La programmation orientée objet (POO)

Capacités attendues

- Écrire la définition d'une classe ;
- Accéder aux attributs et méthodes d'une classe ;
- Utiliser des API (Application Programming Interface) ou des bibliothèques ;
- Exploiter leur documentation ;
- Créer des modules simples et les documenter.

1 Introduction

Toute variable informatique possède un (en Python : `boolean`, `integer`, `float`, `str`, `list`, `dict`, etc.) qui indique à l'interpréteur (ou au compilateur) comment la représenter en mémoire et quelles sont les opérations ou méthodes permises.

Par exemple :

- les entiers, les flottants ou les caractères ne sont pas codés de la même manière en mémoire (cours de 1ère) ;
- l'opérateur '+' permet, selon le contexte, d'additionner des nombres, de concaténer des chaînes de caractères ou d'étendre des listes ;
- les listes possèdent une méthode `append`, mais pas les chaînes de caractères, qui ont une méthode `split` ;
- les chaînes de caractères et les listes sont « itérables » (on peut utiliser la syntaxe 'for el in var' quand var est une chaîne de caractères ou une liste), mais pas les nombres.

Définitions

La programmation orientée objet (POO) est un (une famille) de programmation qui permet en quelque sorte de créer de « nouveaux types », appelés . Dans ce contexte, une variable dont le « type » est associé à une classe est appelée . La création d'un objet s'appelle .

Jusqu'à présent, les paradigmes de programmation que nous avons utilisés sont la programmation **impérative** (suite d'instructions qui s'exécutent dans l'ordre du script) et la programmation **procédurale** qui repose sur l'utilisation de fonctions notamment.

La programmation orientée objet est considérée plus intuitive car plus proche du raisonnement humain.

Point Histoire

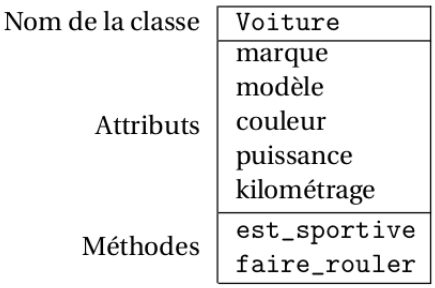
Le paradigme de programmation qu'est la POO a été défini par les norvégiens Ole-Johan Dahl et Kristen Nygaard au début de la décennie 1960. Le premier langage de programmation initiant une forme de programmation orientée objet fut Simula, créé à partir de 1962 ; ce langage servait à faciliter la programmation de logiciels de simulation.
Plus tard, leurs travaux furent repris et amendés dans les années 1970 par l'américain Alan Kay. Le premier langage de programmation réellement fondé sur la programmation orientée objet fut Smalltalk 71, créé au début des années 70.

2 Vocabulaire et théorie

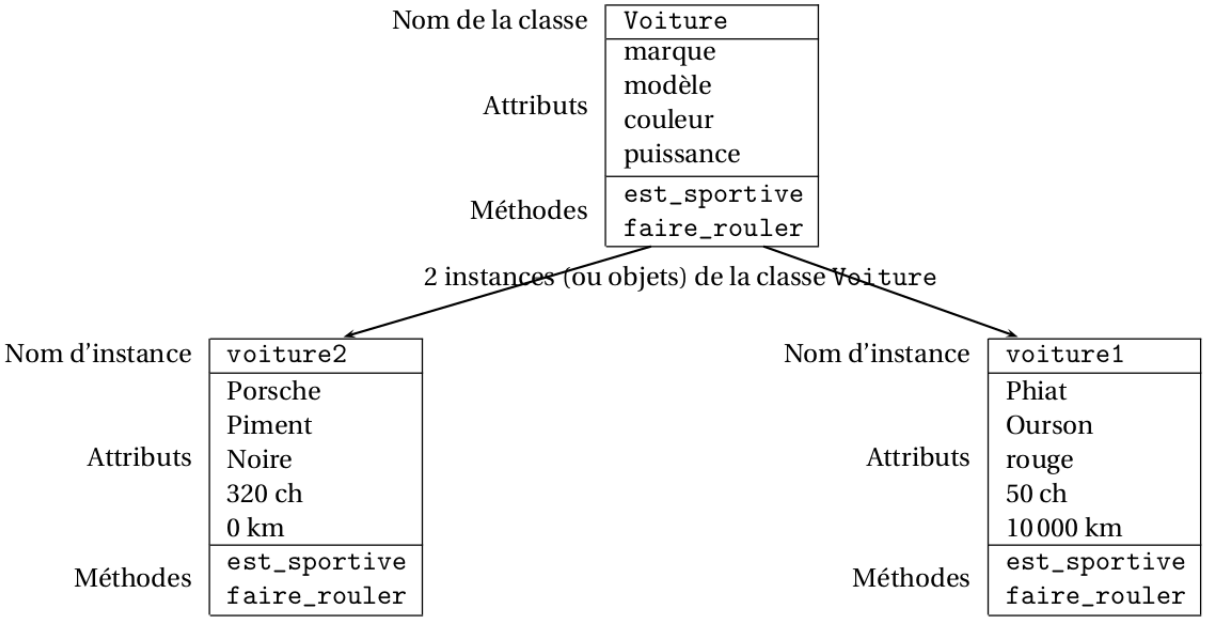
Considérons des voitures caractérisées par une marque, un modèle, une couleur, une puissance et un kilométrage. À l'aide de ces informations, on souhaite déterminer si une voiture est une voiture sportive (si la puissance du moteur est supérieure à 180 ch) ou la faire rouler (en faisant évoluer son kilométrage).

Pour représenter des voitures, on peut créer une `Voiture` qui servira à créer des ayant des (caractéristiques de la voiture : couleur, marque, modèle, etc.) et sur lesquels ont pourra faire des traitements à l'aide de (fonctions qui appartiennent à une classe).

En voici une représentation graphique :



Cette classe `Voiture` permet de créer différentes voitures appelées **objets** ou **instances** de la classe `Voiture`. Chacun des objets créés a des valeurs des attributs qui lui sont propres :



Définitions

La programmation orientée objet (POO) est un **paradigme** de programmation qui permet en quelque sorte de créer de « nouveaux types », appelés **classes**. Dans ce contexte, une variable dont le « type » est associé à une classe est appelée **objet**. La création d'un objet s'appelle **instanciation**.
On peut résumer en disant qu'**un objet est une instance d'une classe**.
Une classe définit :

- la fonction spéciale obligatoire appelée pour créer une instance : c'est le ;
- des variables propres à chaque instance : ce sont les **attributs** ;
- des fonctions qui s'appliquent à l'instance considérée : ce sont les **méthodes**.

L'ensemble des attributs et méthodes (visibles) d'une classe en constituent les **membres**.

3 En Python

3.1 Les classes et leurs attributs

Voici comment la classe `Voiture` s'écrira en Python :

```
1  class Voiture:
2      '''Définition d'une voiture'''
3      def __init__(self, marque, modele, couleur, puissance, km = 0):
4          '''
5          Permet de créer les attributs marque, modèle, couleur, puissance et km
6          à partir des paramètres saisis.
7          '''
8          self.marque = marque
9          self.modele = modele
10         self.couleur = couleur
11         self.puissance = puissance
12         self.km = km
13
14 # Création d'une instance de la classe Voiture ou objet
15 ma_voiture = Voiture('Porsche', 'Piment', 'Noire', 320, 0)
```

Écrire une classe en Python

- On définit une classe en suivant la syntaxe `class NomClasse`. Le nom de la classe s'écrit, par convention, en CamelCase (ligne 1) ;
- On documente la classe avec un *docstring* (ligne 2) ;
- Le mot-clef `self` fait référence à l'objet courant (signife « soi-même » en anglais) ;
- On écrit obligatoirement la méthode `__init__` appelée **constructeur** et qui prend en paramètres `self` et les attributs souhaités (ligne 3) ;
- On initialise les attributs dans le constructeur (lignes 8 à 12) ;
- Un attribut peut avoir une **valeur par défaut** comme `km` ici, qui est initialisé à `0`. Les paramètres par défaut doivent être situés à la fin de la liste des paramètres.

Instancier un objet et le manipuler en Python

- On crée un objet (une instance de classe) avec l'instruction :
  `nom_objet = NomClasse(param1, param2, etc.)`
- On accède aux attributs définis dans le constructeur de la classe en suivant la syntaxe :
  `nom_objet.nom_attribut`
- On modifie les attributs d'une instance avec l'instruction :
  `nom_objet.nom_attribut = valeur`
- Attention, l'instruction `print(nom_objet)` renvoie l'adresse à laquelle l'objet est stocké en mémoire.

3.2 Les méthodes

Dans le paradigme de la programmation orientée objet, la notion de classe va de paire avec à la notion d'**encapsulation** : un programme manipulant un objet n'est pas censé accéder librement à la totalité de son contenu. La manipulation de l'objet se fait par une **interface** constituée de fonctions dédiées.

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure. Ceci permet d'assurer l'intégrité des objets créés (ils partagent tous les mêmes caractéristiques communes) et d'en cacher potentiellement une partie à l'utilisateur.

Définition : interface

En POO, l'interface d'une classe est l'ensemble des méthodes associées à cette classe.

Créer et utiliser une méthode

- Une méthode est une fonction encapsulée dans la classe qui permet de travailler sur l'objet et ses attributs ;
- Les méthodes se définissent comme des fonctions, avec le mot-clef `def`, et leurs corps se trouvent dans le corps de la classe ;
- Les méthodes prennent en **premier paramètre** `self`, qui représente l'objet lui-même, sur lequel la méthode s'applique ;
- Pour définir une méthode, on utilise la syntaxe suivante :

```
def nom_methode(self, parametre1, parametre2):  # autant de paramètres qu'on veut
  # instructions
```

- Pour utiliser une méthode, on utilise la syntaxe suivante :

```
nom_objet.nom_methode(parametre1, parametre2) # pas de paramètre self à l'appel !
```

- Si la méthode a pour seul paramètre `self`, l'appel de la méthode se fait avec l'instruction :

```
nom_objet.nom_methode()
```

Remarque : si `lst` est une variable Python de type `list`, on peut utiliser l'instruction `liste.append(val)` qui permet d'ajouter la valeur `val` à la fin de la liste `lst`.Cette instruction correspond en fait à l'utilisation de la méthode `append` avec l'objet `lst` instancié depuis la classe `list`.

Exemple :

On veut ajouter deux méthodes à la classe `Voiture` précédemment créée. Voici le code Python correspondant :

```
class Voiture:
    '''Définition d'une voiture'''
    def __init__(self, marque, modele, couleur, puissance, km = 0):
        '''
        Permet de créer les attributs marque, modèle, couleur, puissance et km
        à partir des paramètres saisis.
        '''
        self.marque = marque
        self.modele = modele
        self.couleur = couleur
        self.puissance = puissance
        self.km = km

    def est_sportive(self):
        '''
        Renvoie True si la voiture passée en paramètre est sportive et False sinon
        '''
        return self.puissance >= 180

    def rouler(self, vitesse, duree):
        '''
        Fait rouler une voiture à une vitesse en km/h et pendant un temps en h
        et modifie le kilométrage.
        param vitesse: en km/h - type entier
        param duree: en h - type float
        return: nouveau kilométrage de la voiture
        '''
        self.km += self.km + vitesse * duree

# Création d'une instance de la classe objet
voiture1 = Voiture('Porsche', 'Piment', 'Noire', 340, 0)

# Exécution méthode est_sportive
print(voiture1.est_sportive())

# Exéctuion méthode rouler
voiture1.rouler(100, 1)
```

3.3 Les méthodes spéciales

Le constructeur `__init__(self, [parametres])` est une méthode qui fonctionne de manière particulière : elle est exécutée automatiquement lors de l'instanciation de l'objet.

Il existe d'autres méthodes spéciales.

La méthode `__str__`

- reçoit comme unique paramètre self ;
- renvoie une chaîne de caractères ;
- peut être appelée avec la syntaxe `objet.__str__()` mais aussi avec l'instruction `print(objet)`.

Exemple :

On ajoute la méthode `__str__` dans la définition de notre classe `Voiture` :

```
class Voiture:
    '''Définition d'une voiture'''
    def __init__(self, marque, modele, couleur, puissance, km = 0):
        '''
        Permet de créer les attributs marque, modèle, couleur, puissance et km
        à partir des paramètres saisis.
        '''
        self.marque = marque
        self.modele = modele
        self.couleur = couleur
        self.puissance = puissance
        self.km = km

    def est_sportive(self):
        '''
        Renvoie True si la voiture passée en paramètre est sportive et False sinon
        '''
        return self.puissance >= 180

    def rouler(self, vitesse, duree):
        '''
        Fait rouler une voiture à une vitesse en km/h et pendant un temps en h
        et modifie le kilométrage.
        param vitesse: en km/h - type entier
        param duree: en h - type float
        return: nouveau kilométrage de la voiture
        '''
        self.km += self.km + vitesse * duree

    def __str__(self):
        '''
        Affichage de la marque et du modèle de la voiture
        '''
        texte = "Marque %s, Modèle : %s" % (self.marque, self.modele)
        return texte

    print(voiture1)
```

Ceci affiche :

```
>>> %Run poo.py
Marque Porsche, Modèle : Piment
```

Les autres méthodes spéciales

méthode	appel	effet
`__lt__(self, b)`	`a < b`	Renvoie `True` si `a` est strictement inférieur à `b` et `False` sinon.
`__eq__(self, b)`	`a == b`	Renvoie `True` ssi `a` est égal à `b` et `False` sinon.
`__len__(self)`	`len(a)`	Renvoie un entier décrivant la taille de l'objet `a`.
`__getitem__(self, i)`	`a[i]`	Renvoie `i` ^{e} élément de l'objet `a`.
`__contains__(self, b)`	`a in b`	Renvoie `True` ssi `u` est dans l'objet `a` et `False` sinon.
`__add__(self, b)`	`a + b`	redéfini l'opérateur +
`__mul__(self, b)`	`a * b`	redéfini l'opérateur $*$

Remarque : Pour obtenir la liste complète des méthodes que l'on peut utiliser avec une structure de données ou une classe, on utilise l'instruction `dir(nom_objet)`. On peut remarquer que lors de la création de l'objet, il existe un ensemble de méthodes spéciales définies par défaut.

4 Encapsulation et attributs privés

Par défaut, les attributs de classe, tels qu'on les a vus, sont **publics**. Ceci signifie qu'on peut instancier une voiture `voiture1` dans un script Python et faire ensuite un `print(voiture1.km)` qui affichera la valeur associée à l'attribut `km`.

Dans la philosophie de la programmation orientée objet, l'interaction avec les objets d'une classe se fait à travers les méthodes. L'accès direct aux attributs avec l'instruction `nom_objet.nom_attribut` est déconseillé car on ne veut pas forcément que l'utilisateur ait accès à la représentation interne des classes. Pour utiliser ou modifier les attributs, on utilisera de préférence des méthodes dédiées, appelées les **accesseurs** et les **mumtateurs**, dont le rôle est de faire l'interface entre l'utilisateur de l'objet et la représentation interne de l'objet (ses attributs). Aussi, pour éviter un accès direct aux attributs sans passer par ces méthodes, il est recommandé de créer des attributs **privés**.

Remarque : L'utilisation d'une classe par un utilisateur extérieur se fait en consultant sa documentation. Pour y accéder, on utilise l'instruction `help(nom_classe)`. C'est pour cette raison qu'il est important d'utiliser les *docstrings* pour documenter son programme car c'est cette partie qui sera affichée via la fonction `help`.

Attributs privés, accesseurs et mutateurs

- On réalise la protection des attributs d'une classe en commençant le nom des attributs par un double *underscore* : `__mon_attribut_prive`. De tels attributs sont dits **privés** ;
- Pour accéder aux valeurs d'attributs privés, on écrit des méthode appelées **accesseurs** ou ***getters***. Par convention, ces méthodes sont nommées de la façon suivante : `get_xxx` ;
- Pour modifier les valeurs d'attributs privés, on écrit des méthodes appelées **mutateurs** ou ***setters***. Par convention, ces méthodes sont nommées de la façon suivante : `set_xxx`.

Exemple :

On redéfinit la classe Voiture avec le constructeur suivant :

```
class Voiture:
    '''Définition d'une voiture'''
    def __init__(self, marque, modele, couleur, puissance, km=0):
        '''
        Permet de créer les attributs marque, modèle, couleur, puissance et km
        à partir des paramètres saisis.
        '''
        self.__marque = marque
        self.__modele = modele
        self.__couleur = couleur
        self.__puissance = puissance
        self.__km = km

# Création d'une instance de la classe Voiture
voiture1 = Voiture('Porsche', 'Piment', 'Noire', 340)
```

Avec une telle définition de la classe :

- L'instruction `voiture.__marque` renvoie un message d'erreur ;
- L'instruction `voiture.__marque = 'Phiat'` ne modifie pas la valeur de l'attribut `__marque` mais en crée un nouveau dont la valeur est `Phiat`.

Pour accéder à cet attribut ou le modifier, il faut écrire et utiliser un accesseur et un mutateur :

```
class Voiture:
    '''Définition d'une voiture'''
    def __init__(self, marque, modele, couleur, puissance, km=0):
        '''
        Permet de créer les attributs marque, modèle, couleur, puissance et km
        à partir des paramètres saisis.
        '''
        self.__marque = marque
        self.__modele = modele
        self.__couleur = couleur
        self.__puissance = puissance
        self.__km = km

    def get_marque(self):
        '''
        Donne la marque de la voiture.
        '''
        return self.__marque

    def set_marque(self, nv_marque):
        '''
        Change la marque de la voiture.
        '''
        self.__marque = nv_marque

# Création d'une instance de la classe Voiture
voiture1 = Voiture('Porsche', 'Piment', 'Noire', 340)
print(voiture1.get_marque())
voiture1.set_marque("Phiat")
print(voiture1.get_marque())
```