Programmation orientée objet

1. Objet

1.1. Définition

Un objet est caractérisé par trois éléments :

- Une identité unique (son nom)
- Des données propres (son état), appelées aussi attributs ou propriétés
- Des traitements dont il est responsable (des fonctions pouvant modifier son état ou s'appliquer à d'autres objets, appelées méthodes)

Exemple : objet eleve

- Identité : alex
- Données : nom complet, liste de notes, moyenne
- Traitements (méthodes) :
  - Lire les données (un par donnée : nom complet, notes, moyenne)
  - Ajouter une note
  - Calculer la moyenne

1.2. Cas particuliers

- Objet sans identité : objet anonyme. Les autres objets ne pourront pas lui demander de traitement, mais lui pourra demander des traitements aux autres objets.
- Objet sans données : objet stateless. Ces objets réalisent des traitements. Ils sont semblables à des fonctions que l'on peut supprimer, dupliquer, …
- Objet sans traitement : techniquement possible, mais pas très utiles. Généralement, tous les objets ont au moins des traitements d'accès à leurs données (getter : méthode pour lire les données, et setter : méthode pour écrire dans les données).

1.3. Programmer des objets

On ne programme pas directement des objets en principe, on passe par les classes.

2. Classe

2.1. Définition

Une classe a un nom unique dans le programme.
Elle définit les données associées à ses objets : les propriétés (attributs).
Elle définit les traitements des objets : les méthodes.
Un objet est une instance d'une classe.

On programme une classe. La classe crée une instance (un objet) en mémoire à l'exécution du code. La création d'un objet à partir d'une classe s'appelle l'instanciation.

alex = Eleve() permet d'instancier l'objet alex à partir de la classe Eleve.
On note alex : Eleve pour indiquer que l'objet alex est une instance de la classe Eleve.

2.2. Propriétés

Une propriété est définie par

- un nom unique parmi les propriétés,
- éventuellement, selon le langage, un type qui définit la structure de données,
- éventuellement une valeur par défaut donnée lors de la création.

2.3. Méthodes

Une méthode est définie par

- une signature unique dans chaque classe : nom, paramètres d'entrée et de sortie,
- un corps (body) : le code du traitement (this ou self référence l'objet lui-même).

Le paramètre self est obligatoire en Python dans les méthodes agissant sur un objet. La plupart des langages considèrent que puisqu'on est dans une classe, on va manipuler l'objet avec self (ou this en Javascript) et ne demandent pas qu'on le passe en paramètre.

3. Principes de la POO

3.1. Encapsulation

Un objet protège ses données. Il peut laisser d'autres objets lire ses données, mais il est le seul à modifier ses données. Idéalement, si on est pur, les autres objets n'ont aucun accès aux données qui sont à l'intérieur de l'objet, ils savent juste que cet objet propose des traitements.

3.2. Responsabilité

Un objet est responsable des traitements qu'il propose. Il a donc toutes les données nécessaires et suffisantes. Il peut utiliser les traitements d'autres objets, mais il est garant du bon fonctionnement de la méthode.

3.3. Cohérence

Un objet est spécialiste : les méthodes qu'il propose sont toutes relatives à un seul objectif (méthodes cohérentes). Un objet propose peu de traitements. Un objet proposant beaucoup de traitements manque de cohérence.
Si on peut le couper en deux (certaines méthodes agissant sur des données, d'autres méthodes sur d'autres données), alors notre objet n'est pas cohérent. On doit faire deux objets, ou plutôt deux classes puisqu'on programme les classes. Une classe doit être insécable. Si vous avez du mal à nommer vos classes, c'est souvent qu'il y a un problème de cohérence.

3.4. Couplages

Un objet #1 doit avoir la référence d'un objet #2 dans ses données (ou méthodes) pour communiquer avec lui. On dit alors que #1 est couplé avec #2.
Plus il y a de couplage, plus les objets sont liés, ce qui complique les tests.
Il est préférable qu'un objet communique avec peu d'autres objets, mais dans ce cas, il risque d'avoir besoin de beaucoup de traitements, ce qui crée des problèmes de cohérence.

Une difficulté de la POO est de trouver un bon équilibre couplage-cohérence.
On peut faire un graphe de dépendance entre les objets, et il faut limiter les chaînes et interdire les boucles dans ce graphe (des objets qui s'appellent en boucle). Sinon, on viole le principe de responsabilité.

3.5. Communication synchrone et monothread

Les objets communiquent par échange de messages, avec un seul type de message : la demande de réalisation de méthode (de traitement).
Dans le modèle de base de la POO, la sémantique est celle de l'échange synchrone et monothread (« monothrédé », un seul objet actif) :

1. L'objet actif peut envoyer une demande de traitement par un autre objet
2. L'envoyeur n'est plus actif
3. Le receveur devient actif et effectue le traitement demandé
4. L'envoyeur redevient actif avec la réponse

Les objets se transmettent l'activité. On peut créer un modèle asynchrone, avec deux threads et donc deux objets actifs simultanément.

3.6. Dynamicité

Si un objet ne sert qu'à un moment précis, on préférera généralement le construire à ce moment plutôt qu'à l'initialisation. C'est le caractère dynamique d'un code.

4. Application Orientée Objet

Une application Orientée Objet est constituée de plusieurs objets qui communiquent.
Certains communiquent avec l'extérieur (clic de souris, clavier, affichage sur un écran, envoi de requêtes sur le réseau, création de nouveaux objets…).
La décomposition en objets/classes structure le code.
Il faut une configuration initiale, le main : fonction d'initialisation permettant de construire les objets nécessaires au démarrage de l'application.

Les questions à se poser :

- Quels objets ? Identité, données, traitements ?
- Quelles cohérences ? But de chaque objet ?
- Quels couplages ? Rôle de chaque objet par rapport aux autres, relations d'appels entre les objets, pas de chaine trop longue ou de cycle de dépendance
- Quelle dynamicité ? Configuration initiale (main) et comment construire ou supprimer des objets, ce qui fait toute la dynamique de l'application

Concevoir une AOO n'est pas simple car il y a un équilibre à trouver entre le nombre d'objets, leur cohérence, leur couplage et comment ils se construisent. Ces difficultés sont à aborder avant de programmer. La programmation revient à écrire dans le code ce qu'on a déjà conçu.

5. Exemple de programmation en Python

```
class Eleve:
    def __init__(self, nom):
        self.nom = nom
        self.notes = []
        self.moyenne = 'undefined'

    def get_nom(self):
        return self.nom

    def get_notes(self):
        return self.notes

    def get_moyenne(self):
        if self.moyenne=='undefined' and self.notes:
            self.calculer_moyenne()
        return self.moyenne

    def ajouter_note(self, note):
        self.notes.append(note)

    def calculer_moyenne(self):
        som = 0
        for n in self.notes:
            som += n
        self.moyenne = som/len(self.notes)

if __name__ == '__main__':
    alex = Eleve('Alexis Baronville')
    alex.ajouter_note(14)
    alex.ajouter_note(10)
    print(alex.get_moyenne())
```

Le constructeur est la méthode __init__ appelée lorsqu'on construit un objet. Il contient la structuration et les valeurs par défaut.

La méthode __repr__ renvoie la représentation de l'objet sous forme de chaine de caractères, celle qui sera affichée si on appelle l'objet dans la console (ou avec print car elle redéfinit la méthode __str__ appelée par print).
Pour que l'appelle d'une instance d'Eleve affiche le nom d'un élève, on ajoute dans la classe Eleve :

```
    def __repr__(self):
        return self.get_nom()
```

Getter et setter

On considère qu'on ne viole pas le principe de l'encapsulation via un getter (accesseur en français) puisque l'objet contrôle l'accès aux données. Par contre, les setters (méthode pour donner des valeurs aux données) ne sont pas recommandés. Si on est puriste, on évite les deux. Le principe d'encapsulation dit qu'on ne peut pas voir l'intérieur d'un objet.

La méthode get_nom de la classe Eleve est un getter.
La méthode set_nom ci-dessous est un setter (non recommandé).

```
def set_nom(self, nnom):
    self.nom = nnom
```

À propos de l'encapsulation

La classe précise les règles d'accès aux données des objets :

- public : accès total, ne respecte pas la propriété d'encapsulation. N'importe quel objet peut modifier les données… c'est l'anarchie !
- private : accès interdit

Certains langages permettent des règles beaucoup plus fines : public, private, package (accès qu'aux objets du même package)… accès en lecture seule…
Python n'est pas bon en encapsulation. Par défaut, la donnée est publique. On met un underscore_ devant le nom de l'objet pour indiquer un caractère privé (qui n'est pas respectée techniquement), deux underscores __ pour rendre la donnée privée. En pratique, il n'y a pas de véritable contrôle d'accès (contrairement à Java ou C++), l'interpréteur Python renomme simplement cette variable en la préfixant par le nom de la classe !

Exemple

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

__nom devient _Eleve__nom et reste accessible avec ce nouveau nom.

6. Compléments hors-programme : héritage et polymorphisme

Un principe de programmation est DRY : Don't Repeat Yourself. On ne va jamais coder deux fois la même chose. On utilise des classes réutilisables, et si deux classes ont plusieurs méthodes communes, plutôt que de les coder deux fois, on les code une fois dans une superclasse, puis on crée deux sous-classes qui vont hériter des méthodes de la superclasse.
Le polymorphisme, c'est redéfinir certaines méthodes de la superclasse dans la sous-classe. Elles n'auront alors un effet différent avec un objet de la superclasse et avec un objet de la sous-classe.

Exemple

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

    def get_nom(self):
        return self.__nom

class Eleve(Personne):
    def __init__(self, nom, classe):
        super().__init__(nom)
        self.__classe = classe

    def get_classe(self):
        return self.__classe

class Prof(Personne):
    def __init__(self, nom, matiere):
        super().__init__(nom)
        self.__matiere = matiere

    def get_matiere(self):
        return self.__matiere
```

Le mot clé super() permet d'accéder au constructeur __init__ de la classe mère.

7. Projet POO

Écrire une application de gestion d'un parking.
Le parking a 5 niveaux, avec 80 places par niveau, numérotée 1 à 480 (le chiffre des centaines étant le numéro du niveau). On créera une classe Parking et une classe Voiture. Un objet voiture contient le numéro d'immatriculation de la voiture, sa marque, le nom de son propriétaire et un booléen indiquant si le propriétaire est abonné, auquel cas il lui est attribué un numéro de place.
L'objet parking contient la liste des abonnées et la liste des places. Des méthodes permettent :

- de récupérer les différentes informations relatives à une voiture (immatriculation, marque, propriétaire, abonnement)
- d'abonner ou d'annuler un abonnement
- d'affecter une voiture à une place
- de savoir si une place donnée est libre, ou quelle voiture l'occupe
- de renvoyer la liste des places d'abonnées occupées par d'autres voitures
- de connaitre le nombre de places libres sans compter les places réservées aux abonnées
- de représenter un niveau (en mode texte)

Comme toujours, tester vos morceaux de codes au fur et à mesure !