Chapitre:
 def __init__(self, titre, contenu):
 if not contenu.strip():
 raise ChapitreVideError("Le contenu du chapitre ne peut pas être vide.")
 self.titre = titre
 self.contenu = contenu
```

 **3. Classe Livre**

```
class Livre:
 def __init__(self, titre):
 self.titre = titre
 self.chapitres = {}

 def ajouter_chapitre(self, titre, contenu):
 if titre in self.chapitres:
 raise ChapitreExisteDejaError(f"Le chapitre '{titre}' existe déjà.")
 chapitre = Chapitre(titre, contenu)
 self.chapitres[titre] = chapitre

 def afficher_chapitres(self):
 print(f"\nChapitres du livre : {self.titre}")
 for titre in self.chapitres:
 print(f" - {titre}")
```

 **4. Classe Bibliotheque**

```
class Bibliotheque:
 def __init__(self):
 self.livres = {}

 def ajouter_livre(self, titre):
 if titre in self.livres:
 raise LivreExisteDejaError(f"Le livre '{titre}' existe déjà.")
 self.livres[titre] = Livre(titre)

 def get_livre(self, titre):
 if titre not in self.livres:
 raise LivreInexistantError(f"Le livre '{titre}' n'existe pas.")
 return self.livres[titre]

 def afficher_livres(self):
 print("\nLivres disponibles :")
 for titre in self.livres:
 print(f" - {titre}")
```

 **5. Exemple d'utilisation avec gestion des erreurs**

```
biblio = Bibliotheque()

try:
 biblio.ajouter_livre("Python pour débutants")
 biblio.ajouter_livre("Python pour débutants") # Erreur : déjà ajouté
except LivreExisteDejaError as e:
 print(f"[ERREUR] {e}")

try:
 livre = biblio.get_livre("Python pour débutants")
 livre.ajouter_chapitre("Introduction", " ") # Erreur : contenu vide
except ChapitreVideError as e:
 print(f"[ERREUR] {e}")

try:
 livre.ajouter_chapitre("Introduction", "Bienvenue dans le monde Python.")
 livre.ajouter_chapitre("Introduction", "Chapitre en double.") # Erreur
except ChapitreExisteDejaError as e:
 print(f"[ERREUR] {e}")

try:
 livre_inexistant = biblio.get_livre("Inexistant")
except LivreInexistantError as e:
 print(f"[ERREUR] {e}")

Affichage
biblio.afficher_livres()
livre.afficher_chapitres()
```

### Résultat attendu

[ERREUR] Le livre 'Python pour débutants' existe déjà.

[ERREUR] Le contenu du chapitre ne peut pas être vide.

[ERREUR] Le chapitre 'Introduction' existe déjà.

[ERREUR] Le livre 'Inexistant' n'existe pas.

### Livres disponibles :

- Python pour débutants

### Chapitres du livre : Python pour débutants

- Introduction

### Concepts illustrés

| Élément                   | Application dans le TP                                             |
|---------------------------|--------------------------------------------------------------------|
| Exceptions personnalisées | Gestion fine des erreurs métiers (LivreExisteDejaError, etc.)      |
| Encapsulation             | Les objets cachent leurs données internes                          |
| Composition               | Un Livre contient des Chapitre, un Bibliotheque contient des Livre |
| Validation de données     | Empêche l'ajout de chapitres vides ou en doublon                   |

- Gérer les erreurs dans une **application console interactive** ?

Voici comment **gérer les erreurs proprement** dans une **application console interactive Python**, tout en utilisant les **exceptions personnalisées** vues dans le TP précédent.

### Objectif

Créer une **interface en ligne de commande** permettant à un utilisateur :

- d'ajouter des livres,
- d'ajouter des chapitres,
- d'afficher les livres et chapitres,
- avec une **gestion robuste des erreurs**.

 **Rappel : Exceptions personnalisées**

```
class LivreExisteDejaError(Exception): pass
class LivreInexistantError(Exception): pass
class ChapitreVideError(Exception): pass
class ChapitreExisteDejaError(Exception): pass
```

 **Structure simplifiée des classes**

```
class Chapitre:
 def __init__(self, titre, contenu):
 if not contenu.strip():
 raise ChapitreVideError("Le contenu du chapitre ne peut pas être vide.")
 self.titre = titre
 self.contenu = contenu

class Livre:
 def __init__(self, titre):
 self.titre = titre
 self.chapitres = {}

 def ajouter_chapitre(self, titre, contenu):
 if titre in self.chapitres:
 raise ChapitreExisteDejaError(f"Le chapitre '{titre}' existe déjà.")
 self.chapitres[titre] = Chapitre(titre, contenu)

 def afficher_chapitres(self):
 if not self.chapitres:
 print("Aucun chapitre.")
 for titre in self.chapitres:
 print(f"- {titre}")

class Bibliotheque:
 def __init__(self):
 self.livres = {}

 def ajouter_livre(self, titre):
 if titre in self.livres:
 raise LivreExisteDejaError(f"Le livre '{titre}' existe déjà.")
 self.livres[titre] = Livre(titre)

 def get_livre(self, titre):
 if titre not in self.livres:
 raise LivreInexistantError(f"Le livre '{titre}' n'existe pas.")
 return self.livres[titre]

 def afficher_livres(self):
 if not self.livres:
 print("Aucun livre disponible.")
 for titre in self.livres:
 print(f"- {titre}")
```

## ¶ Interface console interactive

```

def menu():
 print("\n--- MENU ---")
 print("1. Ajouter un livre")
 print("2. Ajouter un chapitre à un livre")
 print("3. Afficher les livres")
 print("4. Afficher les chapitres d'un livre")
 print("5. Quitter")

def lancer_console():
 biblio = Bibliotheque()

 while True:
 menu()
 choix = input("Choisissez une option : ").strip()

 try:
 if choix == "1":
 titre = input("Titre du livre : ")
 biblio.ajouter_livre(titre)
 print(f"Livre '{titre}' ajouté.")

 elif choix == "2":
 titre_livre = input("Nom du livre : ")
 livre = biblio.get_livre(titre_livre)
 titre_chapitre = input("Titre du chapitre : ")
 contenu = input("Contenu du chapitre : ")
 livre.ajouter_chapitre(titre_chapitre, contenu)
 print(f"Chapitre '{titre_chapitre}' ajouté au livre '{titre_livre}'. ")

 elif choix == "3":
 biblio.afficher_livres()

 elif choix == "4":
 titre_livre = input("Nom du livre : ")
 livre = biblio.get_livre(titre_livre)
 livre.afficher_chapitres()

 elif choix == "5":
 print("Au revoir !")
 break

 else:
 print("Choix invalide.")

 except (LivreExisteDejaError, LivreInexistantError,
 ChapitreVideError, ChapitreExisteDejaError) as e:
 print(f"[Erreur] {e}")

```

```
except Exception as e:
 print(f"[Erreur inattendue] {e}")
```

## ▶ Lancer le programme

```
if __name__ == "__main__":
 lancer_console()
```

## ✓ Avantages de cette approche

| Fonctionnalité           | Description                                                         |
|--------------------------|---------------------------------------------------------------------|
| Interface interactive    | Utilisateur guidé via menus clairs                                  |
| Gestion d'erreurs propre | Utilisation d'exceptions personnalisées avec messages explicites    |
| Modularité               | Séparation du modèle (classes) et du contrôleur (interface console) |
| Extensible               | Facile à transformer en interface graphique ou API plus tard        |

## • Gestión d'exceptions en Python

La **gestion des exceptions** en Python permet de traiter proprement les erreurs qui pourraient survenir à l'exécution du programme, sans le faire crasher brutalement. C'est un pilier de la **programmation robuste**.

### 💡 1. Qu'est-ce qu'une exception ?

Une **exception** est une erreur détectée pendant l'exécution. Par exemple :

```
print(10 / 0) # ➤ ZeroDivisionError
```

### 🛠 2. Gestion avec try / except

#### ◆ Structure de base

```
try:
 # Code susceptible de provoquer une erreur
 x = int(input("Entrez un nombre : "))
 print(10 / x)
except ZeroDivisionError:
 print("Erreur : division par zéro.")
except ValueError:
 print("Erreur : valeur invalide.")
```

### 3. Blocs else et finally

| Bloc    | Utilisation                                               |
|---------|-----------------------------------------------------------|
| else    | S'exécute si <b>aucune exception</b> n'est levée          |
| finally | S'exécute <b>dans tous les cas</b> (utile pour nettoyage) |

- ◆ Exemple :

```
try:
 x = int(input("Nombre positif : "))
 assert x > 0, "Le nombre doit être positif"
except ValueError:
 print("Entrée invalide.")
except AssertionError as e:
 print("Erreur :", e)
else:
 print("Valeur correcte.")
finally:
 print("Fin du traitement.")
```

### 4. Exceptions courantes en Python

| Exception             | Cause typique                                |
|-----------------------|----------------------------------------------|
| ZeroDivisionError     | Division par zéro                            |
| ValueError            | Conversion invalide (ex: int("abc"))         |
| TypeError             | Mauvais type (ex: addition entre int et str) |
| KeyError              | Clé absente dans un dictionnaire             |
| IndexError            | Indice hors limites dans une liste           |
| FileNotFoundException | Fichier introuvable                          |

### 5. Lever des exceptions avec raise

```
def division(a, b):
 if b == 0:
 raise ZeroDivisionError("Division par zéro interdite.")
 return a / b

try:
 print(division(5, 0))
except ZeroDivisionError as e:
 print(f"Erreur détectée : {e}")
```

## 6. Exceptions personnalisées

```
class NombreNegatifError(Exception):
 pass

def racine(x):
 if x < 0:
 raise NombreNegatifError("Impossible de calculer la racine d'un nombre négatif.")
 return x ** 0.5
```

### Bonnes pratiques

- Gérer **seulement les exceptions attendues**.
- Utiliser finally pour libérer des ressources (ex: fermer un fichier).
- Créer des **exceptions personnalisées** pour les règles métier.
- Ne pas utiliser except: sans préciser l'exception (trop large).

### • Créer ses propres exceptions

Créer ses **propres exceptions personnalisées** en Python permet de mieux structurer son code, de le rendre **plus lisible, plus expressif, et plus facile à maintenir**, notamment dans des cas métiers spécifiques.



## 1. Pourquoi créer des exceptions personnalisées ?

| Raison                             | Exemple                                                            |
|------------------------------------|--------------------------------------------------------------------|
| Identifier des erreurs métiers     | SoldeInsuffisantError dans une app bancaire                        |
| Clarifier les messages d'erreurs   | Message explicite au lieu d'un ValueError générique                |
| Permettre un traitement spécifique | Différencier UtilisateurInexistantError de DroitsInsuffisantsError |



## 2. Comment créer une exception personnalisée ?

Il suffit de **créer une classe** qui hérite de la classe `Exception` (ou une sous-classe).

### ◆ Exemple simple :

```
class MonException(Exception):
 pass
```

◆ Exemple avec message personnalisé :

```
class SoldeInsuffisantError(Exception):
 def __init__(self, message="Solde insuffisant pour effectuer l'opération."):
 super().__init__(message)
```



## 3. Utilisation dans un programme

```
def retirer(solde, montant):
 if montant > solde:
 raise SoldeInsuffisantError(f"You avez {solde}€, vous essayez de
retirer {montant}€.")
 return solde - montant

try:
 nouveau_solde = retirer(100, 150)
except SoldeInsuffisantError as e:
 print(f"Erreur : {e}")
```

= Résultat :

Erreur : Vous avez 100€, vous essayez de retirer 150€.



## 4. Bonnes pratiques

- Nommez clairement vos exceptions (`NomMéthierError`)
- Héritez toujours d'`Exception`, sauf cas avancé
- Documentez si besoin avec une docstring :

```
class FichierInvalideError(Exception):
 """Exception levée quand un fichier ne respecte pas le format attendu."""
 pass
```



## Exemple complet : formulaire utilisateur

```
class ChampObligatoireError(Exception):
 pass

class EmailInvalideError(Exception):
 pass

def valider_utilisateur(nom, email):
 if not nom:
```

```

 raise ChampObligatoireError("Le nom est obligatoire.")
if "@" not in email:
 raise EmailInvalideError("L'email est invalide.")
print("Utilisateur valide.")

try:
 valider_utilisateur("", "example.com")
except (ChampObligatoireError, EmailInvalideError) as e:
 print(f"[Erreur] {e}")

```

## ⊕ Optionnel : Regrouper vos exceptions dans une hiérarchie

```

class ErreurApplication(Exception): pass
class ConnexionError(ErreurApplication): pass
class AuthentificationError(ErreurApplication): pass

```

Cela vous permet de **capturer toute erreur métier d'un bloc** si nécessaire.

- **TP : Système de réservation avec vérification des données et exceptions**

Parfait ! Voici un **TP complet** pour un **système de réservation** en Python, avec :

- Vérification des données entrées par l'utilisateur
- Gestion des erreurs avec **exceptions personnalisées**
- Console interactive simple
- Application claire des concepts POO et robustesse

### Objectif pédagogique

Créer une application qui permet de :

- Réserver une place à un événement
- Vérifier la validité des données
- Gérer les erreurs de manière **propre et explicite**
- Utiliser des **exceptions personnalisées** pour les cas métier

### Étapes du TP

#### 1. Créer des exceptions personnalisées

#### 2. Créer les classes : **Reservation**, **Evenement**, **SystèmeReservation**

### 3. Intégrer la vérification des données

#### 4. Ajouter une interface console interactive

- ◆ 1. Exceptions personnalisées

```
class ReservationError(Exception): pass

class PlaceDejaReserveeError(ReservationError):
 pass

class DonneesInvalidesError(ReservationError):
 pass

class EvenementInexistantError(ReservationError):
 pass
```

#### 2. Classe Reservation

```
class Reservation:
 def __init__(self, nom, email, place):
 if not nom.strip() or "@" not in email:
 raise DonneesInvalidesError("Nom ou email invalide.")
 self.nom = nom
 self.email = email
 self.place = place
```

#### 3. Classe Evenement

```
class Evenement:
 def __init__(self, titre, nb_places):
 self.titre = titre
 self.nb_places = nb_places
 self.reservations = {}

 def reserver(self, nom, email, place):
 if place in self.reservations:
 raise PlaceDejaReserveeError(f"La place {place} est déjà réservée.")
 if not (1 <= place <= self.nb_places):
 raise DonneesInvalidesError(f"Place invalide. Choisissez entre 1 et {self.nb_places}.")
 reservation = Reservation(nom, email, place)
 self.reservations[place] = reservation
 print(f"✓ Réservation confirmée pour {nom}, place {place}.")

 def afficher_reservations(self):
 if not self.reservations:
 print("Aucune réservation.")
 for p, r in self.reservations.items():
 print(f"- Place {p} : {r.nom} ({r.email})")
```

 **4. Classe SystemeReservation**

```
class SystemeReservation:
 def __init__(self):
 self.evenements = {}

 def ajouter_evenement(self, titre, nb_places):
 self.evenements[titre] = Evenement(titre, nb_places)

 def reserver_place(self, titre_evenement, nom, email, place):
 if titre_evenement not in self.evenements:
 raise EvenementInexistantError(f"L'événement '{titre_evenement}' n'existe pas.")
 evenement = self.evenements[titre_evenement]
 evenement.reserver(nom, email, place)

 def afficher_evenements(self):
 for titre, event in self.evenements.items():
 print(f" {titre} - {event.nb_places} places")
```

 **5. Interface console interactive**

```
def lancer_console():
 systeme = SystemeReservation()
 systeme.ajouter_evenement("Concert de jazz", 5)
 systeme.ajouter_evenement("Théâtre", 3)

 while True:
 print("\n--- MENU ---")
 print("1. Voir les événements")
 print("2. Réserver une place")
 print("3. Voir les réservations")
 print("4. Quitter")

 choix = input("Votre choix : ").strip()

 if choix == "1":
 systeme.afficher_evenements()

 elif choix == "2":
 try:
 titre = input("Nom de l'événement : ")
 nom = input("Votre nom : ")
 email = input("Votre email : ")
 place = int(input("Numéro de place : "))
 systeme.reserver_place(titre, nom, email, place)
 except (ReservationError, ValueError) as e:
 print(f"[Erreur] {e}")

 elif choix == "3":
 titre = input("Nom de l'événement : ")
 try:
```

```

event = systeme.evenements[titre]
event.afficher_reservations()
except KeyError:
 print("[Erreur] Événement inconnu.")

elif choix == "4":
 print("👋 Merci et à bientôt.")
 break
else:
 print("Choix invalide.")

```

### Lancer le programme

```

if __name__ == "__main__":
 lancer_console()

```

### Résumé des concepts utilisés

| Concept                   | Illustration dans le TP                                        |
|---------------------------|----------------------------------------------------------------|
| Exceptions personnalisées | Gèrent les cas métier spécifiques                              |
| Validation de données     | Vérifie nom/email/place                                        |
| POO (composition)         | Un système contient des événements, eux-mêmes des réservations |
| Console interactive       | Dialogue utilisateur clair et protégé contre les erreurs       |

## ◆ Semaine 10 – Organisation modulaire du code

### ❖ Définition : Organisation modulaire du code en Python

L'**organisation modulaire du code** en Python consiste à **diviser un programme en plusieurs fichiers (modules) et dossiers (packages)** logiques, plutôt que tout écrire dans un seul fichier. Cela permet d'écrire du code plus propre, réutilisable, maintenable et testable.

### Pourquoi organiser son code en modules ?

| Avantage | Description                                            |
|----------|--------------------------------------------------------|
|          | Chaque fichier a un rôle clair                         |
|          | Les fonctions/classes peuvent être importées ailleurs  |
|          | On peut tester chaque module individuellement          |
|          | Cacher les détails internes, exposer une API propre    |
|          | Plusieurs développeurs peuvent travailler en parallèle |

### Exemple d'organisation d'un projet

```
mon_projet/
|
├── main.py # Point d'entrée principal
├── gestion_utilisateur.py # Fonctions liées aux utilisateurs
└── reservation/
 ├── __init__.py
 ├── evenement.py # Classe Evenement
 ├── systeme.py # Classe SystemeReservation
 └── exceptions.py # Exceptions personnalisées
 └── utils/
 ├── __init__.py
 └── validation.py # Fonctions utilitaires (email, date...)
└── tests/
 ├── test_reservation.py
 └── test_utilisateur.py
```

### Exemple concret

#### 1. reservation/exceptions.py

```
class ReservationError(Exception): pass
class PlaceDejaReserveeError(ReservationError): pass
```

#### 2. reservation/evenement.py

```
from .exceptions import PlaceDejaReserveeError
```

```

class Evenement:
 def __init__(self, titre, nb_places):
 self.titre = titre
 self.nb_places = nb_places
 self.reservations = {}

 def reserver(self, place):
 if place in self.reservations:
 raise PlaceDejaReserveeError("Place déjà réservée.")
 self.reservations[place] = True

```

### 3. main.py

```

from reservation.evenement import Evenement

event = Evenement("Conférence", 10)
event.reserver(1)

```

#### Importation dans un projet modulaire

- Depuis le même dossier :
- import reservation.evenement
- Depuis un fichier du même package :
- from .exceptions import ReservationError

#### Bonnes pratiques

| Pratique                                                                                                             | Détail                                          |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|
|  1 module = 1 responsabilité      | utilisateur.py, paiement.py, etc.               |
|  Utiliser __init__.py             | Pour marquer les dossiers comme packages Python |
|  Isoler les tests                 | Dans un dossier tests/                          |
|  Éviter les fichiers > 300 lignes | Favoriser la clarté et la lisibilité            |
|  Ajouter un README.md ou doc      | Décrire la structure du projet                  |

- **Modules et packages Python**

#### Modules et Packages en Python : Explication claire et exemples

##### ◆ 1. Qu'est-ce qu'un module en Python ?

Un **module** est un **fichier Python (.py)** contenant du code — fonctions, classes, variables — que l'on peut **importer** dans d'autres fichiers.



Exemple simple : module `math_utils.py`

```
math_utils.py
def carre(x):
 return x * x

def cube(x):
 return x ** 3
❖ Utilisation dans un autre fichier :
main.py
import math_utils

print(math_utils.carre(4)) # 16
```

◆ 2. Qu'est-ce qu'un package ?

Un **package** est un **dossier** qui contient plusieurs **modules** et un fichier spécial `__init__.py` (qui peut être vide).

Il sert à **organiser logiquement** un ensemble de modules liés entre eux.

Structure d'un package utilitaires

```
mon_projet/
 |-- utilitaires/
 | |-- __init__.py
 | |-- math_utils.py
 | |-- string_utils.py
 |
 |-- main.py
```

❖ Utilisation :

```
main.py
from utilitaires.math_utils import carre

print(carre(5)) # 25
```

Remarque : `__init__.py` permet à Python de reconnaître le dossier comme un **package** importable.

❖ 3. Importer un module ou un élément spécifique

| Syntaxe                                  | Utilisation                                 |
|------------------------------------------|---------------------------------------------|
| <code>import module</code>               | <code>module.nom_fonction()</code>          |
| <code>from module import fonction</code> | <code>fonction()</code>                     |
| <code>from module import *</code>        | Importe tout (⚠ à éviter en pratique)       |
| <code>import package.module</code>       | Accès via <code>package.module.nom()</code> |

## Bonnes pratiques

| Bonne pratique                                                         | Pourquoi ?                                       |
|------------------------------------------------------------------------|--------------------------------------------------|
| <input checked="" type="checkbox"/> Regrouper le code par thème        | Ex. paiement.py, utilisateur.py, base_donnees.py |
| <input checked="" type="checkbox"/> Ne pas surcharger les modules      | < 300 lignes ≈ + lisibilité                      |
| <input checked="" type="checkbox"/> Utiliser __init__.py               | Pour structurer vos packages                     |
| <input checked="" type="checkbox"/> Privilégier les imports explicites | Meilleure clarté et auto-complétion              |

## Exemple complet

### Arborescence

```
bibliotheque/
├── __init__.py
├── livre.py
├── chapitre.py
└── erreurs.py
main.py
```

### Contenu de erreurs.py

```
class LivreExisteError(Exception): pass
class ChapitreVideError(Exception): pass
```

### Contenu de livre.py

```
from .erreurs import LivreExisteError
```

```
class Livre:
 def __init__(self, titre):
 self.titre = titre
 self.chapitres = []
```

### Dans main.py

```
from bibliotheque.livre import Livre

livre = Livre("Python Avancé")
```

## Résumé

| Terme   | Définition                                           |
|---------|------------------------------------------------------|
| Module  | Fichier .py contenant du code importable             |
| Package | Dossier avec __init__.py contenant plusieurs modules |

Voici une **structure de projet modulaire en Python**, adaptée à un projet typique (ex. : système de réservation ou gestion d'application), **avec les fichiers essentiels déjà nommés et organisés**. Tu pourras l'utiliser comme base pour presque tout type de projet Python.

 **Structure proposée : projet\_reservation**

```

projet_reservation/
└── main.py # Point d'entrée du programme
└── requirements.txt # Dépendances éventuelles
└── README.md # Présentation du projet
└── reservation/
 ├── init_.py
 ├── systeme.py # Classe SystemeReservation
 ├── evenement.py # Classe Evenement
 ├── reservation.py # Classe Reservation
 └── exceptions.py # Exceptions personnalisées
└── utils/ # Fonctions utilitaires (facultatif)
 ├── init_.py
 └── validation.py # Email, saisie, etc.
└── tests/ # Tests unitaires (avec pytest ou unittest)
 ├── init_.py
 ├── test_evenement.py
 └── test_reservation.py

```

 **Détails des fichiers**

◆ **main.py**

```

from reservation.systeme import SystemeReservation

def main():
 systeme = SystemeReservation()
 # Lancer interface console ici

 if __name__ == "__main__":
 main()

```

◆ **reservation/exceptions.py**

```

class ReservationError(Exception): pass
class PlaceDejaReserveeError(ReservationError): pass
class DonneesInvalidesError(ReservationError): pass
class EvenementInexistantError(ReservationError): pass

```

◆ **reservation/evenement.py**

```
from .exceptions import PlaceDejaReserveeError, DonneesInvalidesError
from .reservation import Reservation

class Evenement:
 def __init__(self, titre, nb_places):
 self.titre = titre
 self.nb_places = nb_places
 self.reservations = {}

 def reserver(self, nom, email, place):
 # Vérification et ajout de réservation
 pass
```

◆ **reservation/systeme.py**

```
from .evenement import Evenement
from .exceptions import EvenementInexistantError

class SystemeReservation:
 def __init__(self):
 self.evenements = {}

 def ajouter_evenement(self, titre, nb_places):
 self.evenements[titre] = Evenement(titre, nb_places)
```

◆ **utils/validation.py (optionnel)**

```
import re

def email_valide(email):
 return re.match(r"^[^@]+@[^@]+\.[^@]+", email) is not None
```

◆ **tests/test\_evenement.py**

```
import pytest
from reservation.evenement import Evenement

def test_evenement_initialisation():
 event = Evenement("Concert", 10)
 assert event.nb_places == 10
```

Pour exécuter les tests :

```
pytest tests/
```



**Fichier requirements.txt (si besoin de bibliothèques externes)**

```
pytest
```



### Pour créer rapidement cette structure :

Tu peux copier cette commande dans un terminal :

```
mkdir -p projet_reservation/reservation projet_reservation/utils projet_reservation/tests
touch projet_reservation/main.py projet_reservation/requirements.txt projet_reservation/README.md
touch projet_reservation/reservation/__.init__.py,systeme.py,evenement.py,reservation.py,exceptions.py
touch projet_reservation/utils/__.init__.py,validation.py
touch projet_reservation/tests/__.init__.py,test_evenement.py,test_reservation.py}
```

-  Des exemples de **tests unitaires** pour modules/packages ?

Voici plusieurs **exemples concrets de tests unitaires** en Python pour un projet modulaire (comme un **système de réservation**) avec pytest ou unittest.

### Prérequis

Installe **pytest** (recommandé pour sa simplicité) :

```
pip install pytest
```

Place tes fichiers de test dans un dossier tests/, **en nommant les fichiers et fonctions avec le préfixe test\_**.

### Structure récapitulative

```
projet_reservation/
├── reservation/
│ ├── evenement.py
│ └── exceptions.py
└── tests/
 └── test_evenement.py
```

### Exemple 1 – Test de la classe Evenement

- ◆ Fichier : **reservation/evenement.py (extrait)**

```
from .exceptions import PlaceDejaReserveeError

class Evenement:
 def __init__(self, titre, nb_places):
 self.titre = titre
 self.nb_places = nb_places
 self.reservations = {}

 def reserver(self, place, nom):
 if place in self.reservations:
 raise PlaceDejaReserveeError("Cette place est déjà réservée.")
 self.reservations[place] = nom
```

◆ Fichier de test : `tests/test_evenement.py`

```
import pytest
from reservation.evenement import Evenement
from reservation.exceptions import PlaceDejaReserveeError

def test_initialisation_evenement():
 event = Evenement("Concert", 10)
 assert event.titre == "Concert"
 assert event.nb_places == 10
 assert event.reservations == {}

def test_reservation_place_valide():
 event = Evenement("Concert", 10)
 event.reserver(1, "Alice")
 assert event.reservations[1] == "Alice"

def test_reservation_place_deja_reservee():
 event = Evenement("Concert", 10)
 event.reserver(1, "Alice")
 with pytest.raises(PlaceDejaReserveeError):
 event.reserver(1, "Bob")
```

 Exemple 2 – Tester une exception personnalisée

◆ Fichier : `reservation/exceptions.py`

```
class ReservationError(Exception): pass
class DonneesInvalidesError(ReservationError): pass
```

◆ Test rapide :

```
from reservation.exceptions import DonneesInvalidesError
import pytest

def test_exception_personnalisee():
 with pytest.raises(DonneesInvalidesError):
 raise DonneesInvalidesError("Données invalides.")
```

 Exemple 3 – Tester un module `utils/validation.py`

◆ Fichier : `utils/validation.py`

```
def email_valide(email):
 return "@" in email and "." in email
```

◆ `tests/test_validation.py`

```
from utils.validation import email_valide

def test_email_valide():
 assert email_valide("test@example.com") is True
```

```
assert email_valide("testexample.com") is False
```

- 💡 Exécuter tous les tests :

`pytest`

- 📝 ou de façon ciblée :

`pytest tests/test_evenement.py`

### 🧠 Bonnes pratiques pour les tests unitaires

| Conseil                        | Pourquoi ?                                 |
|--------------------------------|--------------------------------------------|
| ✓ 1 fonctionnalité = 1 test    | Facile à isoler et comprendre              |
| ⌚ Utiliser pytest.raises()     | Pour tester les exceptions                 |
| 💡 Utiliser des noms explicites | Ex. test_reservation_place_deja_reservee() |
| 📁 Garder les tests dans /tests | Organisation claire du code                |

- 📋 Un script automatisé pour créer l'arborescence de projet ?

Voici un **script Python automatisé** qui crée **l'arborescence complète d'un projet modulaire** avec tous les dossiers et fichiers essentiels. Tu peux l'exécuter une fois pour générer la structure d'un projet prêt à développer (par exemple, un **système de réservation**).

### 🛠️ Script Python : generateur\_projet.py

```
import os

Nom du projet
nom_projet = "projet_reservation"

Structure à créer
structure = {
 nom_projet: [
 "main.py",
 "README.md",
 "requirements.txt",
 ("reservation", [
 "__init__.py",
 "systeme.py",
 "evenement.py",
 "reservation.py",
 "exceptions.py"
]),
 ("utils", [
 "__init__.py",
 "validation.py"
]),
]
}
```

```

("tests", [
 "__init__.py",
 "test_evenement.py",
 "test_reservation.py"
])
]
}

def creer_structure(base, contenu):
 for element in contenu:
 if isinstance(element, str):
 # Créer fichier
 chemin = os.path.join(base, element)
 with open(chemin, "w", encoding="utf-8") as f:
 f.write("") # Crée un fichier vide
 elif isinstance(element, tuple):
 dossier, sous_contenu = element
 chemin_dossier = os.path.join(base, dossier)
 os.makedirs(chemin_dossier, exist_ok=True)
 creer_structure(chemin_dossier, sous_contenu)

def main():
 print(f"📁 Création de la structure du projet : {nom_projet}")
 creer_structure(".", structure[nom_projet])
 print("✅ Structure générée avec succès.")

if __name__ == "__main__":
 main()

```

## 📁 Structure générée

```

projet_reservation/
├── main.py
├── README.md
├── requirements.txt
├── reservation/
│ ├── __init__.py
│ ├── systeme.py
│ ├── evenement.py
│ ├── reservation.py
│ └── exceptions.py
└── utils/
 ├── __init__.py
 └── validation.py
└── tests/
 ├── __init__.py
 ├── test_evenement.py
 └── test_reservation.py

```

## Utilisation

1. Copie le script dans un fichier generateur\_projet.py.
2. Exécute-le :

```
python generateur_projet.py
```

3. Tu peux ensuite remplir les fichiers générés avec le code souhaité.

Remplissage **automatique** de chaque fichier avec une **version de base fonctionnelle** du système de réservation,

 Voici le script complet que vous pouvez copier et exécuter localement sur votre machine :

```
import os

nom_projet = "projet_reservation"

fichiers_contenu = {
 "main.py": """from reservation.systeme import SystemeReservation

def main():
 systeme = SystemeReservation()
 systeme.ajouter_evenement("Concert", 5)

 while True:
 print("\n--- MENU ---")
 print("1. Réserver une place")
 print("2. Voir les événements")
 print("3. Quitter")

 choix = input("Votre choix : ").strip()

 if choix == "1":
 titre = input("Nom de l'événement : ")
 nom = input("Votre nom : ")
 email = input("Votre email : ")
 try:
 place = int(input("Numéro de place : "))
 systeme.reserver_place(titre, nom, email, place)
 except Exception as e:
 print(f"[Erreur] {e}")
 elif choix == "2":
 systeme.afficher_evenements()
 elif choix == "3":
 break
 else:
```

```

print("Choix invalide.")

if __name__ == "__main__":
 main()
",

"README.md": "# Projet de Réservation Python\n\nCe projet permet de gérer des réservations
d'événements en console.",
"requirements.txt": "pytest",

"reservation/__init__.py": "",

"reservation/systeme.py": "from .evenement import Evenement
from .exceptions import EvenementInexistantError

class SystemeReservation:
 def __init__(self):
 self.evenements = {}

 def ajouter_evenement(self, titre, nb_places):
 self.evenements[titre] = Evenement(titre, nb_places)

 def reserver_place(self, titre_evenement, nom, email, place):
 if titre_evenement not in self.evenements:
 raise EvenementInexistantError(f'L'événement "{titre_evenement}" n'existe pas.')
 self.evenements[titre_evenement].reserver(nom, email, place)

 def afficher_evenements(self):
 for titre, evenement in self.evenements.items():
 print(f"- {titre} ({evenement.nb_places} places)")
",

"reservation/evenement.py": "from .reservation import Reservation
from .exceptions import PlaceDejaReserveeError, DonneesInvalidesError

class Evenement:
 def __init__(self, titre, nb_places):
 self.titre = titre
 self.nb_places = nb_places
 self.reservations = {}

 def reserver(self, nom, email, place):
 if place in self.reservations:
 raise PlaceDejaReserveeError(f"La place {place} est déjà réservée.")
 if not (1 <= place <= self.nb_places):
 raise DonneesInvalidesError("Numéro de place invalide.")
 self.reservations[place] = Reservation(nom, email, place)
 print(f"✓ {nom} a réservé la place {place} pour {self.titre}.")
",

"reservation/reservation.py": "from .exceptions import DonneesInvalidesError

```

```

class Reservation:
 def __init__(self, nom, email, place):
 if not nom.strip() or "@" not in email:
 raise DonneesInvalidesError("Nom ou email invalide.")
 self.nom = nom
 self.email = email
 self.place = place
 """

 "reservation/exceptions.py": "class ReservationError(Exception): pass
class PlaceDejaReserveeError(ReservationError): pass
class DonneesInvalidesError(ReservationError): pass
class EvenementInexistantError(ReservationError): pass
",

"utils/__init__.py": "",

"utils/validation.py": "def email_valide(email):
 return "@" in email and "." in email
",

"tests/__init__.py": "",

"tests/test_evenement.py": "import pytest
from reservation.evenement import Evenement
from reservation.exceptions import PlaceDejaReserveeError

def test_reservation_succes():
 event = Evenement("Concert", 5)
 event.reserver("Alice", "alice@mail.com", 1)
 assert 1 in event.reservations

def test_reservation_place_deja_reservee():
 event = Evenement("Concert", 5)
 event.reserver("Bob", "bob@mail.com", 2)
 with pytest.raises(PlaceDejaReserveeError):
 event.reserver("Charlie", "charlie@mail.com", 2)
",

"tests/test_reservation.py": "import pytest
from reservation.reservation import Reservation
from reservation.exceptions import DonneesInvalidesError

def test_reservation_valide():
 r = Reservation("Jean", "jean@mail.com", 1)
 assert r.nom == "Jean"

def test_email_invalide():
 with pytest.raises(DonneesInvalidesError):
 Reservation("Luc", "email-invalide", 2)
"

```

```

"""
}

def creer_structure(chemin_base, fichiers):
 for chemin_relatif, contenu in fichiers.items():
 chemin_complet = os.path.join(chemin_base, chemin_relatif)
 os.makedirs(os.path.dirname(chemin_complet), exist_ok=True)
 with open(chemin_complet, "w", encoding="utf-8") as f:
 f.write(contenu)

 creer_structure(nom_projet, fichiers_contenu)
 print("☑ Structure de projet générée avec contenu fonctionnel.")

```

### Que faire ensuite ?

1. Copie ce code dans un fichier generateur.py.
2. Exécute-le dans un terminal avec :

python generateur.py

Cela créera un dossier projet\_reservation/ avec tous les fichiers prêts à l'emploi.

- **Structure d'un projet Python orienté objet**

Voici une **structure complète et professionnelle** d'un **projet Python orienté objet (POO)**. Elle est conçue pour un code **modulaire, testable et maintenable**, que ce soit pour une application console, API ou outil de traitement.

### Exemple : Projet OO de gestion (ex. : bibliothèque, réservation, boutique...)

```

mon_projet/
├── main.py # Point d'entrée de l'application
├── requirements.txt # Dépendances
└── README.md # Description du projet
|
└── mon_projet/ # Package principal de l'application
 ├── __init__.py
 ├── models/ # Entités (orienté objet)
 │ ├── __init__.py
 │ ├── utilisateur.py # Classe Utilisateur
 │ ├── livre.py # Classe Livre
 │ └── reservation.py # Classe Reservation
 |
 ├── services/ # Logique métier
 │ ├── __init__.py
 │ ├── gestion_livres.py # Méthodes de gestion de livres
 │ └── gestion_users.py # Méthodes de gestion utilisateurs

```

```

 | └── utils/ # Fonctions utilitaires
 | ├── __init__.py
 | └── validation.py
 |
 | └── exceptions/ # Exceptions personnalisées
 | ├── __init__.py
 | └── erreurs.py
 |
 └── interface/ # UI console ou interface API
 ├── __init__.py
 └── console.py
 |
 └── tests/ # Tests unitaires
 ├── __init__.py
 ├── test_utilisateur.py
 ├── test_livre.py
 └── test_services.py

```

### Description des couches

| Dossier     | Rôle                                                                 |
|-------------|----------------------------------------------------------------------|
| models/     | Représente les objets métier (classes avec attributs et méthodes)    |
| services/   | Gère les règles métiers (ex: emprunter un livre, valider une action) |
| utils/      | Contient des fonctions d'aide génériques                             |
| exceptions/ | Centralise les erreurs personnalisées                                |
| interface/  | Gère les entrées/sorties (console, API, GUI, etc.)                   |
| tests/      | Contient les tests unitaires                                         |

### Exemple de classe orientée objet (models/livre.py)

```

class Livre:
 def __init__(self, titre, auteur):
 self.titre = titre
 self.auteur = auteur
 self.disponible = True

 def emprunter(self):
 if not self.disponible:
 raise Exception("Livre non disponible")
 self.disponible = False

 def retourner(self):
 self.disponible = True

```

### Exemple de service (services/gestion\_livres.py)

```
from mon_projet.models.livre import Livre
```

```

class GestionLivres:
 def __init__(self):
 self.catalogue = []

 def ajouter_livre(self, titre, auteur):
 livre = Livre(titre, auteur)
 self.catalogue.append(livre)
 return livre

 def lister_livres(self):
 return [f"{livre.titre} par {livre.auteur}" for livre in self.catalogue]

```



### Bonnes pratiques

| Bonne pratique                        | Pourquoi ?                       |
|---------------------------------------|----------------------------------|
| 1 classe = 1 fichier                  | Facilite la lecture et les tests |
| Séparer métier et interface           | Indépendance UI / logique métier |
| Utiliser __init__.py                  | Pour déclarer les packages       |
| Éviter la logique métier dans main.py | Garder main.py propre            |
| Ajouter des tests unitaires           | Pour fiabiliser l'architecture   |

- **TP : Organisation d'un mini-projet en plusieurs fichiers/modules**

Parfait ! Voici un **TP complet** pour t'entraîner à organiser un mini-projet Python **en plusieurs fichiers/modules**, tout en pratiquant l'**approche orientée objet** et la **gestion modulaire**.

**Objectif : Créer une mini application de gestion d'une bibliothèque**

**Fonctionnalités :**

- Ajouter un livre.
- Emprunter un livre.
- Retourner un livre.
- Afficher les livres disponibles.
- Gérer les erreurs (livre introuvable, déjà emprunté...).



## Structure de projet proposée

```

bibliotheque_tp/
├── main.py # Point d'entrée
├── models/
│ ├── __init__.py
│ └── livre.py # Classe Livre
├── services/
│ ├── __init__.py
│ └── gestion_livres.py # Logique métier
├── exceptions/
│ ├── __init__.py
│ └── erreurs.py # Exceptions personnalisées
└── tests/
 └── test_gestion_livres.py # Test unitaire simple (pytest)

```



### Contenu des fichiers

#### **models/livre.py**

```

class Livre:
 def __init__(self, titre, auteur):
 self.titre = titre
 self.auteur = auteur
 self.disponible = True

 def __str__(self):
 return f"{self.titre} par {self.auteur} - {'Disponible' if self.disponible else 'Emprunté'}"

```

#### **exceptions/erreurs.py**

```

class LivreIntrouvableError(Exception):
 pass

class LivreDejaEmprunteError(Exception):
 pass

```

## services/gestion\_livres.py

```
from models.livre import Livre
from exceptions.erreurs import LivreIntrouvableError, LivreDejaEmprunteError

class GestionLivres:
 def __init__(self):
 self.catalogue = []

 def ajouter_livre(self, titre, auteur):
 self.catalogue.append(Livre(titre, auteur))

 def emprunter_livre(self, titre):
 livre = self._trouver_livre(titre)
 if not livre.disponible:
 raise LivreDejaEmprunteError(f"Le livre '{titre}' est déjà emprunté.")
 livre.disponible = False

 def retourner_livre(self, titre):
 livre = self._trouver_livre(titre)
 livre.disponible = True

 def lister_livres(self):
 return [str(livre) for livre in self.catalogue]

 def _trouver_livre(self, titre):
 for livre in self.catalogue:
 if livre.titre.lower() == titre.lower():
 return livre
 raise LivreIntrouvableError(f"Le livre '{titre}' n'existe pas.")
```

## main.py

```
from services.gestion_livres import GestionLivres
from exceptions.erreurs import *

def afficher_menu():
 print("\n--- Bibliothèque ---")
 print("1. Ajouter un livre")
 print("2. Emprunter un livre")
 print("3. Retourner un livre")
 print("4. Afficher tous les livres")
 print("5. Quitter")

def main():
 gestion = GestionLivres()

 while True:
 afficher_menu()
 choix = input("Choix : ").strip()
```

```

if choix == "1":
 titre = input("Titre : ")
 auteur = input("Auteur : ")
 gestion.ajouter_livre(titre, auteur)

elif choix == "2":
 titre = input("Titre à emprunter : ")
 try:
 gestion.emprunter_livre(titre)
 print("Livre emprunté.")
 except Exception as e:
 print(f"[Erreur] {e}")

elif choix == "3":
 titre = input("Titre à retourner : ")
 try:
 gestion.retourner_livre(titre)
 print("Livre retourné.")
 except Exception as e:
 print(f"[Erreur] {e}")

elif choix == "4":
 for livre in gestion.lister_livres():
 print(livre)

elif choix == "5":
 break
else:
 print("Choix invalide.")

if __name__ == "__main__":
 main()

```

---

## tests/test\_gestion\_livres.py

```

import pytest
from services.gestion_livres import GestionLivres
from exceptions.erreurs import LivreIntrouvableError

def test_ajout_et_recherche():
 gestion = GestionLivres()
 gestion.ajouter_livre("1984", "George Orwell")
 livres = gestion.lister_livres()
 assert any("1984" in livre for livre in livres)

def test_emprunter_livre_introuvable():

```

```
gestion = GestionLivres()
with pytest.raises(LivreIntrouvableError):
 gestion.emprunter_livre("Inconnu")
```

▶ Pour exécuter le projet :

```
python main.py
```

Et pour les tests :

```
pytest tests/
```

## ◆ Semaine 11 – Test unitaire et documentation

Voici une définition claire et concise de ces deux concepts fondamentaux en Python :

### Test unitaire (unit test)

#### Définition :

Un **test unitaire** est un **code qui vérifie automatiquement le bon fonctionnement d'une unité de code**, généralement une fonction ou une méthode.

 Le but est de **déetecter rapidement les erreurs** dans les petites briques logiques d'un programme, indépendamment du reste du système.

#### Exemple simple avec unittest :

```
fichier: maths.py
def addition(a, b):
 return a + b
fichier: test_maths.py
import unittest
from maths import addition

class TestMaths(unittest.TestCase):
 def test_addition(self):
 self.assertEqual(addition(2, 3), 5)

if __name__ == '__main__':
 unittest.main()
```

#### Autres outils populaires :

- pytest (plus concis, recommandé pour la plupart des projets)
- doctest (tests intégrés à la docstring)

### Documentation en Python

#### Définition :

La **documentation** en Python est un ensemble de textes (docstrings, fichiers README, etc.) qui **explique le fonctionnement, les paramètres et les objectifs** du code pour les développeurs.

## Docstring (documentation inline)

Un **docstring** est une chaîne de caractères spéciale placée directement dans une fonction, classe ou module pour décrire ce qu'elle fait.

```
def aire_rectangle(longueur, largeur):
 """
 Calcule l'aire d'un rectangle.

 Paramètres:
 - longueur (float): la longueur du rectangle
 - largeur (float): la largeur du rectangle

 Retourne:
 - float: aire calculée
 """
 return longueur * largeur
```

## Bonnes pratiques de documentation :

| Élément       | Outil conseillé           |
|---------------|---------------------------|
| Code Python   | Docstring ("""" ... """") |
| Projet        | README.md                 |
| API / Modules | Sphinx, MkDocs, pdoc      |

## Pour afficher la doc :

```
help(aire_rectangle)
print(aire_rectangle.__doc__)
```

## En résumé

| Concept       | Description courte                                 |
|---------------|----------------------------------------------------|
| Test unitaire | Vérifie automatiquement une fonction/méthode       |
| Documentation | Explique le code à travers des commentaires utiles |

- **Tests avec unittest**

Voici un guide complet pour écrire des **tests unitaires avec unittest** en Python.

### 1. Qu'est-ce que unittest ?

unittest est le **module standard de Python** pour l'écriture de **tests unitaires**. Il permet de vérifier que chaque fonction/méthode fonctionne correctement de manière isolée.

 **2. Exemple de projet simple**

► Fichier à tester : **calculs.py**

```
calculs.py
def addition(a, b):
 return a + b

def division(a, b):
 if b == 0:
 raise ValueError("Division par zéro impossible")
 return a / b
```

 **3. Créer un fichier de test : test\_calculs.py**

```
test_calculs.py
import unittest
from calculs import addition, division

class TestCalculs(unittest.TestCase):

 def test_addition(self):
 self.assertEqual(addition(2, 3), 5)
 self.assertEqual(addition(-1, 1), 0)

 def test_division(self):
 self.assertEqual(division(10, 2), 5)
 self.assertAlmostEqual(division(7, 3), 2.333, places=3)

 def test_division_par_zero(self):
 with self.assertRaises(ValueError):
 division(5, 0)

if __name__ == '__main__':
 unittest.main()
```

 **4. Exécuter les tests**

Dans ton terminal :

```
python test_calculs.py
Sortie typique si tous les tests passent :
...

Ran 3 tests in 0.001s

OK
```

## 5. Méthodes utiles de unittest

| Méthode                    | Utilité                                        |
|----------------------------|------------------------------------------------|
| assertEqual(a, b)          | Vérifie que $a == b$                           |
| assertNotEqual(a, b)       | Vérifie que $a != b$                           |
| assertTrue(x)              | Vérifie que $x$ est vrai                       |
| assertFalse(x)             | Vérifie que $x$ est faux                       |
| assertRaises(exception)    | Vérifie qu'une exception est levée             |
| assertAlmostEqual(a, b, x) | Vérifie que $a \approx b$ à $x$ décimales près |

## 6. Organisation recommandée

```
mon_projet/
├── calculs.py
└── test_calculs.py
```

Ou, dans un projet plus grand :

```
mon_projet/
├── src/
│ └── calculs.py
└── tests/
 └── test_calculs.py
```

- Documentation des classes et modules (docstrings, PEP 257)**

Excellente question ! Documenter ton code avec des **docstrings** claires et bien structurées est essentiel pour sa lisibilité, sa maintenance et sa réutilisation.

### Qu'est-ce qu'un docstring en Python ?

Un **docstring** est une chaîne de caractères placée juste après la déclaration d'un **module**, d'une **classe** ou **d'une fonction** pour expliquer son rôle.

C'est la méthode recommandée par Python pour documenter le code, conformément à la **PEP 257**.

### Règles de base de la PEP 257

| Règle PEP 257                             | Explication rapide                                    |
|-------------------------------------------|-------------------------------------------------------|
| Utiliser des chaînes triple-quote """     | Toujours avec triple guillemets (même pour une ligne) |
| Première phrase = résumé                  | Brève, sur une seule ligne, compréhensible seule      |
| Ligne vide après le résumé                | Pour séparer le résumé des détails supplémentaires    |
| Docstring placée juste après def ou class | Sinon, elle ne sera pas reconnue comme documentation  |

 **Exemple de documentation d'un module, classe et méthode**

◆ **Module : calcul.py**

```
"""
```

*Module calcul.py*

*Fournit des fonctions mathématiques de base : addition, soustraction, etc.*

```
"""
```

```
def addition(a, b):
 """Retourne la somme de a et b."""
 return a + b
```

◆ **Classe documentée**

```
class CompteBancaire:
```

```
"""
```

*Classe représentant un compte bancaire simple.*

*Attributs :*

*solde (float) : montant courant sur le compte.*

```
"""
```

```
def __init__(self, solde_initial=0):
```

```
"""
```

*Initialise le compte avec un solde donné.*

*Args:*

*solde\_initial (float) : montant de départ.*

```
"""
```

*self.solde = solde\_initial*

```
def deposer(self, montant):
```

```
"""
```

*Ajoute un montant au solde.*

*Args:*

*montant (float) : la somme à déposer.*

```
"""
```

*self.solde += montant*

 **Formats de docstring populaires**

Tu peux adopter un **style de docstring** standardisé pour que les outils comme Sphinx ou pdoc puissent générer automatiquement ta documentation :

✓ **Style Google**

```
def fonction(x, y):
 """
```

*Calcule quelque chose.*

**Args:**

*x (int): un entier  
y (float): un nombre décimal*

**Returns:**

*float: le résultat du calcul  
.....*

**✓ Style NumPy**

```
def fonction(x, y):
 """
 """
```

*Calcule quelque chose.*

**Parameters**

-----

*x : int*

*Un entier*

*y : float*

*Un nombre décimal*

**Returns**

-----

*float*

*Le résultat*

.....

**🔍 Accéder aux docstrings dans Python**

```
from calcul import addition
help(addition)
print(addition.__doc__)
```

**✓ En résumé**

| Élément     | Ce qu'il faut faire                                |
|-------------|----------------------------------------------------|
| Fonctions   | Résumé + paramètres + type de retour               |
| Classes     | Résumé + attributs + méthodes                      |
| Modules     | Résumé général au tout début du fichier            |
| Conventions | Suivre PEP 257 + style cohérent (Google, NumPy...) |

**• 🏠 Un template de classe Python documenté prêt à copier ?**

Bien sûr ! Voici un **template complet de classe Python** bien structuré et **documenté avec des docstrings** selon la [PEP 257](#) et le **style Google** — idéal pour vos projets orientés objet.

**📦 Template : Classe Python documentée**

```
class CompteBancaire:
```

```
 """
```

*Représente un compte bancaire simple.*

*Attributes:*

*titulaire (str): Le nom du titulaire du compte.*

*solde (float): Le montant actuellement disponible sur le compte.*

```
 """
```

```
def __init__(self, titulaire: str, solde_initial: float = 0.0):
```

```
 """
```

*Initialise un nouveau compte bancaire.*

*Args:*

*titulaire (str): Le nom du titulaire du compte.*

*solde\_initial (float, optional): Le solde de départ du compte. Par défaut à 0.0.*

```
 """
```

*self.titulaire = titulaire*

*self.solde = solde\_initial*

```
def déposer(self, montant: float):
```

```
 """
```

*Dépose un montant sur le compte.*

*Args:*

*montant (float): Le montant à déposer. Doit être positif.*

*Raises:*

*ValueError: Si le montant est négatif.*

```
 """
```

*if montant <= 0:*

*raise ValueError("Le montant à déposer doit être positif.")*

*self.solde += montant*

```
def retirer(self, montant: float):
```

```
 """
```

*Retire un montant du compte.*

*Args:*

*montant (float): Le montant à retirer.*

*Raises:*

*ValueError: Si le montant est négatif ou si le solde est insuffisant.*

```
 """
```

*if montant <= 0:*

*raise ValueError("Le montant à retirer doit être positif.")*

*if montant > self.solde:*

*raise ValueError("Fonds insuffisants.")*

*self.solde -= montant*

```
def afficher_solde(self) -> float:
```

```
"""
 Retourne le solde actuel du compte.

 Returns:
 float: Le solde du compte.
 """
 return self.solde

def __str__(self):
 """
 Représentation lisible du compte bancaire.

 Returns:
 str: Description du compte avec le titulaire et le solde.
 """
 return f"Compte de {self.titulaire} - Solde: {self.solde:.2f} €"
```

Ce que ce template respecte :

| Bonnes pratiques                      | Appliquées <input checked="" type="checkbox"/> |
|---------------------------------------|------------------------------------------------|
| Docstring de classe avec attributs    | <input checked="" type="checkbox"/>            |
| Docstring de chaque méthode avec Args | <input checked="" type="checkbox"/>            |
| Gestion des erreurs avec Raises       | <input checked="" type="checkbox"/>            |
| Types de retour (Returns) documentés  | <input checked="" type="checkbox"/>            |
| Annotations de types (: float)        | <input checked="" type="checkbox"/>            |
| Lisibilité et modularité              | <input checked="" type="checkbox"/>            |

- Un exemple de projet complet avec docstrings + génération de doc HTML ?

Parfait ! Voici un **exemple de projet Python complet** qui inclut :

- ✓ Code organisé en modules et classes,
- ✓ Docstrings bien structurés (PEP 257, style **Google**),
- ✓ ■ **Génération de documentation HTML avec Sphinx.**

🎯 Objectif du projet : gestion de comptes bancaires

🏗 Structure du projet

```
banque_project/
 |-- banque/
 | |-- __init__.py
 | '-- compte.py # Classe principale avec docstrings
```

```

├── main.py # Fichier d'utilisation
├── docs/ # Dossier de documentation (Sphinx)
├── requirements.txt # Dépendances (inclut sphinx)
└── README.md

```

## 1. Fichier banque/compte.py

....

Module de gestion de comptes bancaires.

Contient la classe `CompteBancaire` avec dépôt, retrait et solde.

....

*class CompteBancaire:*

....

*Représente un compte bancaire avec opérations de base.*

*Attributes:*

*titulaire (str): Nom du titulaire du compte.*

*solde (float): Solde courant du compte.*

....

*def \_\_init\_\_(self, titulaire: str, solde\_initial: float = 0.0):*

....

*Initialise le compte avec un titulaire et un solde initial.*

*Args:*

*titulaire (str): Nom du titulaire.*

*solde\_initial (float): Solde de départ.*

....

*self.titulaire = titulaire*

*self.solde = solde\_initial*

*def déposer(self, montant: float):*

....

*Dépose un montant sur le compte.*

*Args:*

*montant (float): Montant à ajouter.*

*Raises:*

*ValueError: Si le montant est négatif.*

....

*if montant <= 0:*

*raise ValueError("Le montant doit être positif.")*

*self.solde += montant*

```
def retirer(self, montant: float):
 """
 Retire un montant du compte.

 Args:
 montant (float): Montant à retirer.

 Raises:
 ValueError: Si le montant dépasse le solde.

 """
 if montant > self.solde:
 raise ValueError("Fonds insuffisants.")
 self.solde -= montant

def afficher_solde(self) -> float:
 """
 Retourne le solde actuel.

 Returns:
 float: Solde disponible.

 """
 return self.solde
```

## ► 2. Fichier main.py

```
from banque.compte import CompteBancaire

def main():
 compte = CompteBancaire("Alice", 100)
 compte.deposer(50)
 compte.retirer(30)
 print(f"Solde de {compte.titulaire} : {compte.afficher_solde()} €")

if __name__ == "__main__":
 main()
```

## ► 3. Générer la documentation HTML avec Sphinx

Étapes simples :

1. **Installer sphinx :**
  2. pip install sphinx
  3. **Créer la structure de doc :**
  4. sphinx-quickstart docs
- Réponds aux questions (le nom du projet, l'auteur, etc.)
5. **Activer l'autodoc dans docs/conf.py :**
  6. extensions = ['sphinx.ext.autodoc']

7. **Créer un fichier index.rst ou modifier celui généré :**

8. Bienvenue dans la documentation de Banque

9. =====

10.

11. .. automodule:: banque.compte

12. :members:

13. :undoc-members:

14. :show-inheritance:

15. **Générer la documentation HTML :**

16. sphinx-apidoc -o docs ./banque

17. cd docs

18. make html

19. **Consulter la doc générée :**

Ouvre docs/\_build/html/index.html dans un navigateur 

### Résultat

Tu obtiens une documentation professionnelle au format HTML, avec toutes les classes et méthodes bien listées et commentées, automatiquement extraites des **docstrings**.

### Fichier requirements.txt

sphinx>=7.0

- TP : Ajouter des tests et documenter un mini-projet

◆ **Semaine 12 – Projet final (partie 1)**

- Lancement du projet final (par binôme ou groupe)
- Sujet au choix : gestion bibliothèque, réseau social, mini-jeu, gestion d'inventaire, etc.
- Cahier des charges, modélisation UML, premières classes

 **Évaluation**

-  TP réguliers : 30 %
-  QCMs/interrogations courtes : 10 %
-  Projet final (code + rapport + soutenance) : 40 %
-  Examen final (théorie + pratique) : 20 %