POO sous Python séance num 4

Enseignant : GHNEM

Séance numéro 4 : Fondamentaux de la POO

Sommaire

1) Méthodes d'instance : Définition et appel de méthodes, Utilisation de self pour accéder aux attributs, Exemples de méthodes métiers

2) Encapsulation et visibilité : Notion d'attributs privés (__attribut), Getter et Setter, Contrôle d'accès aux données

3) Méthodes **str**, **repr** : Représentation des objets

4) Abstraction : Cacher la complexité, Interface publique vs implémentation interne, Exemples concrets

5) Échange de données entre instances de classes différentes

1) Méthodes d'instance - Définition et appel de méthodes

- Une méthode d'instance est une fonction définie dans une classe
- Elle prend toujours self comme premier paramètre
- Elle peut accéder et modifier les attributs de l'objet
- Appel : objet.nom_méthode(paramètres)

- **Exemple : Méthodes d'instance**

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

    def afficher_infos(self):
        print(f"Produit: {self.nom}, Prix: {self.prix} DH")

    def appliquer_remise(self, pourcentage):
        remise = self.prix * pourcentage / 100
        self.prix -= remise
```

- **Utilisation des méthodes**

```
# Création d'un objet
p1 = Produit("Laptop", 800)

# Appel des méthodes
p1.afficher_infos()       # Produit: Laptop, Prix: 800 DH
p1.appliquer_remise(10)   # Application de 10% de remise
p1.afficher_infos()       # Produit: Laptop, Prix: 720 DH
```

- **Exercice : Méthodes d'instance**
  Créez une classe CompteBancaire avec :
- Attributs : numero_compte, solde
- Méthodes : deposer(montant), retirer(montant), afficher_solde()
  La méthode retirer doit vérifier si le solde est suffisant

- **Solution : Méthodes d'instance**

```
class CompteBancaire:
    def __init__(self, numero_compte, solde=0):
        self.numero_compte = numero_compte
        self.solde = solde

    def deposer(self, montant):
        self.solde += montant

    def retirer(self, montant):
        if montant <= self.solde:
            self.solde -= montant
        else:
            print("Solde insuffisant")

    def afficher_solde(self):
        print(f"Solde: {self.solde} DH")
```

- **un exemple d'utilisation de la classe CompteBancaire :**

```
compte1 = CompteBancaire("FR7630001007941234567890185", 1000)


print("=== Solde initial ===")
compte1.afficher_solde()



print("\n=== Après dépôt de 500 DH ===")
compte1.deposer(500)
compte1.afficher_solde()

print("\n=== Après retrait de 300 DH ===")
compte1.retirer(300)
compte1.afficher_solde()

print("\n=== Tentative de retrait de 2000 DH ===")
compte1.retirer(2000)

print("\n=== Création d'un deuxième compte ===")
compte2 = CompteBancaire("FR7630001007949876543210987")
compte2.afficher_solde()

print("\n=== Dépôt de 2000 DH sur le deuxième compte ===")
compte2.deposer(2000)
compte2.afficher_solde()
```

2) Encapsulation - Attributs privés

- Les attributs privés commencent par __ (double underscore)
- Ils ne sont pas accessibles directement depuis l'extérieur
- Protection des données contre les modifications non contrôlées
- Exemple : self.__note au lieu de self.note

- **Exemple : Attributs privés**

```
class Note:
    def __init__(self, nom_etudiant, note):
        self.nom_etudiant = nom_etudiant  # Public
        self.__note = note                # Privé

    def get_note(self):
        return self.__note

    def set_note(self, nouvelle_note):
        if 0 <= nouvelle_note <= 20:
            self.__note = nouvelle_note
        else:
            print("Erreur: Note invalide")
```

- **Test de l'encapsulation**

```
n1 = Note("Ahmed", 15)
print(n1.nom_etudiant)  # Accessible
print(n1.__note)        # Erreur!
print(n1.get_note())    # 15 (via getter)

n1.set_note(18)         #  Modification contrôlée
n1.set_note(25)         # Message d'erreur
```

Exercice : Encapsulation
Créez une classe Temperature avec :

- Attribut privé : __celsius
- Méthodes : get_celsius(), set_celsius(valeur), get_fahrenheit()
- Contrôle : la température ne peut pas descendre en dessous de -273.15°C

Remarque :La formule pour transformer des degrés Celsius en degrés Fahrenheit est :

$$°F = (°C \times 9/5) + 32$$

- **Solution : Encapsulation**

```
class Temperature:
    def __init__(self, celsius):
        self.__celsius = celsius

    def get_celsius(self):
        return self.__celsius

    def set_celsius(self, valeur):
        if valeur >= -273.15:
            self.__celsius = valeur
        else:
            print("Temperature impossible!")

    def get_fahrenheit(self):
        return (self.__celsius * 9/5) + 32
```

- une utilisation de la classe Temperature :

```
temp = Temperature(25)

print(f"Température en Celsius: {temp.get_celsius()}°C")
print(f"Température en Fahrenheit: {temp.get_fahrenheit()}°F")

print("\n=== Modification à 30°C ===")
temp.set_celsius(30)
print(f"Nouvelle température: {temp.get_celsius()}°C")
print(f"En Fahrenheit: {temp.get_fahrenheit()}°F")

print("\n=== Tentative de -300°C ===")
temp.set_celsius(-300)

print("\n=== Température -10°C ===")
temp.set_celsius(-10)
print(f"Température: {temp.get_celsius()}°C")
print(f"En Fahrenheit: {temp.get_fahrenheit()}°F")
```

3) Méthodes str et repr

- __str__ : représentation lisible pour les humains
- __repr__ : représentation technique pour les développeurs
- Appel automatique avec print() et str()
- Améliore le débogage et l'affichage

- **Exemple : str et repr**

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

    def __str__(self):
        return f"Produit: {self.nom} - {self.prix} DH"

    def __repr__(self):
        return f"Produit('{self.nom}', {self.prix})"
```

- **Utilisation des méthodes spéciales**

```
p = Produit("Ordinateur", 800)

print(str(p))   # Produit: Ordinateur - 800 DH
print(repr(p))  # Produit('Ordinateur', 800)

# Appel automatique
print(p)        # Produit: Ordinateur - 800 DH

# Pour le débogage
print([p])      # [Produit('Ordinateur', 800)]
```

- **Exercice : str et repr**
  Ajoutez les méthodes __str__ et __repr__ à la classe CompteBancaire :
- __str__ : "Compte XXXX - Solde: XXXX DH"
- __repr__ : "CompteBancaire('XXXX', XXXX)"

- **Solution : str et repr**

```
class CompteBancaire:
    def __init__(self, numero_compte, solde=0):
        self.numero_compte = numero_compte
        self.solde = solde

    def __str__(self):
        return f"Compte {self.numero_compte} - Solde: {self.solde} DH"

    def __repr__(self):
        return f"CompteBancaire('{self.numero_compte}', {self.solde})"
```

- utilisation de la classe CompteBancaire avec les méthodes __str__ et __repr__ :

```
compte1 = CompteBancaire("FR7630001007941234567890185", 1500)
compte2 = CompteBancaire("FR7630001007949876543210987")

print("=== Affichage avec print() ===")
print(compte1)
print(compte2)

print("\n=== Affichage dans une liste ===")
comptes = [compte1, compte2]
for compte in comptes:
    print(compte)

print("\n=== Représentation technique ===")
print(repr(compte1))
print(repr(compte2))

print("\n=== Affichage direct (simulation interpréteur) ===")
print(f">>> compte1")
print(compte1)
print(f">>> compte2")
print(compte2)
```

4) Abstraction - Cacher la complexité

- Montrer seulement ce qui est nécessaire
- Cacher les détails techniques internes
- Interface simple pour l'utilisateur
- Implémentation complexe masquée

- **Exemple : Abstraction avec une voiture**

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

    def demarrer(self):
        self.__verifier_carburant()
        self.__allumer_moteur()
        self.__verifier_ceintures()
        print("Voiture démarrée!")

    def __verifier_carburant(self):
        print("Vérification carburant...")

    def __allumer_moteur(self):
        print("Allumage moteur...")

    def __verifier_ceintures(self):
        print("Vérification ceintures...")
```

- **Utilisation de l'abstraction**

```
ma_voiture = Voiture("Fiat", "2025")

# L'utilisateur voit seulement ça :
ma_voiture.demarrer()

# Il ne voit pas les étapes internes :
# __verifier_carburant()
# __allumer_moteur()
# __verifier_ceintures()
```

Exercice : Abstraction
Créez une classe Cafetiere avec :

- Méthode publique : preparer_cafe()
- Méthodes privées : __chauffer_eau(), __moudre_grains(), __infuser()
- L'utilisateur appelle seulement preparer_cafe()

- **Solution : Abstraction**

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

    def preparer_cafe(self):
        self.__chauffer_eau()
        self.__moudre_grains()
        self.__infuser()
        print("Café prêt! ")

    def __chauffer_eau(self):
        print("Chauffage de l'eau...")

    def __moudre_grains(self):
        print("Broyage des grains...")

    def __infuser(self):
        print("Infusion en cours...")
```

- **Instanciation et test de la cafetière**

```
# Création d'une cafetière
ma_cafetiere = Cafetiere("Philips")

# Utilisation simple
ma_cafetiere.preparer_cafe()
```

- **Résultat de l'exécution**

Chauffage de l'eau...

Broyage des grains...

Infusion en cours...

Café prêt!

- **Avantage de l'abstraction**
  L'utilisateur voit seulement :

ma_cafetiere.preparer_cafe() # → "Café prêt! "

- Il ne sait pas comment :
- L'eau est chauffée
- Les grains sont moulus
- L'infusion se fait

5) Échange de données entre instances de classes différentes

- Les objets de classes différentes peuvent communiquer entre eux
- Passage d'objets en paramètres de méthodes
- Lecture et modification des attributs d'autres objets
- Création de systèmes complexes avec interactions

- **Exemple 1 : Gestion de Bibliothèque - Définition des classes**

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

    def changer_statut_emprunt(self):
        self.est_emprunte = not self.est_emprunte

class Membre:
    def __init__(self, nom, limite_emprunts=3):
        self.nom = nom
        self.limite_emprunts = limite_emprunts
        self.livres_empruntes = []
```

- **Méthode de vérification d'emprunt**

```
def verifier_emprunt_possible(self, livre):
    # Méthode qui reçoit un objet Livre en argument
    if len(self.livres_empruntes) >= self.limite_emprunts:
        print(f" {self.nom} a atteint la limite d'emprunts")
        return False

    if livre.est_emprunte:
        print(f" Le livre '{livre.titre}' est déjà emprunté")
        return False

    # Calcul et vérification réussie
    livres_restants = self.limite_emprunts - len(self.livres_empruntes)
    print(f" {self.nom} peut emprunter '{livre.titre}'")
    return True
```

- **Utilisation du système bibliothèque**

```
# Création des objets
livre1 = Livre("La Boîte à merveilles", "Ahmed Sefrioui")
livre2 = Livre("Antigone", "Jean Anouilh", est_emprunte=True)

membre = Membre("Mohammed-Al Barae Abdoussi", limite_emprunts=3)

# Communication entre objets de classes différentes
membre.verifier_emprunt_possible(livre1) # Mohammed peut emprunter
membre.verifier_emprunt_possible(livre2) # Livre déjà emprunté
```

```
class Livre:

    def __init__(self, titre, auteur, est_emprunte=False):

        self.titre = titre

        self.auteur = auteur

        self.est_emprunte = est_emprunte



    def changer_statut_emprunt(self):

        self.est_emprunte = not self.est_emprunte



class Membre:

    def __init__(self, nom, limite_emprunts=3):

        self.nom = nom

        self.limite_emprunts = limite_emprunts

        self.livres_empruntes = []



    def verifier_emprunt_possible(self, livre):

        if len(self.livres_empruntes) >= self.limite_emprunts:

            print(f" {self.nom} a atteint la limite d'emprunts")

            return False



        if livre.est_emprunte:

            print(f" Le livre '{livre.titre}' est déjà emprunté")

            return False



        livres_restants = self.limite_emprunts - len(self.livres_empruntes)

        print(f" {self.nom} peut emprunter '{livre.titre}'")

        return True



livre1 = Livre("La Boîte à merveilles", "Ahmed Sefrioui")

livre2 = Livre("Antigone", "Jean Anouilh", est_emprunte=True)



membre = Membre("Mohammed-Al Barae Abdoussi", limite_emprunts=3)

membre.verifier_emprunt_possible(livre1) # Mohammed peut emprunter

membre.verifier_emprunt_possible(livre2) # Livre déjà emprunté
```

Exercice : Système bibliothèque
Améliorez la classe Membre en ajoutant :

- Méthode emprunter_livre(livre) qui :
  - Vérifie si l'emprunt est possible
  - Ajoute le livre à livres_empruntes
  - Change le statut du livre
  - Affiche un message de confirmation

- **Solution : Système bibliothèque amélioré**

```
def emprunter_livre(self, livre):
    if self.verifier_emprunt_possible(livre):
        self.livres_empruntes.append(livre)
        livre.changer_statut_emprunt()
        print(f"{self.nom} a emprunté '{livre.titre}'")
        return True
    return False
```

```python
class Livre:

  def __init__(self, titre, auteur, est_emprunte=False):

    self.titre = titre

    self.auteur = auteur

    self.est_emprunte = est_emprunte


  def changer_statut_emprunt(self):

    self.est_emprunte = not self.est_emprunte


class Membre:

  def __init__(self, nom, limite_emprunts=3):

    self.nom = nom

    self.limite_emprunts = limite_emprunts

    self.livres_empruntes = []


  def verifier_emprunt_possible(self, livre):

    if len(self.livres_empruntes) >= self.limite_emprunts:

      print(f" {self.nom} a atteint la limite d'emprunts")

      return False


    if livre.est_emprunte:

      print(f" Le livre '{livre.titre}' est déjà emprunté")

      return False


    livres_restants = self.limite_emprunts - len(self.livres_empruntes)

    print(f" {self.nom} peut emprunter '{livre.titre}'")

    return True

  def emprunter_livre(self, livre):

    if self.verifier_emprunt_possible(livre):

      self.livres_empruntes.append(livre)

      livre.changer_statut_emprunt()

      print(f"{self.nom} a emprunté '{livre.titre}'")

      return True

    return False


livre1 = Livre("La Boîte à merveilles", "Ahmed Sefrioui")

livre2 = Livre("Antigone", "Jean Anouilh", est_emprunte=True)


membre = Membre("Mohammed-Al Baraa Abdoussi", limite_emprunts=3)

membre.emprunter_livre(livre1)

membre.emprunter_livre(livre2)
```

- **Exemple 2 : Gestion de Commandes - Définition des classes**

```
class Produit:
    def __init__(self, nom, prix, stock):
        self.nom = nom
        self.prix = prix
        self.stock = stock

    def reduire_stock(self, quantite):
        if self.stock >= quantite:
            self.stock -= quantite
            return True
        return False

class Panier:
    def __init__(self, client_nom):
        self.client_nom = client_nom
        self.total = 0
```

- **Méthode d'ajout au panier**

```
def ajouter_produit(self, produit, quantite):
    # Méthode qui reçoit un objet Produit en argument
    if produit.stock >= quantite:
        # Calcul du sous-total
        sous_total = produit.prix * quantite
        self.total += sous_total

        # Lecture des attributs de l'objet Produit
        print(f" {self.client_nom} a ajouté {quantite} x '{produit.nom}'")
        print(f" Prix unitaire: {produit.prix}€ - Sous-total: {sous_total}€")
        print(f" Stock restant: {produit.stock - quantite} unités")
        print(f" Total panier: {self.total}€")
        print("-" * 40)

        return True
    else:
        print(f" Stock insuffisant pour '{produit.nom}'")
        return False
```

- **Utilisation du système de commandes**

```
# Création des produits
produit1 = Produit("Ordinateur Portable", 800, 5)
produit2 = Produit("Souris USB", 25, 10)

panier_client = Panier("Nouhaila Zouine")

# Communication entre objets
panier_client.ajouter_produit(produit1, 1) # Ajout réussi
panier_client.ajouter_produit(produit2, 15) # Stock insuffisant
```

```python
class Produit:
    def __init__(self, nom, prix, stock):
        self.nom = nom
        self.prix = prix
        self.stock = stock

    def reduire_stock(self, quantite):
        if self.stock >= quantite:
            self.stock -= quantite
            return True
        return False

class Panier:
    def __init__(self, client_nom):
        self.client_nom = client_nom
        self.total = 0
    def ajouter_produit(self, produit, quantite):

        if produit.stock >= quantite:

            sous_total = produit.prix * quantite
            self.total += sous_total


            print(f" {self.client_nom} a ajouté {quantite} x '{produit.nom}'")
            print(f" Prix unitaire: {produit.prix}€ - Sous-total: {sous_total}€")
            print(f" Stock restant: {produit.stock - quantite} unités")
            print(f" Total panier: {self.total}€")
            print("-" * 40)

            return True
        else:
            print(f" Stock insuffisant pour '{produit.nom}'")
            return False


produit1 = Produit("Ordinateur Portable", 800, 5)
produit2 = Produit("Souris USB", 25, 10)

panier_client = Panier("Nouhaila Zouine")

panier_client.ajouter_produit(produit1, 1) # Ajout réussi
panier_client.ajouter_produit(produit2, 15) # Stock insuffisant


k=input(")
```

Exercice : Système de commandes amélioré
Ajoutez à la classe Panier :

- Méthode finaliser_commande() qui :
  - Réduit le stock de tous les produits ajoutés
  - Vide le panier
  - Affiche un récapitulatif de la commande

- **Solution : Finalisation de commande**

```
def finaliser_commande(self):
    print(f"Commande finalisée pour {self.client_nom}")
    print(f"Total à payer: {self.total}DH")
    print("Merci pour votre achat!")
    self.total = 0
```

Exercice global : Système de Réservation de Billets de Train (ONCF)

Créez un système de réservation de billets de train avec 3 classes :

- Train : représente un train avec son numéro, destination et nombre de places
- Billet : représente un billet avec son numéro, le train et le passager
- Passager : représente un passager avec son nom et ses billets réservés
- **Consignes :**
- La classe Train doit avoir une méthode reserver_place() qui diminue le nombre de places disponibles
- La classe Passager doit avoir une méthode reserver_billet(train) qui :
  - Vérifie s'il reste des places dans le train
  - Crée un billet et l'ajoute aux billets du passager
  - Réserve une place dans le train
- Utilisez l'encapsulation pour protéger les attributs critiques

- **Exercice global : Système ONCF**

```
class Train:
    def __init__(self, numero, destination, places_total):
        self.numero = numero
        self.destination = destination
        self.__places_disponibles = places_total

    def reserver_place(self):
        if self.__places_disponibles > 0:
            self.__places_disponibles -= 1
            return True
        return False

    def get_places_disponibles(self):
        return self.__places_disponibles
```

- **Suite de la solution**

```
class Billet:
    def __init__(self, numero_billet, train, passager):
        self.numero_billet = numero_billet
        self.train = train
        self.passager = passager

    def __str__(self):
        return f"Billet {self.numero_billet} - Train {self.train.numero} vers {self.train.destination}"

class Passager:
    def __init__(self, nom):
        self.nom = nom
        self.billets = []

    def reserver_billet(self, train):
        if train.reserver_place():
            numero_billet = f"B{len(self.billets)+1:03d}"
            billet = Billet(numero_billet, train, self)
            self.billets.append(billet)
            print(f" {self.nom} a réservé un billet pour {train.destination}")
            return billet
        else:
            print(f" Plus de places disponibles pour le train vers {train.destination}")
            return None
```

- **Utilisation du système**

```
# Création des trains
train1 = Train("TGV123", "Casablanca", 2)
train2 = Train("TL12", "Marrakech", 1)

# Création des passagers
passager1 = Passager("Ahmed")
passager2 = Passager("Fatima")

# Réservations
b1 = passager1.reserver_billet(train1)  #  Réservation réussie
b2 = passager2.reserver_billet(train1)  # Réservation réussie
b3 = passager1.reserver_billet(train1)  # Plus de places
b4 = passager2.reserver_billet(train2)  # Réservation réussie

print(f"\nPlaces restantes Casablanca: {train1.get_places_disponibles()}")
print(f"Places restantes Marrakech: {train2.get_places_disponibles()}")
```

- **Résultat de l'exécution**

- Ahmed a réservé un billet pour Casablanca
- Fatima a réservé un billet pour Casablanca
- Plus de places disponibles pour le train vers Casablanca
- Fatima a réservé un billet pour Marrakech

- Places restantes Casablanca: 0
- Places restantes Marrakech: 0