Exercice 1
Gestion d’un catalogue de produits

```
# Cet exercice vise à mettre en pratique les notions de classes, encapsulation,
validation des
# données et manipulation de listes d’objets.
#
# 1. Créer une classe Produit avec les attributs privés suivants :
#    - nom : nom du produit (chaîne de caractères)
#    - prix : prix du produit (nombre réel)
#    - quantite : quantité disponible en stock (entier)
#
# 2. Définir un constructeur avec paramètres pour initialiser tous les attributs.
#
# 3. Implémenter les getters et setters pour chaque attribut, en garantissant
que :
#    - le prix est strictement positif,
#    - la quantité est un entier positif ou nul.
#
# 4. Ajouter une méthode afficher_infos() qui affiche les informations complètes
du produit.
#
# 5. Créer une classe CatalogueProduits contenant une liste de produits et
permettant :
#    - d’ajouter un produit,
#    - d’afficher tous les produits,
#    - de calculer et retourner la valeur totale du stock.
# 6. Écrire un programme principal pour tester toutes les fonctionnalités.

class Produit:
    def __init__(self, nom, prix, quantite):
        self._nom = nom
        if prix > 0:
            self._prix = prix
        else:
            raise ValueError("Le prix doit être strictement positif.")
        if isinstance(quantite, int) and quantite >= 0:
            self._quantite = quantite
        else:
            raise ValueError("La quantité doit être un entier positif ou nul.")

    # Getters
    def get_nom(self):
```

```
        return self._nom

    def get_prix(self):
        return self._prix

    def get_quantite(self):
        return self._quantite

    # Setters
    def set_nom(self, nom):
        self._nom = nom

    def set_prix(self, prix):
        if prix > 0:
            self._prix = prix
        else:
            print("Erreur : Le prix doit être strictement positif.")

    def set_quantite(self, quantite):
        if isinstance(quantite, int) and quantite >= 0:
            self._quantite = quantite
        else:
            print("Erreur : La quantité doit être un entier positif ou nul.")

    def afficher_infos(self):
        print(f"Produit: {self._nom}")
        print(f"Prix: {self._prix} €")
        print(f"Quantité en stock: {self._quantite}")
        print("-" * 20)


class CatalogueProduits:
    def __init__(self):
        self.produits = []

    def ajouter_produit(self, produit):
        self.produits.append(produit)
        print(f"Produit '{produit.get_nom()}' ajouté au catalogue.")

    def afficher_produits(self):
        if not self.produits:
            print("Le catalogue est vide.")
            return
        print("--- Contenu du Catalogue ---")
        for produit in self.produits:
```

```
            produit.afficher_infos()
        print("--------------------------")

    def valeur_totale_stock(self):
        valeur_totale = sum(p.get_prix() * p.get_quantite() for p in
self.produits)
        return valeur_totale

# Programme principal pour tester
# Création de produits
try:
    produit1 = Produit("Ordinateur Portable", 1200.50, 10)
    produit2 = Produit("Smartphone", 800.00, 25)
    produit3 = Produit("Clavier", 75.99, 50)
except ValueError as e:
    print(f"Erreur lors de la création d'un produit : {e}")
    exit()

# Création du catalogue
catalogue = CatalogueProduits()

# Ajout des produits au catalogue
catalogue.ajouter_produit(produit1)
catalogue.ajouter_produit(produit2)
catalogue.ajouter_produit(produit3)
print("\n")

# Affichage de tous les produits du catalogue
catalogue.afficher_produits()

# Calcul et affichage de la valeur totale du stock
valeur_stock = catalogue.valeur_totale_stock()
print(f"La valeur totale du stock est de : {valeur_stock:.2f} €")

# Test des setters avec des valeurs invalides
print("\n--- Test des setters avec des valeurs invalides ---")
produit1.set_prix(-50)
produit2.set_quantite(-5)
produit2.set_quantite(15.5)
print("------------------------------------------------")

# Affichage des informations après tentative de modification
print("\nInformations du produit 1 après tentative de modification invalide du
prix:")
produit1.afficher_infos()
```

Exercice 2
Gestion d’une bibliothèque :

```
# On souhaite développer une application Python pour gérer des personnes, des adhérents, des
# auteurs et des livres dans une bibliothèque.
# 1. Créer une classe Personne avec les attributs privés : nom, prenom, email, tel et age.
#    - Ajouter un constructeur avec paramètres.
#    - Définir la méthode __str__().
#
# 2. Créer une classe Adherent héritant de Personne, avec un attribut num_adherent, et
#    redéfinir la méthode __str__().
#
# 3. Créer une classe Auteur héritant de Personne, avec un attribut num_auteur, et redéfinir la
#    méthode __str__().
#
# 4. Créer une classe Livre avec les attributs : isbn, titre et auteur (objet Auteur).
#    - Définir la méthode __str__().
#
# 5. Écrire un programme principal qui :
#    - crée un adhérent,
#    - crée un auteur,
#    - crée un livre écrit par cet auteur,
#    - affiche les informations de l’adhérent et du livre.

class Personne:
    def __init__(self, nom, prenom, email, tel, age):
        self._nom = nom
        self._prenom = prenom
        self._email = email
        self._tel = tel
        self._age = age

    def __str__(self):
        return f"Nom: {self._nom}, Prénom: {self._prenom}, Email: {self._email}, Tel: {self._tel}, Age: {self._age}"
```

```python
class Adherent(Personne):
    def __init__(self, nom, prenom, email, tel, age, num_adherent):
        super().__init__(nom, prenom, email, tel, age)
        self._num_adherent = num_adherent

    def __str__(self):
        return f"Adhérent N°{self._num_adherent} - {super().__str__()}"


class Auteur(Personne):
    def __init__(self, nom, prenom, email, tel, age, num_auteur):
        super().__init__(nom, prenom, email, tel, age)
        self._num_auteur = num_auteur

    def __str__(self):
        return f"Auteur N°{self._num_auteur} - {super().__str__()}"


class Livre:
    def __init__(self, isbn, titre, auteur):
        self._isbn = isbn
        self._titre = titre
        self._auteur = auteur

    def __str__(self):
        return f"Livre: {self._titre}\nISBN: {self._isbn}\n{self._auteur}"


# Programme principal
# Crée un adhérent
adherent1 = Adherent("Dupont", "Jean", "jean.dupont@email.com", "0123456789", 35,
"AD001")

# Crée un auteur
auteur1 = Auteur("Hugo", "Victor", "victor.hugo@email.com", "9876543210", 83,
"AU001")

# Crée un livre écrit par cet auteur
livre1 = Livre("978-2-07-041311-9", "Les Misérables", auteur1)

# Affiche les informations de l’adhérent et du livre
print("--- Informations de l'adhérent ---")
print(adherent1)
print("\n--- Informations du livre ---")
print(livre1)
```

Exercice 3
Hiérarchie de véhicules

```
# On souhaite modéliser différents types de véhicules afin d'illustrer l'héritage
et le
# polymorphisme.
#
# 1. Créer une classe de base Vehicule avec :
#    - nom : nom du véhicule
#    - prix : prix du véhicule
#    - emettre_son() : affiche un son générique
#    - afficher_informations() : affiche le nom et le prix
# 2. Créer les sous-classes suivantes :
#    - Voiture : attributs modele et annee, redéfinit emettre_son()
#    - Moto : attributs marque et puissance, redéfinit emettre_son()
#    - Avion : attributs compagnie et vitesse_max, redéfinit emettre_son()
# 3. Écrire un programme principal qui :
#    - crée des objets Voiture, Moto et Avion,
#    - appelle emettre_son() pour chaque véhicule,
#    - affiche toutes les informations.

class Vehicule:
    def __init__(self, nom, prix):
        self._nom = nom
        self._prix = prix

    def emettre_son(self):
        print("Le véhicule émet un son générique.")

    def afficher_informations(self):
        print(f"Nom: {self._nom}")
        print(f"Prix: {self._prix:.2f} €")


class Voiture(Vehicule):
    def __init__(self, nom, prix, modele, annee):
        super().__init__(nom, prix)
        self._modele = modele
        self._annee = annee

    def emettre_son(self):
        print("Vroum vroum !")
```

```python
    def afficher_informations(self):
        print("--- Informations de la Voiture ---")
        super().afficher_informations()
        print(f"Modèle: {self._modele}")
        print(f"Année: {self._annee}")


class Moto(Vehicule):
    def __init__(self, nom, prix, marque, puissance):
        super().__init__(nom, prix)
        self._marque = marque
        self._puissance = puissance

    def emettre_son(self):
        print("Brap brap !")

    def afficher_informations(self):
        print("--- Informations de la Moto ---")
        super().afficher_informations()
        print(f"Marque: {self._marque}")
        print(f"Puissance: {self._puissance} ch")


class Avion(Vehicule):
    def __init__(self, nom, prix, compagnie, vitesse_max):
        super().__init__(nom, prix)
        self._compagnie = compagnie
        self._vitesse_max = vitesse_max

    def emettre_son(self):
        print("Fshhhhhhhhhhhhh !")

    def afficher_informations(self):
        print("--- Informations de l'Avion ---")
        super().afficher_informations()
        print(f"Compagnie: {self._compagnie}")
        print(f"Vitesse maximale: {self._vitesse_max} km/h")


# Programme principal
# Création des objets
ma_voiture = Voiture("Voiture de sport", 50000.00, "Porsche 911", 2023)
ma_moto = Moto("Moto de course", 15000.00, "Ducati", 200)
mon_avion = Avion("Avion de ligne", 100000000.00, "Air France", 950)

vehicules = [ma_voiture, ma_moto, mon_avion]
```

```python
# Appel des méthodes pour chaque véhicule
for vehicule in vehicules:
    vehicule.afficher_informations()
    vehicule.emettre_son()
    print("-" * 30)
```

Exercice 4
Gestion des paiements (Polymorphisme)

```
# On souhaite développer une application permettant de gérer différents moyens de
paiement.
#
# 1. Créer une classe de base Paiement avec la méthode
effectuer_paiement(montant).
#
# 2. Créer les classes dérivées CarteCredit et PayPal héritant de Paiement :
#    - CarteCredit possède un attribut numero_carte
#    - PayPal possède un attribut email
#    - Chaque classe redéfinit effectuer_paiement(montant)
#
# 3. Créer une classe Commande avec :
#    - montant_commande
#    - moyen_paiement (objet de type Paiement)
#    - une méthode process_payment() utilisant le polymorphisme
#
# 4. Dans un programme principal :
#    - créer des commandes avec différents moyens de paiement
#    - appeler process_payment() et observer le comportement polymorphe.

from abc import ABC, abstractmethod

# 1. Classe de base Paiement
class Paiement(ABC):
    @abstractmethod
    def effectuer_paiement(self, montant):
        """
        Méthode abstraite pour effectuer un paiement.
        Cette méthode doit être implémentée par les classes filles.
        """
        pass

# 2. Classes dérivées
class CarteCredit(Paiement):
    def __init__(self, numero_carte, nom_titulaire):
        self.numero_carte = numero_carte
        self.nom_titulaire = nom_titulaire

    def effectuer_paiement(self, montant):
        print(f"Paiement de {montant:.2f} € effectué par carte de crédit.")
```

```
        print(f"Numéro de carte : **** **** **** {self.numero_carte[-4:]}")
        print(f"Titulaire : {self.nom_titulaire}")

class PayPal(Paiement):
    def __init__(self, email):
        self.email = email

    def effectuer_paiement(self, montant):
        print(f"Paiement de {montant:.2f} € effectué via PayPal.")
        print(f"Compte PayPal : {self.email}")

# 3. Classe Commande
class Commande:
    def __init__(self, montant_commande, moyen_paiement: Paiement):
        self.montant_commande = montant_commande
        self.moyen_paiement = moyen_paiement

    def process_payment(self):
        print(f"\nTraitement d'une commande de {self.montant_commande:.2f} €...")
        self.moyen_paiement.effectuer_paiement(self.montant_commande)

# 4. Programme principal
# Création des moyens de paiement
paiement_cc = CarteCredit(numero_carte="1234567890123456", nom_titulaire="Jean
Dupont")
paiement_paypal = PayPal(email="jean.dupont@example.com")

# Création des commandes avec différents moyens de paiement
commande1 = Commande(montant_commande=150.75, moyen_paiement=paiement_cc)
commande2 = Commande(montant_commande=89.99, moyen_paiement=paiement_paypal)

# Appeler process_payment() et observer le comportement polymorphe
commande1.process_payment()
commande2.process_payment()
```