Programmation orientée objet

Activité – Terminale NSI

① Pourquoi des objets ?

Exercice 1 *Représenter une voiture sans objet*

On souhaite stocker, dans un programme Python, plusieurs voitures avec leurs caractéristiques (marque, modèle, kilométrage, vitesse actuelle).

Première approche. Utiliser plusieurs variables ou un tuple :

```
voiture_1 = ("Peugeot", "208", 45000, 0)
voiture_2 = ("Renault", "Clio", 12000, 90)
voiture_3 = ("Citroën", "C3", 78000, 50)
```

1. Pour augmenter la vitesse de `voiture_2` de 20 km/h, comment fait-on ? *Indication : les tuples ne sont pas modifiables.*
2. Si l'on souhaite ajouter à chaque voiture une nouvelle caractéristique (sa couleur), combien de variables faut-il modifier ?
3. Pour écrire la fonction « rouler(n km) » qui simule un trajet, comment passerait-on la voiture en paramètre ? Combien d'arguments la fonction renverrait-elle ?

Remarque 1. Les tuples (et les listes) ne suffisent pas dès qu'un « objet du monde réel » possède plusieurs caractéristiques liées entre elles. La **programmation orientée objet** (POO) permet de regrouper ces caractéristiques (**attributs**) avec les opérations qui s'y appliquent (**méthodes**) sous une seule entité : la **classe**.

② Définir une classe et créer des objets

Définition 1 **(Classe et instance)** Une **classe** est un *patron* (un « moule ») à partir duquel on crée des **objets** (aussi appelés *instances*). Chaque objet possède ses propres **attributs** (les données) et utilise les **méthodes** de sa classe (les opérations).

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
        self.km = 0
        self.vitesse = 0
```

La méthode spéciale `__init__` est le **constructeur** : elle est appelée automatiquement lors de la création d'un objet. Le premier paramètre, `self`, désigne *l'objet en cours de création*.

Exercice 2 *Créer une voiture*

On utilise la classe `Voiture` ci-dessus.

1. Que crée la ligne `ma_voiture = Voiture("Peugeot", "208")` ?
2. Que valent `ma_voiture.marque`, `ma_voiture.modele`, `ma_voiture.km`, `ma_voiture.vitesse` ?
3. On exécute `ma_voiture.km = 1500`. Que se passe-t-il ?
4. Si l'on crée une seconde voiture `autre = Voiture("Renault", "Clio")`, modifier la marque de `ma_voiture` change-t-il quelque chose pour `autre` ? Pourquoi ?

③ Ajouter des méthodes

Définition 2 **(Méthode)** Une **méthode** est une fonction définie *à l'intérieur* d'une classe. Son premier paramètre, `self`, désigne l'objet sur lequel la méthode est appelée. L'appel se fait avec la *notation pointée* : `objet.methode(arguments)`.

Exercice 3 *Enrichir la classe `Voiture`*

On souhaite enrichir la classe avec deux méthodes :

— `accelerer(delta)` qui ajoute `delta` km/h à la vitesse ;

— `rouler(duree)` qui simule un trajet de `duree` heures à la vitesse actuelle, et met à jour le kilométrage.

1. Compléter la classe :

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
        self.km = 0
        self.vitesse = 0

    def accelerer(self, delta):
        self.vitesse = ............

    def rouler(self, duree):
        distance = ............
        self.km = ............
```

2. Que se passe-t-il après les appels suivants ?

```
v = Voiture("Peugeot", "208")
v.accelerer(50)
v.rouler(2)
print(f"v.km = {v.km}, v.vitesse = {v.vitesse}")
```

3. Pourquoi écrit-on `v.accelerer(50)` et non `Voiture.accelerer(50)` ?

④ Afficher proprement : la méthode `__str__`

Remarque 2 (Méthodes spéciales). Python définit des *méthodes spéciales* (entourées de doubles `__`) qui sont appelées automatiquement dans certains contextes. La plus utile est `__str__`, qui contrôle l'affichage d'un objet par `print` ou par `str`.

Exercice 4 *Une méthode `__str__` pour `Voiture`*

1. Sans `__str__`, qu'affiche `print(Voiture("Peugeot", "208"))` ?
2. Compléter la méthode `__str__` pour qu'elle renvoie une chaîne du type « Peugeot 208 (12000 km, vitesse 90 km/h) ».

```
def __str__(self):
    return f"{...} {...} ({...} km, vitesse {...} km/h)"
```

3. Ajouter cette méthode à la classe et tester avec une voiture créée et modifiée. Que renvoie maintenant `print(v)` ?

⑤ L'héritage

Définition 3 **(Héritage)** Une classe peut **hériter** d'une autre classe en reprenant ses attributs et ses méthodes, et en ajoutant ou en modifiant ce qui doit l'être. La classe parente est appelée **classe de base** ; la classe enfant, **classe dérivée**.

```
class VoitureElectrique(Voiture):
    def __init__(self, marque, modele, autonomie):
        super().__init__(marque, modele)   # appel du constructeur parent
        self.autonomie = autonomie
```

L'expression `super().__init__(...)` appelle le constructeur de la classe parente.

Exercice 5 *Une voiture électrique*

1. Que représente le paramètre `autonomie` ?
2. Si l'on crée `tesla = VoitureElectrique("Tesla", "Model 3", 500)`, quels sont les attributs de `tesla` ?
3. La méthode `accelerer` héritée de `Voiture` fonctionne-t-elle sur une `VoitureElectrique` ? Justifier.
4. Que faudrait-il faire pour qu'une voiture électrique consomme de l'autonomie quand elle roule ? *Donner uniquement le principe, pas le code complet.*

⑥ Pourquoi (et quand) utiliser la POO ?

Exercice 6 *Bilan critique*

1. Reprendre les trois objections de la **Partie 1** (modification, ajout d'une caractéristique, fonction `rouler`). En quoi la POO les résout-elle ?

2. Donner deux exemples concrets, hors voitures, où la POO serait pertinente.
3. Donner un exemple où la POO serait inutile (et où une fonction simple suffirait).

7 Ce qu'il faut retenir

Propriété 1 **(Vocabulaire de la POO)**

— **Classe** : patron qui décrit un type d'objet (ses attributs et ses méthodes).

— **Objet** (ou *instance*) : une réalisation concrète d'une classe.

— **Attribut** : une donnée propre à un objet (`self.attribut`).

— **Méthode** : une fonction définie dans une classe, dont le premier paramètre est `self`.

— **Constructeur** (`__init__`) : méthode appelée automatiquement à la création d'un objet.

Propriété 2 **(Notation pointée)** On accède à un attribut par `objet.attribut` et on appelle une méthode par `objet.methode(arguments)`. Python passe automatiquement l'objet comme premier argument (`self`).

Propriété 3 **(Héritage)** `class Fille(Mere):` indique que `Fille` hérite de `Mere`. La classe fille reçoit gratuitement tous les attributs et méthodes de la mère, qu'elle peut compléter ou redéfinir. `super().__init__(...)` appelle le constructeur parent.

Remarque 3. La méthode spéciale `__str__` contrôle l'affichage par `print`. Sans elle, Python affiche l'adresse mémoire de l'objet, peu lisible. C'est l'une des premières méthodes à ajouter à une classe utile.

Solutions des exercices

Corrigé de l'exercice 1.

1. Les tuples étant immuables, il faut créer un *nouveau* tuple : `voiture_2 = (voiture_2[0], voiture_2[1], voiture_2[2], voiture_2[3] + 20)`. Très lourd et risqué (il faut connaître la position de chaque champ).
2. Toutes les voitures du programme doivent être recréées avec un champ supplémentaire. Toute fonction qui manipulait les voitures doit aussi être adaptée.
3. Il faut passer les 4 champs de la voiture *séparément*, et la fonction doit renvoyer un nouveau tuple à 4 champs représentant la voiture après le trajet.

Corrigé de l'exercice 2.

1. Cette ligne crée un *objet* de la classe `Voiture`. Le constructeur reçoit `"Peugeot"` et `"208"` et initialise les quatre attributs.
2. Marque : `"Peugeot"` ; modèle : `"208"` ; km : `0` ; vitesse : `0`.
3. L'attribut `km` de l'objet est modifié ; il vaut désormais 1 500.
4. Non. Chaque objet a ses propres attributs. Modifier un attribut de `ma_voiture` n'a aucun effet sur `autre`.

Corrigé de l'exercice 3.

1. Méthodes complétées :

```
def accelerer(self, delta):
    self.vitesse = self.vitesse + delta

def rouler(self, duree):
    distance = self.vitesse * duree
    self.km = self.km + distance
```

2. Après `v.accelerer(50)` : vitesse passe à 50. Après `v.rouler(2)` : distance = $50 \times 2 = 100$ km, kilométrage passe à 100. Le programme affiche `v.km = 100, v.vitesse = 50`.
3. Parce que la méthode opère sur un *objet précis* (ici `v`). En écrivant `v.accelerer(50)`, Python passe automatiquement `v` comme premier argument (`self`). `Voiture.accelerer(50)` ne précise pas sur quelle voiture la méthode doit agir.

Corrigé de l'exercice 4.

1. Sans `__str__`, Python affiche quelque chose comme `<__main__.Voiture object at 0x10c8d4f10>` : la classe et l'adresse mémoire de l'objet, illisible pour un humain.
2. Méthode complétée :

```
def __str__(self):
    return f"{self.marque} {self.modele} ({self.km} km, vitesse {self.vitesse} km/h)
    "
```

3. Avec cette méthode, `print(v)` affiche par exemple `Peugeot 208 (100 km, vitesse 50 km/h)`.

Corrigé de l'exercice 5.

1. L'autonomie en kilomètres restants avant de devoir recharger.
2. Quatre attributs hérités de `Voiture` (`marque`, `modele`, `km`, `vitesse`) plus l'attribut spécifique `autonomie`.
3. Oui : `VoitureElectrique` hérite de `Voiture`, donc elle dispose de toutes les méthodes de la classe parente.
4. Il faut *redéfinir* la méthode `rouler` dans `VoitureElectrique` pour qu'elle décrémente `self.autonomie` en plus de mettre à jour `self.km`. C'est ce qu'on appelle la *redéfinition* (*override*).

Corrigé de l'exercice 6.

1. — Modification : `ma_voiture.vitesse += 20` suffit (les objets sont *mutables*).
   — Ajout d'une caractéristique : il suffit de modifier le constructeur en un seul endroit, toutes les voitures sont automatiquement à jour.
   — Fonction `rouler` : c'est désormais une méthode (`ma_voiture.rouler(2)`). Elle n'a pas à renvoyer une nouvelle voiture : elle modifie l'objet directement.
2. Par exemple : un `Compte` bancaire (avec attributs `solde`, `titulaire` et méthodes `depot`, `retrait`) ; un `Eleve` (avec attributs `prenom`, `notes` et méthodes `ajouter_note`, `moyenne`). La POO sert à représenter toute entité dotée de plusieurs caractéristiques liées et d'opérations.
3. Calculer la racine carrée d'un nombre : une simple fonction `racine(x)` suffit. Pas besoin de définir une classe pour des opérations simples sur des valeurs basiques.