

1 Introduction

Jusqu'ici, lorsque nous avons des données à représenter, nous avons utilisé les types entier, réel et chaîne de caractères intégrés à Python.

Ceci dit, lors de l'élaboration d'un programme plus grand, on veut souvent représenter des données plus complexes. Par exemple, si je souhaite réaliser un programme pour un concessionnaire automobile, je vais avoir envie de représenter chacune de mes voitures. Or, une voiture possède plusieurs propriétés (sa marque, son modèle, son nombre de portes, ...), qui ne peuvent pas être représentées par une seule donnée.

Nous allons donc créer des **classes**, qui permettront, entre autres, de regrouper ensemble toutes les données représentant un objet quelconque (dans notre cas, une voiture).

On peut donc, dans un premier temps, voir une classe comme un nouveau type de données. On commence par le définir, puis on pourra l'utiliser par la suite dans notre programme.

2 Classes et attributs

2.1 Différence entre classe et objet

Comme expliqué plus haut, les classes peuvent être vues comme un nouveau type (en fait, un peu plus complexe qu'un type, nous y reviendrons plus tard). Ce qui veut dire que tout le code que l'on met dans une classe n'est pas immédiatement exécuté, c'est une "définition" de ce que cette classe va représenter.

Une fois la classe définie, si on veut l'utiliser dans notre programme, il va falloir en créer un exemplaire. On peut créer autant d'exemplaires que l'on veut d'une classe donnée (ex : une fois que j'ai défini la classe "Voiture", je peux créer autant de voitures que je veux). Un exemplaire d'une classe s'appelle un **objet**. La création d'un objet s'appelle **l'instanciation**.

2.2 Définir une classe

En python, la définition d'une classe se fait grâce au mot **class**. On fait **toujours** commencer le nom d'une classe par une majuscule. Ex :

```
class Voiture:
    # Vous mettez ici le contenu de la classe voiture
```

2.3 Utiliser une classe

Lorsque vous voudrez utiliser une classe, vous allez devoir **l'instancier** (= créer un nouvel **objet**). On peut instancier une classe autant de fois que l'on veut. Pour ceci, on utilise la syntaxe suivante :

```
ma_premiere_voiture = Voiture()
voiture2 = Voiture()
...
```

2.4 Le constructeur

En réalité, lorsque l'on instancie une classe, python appelle une méthode spéciale, appelée le **constructeur**. Toutes les classes ont un constructeur par défaut, qui permet de créer l'objet. Cette méthode s'appelle `__init__`. Elle possède au moins toujours 1 paramètre, qui doit s'appeler `self`. Ce `self` représente l'instance courante de la classe, celle que l'on est en train de manipuler.

On peut modifier ce constructeur afin d'initialiser certains **attributs** de notre classe. Les **attributs** sont des variables qui représentent l'état d'un objet. Par exemple, une voiture peut avoir comme attributs son modèle, sa couleur, etc. On préfixe ces attributs par `self`, qui permet de dire que ce sont les caractéristiques de l'instance que l'on manipule (ex : la couleur sera propre à chaque voiture, elle n'est pas commune à toutes les voitures existantes).

Dans ce cours, nous vous demanderons de prendre l'habitude de toujours écrire un constructeur, qui initialisera tous les attributs de votre classe. Exemple :

```
class Voiture:
    def __init__(self):
        self.km = 0
        self.modele = None
        self.couleur = None

# On pourra ensuite créer des voitures. Chaque voiture nouvellement
# créée aura 0 km au compteur et n'aura pas de modèle ni de
# marque
v1 = Voiture()
...
```

On peut également rajouter des paramètres à la méthode `__init__`, qui vont nous aider à faire l'initialisation de la voiture. Par exemple, si je veux forcer toutes mes voitures à avoir un modèle et une couleur dès que je les crée, je ferai ceci :

```
class Voiture:
    def __init__(self, mon_modele, ma_couleur):
        self.km = 0
        self.modele = mon_modele
        self.couleur = ma_couleur

# Maintenant, quand on crée des voitures, on doit leur donner un
# modèle et une couleur
v1 = Voiture("Zoé", "Bleue")
v2 = Voiture("208", "Grise")
...
```

2.5 Manipulation des attributs

On peut accéder aux attributs d'un objet en utilisant le point ("."). Il est possible de récupérer leur valeur ou des la modifier de cette façon. Par exemple :

```
v1 = Voiture("Zoé", "Bleue")

print(v1.couleur) # Affiche "Bleue"
v1.couleur = "Verte" # Change la couleur de la voiture 1
print(v1.couleur) # Affiche "Verte"
```

3 Les méthodes

En plus de pouvoir regrouper un certain nombre d'attributs, les classes peuvent également posséder des **méthodes**. Il existe plusieurs types de méthodes mais nous n'en aborderons dans ce cours qu'un seul (appelé "méthode d'instance").

Ces méthodes sont des fonctions qui sont liées à une classe. Pour les appeler, il faudra au préalable avoir créé une instance de la classe en question. On appellera ensuite la méthode grâce à l'intermédiaire du symbole point (comme pour les attributs).

Chaque méthode va spécifier un comportement de la classe en question. Par exemple, je pourrais écrire une méthode "avancer" dans ma classe Voiture, qui permettrait à une voiture d'avancer.

Comme les fonctions, les méthodes peuvent avoir différents paramètres, et renvoyer ou non une valeur. La seule différence est qu'il faut nécessairement que le premier paramètre de la méthode soit **self** (encore lui...)

```
class Voiture:
    def __init__(self):
        self.km = 0
        self.demarree = False

    # Méthode qui permet de démarrer une voiture
    def demarrer(self):
        self.demarree = True

    # Méthode qui permet de faire avancer une voiture
    def avancer(self, distance_km):
        if self.demarree:
            self.km = self.km + distance_km
            print("La voiture a avancé de ", distance_km, "km")
        else:
            print("La voiture n'est pas démarrée, elle ne peut pas avancer !")

v1 = Voiture()
v2 = Voiture()

v1.avancer(42) # N'avance pas car v1 n'est pas démarrée

v2.demarrer() # On démarre v2
v2.avancer(40) # On la fait avancer de 40 km
print(v2.km) # Affiche 40
v2.avancer(50) # On fait de nouveau avancer v2 de 50 km
print(v2.km) # Affiche 90

print(v1.km) # Affiche 0 (v1 et v2 sont bien 2 instances distinctes
, leurs km ne sont pas partagés !)
```

4 Pour aller plus loin : la surcharge d'opérateurs (Hors programme) *

Attention, assurez-vous d'avoir parfaitement compris les bases de la programmation orientée objet avant de lire cette rubrique. Elle pourrait sinon amener des confusions sur les notions de base des classes/ob-

jets/méthodes.

Par défaut, les opérateurs de python ne fonctionnent pas pour nos classes. C'est assez logique car si Python sait très bien comment comparer, par exemple, 2 entiers, il ne peut pas savoir comment comparer des objets de notre invention (quels critères choisir ?). De même, il sait très bien comment afficher un entier à l'écran, mais comment afficher un objet de type voiture ou rectangle ? Qu'afficher, comment le disposer, etc ?

Lorsque l'on définit une classe, python permet donc ce qu'on appelle la surcharge d'opérateurs. Ce mécanisme permet de définir, pour notre classe, comment un opérateur doit se comporter pour des instances de la classe donnée.

Pour ceci, il faut définir une méthode avec un nom spécial pour chaque opérateur (ainsi que les paramètres appropriés), et Python appliquera cette méthode au moment d'utiliser l'opérateur en question.

Ainsi, si je définis une méthode d'addition pour mes voitures (méthode `__add__`), j'aurai ensuite le droit d'écrire `v1 + v2`. Python appellera en fait `v1.__add__(v2)`.

Ex :

```
class Voiture:
    def __init__(self, mon_modele, mes_km):
        self.modele = mon_modele
        self.km = mes_km

    # L'opérateur str définit la façon dont l'objet est transformé
    # en chaîne de caractères.
    # Il est notamment appelé implicitement par la fonction print
    def __str__(self):
        return "Cette voiture est un(e) " + self.modele + " et poss
            ède " + str(self.km) + " km"

    # L'opérateur lt représente l'opération "<"
    # (lesser than = inférieur à)
    # Ici, je décide qu'une voiture est inférieure à une autre
    # si elle a un plus petit kilométrage. NB : je pourrais
    # choisir l'inverse, ou n'importe quel autre critère
    def __lt__(self, v2):
        return self.km < v2.km

v = Voiture("Zoe", 5)
v2 = Voiture("208", 25)

# Comme nous avons défini str et lt, nous avons le droit
# d'écrire les deux lignes suivantes. Sinon, elles auraient
# toutes deux généré une erreur !

print(v) # Affiche les caractéristiques de notre voiture !
print(v<v2) # Affiche True
```

Voici un petit nombre de noms de méthodes représentant des opérateurs. Si vous cherchez le nom de la méthode correspondant à un opérateur en particulier, il est très facile de le trouver dans la documentation sur internet.

Remarque : Lorsque ces méthodes ont besoin d'un paramètre supplémentaire, vous pouvez le nommer

comme vous le voulez (ici *other*).

Exemples :

- Addition : `__add__(self, other)`
- Soustraction : `__sub__(self, other)`
- Conversion en chaîne de caractères : `__str__(self)`
- Inférieur à : `__lt__(self, other)`
- Supérieur à : `__gt__(self, other)`
- Égal à : `__eq__(self, other)`
- ...