Python objet – TP 2

1. Attributs et méthodes de classe

Attributs de classe

Les attributs de classe sont simplement des attributs non déclarés dans les méthodes, mais directement dans la classe. Toutes les instances de la classe peuvent y accéder et la modifier, mais pour la modifier de manière « globale », il faut l'appeler avec le nom de la classe.

```
class Etudiant():
    counter = 0

    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age
        Etudiant.counter +=1
```

Et le code pour utiliser la classe :

```
>>> Etudiant.counter
0
>>> etu = Etudiant('Lennon','John',25)
>>> Etudiant.counter
1
>>> etu.counter
1
>>> etu.counter += 1
>>> etu.counter
2
>>> Etudiant.counter
1
```

Méthodes de classe

De la même manière que pour les attributs, on peut définir des méthodes de classe, qui n'ont pas forcément leur place dans une instance. Exemple : on peut définir une méthode de classe nbEtudiants() qui sera une méthode de classe. Attention, on ne met pas self dans les arguments de la méthode ! Par contre, elle ne peut pas être appelée par une instance.

```
Class Etudiant() :
        def nbEtudiants() :
                return Etudiant.counter
```

2. Héritage simple et multiple

Héritage simple

Si MaClasseFille hérite de MaClasseMere :

```
class MaClasseFille(MaClasseMere):
    def __init__(self):
        MaClasseMere.__init__(self)
```

La classe fille peut toujours appeler les méthodes de la classe mère en utilisant MaClasseMere.methode().

Héritage multiple

L'héritage multiple existe mais est peu répandu.
Pour connaître l'ordre de résolution, on peut utiliser l'outil **MRO** (Method Resolution Order) :

```
class A():
    pass

class B():
    pass

class C(A, B):
    pass

print(C.__mro__)
```

L'exécution du code renvoie :
```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)
```

3. Visibilité

Visibilité des attributs et méthodes

Par principe, en Python, **tout est public**. On estime que l'indication de la manière dont on va utiliser les objets peut être simplement « suggéré » par le programmeur.
On peut donc marquer _mon_attribut avec un underscore pour dire « je pense que vous ne devriez pas accéder à cet attribut à l'extérieur de la classe », mais rien n'empêchera un utilisateur de le faire.

Les properties Python

La lisibilité doit être considérée comme beaucoup plus importante que la durée d'écriture de code. Un code est écrit une fois, mais sera certainement lu des dizaines voire des centaines de fois dans la vie d'un logiciel.

Dans les objets Python il n'existe pas de propriété privée et nous n'avons jamais besoin des getters ou des setters. Tous les objets sont fait pour qu'on accède directement à leurs propriétés. J'entends déjà les cris des développeurs JAVA : "Mais et l'encapsulation des données ? Si on veut refactoriser et contrôler la propriété comment on fait sans getter et setter ?".

En Python il existe ce qu'on appelle les **properties.** Voici les avantages de ce système :

- Pas besoin de créer (ou de générer) vos getters et setters.
- La syntaxe est exactement la même que pour accéder à une propriété.
- Vous pouvez les rajouter quand vous voulez.
- Le code Python utilisant la classe est plus lisible.

La pratique

Pour les manipulations sur les objets suivants, nous définissons une classe Celsius() qui peut instancier des températures, et les convertir en degrés Fahrenheit.

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

Si on veut modifier la température, le code ressemble à ça :

```
>>> t = Celsius(35)
>>> t.temperature -= 800
>>> print(t.temperature)
```

Attention, si la température est inférieure à -273, on doit lancer une erreur. Une solution simple est de rajouter des accesseurs/mutateurs et faire un test quand la valeur est modifiée. Un souci est alors que le code ressemble maintenant à ça (ce qui est bien moins lisible, et moins maintenable, car le code est modifié) :

```
>>> t = Celsius(35)
>>> t.set_temp() = t.get_temp() - 800
>>> print(t.get_temp())
```

La solution est alors d'utiliser la fonction **property**(), qui va « cacher » les accesseurs et mutateurs à l'utilisateur : le code reste alors le même !

```
class Celsius:
    def __init__(self, temperature = 0):
        self._temperature = temperature # avec underscore !

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temp(self):
        print("Getting value")
        return self._temperature # avec underscore !

    def set_temp(self, value):
        if value < -273:
            print("Temperature below -273 is not possible")
        else:
            print("Setting value")
            self._temperature = value # avec underscore !

    temperature = property(get_temp,set_temp)
```

On remarque :

- Il faut écrire la ligne property… après la déclaration des deux méthodes get_temp () et set_temp ().
- Il faut penser à transformer tous les self.temperature en self._temperature (avec le underscore) pour deux raisons : pour montrer au programmeur que l'attribut est privé, et pour éviter les boucles sans fin dans les fonctions get_temp() et set_temp()