FORMATION PYTHON

POO AVANCÉ - Partie 1

Introduction

L'*orientation objet* est l'approche de programmation qui consiste à grouper les données et les traitements associés au sein d'entités cohérentes appelées "objets".

On utilise des classes d'objet pour cela.

Syntaxe de définition d'une classe d'objets

```
class MaClasse:
    """ Documentation de la classe """
    # corps de la classe

    def __init__(self, param1, param2, ...):
        pass

    ...
```

Note : La fonction `__init__()` effectue l'initialisation de l'objet. Elle est appelée automatiquement lorsqu'un objet de cette classe est créé.

Exemple

Une classe pour gérer des personnes, avec les informations de nom et d'age :

```
>>> class Personne:
...     def __init__(self, nom):
...         self.nom = nom
...         self.age = 0
```

Instances et attributs

- On peut créer un objet `p1` à partir de la classe `Personne`, en l'appelant avec les paramètres nécessaires. On parle d'*instance de la classe* pour `p1`, et d'*instanciation* pour l'opération de création de l'objet.

```
>>> p1 = Personne('Jean Dupont')
```

- La fonction `type()` permet de confirmer que l'objet est de la classe `Personne`.

```
>>> type(p1)
```

- Dans la définition du `__init__()`, le paramètre `self` référence le nouvel objet créé.
- Dans l'exemple, les éléments 'nom' et 'age' sont appelés les **attributs d'instance**.
- Une fois l'instance `p1` créée, on peut lire la valeur de ses attributs :

```
>>> p1.nom
'Jean Dupont'
>>> p1.age
0
```

- Créons une autre instance :

```
>>> p2 = Personne('Marie Curie')
```

- On peut changer la valeur d'un attribut sur l'objet ; utiliser l'opérateur d'accès '.' de la même manière que pour lire la valeur :

Si on donne une nouvelle valeur à l'age de la personne...

```
>>> p1.age = 32
```

... cette nouvelle valeur est accessible :

```
>>> p1.age
32
```

Introspection d'objets

- La fonction prédéfinie `dir()` est un outil très utile pour introspecter tout objet.
- Permet d'obtenir la liste des attributs d'un objet.

```
>>> a = 50
>>> dir(a)
```

Méthodes

- Attributs "spéciaux" similaires aux fonctions.
- Une méthode a un premier paramètre obligatoire, le paramètre `self` qui référence l'objet manipulé (l'instance de la classe), et éventuellement d'autres paramètres.
- `__init__()`, que nous avons déjà vu, est une méthode.
- Ajoutons la méthode `.separeNom()` à la classe dans notre exemple :

```
class Personne:
    def __init__(self, nom):
        self.nom = nom
        self.age = 0

    def separeNom(self):
        nom_complet = self.nom
        return nom_complet.split()
```

- Appelons la méthode sur un objet :

```
>>> p1 = Personne('Jean Dupont')
>>> p1.separeNom()
['Jean', 'Dupont']
```

Attributs de classe vs. Attributs d'instance

En dehors des **attributs d'instance**, il est possible d'avoir des attributs appartenant à la classe elle-même.

Faisons évoluer notre exemple pour illustrer ce cas !

```
class Personne:

    pays = ['france', 'suisse', 'allemagne']

    def __init__(self, nom):
        self.nom = nom
        self.age = 0
```

On se sert de l'attribut de classe en dehors de toute création d'instance :

```
liste_pays = Personne.pays
print(liste_pays)
```

Dans cet exemple, on se sert de l'attribut de classe comme une constante. Mais il est également possible de modifier les attributs de classe.

```
class Personne:

    pays = ['france', 'suisse', 'allemagne']

    compteur = 0

    def __init__(self, nom):
        self.nom = nom
        self.age = 0

        self.__class__.compteur += 1

# Testons la classe...
print(f'Valeur du compteur au debut : {Personne.compteur}')

p1 = Personne('Jean Dupont')
print(p1, f'Nouvelle valeur du compteur : {Personne.compteur}')

print(f'Valeur du compteur : {Personne.compteur}')

p2 = Personne('John Doe')
print(p2, f'Nouvelle valeur du compteur : {Personne.compteur}')

print(f'Valeur du compteur a la fin : {Personne.compteur}')
```

Getter et Setter

- La programmation objet permet de passer par des propriétés pour manipuler la valeur des attibuts des objets, et garantir l'encapsulation des données.
- Par convention, bien que non obligatoire, on utilise des méthodes :
    - **getter** (ou accesseur),
    - **setter** (ou mutateur) pour changer la valeur d'un attribut.

Getter

Permet d'accéder à la valeur de l'attribut.

Exemple : `get_age(self)` pour l'attribut `_age`, sur une classe.

```
>>> class Customer:
...     def __init__(self, age):
...         self._age = 20
...
...     def get_age(self):
...         return self._age
...
>>> c = Customer()
>>> print(c.get_age())
20
```

Setter

- Permet de modifier la valeur de l'attribut.
- La méthode "setter" prend comme second argument la nouvelle valeur.

Exemple : `set_age(self)` pour l'attribut `_age`, sur la même classe.

```
>>> class Customer:
...     def __init__(self, age):
...         self._age = 20
...
...     def get_age(self):
...         return self._age
...
...     def set_age(self, x):
...         self._age = x
...
>>> c = Customer()
>>> print(c.get_age())
20
>>> c.set_age(25)
>>> print(c.get_age())
25
```

Le décorateur `@property`

L'utilisation du décorateur `@property` nous fournit une syntaxe concise, une facilité d'accès aux instances des attributs et la possiblité de réutiliser le nom de nos attibuts sans définir de nouvelles méthodes.

```
>>> class Customer:
...     def __init__(self, age):
...         self._age = 20
...
...     @property
...     def age(self):
...         return self._age
...
...     @age.setter
...     def age(self, x):
...         self._age = x
...
>>> c = Customer()
>>> print(c.age)
20
>>> c.age = 22
>>> print(c.age)
22
```

Méthodes spéciales

Convention de nommage

`__func__`

Exemples : `__init__`, `__len__`, `__add__`

Fonctionnement

Une méthode spéciale est utile lorsque l'on veut implémenter des types de fonctionnalités ou comportements classiques dans Python.

Une méthode spéciale est appelée dans un des cas suivants :

- Appel d'une fonction prédéfinie.
- Utilisation d'un opérateur arithmétique / relationnel / logique, ou de l'opérateur d'accès aux éléments d'une liste ou d'un dictionnaire (opérateur `[]`).

Méthode spéciale pour Fonction prédéfinie

Soit la fonction prédéfinie `func`, elle correspond à la méthode spéciale `__func__` : l'appel de `func(obj)` correspond à `obj.__func__()`.

```
>>> a = 'Real Python'
>>> len(a)
11
>>> a.__len__()
11
```

```
>>> b = ['Real', 'Python']
>>> str(b)
"['Real', 'Python']"
>>> b.__str__()
"['Real', 'Python']"
>>> print(b)
['Real', 'Python']
```

Méthode spéciale pour Opérateur arithmétique

Soit un opérateur `opr` (par exemple `+` ou `-`), il correspond à la méthode spéciale `__opr__` : `A opr B` équivaut à `A.__opr__(B)` où A et B sont des objets.

Exemples d'opérateurs avec leur méthodes spéciales :

- L'opérateur `+` équivaut à `__add__()`
- L'opérateur `[]` équivaut à `__getitem__()`
- L'opérateur `-` équivaut à `__sub__()`
- L'opérateur `*` équivaut à `__mul__()`

Testons !

```
>>> a = 10
>>> b = 25
>>> a + b
35
>>> a.__add__(b)
>>> 35
```

```
>>> a = "Hello"
>>> b = "Python"
>>> a + " " + b
'Hello Python'
>>> a.__add__(b)
>>> a.__add__(" ").__add__(b)
'Hello Python'
```

Méthode spéciale pour Opérateur d'accès aux éléments d'une liste

La méthode spéciale `__getitem__` est appelée, en interne, lorsque l'on accède à un élément d'une liste par son indice.

```
>>> maliste = ['X', 'Y', 'Z']
>>> maliste[0]
'X'
>>> maliste.__getitem__(0)
'X'
```

Possibilités de surcharge

Surcharge de fonctions prédéfinies

Redéfinition, au niveau de la classe, de la méthode spéciale correspondant à la fonction prédéfinie.

Cas pratique : Avec `__len__` pour `len()` sur une classe que l'on définit. ***Voir exercices.***

Surcharge d'opérateurs

Redéfinition, au niveau de la classe, de la méthode spéciale correspondante.

Cas pratique : Avec l'opérateur `+`, et donc la méthode `__add__()`, sur une classe que l'on définit. ***Voir exercices.***