

Atelier 3 – Introduction à l'encapsulation en Python

Cet atelier est prévu pour 2h.


Petit rappel

Au premier chapitre, nous avons déjà parlé d'encapsulation... Rappelez-vous :

L'encapsulation

Comme une gélule médicinale (= capsule en anglais), un objet présente :

- une partie **transparente** dont on peut voir le contenu depuis l'extérieur
 - son interface, ses éléments de visibilité publique
 - ce à quoi le monde extérieur a accès
- une partie **opaque** dont le contenu est caché
 - sa partie privée, interne
 - ce à quoi le monde extérieur ne peut pas accéder directement



Par ailleurs, dans le labo 2, on avait défini l'encapsulation comme étant le fait de regrouper les données et les méthodes qui permettent de les manipuler au « même endroit » et de ne pas aller modifier/accéder directement ces données.



Quels sont les avantages de l'encapsulation ?

Comment faire pour ne pas directement accéder aux données ?

Plutôt que d'aller chercher directement la valeur dans l'objet, le programmeur va **demander à l'objet** de lui donner cette valeur. → Grâce à un **sélecteur**

Et plutôt que d'aller modifier directement la valeur dans l'objet, le programmeur va **demander à l'objet** de remplacer cette valeur par une nouvelle. → Grâce à un **modificateur**

Sélecteurs (accesseurs, "getters")

- Méthodes de consultation / lecture
- Retour de tout l'état ou d'une partie de l'état d'un objet

Modificateurs (mutateurs, "setters")

- Méthodes d'écriture / de modification
- Modifier tout l'état ou une partie de l'état d'un objet

Les principes d'encapsulations en fonction du langage

Pour commencer, une petite précision : certains langages, comme C++ ou en Java, par exemple, mettent en place des « droits d'accès » dans la définition de classe qui indiquent pour chaque attribut ou méthode si elle est privée ou public.

Pour rappel, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, on ne peut pas. On doit passer par des accesseurs ou mutateurs. Essayer d'utiliser un attribut privé en dehors de sa classe produira une erreur.

En python, la philosophie est un peu différente : on fait globalement confiance aux gens qui vont réutiliser du code. La plupart du temps, les attributs sont totalement publics ! Quelques « subterfuges » permettent tout de même de donner une « illusion » de vie privée. Pour faire respecter l'encapsulation en python, les développeurs utilisent des conventions et comptent sur le bon sens des utilisateurs (qui sont eux-mêmes développeurs) des classes. Si quelqu'un écrit que cet attribut est privé, on part du principe que personne ne cherchera à y accéder depuis l'extérieur... A ses risques et périls...

Les propriétés

La plupart des langages définissent les sélecteurs et modificateurs comme suit :

- Les sélecteurs :
 - o sont nommés `get_<attribut>()`
 - o ne prennent pas de paramètre
 - o retournent la valeur correspondante
- Les modificateurs :
 - o sont nommés `set_<attribut>()`
 - o prennent comme paramètre la nouvelle valeur
 - o mettent à jour cette valeur
 - o ne retournent rien

Python, encore une fois, fait les choses différemment. Il définit ce qu'on appelle des **propriétés** qui jouent le rôle de sélecteurs et modificateurs mais de manière « invisible ».

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe qu'il faut définir les propriétés.

Exemple : classe Human

Il y a plusieurs manières d'écrire les propriétés en Python. Si vous désirez plus d'informations à ce sujet, n'hésitez pas à vous renseigner dans la doc ou notamment via ce lien (<https://www.programiz.com/python-programming/property>).

Voici une manière d'écrire les propriétés parmi tant d'autres. C'est celle que nous utiliserons dans ce cours :

Reprenons la classe Human du chapitre 2 :

| Human |
|------------|
| name |
| age |
| gender |
| say() |
| birthday() |

```
class Human :  
  
    def __init__(self, name) :  
        self.name = name  
        self.age = 0  
        self.gender = random.choice('MF')  
  
    def say(self, message) :  
        print(self.name, ":", message)  
    def birthday(self) :  
        self.age += 1  
        print("Happy Birthday", self.name)
```



Recopiez la classe dans un script en n'oubliant pas de rajouter `import random` au début. Ajoutez un bloc `main` avec le code suivant, puis exécutez le script.

```
bob = Human("Bob")  
print(bob.name)  
bob.name = "Bobby"  
print(bob.name)
```



Maintenant, rendez l'attribut `name` privé en suivant cette démarche : mettez un `_` devant le nom de l'attribut et écrivez son sélecteur et son modificateur.

Pour le sélecteur :

- nommez le `name`
- précédez le du décorateur `@property`.
- mettez uniquement `self` comme paramètre (car il s'agit d'une méthode d'instance)
- il doit renvoyer le nom de l'objet

Pour le modificateur

- nommez le `name` également
- précédez le du décorateur `@name.setter`.
- mettez `self` (car il s'agit d'une méthode d'instance) et le nouveau nom en paramètre
- il doit modifier le nom de l'objet

Si vous rencontrez des difficultés, la solution se trouve à la page suivante...

Comme ceci :

```
class Human :  
  
    def __init__ (self, name) :  
        self._name = name  
        self.age = 0  
        self.gender = random.choice('MF')  
  
    @property  
    def name(self):  
        return self._name  
    @name.setter  
    def name(self, new_name)  
        self._name = new_name  
  
    def say(self, message) :  
        print(self.name, ":", message)  
    def birthday(self) :  
        self.age += 1  
        print("Happy Birthday", self.name)
```

Vous remarquerez que 2 de vos méthodes ont le même nom ! Cela fonctionne uniquement grâce aux décorateurs (`@property` et `@name.setter`).

Par facilité, regroupez toujours le sélecteur et le modificateur dans votre classe.

Pour fonctionner, le décorateur du modificateur (`name.setter` dans notre cas) commence par le nom du sélecteur (qui est à priori le même que l'attribut) suivi de « `.setter` ».



Réexécutez votre code.

Vous voyez que, malgré le fait que vous ayez changé le nom de l'attribut `name` (qui est devenu `_name`), `bob.name` fonctionne toujours. C'est grâce aux propriétés Python. Quelques soient les changements que vous faites dans votre classe et quelques soient les attributs que vous rendez privé, l'utilisation de votre classe ne change pas à l'extérieur. Il y a toujours l'illusion d'accéder directement aux attributs.




Rendez `age` et `gender` privé et écrivez leurs sélecteurs et modificateurs en suivant l'exemple de `name`.




Exécutez le `main` suivant :

```
if __name__ == "__main__" :  
    bob = Human("Bob")  
    print(bob.name)  
    bob.name = "Bobby"  
    print(bob.name)  
  
    bob.age = 10  
    print(bob.age)
```

Réflexions

 Après réflexion, vous vous dites qu'une personne ne peut pas changer d'âge comme elle le souhaite. Après tout, la seule possibilité pour changer d'âge c'est d'avoir son anniversaire et on ne peut pas rajeunir... **enlevez le modificateur de l'âge et réexécutez votre code. Que se passe-t-il ?**

Vous aurez normalement une erreur semblable à ceci : `AttributeError: can't set attribute`. En effet, comme le modificateur n'existe plus, vous ne pouvez plus modifier l'âge...

 Remplacez votre `main` par le code suivant et exécutez le :


```
if __name__ == "__main__" :  
    bob = Human("Bob")  
    bob.say("I am " + str(bob.age) + " years old")
```


Autre réflexion, stocker directement l'âge d'une personne n'est pas vraiment la meilleure solution... l'idéal est de stocker la date de naissance de cette personne. Comme ça, plus besoin d'utiliser la méthode `birthday` pour changer l'âge...


Pour cela, quelques petits changements s'imposent dans votre classe.

Pour manipuler des dates, nous allons utiliser une classe existante de Python : `datetime` : <https://docs.python.org/3/library/datetime.html>. Et plus précisément `datetime.date`.

`Datetime.date` a (entre autre) 3 attributs : `year`, `month` et `day`. Elle a aussi une méthode `today()` qui donne la date du jour.

 Rajoutez `import datetime` au début de votre script.


 Remplacez l'attribut `_age` par `_birthday` qui vaut `datetime.date.today()` et écrivez-lui un sélecteur.

 Supprimez l'ancienne méthode `birthday`.

Le sélecteur de l'âge existe toujours, on va le laisser. Cependant, l'attribut `_age`, lui, n'existe plus. Il va donc falloir le réécrire.

Pour cela, rappelez-vous que `datetime.date.today()` vous donne la date du jour. Il suffit ensuite de soustraire l'année de naissance à l'année d'aujourd'hui pour avoir l'âge.

Attention cependant que si l'anniversaire de la personne n'est pas encore passé cette année, son âge est, en fait, de 1 de moins.

 Modifiez le sélecteur `age`.

Si vous rencontrez des difficultés, un template à compléter se trouve à la page suivante...

Comme ceci : (remplacez les ###)

```
@property
def age(self):
    today = datetime.date.today()
    age = ###

    if (today.month ### self._birthday.month) or (today.month ###
self._birthday.month and today.day ### self._birthday.day):
        age -= 1

    return ###
```



Relancez votre script. Vous verrez que malgré le changement de représentation de l'état de l'objet, il fonctionne toujours.



Pour pouvoir tester un peu plus le programme, modifiez le constructeur de `Human` en permettant de donner la date de naissance (paramètre facultatif) comme ceci :

```
def __init__(self, name, bd = None) :
    self._name = name
    if bd :
        self._birthday = bd
    else :
        self._birthday = datetime.date.today()
    self._gender = random.choice('MF')
```



Relancez votre script.
Rien ne devrait changer.



Modifiez votre `main` comme ceci (en remplaçant `<ANNEE>`, `<MOIS>` et `<JOUR>`) et vérifiez que l'âge est le bon.

```
if __name__=="__main__" :
    bd = datetime.date(<ANNEE>, <MOIS>, <JOUR>)
    bob = Human("Bob",bd)
    bob.say("I am " + str(bob.age) + " years old")
```

Mais tout n'était qu'illusion...



Essayez de modifier la date de naissance comme ceci :

```
if __name__=="__main__" :
    bd = datetime.date(<ANNEE>, <MOIS>, <JOUR>)
    bob = Human("Bob",bd)
    bob.say("I am " + str(bob.age) + " years old")

    new_bd = datetime.date(<ANNEE>, <MOIS>, <JOUR>)
    bob.birthday = new_bd
    bob.say("I am " + str(bob.age) + " years old")
```

Vous aurez encore une erreur qui vous dit que vous ne pouvez pas modifier l'attribut car il est « privé » et il n'y a pas de modificateur.



Ajoutez simplement un `_` devant birthday dans votre code comme ceci :

```
if __name__=="__main__" :
    bd = datetime.date(<ANNEE>, <MOIS>, <JOUR>)
    bob = Human("Bob",bd)
    bob.say("I am " + str(bob.age) + " years old")

    new_bd = datetime.date(<ANNEE>, <MOIS>, <JOUR>)
    bob._birthday = new_bd
    bob.say("I am " + str(bob.age) + " years old")
```

Et oui, ça fonctionne... Vous êtes allé modifier directement l'attribut, sans passer par un modificateur ! L'encapsulation en Python est **totale**ment **illuso**ire ! Encore une fois, Python part du principe que les développeurs sont de bonne volonté quand ils utilisent les classes et ne font pas ce qu'on vient de faire...

Conclusion : l'encapsulation en Python, contrairement à la plupart des langages, est purement illusoire et Python fait totalement confiance au programmeur à ce sujet. **C'est donc le rôle du programmeur de respecter les bonnes pratiques car le langage ne va pas les vérifier.**