UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Licence informatique & vidéoludisme
Semestre 1

Méthodologie de la programmation

Chapitre 2

Programmation orientée objet avec python (2ème partie)

1. Les objets en python

Getters et Setters

Pour accéder aux attributs d'instance, il est de convention d'utiliser des méthodes : les setters et les getters

```
1
2  class Animal(object):
3      def __init__(self, name, age) :
4          self.name = name
5          self.age = age
6
7      def get_name(self):
8          return self.name
9
10     def set_name(self, new_name):
11         self.name = new_name
```

Décorateurs @property

Il existe une manière plus élégante de faire des getters, le décorateur de fonction **@property**.

```
1
2 class Animal(object):
3     def __init__(self, name):
4         self._name = name
5
6     @property
7     def name(self):
8         return self._name
```

Décorateurs @property

Il permet d'accéder à l'attribut comme avant tout en empêchant certains comportements indésirables

```
1
2  class Animal(object):
3      def __init__(self, name):
4          self._name = name
5
6      @property
7      def name(self):
8          return self._name
9
10 a = Animal('Pluto')
11 print(a.name)
12 # 'Pluto'
```

Décorateurs @property

les attributs et les getters décorés par @property ne peuvent pas avoir le même nom. Il est de tradiction de nommer les attributs cachés avec le préfixe "_".

```
1
2  class Animal(object):
3      def __init__(self, name):
4          self._name = name
5
6      @property
7      def name(self):
8          return self._name
9
10 a = Animal('biche')
11 print(a.name)
12 # 'biche'
```

Décorateurs @property

Il est possible grâce au décorateur @property de créer des attributs temporaires qui ne seront calculés qu'à l'appel du setter.

```
1  class Animal(object):
2      def __init__(self, age):
3          self._age = age
4
5      @property
6      def nimp(self):
7          return self._age * 42
8
9  a = Animal(42)
10 print(a.nimp)
```

Décorateurs @attribut.setter

De la même manière, il exite un moyen plus élégant de créer des getters.
Grâce au décorateur @**attribut.setter**.

```
1  class Animal(object):
2      def __init__(self, name):
3      self._name = name
4
5      @property
6      def name(self):
7          return self._name
8
9      @name.setter
10     def name(self, name):
11         self._name = name
```

L'héritage

- Les classes héritent toutes d'une classe parente, ou super-classe.
- Chaque classe est donc la sous-classe d'une autre classe (en Python3), et les sous-classes :
  - Héritent de leur classe-parente leur comportement et leurs attributs de classe ;
  - Implémentent de nouvelles méthodes ;
  - Peuvent ajouter de nouveaux attributs ;
  - Peuvent changer (override, surcharger) le comportement de leur classe-parente.

L'héritage

La sous-classe appelle le constructeur de la classe mère.

```
1 class Animal(object):
2     def __init__(self, name, age):
3         self._name = name
4         self._age = age
5
6 class Chat(Animal):
7     pass
8
9 a = Chat("Felix", 42)
```

L'héritage

On peut implémenter de nouvelles méthodes.

```
1 class Chat(Animal):
2     def speak(self):
3         print("miaou")
4
5 a = Chat("Felix", 42)
6 a.speak()
7 # miaou
```

L'héritage

On peut implémenter de nouvelles méthodes ... mais celles-ci ne seront pas accessibles à l'objet de la classe mère :

```
1 class Chat(Animal):
2     def speak(self):
3         print("miaou")
4
5 a = Animal("Bob", 42)
6 a.speak()
7 # 'Animal' object has no attribute 'speak'
```

L'héritage

- Les sous-classes peuvent avoir des méthodes portant le même nom que celles de la superclasse ;
- Une instance de la classe va d'abord chercher une méthode dans son scope avant de remonter récursivement l'arborescence de ses parent ;
- Les variables de classe sont partagées par toutes les instances de cette classe mais aussi de ses sous-classes.

Sources

- Cours de Pablo Rauzy (lien)
- Cours de Jean-Pascal Palus (lien)
- courspython.com