Programmation orientée objet

Introduction :

Dans ce cours nous nous intéressons à la programmation objet sous l'angle des conteneurs de données. Dans une première partie, nous présenterons les principales caractéristiques du paradigme objet, son vocabulaire et sa syntaxe. Nous aborderons dans une deuxième partie la gestion des états internes des objets, puis de leurs fonctionnalités dans une troisième partie.

1 Principes de la programmation objet

a. Le paradigme objet

L'**approche objet** permet de créer des structures de données et d'y associer un certain nombre de fonctionnalités. Cette approche fait partie des paradigmes majeurs de programmation.

Rappel

Un paradigme est un style de programmation , une manière de modéliser les problèmes et d'écrire des programmes pour les résoudre.

La principale caractéristique du paradigme objet est d'associer étroitement les données et le code pour les traiter, selon le principe de l'**encapsulation**.

Définition

Encapsulation :

L'encapsulation désigne le regroupement des données et du code relatif au traitement de ces données.

Cette encapsulation se matérialise par la définition d'une classe d'objets. Celle-ci établit globalement les types de données et y associe des fonctionnalités communes à tous les objets de cette classe.

Attention

> Certains langages assortissent ce regroupement d'un contrôle très strict de l'accès aux données, mais nous verrons que ce n'est pas le cas en Python.

Les objets individuels sont créés à partir de la classe, par un mécanisme appelé **instanciation**.

Définition

> **Instanciation :**
>
> L'instanciation désigne la création d'un objet appartenant à une classe.
>
> → L'objet devient donc une « instance » de la classe dont il émane.

Les instances partagent des fonctionnalités communes, mais les données sont individuellement propres à chaque objet.

À retenir

> Dans la terminologie objet, les données sont appelées **attributs**, tandis que les fonctions associées sont appelées **méthodes**.

Ce cours aborde la programmation objet sous l'angle des structures de données. Dans ce contexte, nous n'étudierons pas les notions d'héritage et de polymorphisme qui appartiennent également au paradigme de programmation objet.

b. Définition d'une classe

À retenir

> Une classe se définit à partir du mot-clé `class` suivi du nom de celle-ci. Par convention, les noms de classe commencent par une majuscule.

```
    pass
```

Dans le cas où le nom est composé, on utilise la convention dite « *camel case* » (« casse de chameau » en français), qui consiste à juxtaposer les mots en mettant leurs premières lettres en majuscules.

```
class ExempleComportantPlusieursMots:
    pass
```

Les objets encapsulent des propriétés et des méthodes, comme le montre l'exemple ci-après de définition d'une classe.

```
class Eleve:
    """
    Classe représentant des élèves.
    Un élève est caractérisé par :
    - son prénom,
    - son âge.
    """

    def __init__(self, prenom, age):
        """construction d'un objet élève à partir de son prénom et de son âge"""
        self.prenom = prenom self.age = age

    def informations(self):
        """indique le prénom et l'âge de l'élève"""
        return '{} a {} ans'.format(self.prenom, self.age)
```

Cette portion de code ci-dessus ne fabrique aucun objet. Elle se contente de définir les propriétés et fonctionnalités communes à tous les objets qui appartiendront à cette classe.

Notre classe `Eleve` définit les méthodes nommées `__init__` et `informations`.

À retenir

La méthode appelée `__init__` est le constructeur de nos futurs objets. Elle sera automatiquement appelée au moment de la création des objets de la classe `Eleve`. Nous observons qu'elle requiert :

- un paramètre appelé `self` ;
- deux autres paramètres qui sont les attributs des élèves, à savoir leur prénom et leur âge.

Les méthodes prennent un premier paramètre désignant l'objet individuel auquel elles s'appliquent. Par convention ce paramètre est appelé `self` en Python.
Cette convention varie selon les langages : ce paramètre est ainsi nommé `this` en Javascript, et `$this` en PHP.

⚠ Attention

Sa présence est nécessaire à la définition de la méthode au sein de la classe, mais il ne doit pas être passé en argument lors des créations d'objets ou des appels de leurs méthodes.

La méthode appelée `informations` restitue sous forme de chaîne de caractères les informations définies pour un élève donné. Elle ne requiert aucun paramètre hormis `self`.

c. Instanciation d'objets

Les objets sont instanciés (autrement dit créés) à partir de la classe à laquelle ils appartiennent.

```
eleve1 = Eleve('Lisa', 17)
```

Les arguments passés pour la création de l'objet « eleve1 » de la classe `Eleve` sont ceux définis par la méthode `__init__`, c'est-à-dire son prénom et son âge. Cette méthode est appelée automatiquement pour construire l'objet. Le paramètre implicite `self` ne doit pas être précisé.

d. Appel de méthodes

La syntaxe générale pour l'appel d'une méthode s'effectue selon le format suivant :

`objet.methode(arguments)`.

Si la méthode requiert des paramètres (autres que `self` qui n'a pas à être spécifié au moment de l'appel), les arguments correspondants doivent être présents entre les parenthèses, de la même manière qu'en programmation fonctionnelle.

Les objets de notre classe `Eleve` disposent d'une méthode `informations` que nous pouvons appeler avec la notion pointée.

```
print(eleve1.informations())
# affiche Lisa a 17 ans
```

L'appel de la méthode s'effectue sans passer d'arguments, puisqu'elle est définie sans paramètre autre que `self`. Le paramètre `self` utilisé dans la déclaration de la fonction au sein de la classe permet de désigner notre objet au sein de la classe. Nous évitons ainsi le recours à une autre syntaxe, possible mais moins compacte, qui oblige à préciser à la fois la classe et le nom de l'instance pour obtenir le même résultat :

```
print(Eleve.informations(eleve1))
# affiche Lisa a 17 ans
```

> À l'instar des fonctions, les parenthèses sont obligatoires pour appeler une méthode, même si on ne lui passe aucun paramètre.

Créons à présent un nouvel objet de la même classe pour un deuxième élève.

```
eleve2 = Eleve('Paul', 18)
```

```
print(eleve2.informations())
# affiche Paul a 18 ans
```

> Cet exemple montre l'intérêt de la programmation objet : la possibilité de disposer de méthodes communes à des objets distincts dont les valeurs d'attributs peuvent être différentes.

e. Chaînes de documentation

Les chaînes de documentation (*docstrings* en anglais) contribuent, quand elles sont présentes, à informer sur la classe et ses méthodes.

```
help(Eleve)
```

affiche le texte suivant :

```
Help on class Eleve in module __main__:

class Eleve(builtins.object)
 | Eleve(prenom, age)
 |
 | Classe représentant des élèves.
 |
 | Un élève est caractérisé par :
 | - son prénom,
 | - son âge.
 |
 | Methods defined here:
 |
 | __init__(self, prenom, age)
 |construction d'un objet élève à partir de son prénom et de son âge
 |
 | informations(self)
 |indique le prénom et l'âge de l'élève
 |
 | ----------------------------------------------------------------------
 | Data descriptors defined here:
 |
 | __dict__
 |dictionary for instance variables (if defined)
 |
```

```
| __weakref__
|list of weak references to the object (if defined)
```

Maintenant que ces éléments de vocabulaire et de syntaxe sont précisés, intéressons-nous aux états internes des objets et aux modalités d'accès à ceux-ci.

2 | États internes des objets

- Le paradigme de programmation fonctionnelle repose sur l'**absence d'état des fonctions**. Le résultat d'une fonction ne dépend que des arguments fournis en entrée, sans conservation d'états internes une fois le résultat retourné.
- Le paradigme de programmation objet permet pour sa part une **conservation d'états internes** : les objets contiennent non seulement des fonctions, appelées méthodes, mais aussi des variables, appelées attributs.

→ On distingue :

a. les variables d'instance ;

b. et les variables de classe.

a. Variables d'instance

Les objets peuvent contenir tout type de variable, et ces variables sont accessibles aux méthodes afin que ces dernières puissent les manipuler.

Nous avons vu dans la partie précédente qu'elles pouvaient être définies au moment de l'instanciation de l'objet avec la méthode du constructeur, mais il est possible de définir ou de modifier des variables à tout moment. Dans l'exemple ci-dessous, nous modifions la méthode `informations` pour qu'elle génère les initiales d'un élève quand elle est appelée.

```
class Eleve:

    def __init__(self, prenom, nom):
        self.prenom = prenom.title()
        self.nom = nom.title()
```

```
    def informations(self):
        self.initiales = '{}.{}.'.format(self.prenom, self.nom)
        return 'Les initiales de {} {} sont {}'.format(self.prenom, self.nom, self.initiales)
```

Nous instancions un élève et appelons la méthode qui retourne des informations sur celui-ci.

```
tom = Eleve('Thomas', 'Martin')
print(tom.informations())
# affiche : Les initiales de Thomas Martin sont T.M.
```

Les variables des objets, ou attributs d'instance, sont accessibles et modifiables avec la notation pointée.

```
print(tom.prenom)
# affiche Thomas
```

```
tom.nom = 'Dupont'
print(tom.informations())
# affiche Les initiales de Thomas Dupont sont T.D.
```

Contrairement à d'autres langages, Python permet également de déclarer librement des attributs non explicitement définis dans la classe.

```
tom.loisirs = ['natation', 'escalade']
```

Cette faculté permet si on le souhaite d'utiliser une classe comme conteneur de données. La présence de méthodes n'est pas requise pour définir une classe en Python.

```
class Enregistrement():
    pass # ne fait rien
```

```
lisa = Enregistrement() # création d'une instance
lisa.nom = 'Legrand'
lisa.prenom = 'Lisa'
lisa.age = 17
```

Les variables d'instance sont stockées dans l'objet sous forme d'un dictionnaire.

```
print(lisa.__dict__)
# affiche {'nom': 'Legrand', 'prenom': 'Lisa', 'age': 17}
```

b. Variables de classe

Des variables peuvent également être définies au niveau de la classe. Dans ce cas, elles sont communes à tous ses membres, et toute modification d'une variable de classe concernera logiquement l'ensemble des objets qui y sont rattachés.

Astuce

> La syntaxe le reflète avec l'absence de `self` et le rattachement explicite de la variable à la classe dans les méthodes qui manipulent la variable.

Nous l'illustrons avec une classe d'élèves qui comptabilise le nombre d'élèves.

→ Le compteur correspondant est incrémenté à chaque fois qu'un élève est ajouté à la classe par instanciation d'un nouvel objet de cette classe.

```
class Eleve:

    nombre_eleves = 0

    def __init__(self, prenom):
        self.prenom = prenom.title()
        Eleve.nombre_eleves += 1

    def informations(self):
        return '{} est dans une classe comportant {} élève(s)'.format(self.prenom, Eleve.nombre_eleves)

lisa = Eleve('Lisa')
print(lisa.informations())
# affiche Lisa est dans une classe comportant 1 élève(s)

paul = Eleve('Paul')
sarah = Eleve('Sarah')
```

```
print(sarah.informations())
# affiche Sarah est dans une classe comportant 3 élève(s)
```

Nous avons présenté les attributs d'instance et de classe avec des variables de type chaîne de caractères et nombre entiers, mais les autres types de variables sont pareillement utilisables au sein des classes.

c. Accès aux variables

Certains langages obligent à utiliser des méthodes pour lire ou modifier les attributs des objets.

À retenir

> Ces méthodes sont appelées :
>
> - **accesseur** (ou *getter* en anglais) pour une méthode permettant de connaître la valeur d'un attribut ;
> - **mutateur** (ou *setter* en anglais) pour une méthode permettant de modifier un attribut.

Il est possible d'en implémenter en Python, mais cela n'a rien d'obligatoire. L'utilisation d'accesseurs ou de mutateurs peut présenter un intérêt quand les valeurs doivent faire l'objet d'un traitement particulier. Ou bien pour éviter qu'un·e utilisateur·rice de la classe n'introduise des incohérences, en intervenant sur les contenus des variables, hors du cadre des méthodes implémentées.

```
class Eleve:

    def __init__(self, prenom):
        self.prenom = prenom.title().strip()

    def get_prenom(self):
        return self.prenom

    def set_prenom(self, prenom):
        self.prenom = prenom.title().strip()
```

```
lisa = Eleve('lisa ')
lisa.get_prenom()
# affiche Lisa

lisa.set_prenom('eLISAbeth')
lisa.get_prenom()
# affiche Elisabeth
```

On observe que le mutateur applique des traitements à la chaîne de caractère passée en argument :

- transformation si nécessaire de la casse pour mettre en majuscule la première lettre du prénom et les autres en minuscules (méthode *title*) ;
- suppression d'éventuels espaces avant et après le prénom (méthode *strip*).

Nous ne développerons pas davantage cet aspect, mais le langage Python propose différents mécanismes spécifiques d'encapsulation plus ou moins forte, avec des attributs privés ou protégés.

d. Implémentation et interface

L'interface proposée par l'objet peut exposer directement et de manière transparente les structures de données qu'il manipule. Mais cette interface ne reflète pas nécessairement les états ou les types internes des objets.

L'exemple développé ci-après montre un pseudo-dictionnaire implémenté dans une classe :

- dont l'accesseur proposé en interface retourne un dictionnaire ;
- mais dont l'implémentation interne s'appuie sur deux listes mises à jour en parallèle.

```
class PseudoDictionnaire:
    def __init__(self):
        self.keys = []
        self.values = []

    def set_key(self, key, value):
```

```
            self.keys.append(key)
            self.values.append(value)

    def getpseudodictionnaire(self):
        return {key: value for key, value in zip(self.keys, self.values)}

donnees = PseudoDictionnaire()
donnees.set_key('prenom', 'Alice')
donnees.set_key('age', 17)

resultat = donnees.getpseudodictionnaire()
print(resultat)
# affiche {'prenom': 'Alice', 'age': 17}
print(type(resultat))
# affiche
```

Cette implémentation est tout sauf optimale, mais elle illustre la distinction à opérer entre interface et implémentation d'une structure de données.

- L'interface propose une forme d'accès aux données qui n'est pas nécessairement représentative de leur structure interne.
- Cette structure interne peut donc être modifiée tout en conservant à l'identique l'interface.

Nous avons vu le rôle des variables de différents types dans le paradigme objet en tant que structure de données, mais les méthodes que nous allons étudier maintenant contribuent, elles aussi, à l'interface de ces objets.

3 | Fonctionnalités

Rappel

> La programmation objet permet d'implémenter des fonctionnalités, liées aux données qu'elle structure, par l'implémentation de méthodes. Les méthodes sont des fonctions définies au sein de la classe. Elles peuvent accéder et manipuler les attributs des objets, auxquels peuvent être appliqués différents traitements.

À retenir

On distingue deux types de méthodes en Python : les méthodes normales et les méthodes spéciales.

Définition

Méthodes normales :

Les méthodes normales correspondent, dans le paradigme objet, aux fonctions créées en programmation fonctionnelle. La présence des fonctions dans la définition d'une classe les rend accessibles sous forme de méthodes dans les objets qui y sont rattachés.

Méthodes spéciales :

Les méthodes spéciales se distinguent par leur nommage entouré de doubles tirets bas (*underscores* en anglais), sous le format `__methodespeciale__`. Elles permettent d'étendre les fonctionnalités des objets en leur conférant des caractéristiques particulières.

Nous avons déjà rencontré une méthode spéciale, avec le constructeur `__init__`.

Quand cette méthode est présente dans la définition de la classe, elle est exécutée automatiquement à la création de chaque objet. Sa présence n'est toutefois pas indispensable pour créer des objets émanant d'une classe.

Il existe un certain nombre de méthodes spéciales qui permettent d'apporter aux objets que nous créons des fonctionnalités particulières. Nous allons l'illustrer en présentant l'implémentation de deux d'entre elles, dont la mission est, respectivement, de mesurer et de comparer des objets.

a. Longueur d'un objet

Les objets peuvent disposer de nombreuses caractéristiques, à condition d'implémenter les méthodes correspondantes.

Nous concevons une classe poème destinée à analyser des poésies.

```
class Poeme:

    def __init__(self, poeme):
        self.poeme = poeme
```

Nous créons une instance avec un poème de l'auteur japonais Matsuo Basho.

```
petit_poeme_japonais = """Un vieil étang
Une grenouille qui plonge,
Le bruit de l'eau."""

haiku = Poeme(petit_poeme_japonais)
```

Notre objet étant créé, nous souhaitons connaitre sa longueur.

```
print(len(haiku))
# affiche TypeError: object of type 'Poeme' has no len()
```

La notion de longueur n'est pas définie pour notre objet.

Nous pourrions obtenir la longueur de la chaîne de caractères nommée `petit_poeme_japonais` ayant servi à créer notre objet, mais l'idéal serait que cette caractéristique puisse être fournie par l'objet lui-même, en étant appelée en tant que méthode rattachée à l'objet.

L'implémentation de la méthode spéciale `__len__` va nous permettre d'accéder à cette fonctionnalité. Nous redéfinissons la classe en ajoutant cette méthode qui retourne la longueur de la variable d'instance `self.poeme` qui contient le poème :

```
class Poeme:

    def __init__ (self, poeme):
        self.poeme = poeme

    def __len__(self):
        return len(self.poeme)
```

À nouveau, nous créons une instance à partir de cette nouvelle définition de la classe.

```
haiku = Poeme(petit_poeme_japonais)
```

Notre objet étant créé, nous passons la variable qui le référence en argument à la fonction native `len()`.

```
print(len(haiku))
# affiche 60
```

À retenir

> L'existence de la méthode spéciale `__len__` permet à l'interpréteur Python de retourner une longueur pour les objets de la classe ainsi redéfinie.

Notre méthode retourne simplement la longueur de la chaîne, mais nous pouvons également définir une autre manière de déterminer cette longueur.

Considérons les deux poèmes suivants :

1. *Chanson d'automne*, de Paul Verlaine ;
2. *Le Dormeur du val*, d'Arthur Rimbaud.

```
verlaine = """Les sanglots longs
Des violons
De l'automne
Blessent mon coeur
D'une langueur
Monotone."""
```

```
rimbaud = """C'est un trou de verdure où chante une rivière
Accrochant follement aux herbes des haillons
D'argent ; où le soleil, de la montagne fière,
Luit : c'est un petit val qui mousse de rayons"""
```

Notre classe implémentant la longueur de manière littérale à partir de la longueur de la chaîne de caractères, nous obtenons les résultats suivants

en instanciant les deux objets correspondants :

```
automne = Poeme(verlaine)
dormeur = Poeme(rimbaud)
print(len(automne), len(rimbaud))
# affiche 87 185
```

→ Le poème de Verlaine comporte 87 caractères, celui de Rimbaud 185 caractères.

Mais nous pourrions considérer la longueur de ces poèmes différemment, en mesurant plutôt le nombre de lignes qu'ils comportent.

 Exemple

> Cette notion serait par exemple utile à un·e éditeur·rice pour la publication d'un recueil, afin de déterminer le nombre de pages de l'ouvrage.

Il nous suffit de redéfinir notre classe afin que la longueur soit désormais définie par le nombre de lignes du poème.

→ Nous le déterminons avec la longueur de la liste obtenue en divisant la chaîne de caractères du poème sur la base du caractère spécial `\n` de retour à la ligne).

```
class Poeme:

    def __init__(self, poeme):
        self.poeme = poeme

    def __len__(self):
        self.lignes=self.texte.split('\n')

return len(self.lignes)
```

Nous instancions à nouveau deux objets à partir de cette nouvelle définition de notre classe.

Observons à présent leurs longueurs respectives en appelant la fonction native `len()` comme précédemment.

```
automne = Poeme(verlaine)
dormeur = Poeme(rimbaud)
print(len(automne), len(dormeur))
# affiche 6 4
```

→ Nous obtenons cette fois 6 et 4, car le poème de Verlaine comporte 6 lignes, et celui de Rimbaud seulement 4.

Le résultat obtenu est très différent de notre précédente implémentation.

b. Comparaison d'objets

Les méthodes spéciales nous permettent également d'implémenter des comparaisons entre objets, sous réserve de définir les méthodes correspondantes.
Nous allons l'illustrer avec le principe d'un jeu de lettres privilégiant l'usage de lettres rares pour former des mots. Ce principe sert de base à de nombreux jeux de société.

Nous implémentons ce principe avec une classe `Mot`. Cette classe comporte :

1. une variable de classe, de type dictionnaire, définissant le nombre de points attribués pour l'usage de chacune des lettres de l'alphabet ;
2. un constructeur qui convertit le mot proposé en majuscules et conserve uniquement les lettres répertoriées dans le dictionnaire de lettres, et qui en calcule le score par comptage des points.

```
class Mot:

    points = {'A': 1, 'B': 2, 'C': 2, 'D': 2, 'E': 1, 'F': 5, 'G': 2, 'H': 5,
              'I': 1, 'J': 5, 'K': 10, 'L': 1, 'M': 2, 'N': 1, 'O': 1, 'P': 2,
              'Q': 5, 'R': 1, 'S': 1, 'T': 1, 'U': 1, 'V': 2, 'W': 10, 'X': 5,
              'Y': 5, 'Z': 10}
```

```
    def __init__(self, mot):
        self.mot = ''.join(lettre for lettre in mot.upper() if lettre in Mot.points)
        self.score = sum(Mot.points[lettre] for lettre in self.mot)

wok = Mot('wok')
casserole = Mot('casserole')

print(wok.score)
# affiche 21

print(casserole.score)
# affiche 10
```

Nous pouvons calculer les scores des objets, mais pas les comparer entre eux.

```
print(wok < casserole)
```

affiche TypeError: '<' not supported between instances of 'Mot' and 'Mot'[

Redéfinissions notre classe pour implémenter des méthodes de comparaison entre les instances.

Nous déclarons pour cela les méthodes spéciales `__lt__` (pour *lesser than*, soit « plus petit que ») et `__le__` (pour *lesser or equal*, soit « plus petit que ou égal à »).
Ces méthodes requièrent deux paramètres, `self` pour désigner l'objet de comparaison et un second paramètre, conventionnellement nommé `other` (« autre »), pour désigner l'objet auquel le premier est comparé.

Le critère de comparaison est librement défini au sein des méthodes correspondantes. Pour les besoins de notre jeu, nous souhaitons effectuer cette comparaison sur la base des scores.

```
class Mot:

    points = {'A': 1, 'B': 2, 'C': 2, 'D': 2, 'E': 1, 'F': 5, 'G': 2, 'H': 5,
        'I': 1, 'J': 5, 'K': 10, 'L': 1, 'M': 2, 'N': 1, 'O': 1, 'P': 2,
        'Q': 5, 'R': 1, 'S': 1, 'T': 1, 'U': 1, 'V': 2, 'W': 10, 'X': 5,
        'Y': 5, 'Z': 10}
```

```
    def __init__(self, mot):
        self.mot = ''.join(lettre for lettre in mot.upper() if lettre in Mot.points)
        self.score = sum(Mot.points[lettre] for lettre in self.mot)

    def __lt__(self, other):
        return self.score < other.score

    def __le__(self, other):
        return self.score <= other.score

wok = Mot('wok')
casserole = Mot('casserole')

print(wok < casserole)
# affiche False
```

Le résultat obtenu est correct : le score de wok est de 21, il est supérieur à celui de casserole qui est de 10.

→ L'expression évaluée est donc fausse.

Nous n'avons pas besoin de définir en miroir les méthodes spéciales complémentaires `__gt__` (pour *greater than*, soit « plus grand que ») et `__ge__` (pour *greater or equal*, soit « plus grand que ou égal à »), pour évaluer la comparaison associée.

```
print(wok > casserole)
# affiche True
```

Astuce

> Sans avoir recours à une méthode spéciale, il reste possible de comparer le score des deux objets de la manière suivantes :
> `print(wok.score < casserole.score)`

Nous avons montré un aperçu non exhaustif des méthodes spéciales afin d'illustrer l'intérêt de celles-ci pour enrichir les fonctionnalités des objets. Il est également possible de définir les contenus des méthodes appelées

pour la suppression d'un objet, l'addition d'objets entre eux, mais aussi l'itération sur les objets (entre autres).

Nous pouvons ainsi personnaliser le comportement de nos objets en implémentant différentes méthodes spéciales prévues par le langage.

> Conclusion :
>
> Les objets constituent un intéressant paradigme de programmation, associant variables et fonctions sous la forme d'attributs et de méthodes. Nous avons décrit le vocabulaire et la syntaxe de la programmation objet. Nous avons ensuite précisé les différents types de variables ou attributs et leurs modalités d'accès. Nous avons enfin montré qu'il était possible de conférer des fonctionnalités particulières aux objets en implémentant les méthodes correspondantes.