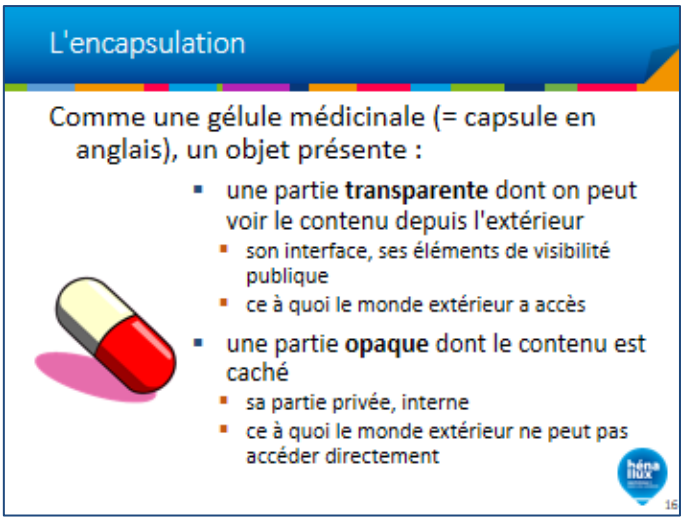
# Atelier 3 – Introduction à l'encapsulation en Python

Cet atelier est prévu pour 2h.

## Petit rappel

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




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 ?

*L'encapsulation regroupe les données et le code d'un objet et considère cet objet comme une boite noir. Elle permet d'utiliser un objet sans connaitre son fonctionnement interne.*
*L'encapsulation implique un* ***couplage faible*** *du code ce qui permet notamment de modifier les structures de données internes de l'objet sans modifier son « interface ». C'est-à-dire la partie avec laquelle le programmeur interagit.*

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

*Le fichier final correspondant à l'énoncé est disponible sur moodle. Des explications supplémentaires sont données à chaque étape*

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 : *(et c'est à priori celle qu'on attendra de vous à l'examen)*

Reprenons la classe Human du chapitre 2 :

| Human |
| --- |
| name<br>age<br>gender |
| say()<br>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.

```
if __name__ == "__main__" :
    bob = Human("Bob")  # crée un Humain appelé "Bob" stocké dans bob
    print(bob.name)     # affiche le nom de l'objet bob
    bob.name = "Bobby"  # change le nom de l'objet bob à "Bobby"
    print(bob.name)     # affiche le nom de l'objet bob
```

*En faisant ça, vous créez un humain appelé Bob et vous affichez son nom. Ensuite vous modifiez son nom en « Bobby » avant d'afficher le nouveau nom.*

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.

→ *Transformer tous les self.name par self._name au sein de la classe. Pour le reste, la réponse se trouve juste après*

Pour le sélecteur :

- nommez le `name` → *def 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 → *def name(...)*
- 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

Comme ceci :

```
class Human :

  def __init__ (self, name) :
    self._name = name       # n'oubliez pas de changer par _name ici
    self.age = 0
    self.gender = random.choice('MF')

  @property
  def name(self):
     return self._name      # n'oubliez pas de changer par _name ici aussi
  @name.setter
  def name(self, new_name)
    self._name = new_name   # n'oubliez pas de changer par _name ici aussi

  def say(self, message) :
    print(self.name, ":", message) # pour bien faire, ici aussi :p
  def birthday(self) :
    self.age += 1
    print("Happy Birthday", self.name) # Ah et là aussi :p
```

*Quand vous utilisez bob.name ou self.name ... bref un objet .name, pour faire simple, python regarde s'il y a un attribut ou un accesseur (ou modificateur selon le contexte) et utilise l'attribut correspondant*

*Ah, au fait, lisez ces remarques, elles sont quand même vraiment intéressantes ;)*

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) suivit de « .setter ».

Réexécutez votre code.

Vous voyez que, malgré le fait que vous ayez change 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`.

*Donc, ça donne ça :*

```
class Human :

    def __init__ (self, name, bd) :
        self._name = name
        self._birthday = bd
        self._gender = random.choice('MF')

    @property
    def name(self):
        return self._name
    @name.setter
    def name(self, new_name):
        self._name = new_name

    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, new_age):
        self._age = new_age

    @property
    def gender(self):
        return self._gender
    @gender.setter
    def gender(self, new_gender):
        self._gender = new_gender
```

Exécutez le `main` suivant :

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

    bob.age = 10      # grace au modificateur
    print(bob.age)    # grace au selecteur
```

## 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 ?**

➔ *bah vous avez une erreur… C'est marqué juste après…*

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 modifiez 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")
```

➔ *ça affiche : "Bob : I am 0 years old"*

---

Autre réflexion, stocker directement l'âge d'une personne n'est pas vraiment la meilleur 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'age…

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`.

```
import random
import datetime

class Human :
    def __init__ (self, name) :
        self._name = name
        self._birthday = datetime.date.today()
        self._gender = random.choice('MF')

    … # sélecteurs/modificateurs de name, age et gender

    @property
    def birthday(self):    #sélecteur de birthday
        return self._birthday

    … # méthode say()
```

Le sélecteur de l'age 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`.

Comme ceci : (remplacez les ###)

```
    @property
     def age(self):
         today = datetime.datetime.now()
         age = today.year - self._birthday.year

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

         return age
```

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

*L'état de l'objet n'est plus représenté de la même façon (on ne stocke plus l'âge mais la date de naissance). Mais grâce à l'encapsulation, ce changement d'état est passé « inaperçu ». Imaginez ça dans un plus gros projet ; vous serez bien content de ne pas devoir chercher tous les endroits à modifier après un changement de représentation d'état ;)*

➔ *Vive l'encapsulation et le couplage faible !*

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 :

*Juste histoire de pouvoir mettre une date de naissance au choix…*

```
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")
```

Par exemple avec :

```
if __name__=="__main__" :
    bd = datetime.date(2000, 4, 21)
    bob = Human("Bob",bd)
    bob.say("I am " + str(bob.age) + " years old")
```

On obtient :

```
Bob : I am 19 years old
```

Avec :

```
if __name__=="__main__" :
    bd = datetime.date(2000, 2, 21)
    bob = Human("Bob",bd)
    bob.say("I am " + str(bob.age) + " years old")
```

On obtient :

```
Bob : I am 20 years old
```

→ ça marche :D

PS: on est en mars 2020 au moment d'écrie cette solution au cas où ;)

## 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 **totalement illusoire** ! 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.**

*La conclusion est quand-même super importante hein !*