Atelier Progressif : Python Orienté Objet Avancé

Exercice 1. Création de classes et d'objets

Crée une classe `Personne` avec les attributs `nom`, `âge`, et `ville`. Ajoute un constructeur pour initialiser ces attributs et une méthode `afficher_informations` qui affiche les détails d'une personne.

Exercice 2. Gestion des attributs

Ajoute un attribut `profession` à la classe `Personne` qui est initialisé par défaut à `"Inconnue"`. Modifie cet attribut pour un objet spécifique via une méthode `modifier_profession`.

Exercice 3. Interaction entre objets

Crée une classe `Adresse` avec les attributs `rue`, `code_postal`, et `ville`. Modifie la classe `Personne` pour inclure un attribut `adresse` (instance de la classe `Adresse`). Implémente une méthode pour afficher les informations complètes d'une personne avec son adresse.

Exercice 4. Encapsulation des attributs

Modifie la classe `Personne` pour rendre les attributs `nom` et `âge` privés. Ajoute des **getters** et **setters** pour y accéder et les modifier.

Exercice 5. Méthodes statiques

Ajoute une méthode statique `est_majeur` dans la classe `Personne` qui prend un âge en paramètre et retourne `True` si l'âge est supérieur ou égal à 18, et `False` sinon.

Exercice 6. Méthode de classe

Ajoute une méthode de classe `creer_personne_vide` qui retourne une instance de `Personne` avec des valeurs par défaut (`nom="Inconnu"`, `âge=0`, `ville="Inconnue"`).

Exercice 7. Héritage - Classe simple

Crée une classe `Etudiant` qui hérite de la classe `Personne`. Ajoute un attribut `niveau_etudes` et une méthode `afficher_etudes` qui affiche le niveau d'études.

Exercice 8. Héritage - Méthodes redéfinies

Redéfinis la méthode `afficher_informations` dans la classe `Etudiant` pour inclure le niveau d'études en plus des informations de la personne.

Exercice 9. Héritage multiple

Crée une classe `Employe` qui hérite de `Personne`. Ajoute un attribut `salaire` et une méthode `afficher_salaire`. Ensuite, crée une classe `EtudiantEmploye` qui hérite à la fois de `Etudiant` et `Employe`. Vérifie les conflits d'héritage si la méthode `afficher_informations` existe dans les deux parents.

Exercice 10. Polymorphisme par inclusion

Crée une méthode générique `presentation` qui accepte un objet de type `Personne` et affiche les informations de l'objet. Teste la méthode avec des instances de `Personne`, `Etudiant`, et `Employe`.

Exercice 11. Polymorphisme par surcharge

Ajoute une méthode `calculer_salaire` dans la classe `Employe`. Surcharge cette méthode pour qu'elle accepte un paramètre supplémentaire `prime` (facultatif). Si une prime est donnée, elle est ajoutée au salaire total.

Exercice 12. Polymorphisme par redéfinition

Ajoute une méthode `etudier` dans la classe `Etudiant` qui affiche `"L'étudiant étudie"`. Redéfinis cette méthode dans une classe dérivée `EtudiantEnLigne` pour afficher `"L'étudiant étudie en ligne"`.

Exercice 13. Gestion des erreurs

Modifie les setters pour lever une exception si l'âge est négatif ou si le salaire d'un employé est inférieur au salaire minimum légal.

Exercice 14. Relations entre classes

Crée une classe `Entreprise` qui contient une liste d'objets `Employe`. Implémente une méthode `ajouter_employe` pour ajouter un employé, et une méthode `afficher_tous_les_employes` pour afficher les détails de tous les employés de l'entreprise.

Exercice 19. Gestion des fichiers

Ajoute une méthode à la classe `Personne` pour sauvegarder ses informations dans un fichier texte. Implémente également une méthode pour charger une personne à partir d'un fichier.

Solutions

Solution 1. Création de classes et d'objets

```
class Personne:
    def __init__(self, nom, age, ville):
        self.nom = nom
        self.age = age
        self.ville = ville

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}, Ville: {self.ville}")

# Exemple d'utilisation
p1 = Personne("Alice", 30, "Paris")
p1.afficher_informations()
```

Solution 2. Gestion des attributs

```
class Personne:
    def __init__(self, nom, age, ville):
        self.nom = nom
        self.age = age
        self.ville = ville
        self.profession = "Inconnue"

    def modifier_profession(self, profession):
        self.profession = profession

# Exemple d'utilisation
p1 = Personne("Bob", 25, "Lyon")
p1.modifier_profession("Ingénieur")
print(p1.profession)
```

Solution 3. Interaction entre objets

```
class Adresse:
    def __init__(self, rue, code_postal, ville):
        self.rue = rue
        self.code_postal = code_postal
        self.ville = ville

class Personne:
    def __init__(self, nom, age, ville):
        self.nom = nom
        self.age = age
        self.adresse = Adresse("Rue inconnue", "00000", ville)

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}, Adresse: {self.adresse.rue},
{self.adresse.code_postal}, {self.adresse.ville}")

# Exemple d'utilisation
p1 = Personne("Charlie", 22, "Marseille")
p1.adresse.rue = "Rue du port"
p1.adresse.code_postal = "13000"
p1.afficher_informations()
```

Solution 4. Encapsulation des attributs

```
class Personne:
    def __init__(self, nom, age):
        self.__nom = nom
        self.__age = age

    def get_nom(self):
        return self.__nom

    def set_nom(self, nom):
        self.__nom = nom

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age < 0:
            raise ValueError("L'âge ne peut pas être négatif.")
        self.__age = age

# Exemple d'utilisation
p1 = Personne("David", 28)
```

```
print(p1.get_nom())
p1.set_age(30)
print(p1.get_age())
```

Solution 5. Méthodes statiques

```
class Personne:
    @staticmethod
    def est_majeur(age):
        return age >= 18

# Exemple d'utilisation
print(Personne.est_majeur(16))  # False
print(Personne.est_majeur(20))  # True
```

---

6. Méthode de classe

```
class Personne:
    def __init__(self, nom, age, ville):
        self.nom = nom
        self.age = age
        self.ville = ville

    @classmethod
    def creer_personne_vide(cls):
        return cls("Inconnu", 0, "Inconnue")

# Exemple d'utilisation
p1 = Personne.creer_personne_vide()
print(p1.nom, p1.age, p1.ville)
```

---

7. Héritage - Classe simple

```
class Etudiant(Personne):
    def __init__(self, nom, age, ville, niveau_etudes):
        super().__init__(nom, age, ville)
        self.niveau_etudes = niveau_etudes

    def afficher_etudes(self):
        print(f"Niveau d'études: {self.niveau_etudes}")

# Exemple d'utilisation
e1 = Etudiant("Eve", 19, "Toulouse", "Licence 1")
e1.afficher_etudes()
```

---

8. Héritage - Méthodes redéfinies

```
# Classe de base Personne
class Personne:
    def __init__(self, nom, age, ville):
        self.nom = nom
        self.age = age
        self.ville = ville

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}, Ville: {self.ville}")

# Classe Etudiant qui hérite de Personne
class Etudiant(Personne):
    def __init__(self, nom, age, ville, niveau_etudes):
        # Appel au constructeur de la classe parente
        super().__init__(nom, age, ville)
        self.niveau_etudes = niveau_etudes

    # Redéfinition de la méthode afficher_informations
    def afficher_informations(self):
        # Appeler la méthode de la classe parente pour réutiliser son affichage
        super().afficher_informations()
        print(f"Niveau d'études: {self.niveau_etudes}")

etudiant1 = Etudiant("Alice", 22, "Lyon", "Master 1")
etudiant1.afficher_informations()
```

---

9. Héritage multiple

```
class Employe(Personne):
    def __init__(self, nom, age, ville, salaire):
        super().__init__(nom, age, ville)
        self.salaire = salaire

    def afficher_salaire(self):
        print(f"Salaire: {self.salaire}")

class EtudiantEmploye(Etudiant, Employe):
    pass

# Exemple d'utilisation
ee1 = EtudiantEmploye("Frank", 21, "Bordeaux", "Master 2")
```

---

10. Polymorphisme par inclusion

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

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}")

class Etudiant(Personne):
    def __init__(self, nom, age, niveau_etudes):
        super().__init__(nom, age)
        self.niveau_etudes = niveau_etudes

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}, Niveau d'études: {self.niveau_etudes}")

class Employe(Personne):
    def __init__(self, nom, age, salaire):
        super().__init__(nom, age)
        self.salaire = salaire

    def afficher_informations(self):
        print(f"Nom: {self.nom}, Âge: {self.age}, Salaire: {self.salaire}")

def presentation(obj):
    obj.afficher_informations()

# Exemple d'utilisation
p1 = Personne("Alice", 30)
e1 = Etudiant("Bob", 20, "Licence")
emp1 = Employe("Charlie", 35, 40000)
```

```
presentation(p1)
presentation(e1)
presentation(emp1)
```

---

11. Polymorphisme par surcharge

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

    def calculer_salaire(self, prime=0):
        return self.salaire + prime

# Exemple d'utilisation
emp1 = Employe("David", 3000)
print(emp1.calculer_salaire())       # 3000
print(emp1.calculer_salaire(500))   # 3500
```

---

12. Polymorphisme par redéfinition

```
class Etudiant:
    def etudier(self):
        print("L'étudiant étudie.")

class EtudiantEnLigne(Etudiant):
    def etudier(self):
        print("L'étudiant étudie en ligne.")

# Exemple d'utilisation
e1 = Etudiant()
e2 = EtudiantEnLigne()

e1.etudier()
e2.etudier()
```

---

13. Gestion des erreurs

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.set_age(age)
```

```
    def set_age(self, age):
        if age < 0:
            raise ValueError("L'âge ne peut pas être négatif.")
        self.age = age

class Employe(Personne):
    def __init__(self, nom, age, salaire):
        super().__init__(nom, age)
        self.set_salaire(salaire)

    def set_salaire(self, salaire):
        if salaire < 1500:
            raise ValueError("Le salaire doit être supérieur ou égal à 1500.")
        self.salaire = salaire

# Exemple d'utilisation
try:
    emp1 = Employe("Alice", 25, 1200)
except ValueError as e:
    print(e)
```

---

14. Relations entre classes

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

class Entreprise:
    def __init__(self):
        self.employes = []

    def ajouter_employe(self, employe):
        self.employes.append(employe)

    def afficher_tous_les_employes(self):
        for emp in self.employes:
            print(f"Nom: {emp.nom}, Salaire: {emp.salaire}")

# Exemple d'utilisation
e1 = Employe("Alice", 3000)
e2 = Employe("Bob", 4000)

entreprise = Entreprise()
entreprise.ajouter_employe(e1)
```

```
entreprise.ajouter_employe(e2)

entreprise.afficher_tous_les_employes()
```

---

15. Gestion des fichiers

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

    def sauvegarder(self, fichier):
        with open(fichier, "w") as f:
            f.write(f"{self.nom}\n{self.age}")

    @staticmethod
    def charger(fichier):
        with open(fichier, "r") as f:
            nom = f.readline().strip()
            age = int(f.readline().strip())
            return Personne(nom, age)

# Exemple d'utilisation
p1 = Personne("Alice", 30)
p1.sauvegarder("personne.txt")

p2 = Personne.charger("personne.txt")
print(p2.nom, p2.age)
```

---

20. Composition et tests

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

class Ecole:
    def __init__(self):
        self.etudiants = []

    def ajouter_etudiant(self, etudiant):
        self.etudiants.append(etudiant)

    def calculer_moyenne_age(self):
        total_age = sum(etudiant.age for etudiant in self.etudiants)
```

```
        return total_age / len(self.etudiants)

# Exemple d'utilisation
e1 = Etudiant("Alice", 20)
e2 = Etudiant("Bob", 22)

ecole = Ecole()
ecole.ajouter_etudiant(e1)
ecole.ajouter_etudiant(e2)

print(ecole.calculer_moyenne_age())
```
