UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Méthodologie de la Programmation

Palus Jean-Pascal

Licence IV - L1 - Semestre I

Bureau A184 - jpp@up8.edu

Séance IV

La programmation orientée objet II

Getters et setters

- Pour accéder aux attributs d'instance, il est de convention d'utiliser des méthodes plutôt que d'exposer directement les attributs.

```
class Animal(object):

    def __init__(self, name, age) :
        self.name = name
        self.age = age

    def get_name(self):
        return self.name

    def set_name(self, new_name):
        self.name = new_name
```

Getters et setters

- Utiliser **classe.get_attribut()** plutot que **classe.attribut** évite les bugs en cas de changement du comportement interne de la classe.

```
class Animal(object):

    def __init__(self, name, age) :
        self.name = name
        self.years = age

    def get_age(self):
        return self.years

    def set_age(self, new_age):
        self.years = new_age
```

Python n'est pas bon pour cacher le comportemet interne des classes.

- Autorise l'accès aux données en dehors de la définition de la classe :

```
print(animal.age)
```

- Autorise l'accès aux attributs :

```
animal.age = 'infini'
```

- Autorise de créer de nouveaux attributs en dehors de la définition de la classe :

```
animal.taille = 'petit'
```

Décorateurs @property

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

```
class Animal(object):
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name
```

Décorateurs @property

- Il permet d'appeler le getter de la même manière qu'un attribut.

```
class Animal(object):
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

a = Animal('foo')
print(a.name)
# 'foo'
```

Décorateurs @property

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

```
class Animal(object):
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

a = Animal('foo')
print(a.name)
# 'foo'
```

Décorateurs @property

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

```
class Animal(object):
    def __init__(self, age):
        self._age = age

    @property
    def nimp(self):
        return self._age * 42

a = Animal(42)
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**.

```
class Animal(object):
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        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).

L'héritage

- 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.

```
class Chat(Animal):
    pass

a = Chat("foo", 42)
```

L'héritage

- On peut implémenter de nouvelles méthodes.

```
class Chat(Animal):
    def speak(self):
        print("miaou")

a = Chat("foo", 42)
a.speak()
# miaou
```

L'héritage

- On peut implémenter de nouvelles méthodes..
- Mais celles-ci ne sont implémentées que dans la sous-classe.

```
class Chat(Animal):
    def speak(self):
        print("miaou")

a = Animal("foo", 42)
a.speak()
# 'Animal' object has no attribute 'speak'
```

L'héritage

- Les sous-classes peuvent avoir des méthodes portant le même nom que leur 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 parents.

L'héritage

- Les variables de classe sont partagées par toutes les instances de cette classe mais aussi de ses sous-classes.

Super()

- La fonction **Super()** permet d'appeler une méthode de la classe parente.

! Elle n'attend pas l'objet instancié comme premier argument.

```
class Humain(Animal):
    def __init__(self, name, age):
        # Appelle le constructeur de Animal
        super().__init__(name, age)
        self._amis = []
```

L'héritage multiple

- Il est possible, bien que déconseillé, de créer une classe héritant de plusieurs superclasses à la fois.

L'héritage multiple

```
class GroundVehicle(object):
    def drive(self):
        print("Drive !")

class FlyingVehicle(object):
    def fly(self):
         print("Fly !")

class FlyingCar(GroundVehicle, FlyingVehicle):
    pass

fc = FlyingCar()
fc.drive()
fc.fly()
```

Method Resolution Order

- Chaque classe dispose d'un attribut **__mro__** permettant de connaître l'ordre dans lequel sera recherché les méthodes et attributs dans l'arborescence de l'héritage.

! Cet attribut n'est accessible qu'au niveau de la classe et non de l'objet.

L'héritage multiple

```
class GroundVehicle(object):
    def drive(self):
        print("Drive !")

class FlyingVehicle(object):
    def fly(self):
         print("Fly !")

class FlyingCar(GroundVehicle, FlyingVehicle):
    pass

fc = FlyingCar()
print(FlyingCar.__mro__)
# (<class '__main__.FlyingCar'>,
# <class '__main__.GroundVehicle'>,
# <class '__main__.FlyingVehicle'>, <class 'object'>)
```