Université Frères Mentouri Constantine 1
Faculté Sciences de la Terre, de la Géographie et de l'Aménagement du Territoire

جامعة الإخوة منتوري قسنطينة ا
Frères Mentouri Constantin I University
Université Frères Mentouri Constantine I

Programmation Orientée Objet

M. A. Bouteljda

Première approche des classes

- Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs et nos premières méthodes. Nous allons aussi essayer de comprendre les mécanismes de la programmation orientée objet en Python.
- Au-delà du mécanisme, l'orienté objet est une véritable philosophie et Python est assez différent des autres langages, en termes de philosophie justement. Restez concentrés, ce langage n'a pas ni de vous étonner !

Pourquoi utiliser des objets ?

- Les premiers langages de programmation n'incluaient pas l'orienté objet. Le langage C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C.
- Java, un langage apparu à peu près en même temps que Python, définit une philosophie assez différente de celle du C++ : contrairement à ce dernier, le Java exige que tout soit rangé dans des classes.

Pourquoi utiliser des objets ?

- En Python, la liberté est plus grande. Après tout, vous avez pu passer une grande partie sans connaître la façade objet de Python. Et pourtant, le langage Python est totalement orienté objet : en Python, tout est objet, vous n'avez pas oublié ? Quand vous croyez utiliser une simple variable, un module, une fonction. . ., ce sont des objets qui se cachent derrière.
- Ce sur quoi je souhaite attirer votre attention, c'est que plusieurs langages intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que le C++ ou le Java, ne tenez pas pour acquis que vous allez retrouver les même mécanismes et surtout, la même philosophie. Gardez autant que possible l'esprit dégagé de tout préjugé sur la philosophie objet de Python.
- Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet. J'irais jusqu‘à dire que ce qu'on a vu jusqu'ici, ce n'était qu'une façon un peu plus esthétique de coder : il est plus simple et plus compréhensible d'écrire **ma_liste.append(5)** que **append_to_list(ma_liste, 5)**. Mais derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

Choix du modèle

- Bon, comme vous vous en souvenez sûrement (du moins, je l'espère), une classe est un peu un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.
- Mais qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. Mais on serait bien limité si on ne pouvait faire ses propres classes.
- Pour l'instant, nous allons modéliser. . . une personne. C'est le premier exemple qui me soit venu à l'esprit, nous verrons bien d'autres exemples avant la n de la partie.

Convention de nommage

- il est préférable d'utiliser pour des noms de classes la convention dite Camel Case.
- Cette convention n'utilise pas le signe souligné _ pour séparer les mots. Le principe consiste à mettre en majuscule chaque lettre débutant un mot, par exemple : MaClasse.
- Pour dénir une nouvelle classe, on utilise le mot-clé class.
- Sa syntaxe est assez intuitive :

```
class NomDeLaClasse:
```

Nos premiers attributs

- N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.
- Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne? Beaucoup de choses, vous en conviendrez.
- Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.
- Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe.

- Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un constructeur, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.
- Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.
- Voyons le code, ce sera plus parlant :

```
1  class Personne: # Définition de notre classe Personne
2      """Classe définissant une personne caractérisée par :
3      - son nom
4      - son prénom
5      - son âge
6      - son lieu de résidence"""
7
8
9      def __init__(self): # Notre méthode constructeur
10         """Pour l'instant, on ne va définir qu'un seul attribut
               """
11         self.nom = "Dupont"
```

- Une docstring commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.
- La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque classique d'une fonction. Elle a pour nom *__init__*, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé *self*.

- Une docstring commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.
- La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque classique d'une fonction. Elle a pour nom __*init*__, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé *self*.

- Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

```
>>> bernard = Personne()
>>> bernard
<__main__.Personne object at 0x00B42570>
>>> bernard.nom
'Dupont'
>>>
```

Quand on tape *Personne()*, on appelle le constructeur de notre classe Personne, d'une façon quelque peu indirecte. Celui-ci prend en paramètre une variable un peu mystérieuse : *self*. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attribut *nom* le plus simplement du monde : *self.nom = "Dupont"*. À la n de l'appel au constructeur, Python renvoie notre objet *self* modifié, avec notre attribut. On va réceptionner le tout dans notre variable bernard.

Étoffons un peu notre constructeur

- Bon, on avait dit quatre attributs, on n'en a fait qu'un. Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois, tout de même !
- C'est juste. Dans un premier temps, on va se contenter de définir les autres attributs, le prénom, l'âge, le lieu de résidence.

```
>>> jean = Personne()
>>> jean.nom
'Dupont'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
'Paris'
>>> # Jean déménage..
... jean.lieu_residence = "Berlin"
>>> jean.lieu_residence
'Berlin'
>>>
```

- Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons. . . le nom et le prénom, pour commencer.

```
def __init__(self, nom, prenom):
    """Constructeur de notre classe"""
    self.nom = nom
    self.prenom = prenom
    self.age = 33
    self.lieu_residence = "Paris"
```

Attributs de classe

- Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs nom, prenom,. . . de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe. Voyons un exemple :

```
1 class Compteur:
2     """Cette classe possède un attribut de classe qui s'incré
          mente à chaque
3     fois que l'on crée un objet de ce type"""
4
5
6     objets_crees = 0 # Le compteur vaut 0 au départ
7     def __init__(self):
8         """À chaque fois qu'on crée un objet, on incrémente le
              compteur"""
9         Compteur.objets_crees += 1
```

```
>>> Compteur.objets_crees
0
>>> a = Compteur() # On crée un premier objet
>>> Compteur.objets_crees
1
>>> b = Compteur()
>>> Compteur.objets_crees
2
>>>
```

- À chaque fois qu'on crée un objet de type Compteur, l'attribut de classe *objets_crees* s'incrémente de 1. Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques.

Les méthodes

- Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions agissant sur l'objet.
- Pour créer nos premières méthodes, nous allons modéliser. . . un tableau noir.
- Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classe ***TableauNoir*** et notre attribut ***surface***, vous ne devriez pas avoir de problème :

```
1  class TableauNoir:
2      """Classe définissant une surface sur laquelle on peut é
           crire,
3      que l'on peut lire et effacer, par jeu de méthodes. L'
           attribut modifié
4      est 'surface'"""
5
6
7      def __init__(self):
8          """Par défaut, notre surface est vide"""
9          self.surface = ""
```

- Nous allons donc écrire notre méthode ***ecrire*** pour commencer.

```
10      def ecrire(self, message_a_ecrire):
11          """Méthode permettant d'écrire sur la surface du
                tableau.
12          Si la surface n'est pas vide, on saute une ligne avant
                de rajouter
13          le message à écrire"""
14
15
16          if self.surface != "":
17              self.surface += "\n"
18          self.surface += message_a_ecrire
```

```
>>> tab = TableauNoir()
>>> tab.surface
''
>>> tab.ecrire("Coooool ! Ce sont les vacances !")
>>> tab.surface
"Coooool ! Ce sont les vacances !"
>>> tab.ecrire("Joyeux Noël !")
>>> tab.surface
"Coooool ! Ce sont les vacances !\nJoyeux Noël !"
>>> print(tab.surface)
Coooool ! Ce sont les vacances !
Joyeux Noël !
>>>
```

Notre méthode ***ecrire*** se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.
On retrouve ici notre paramètre **self**. Il est temps de voir un peu plus en détail à quoi il sert.

Le paramètre self

- Une chose qui a son importance : quand vous créez un nouvel objet, ici un ***tableau noir***, les attributs de l'objet sont propres à l'objet créé. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.
- En revanche, les méthodes sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous tapez ***tab.ecrire(...)***, Python va chercher la méthode ***ecrire*** non pas dans l'objet ***tab***, mais dans la classe ***TableauNoir***.

```
>>> tab.ecrire
<bound method TableauNoir.ecrire of <__main__.TableauNoir
    object at 0x00B3F3F0>>
>>> TableauNoir.ecrire
<function ecrire at 0x00BA5810>
>>> help(TableauNoir.ecrire)
Help on function ecrire in module __main__:
ecrire(self, message_a_ecrire)
    Méthode permettant d'écrire sur la surface du tableau.
    Si la surface n'est pas vide, on saute une ligne avant de
        rajouter
    le message à écrire.
>>> TableauNoir.ecrire(tab, "essai")
>>> tab.surface
'essai'
>>>
```

- Comme vous le voyez, quand vous tapez ***tab.ecrire(...)***, cela revient au même que si vous écrivez ***TableauNoir.ecrire(tab, ...)***. Votre paramètre **self**, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelant **self.surface**.
- Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par self.
- Le nom **self** est une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, qui trouvent qu'écrire **self** à chaque fois est excessivement long, l'abrègent en une unique lettre **s**. Évitez ce raccourci. De manière générale, évitez de changer le nom. Une méthode d'instance travaille avec le paramètre **self**.

- N'est-ce pas effectivement plutôt long de devoir toujours travailler avec **self** à chaque fois qu'on souhaite faire appel à l'objet ?
- Cela peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait au langage Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Mais c'est moins clair et cela peut susciter la confusion. En Python, dès qu'on voit **self**, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

- Bon, voyons nos autres méthodes. Nous devons encore coder ***lire*** qui va se charger d'afficher notre surface et ***effacer*** qui va effacer le contenu de notre surface. Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

```
19      def lire(self):
20          """Cette méthode se charge d'afficher, grâce à print,
21          la surface du tableau"""
22
23
24          print(self.surface)
25      def effacer(self):
26          """Cette méthode permet d'effacer la surface du tableau
                """
27          self.surface = ""
```

```
>>> tab = TableauNoir()
>>> tab.lire()
>>> tab.ecrire("Salut tout le monde.")
>>> tab.ecrire("La forme ?")
>>> tab.lire()
Salut tout le monde.
La forme ?
>>> tab.effacer()
>>> tab.lire()
>>>
```

Et voilà ! Avec nos méthodes bien documentées, un petit coup de ***help(TableauNoir)*** et vous obtenez une belle description de l'utilité de votre classe. C'est très pratique, n'oubliez pas les ***docstrings***.

Méthodes de classe

- Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne travaillent pas sur l'instance **self** mais sur la classe même. C'est un peu plus rare mais cela peut être utile parfois. Notre méthode de classe se définit exactement comme une méthode d'instance, à la différence qu'elle ne prend pas en premier paramètre **self** (l'instance de l'objet) mais **cls** (la classe de l'objet).
- En outre, on utilise ensuite une fonction built-in de Python pour lui faire comprendre qu'il s'agit d'une méthode de classe, pas d'une méthode d'instance.

```
1  class Compteur:
2      """Cette classe possède un attribut de classe qui s'incré
           mente à chaque
3      fois que l'on crée un objet de ce type"""
4
5
6      objets_crees = 0 # Le compteur vaut 0 au départ
7      def __init__(self):
8          """À chaque fois qu'on crée un objet, on incrémente le
               compteur"""
9          Compteur.objets_crees += 1
10     def combien(cls):
11         """Méthode de classe affichant combien d'objets ont été
               créés"""
12         print("Jusqu'à présent, {} objets ont été créés.".
               format(
13                 cls.objets_crees))
14     combien = classmethod(combien)
```

```
>>> Compteur.combien()
Jusqu'à présent, 0 objets ont été créés.
>>> a = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 1 objets ont été créés.
>>> b = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 2 objets ont été créés.
>>>
```

- Une méthode de classe prend en premier paramètre non pas **self** mais **cls**. Ce paramètre contient la classe (ici Compteur).
- Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe. Vous auriez par exemple pu écrire **a.combien()**.
- Enfin, pour que Python reconnaisse une méthode de classe, il faut appeler la fonction **classmethod** qui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

Méthodes statiques

- On peut également définir des méthodes statiques. Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent aucun premier paramètre, ni **self** ni **cls**. Elles travaillent donc indépendamment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.
- Voici la syntaxe permettant de créer une méthode statique.

```
1 class Test:
2     """Une classe de test tout simplement"""
3     def afficher():
4         """Fonction chargée d'afficher quelque chose"""
5         print("On affiche la même chose.")
6         print("peu importe les données de l'objet ou de la
              classe.")
7     afficher = staticmethod(afficher)
```

Rappel : les noms de méthodes encadrés par deux soulignés de part et d'autre sont des méthodes spéciales. Ne nommez pas vos méthodes ainsi. Nous découvrirons plus tard ces méthodes particulières. Exemple de nom de méthode à éviter : **__mamethode__**.

Un peu d'introspection

- Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.
- Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans, non ?
- En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous en aurez besoin ! Pour l'heure donc, voyons plutôt l'effet :

La fonction dir

- La première technique d'introspection que nous allons voir est la fonction **dir**. Elle prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

```
1  class Test:
2      """Une classe de test tout simplement"""
3      def __init__(self):
4          """On définit dans le constructeur un unique attribut
               """
5          self.mon_attribut = "ok"
6
7      def afficher_attribut(self):
8          """Méthode affichant l'attribut 'mon_attribut'"""
9          print("Mon attribut est {0}.".format(self.mon_attribut)
               )
```

```
>>> # Créons un objet de la classe Test
... un_test = Test()
>>> un_test.afficher_attribut()
Mon attribut est ok.
>>> dir(un_test)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '
    __format__', '__g
e__', '__getattribute__', '__gt__', '__hash__', '__init__', '
    __le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__
    ', '__repr__', '_
_setattr__', '__sizeof__', '__str__', '__subclasshook__', '
    __weakref__', 'affich
er_attribut', 'mon_attribut']
>>>
```

- La fonction **dir** renvoie une liste comprenant le nom des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous pouvez remarquer que tout est mélangé, c'est normal : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (callable). La fonction **dir** se contente de renvoyer tout ce qu'il y a dans l'objet, sans distinction.
- C'est quoi tout cela ? On n'a jamais défini toutes ces méthodes ou attributs !
- Non, en effet. Nous verrons plus loin qu'il s'agit de méthodes spéciales utiles à Python.

L'attribut spécial __dict__

- Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial __dict__. Cet attribut est un dictionnaire qui contient en guise de clés les noms des attributs et, en tant que valeurs, les valeurs des attributs.
- Voyez plutôt :

```
>>> un_test = Test()
>>> un_test.__dict__
{'mon_attribut': 'ok'}
>>>
```

Pourquoi attribut spécial ?

- C'est un attribut un peu particulier car ce n'est pas vous qui le créez, c'est Python. Il est entouré de deux signes soulignés __ de part et d'autre, ce qui traduit qu'il a une signification pour Python et n'est pas un attribut standard. Vous verrez plus loin dans ce cours des méthodes spéciales qui reprennent la même syntaxe.

Peut-on modifier ce dictionnaire ?

- Vous le pouvez. Sachez qu'en modifiant la valeur de l'attribut, vous modifiez aussi l'attribut dans l'objet.

```
>>> un_test.__dict__["mon_attribut"] = "plus ok"
>>> un_test.afficher_attribut()
Mon attribut est plus ok.
>>>
```

- De manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre d'écrire **objet.attribut = valeur** que **objet.__dict__[nom_attribut] = valeur**.
- vous découvrirez dans la suite l'utilité des deux méthodes que je vous ai montrées.

En résumé

- On définit une classe en suivant la syntaxe **class NomClasse:**.
- Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.
- Les méthodes d'instance prennent en premier paramètre **self**, l'instance de l'objet manipulé.
- On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée **__init__**.
- On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe : **self.nom_attribut = valeur**.

Les propriétés

- Nous avons appris à créer nos premiers attributs et méthodes. Mais nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais tâcher de lever.
- Nous allons découvrir dans ce qui suit les propriétés, un concept propre à Python et à quelques autres langages, comme le Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation.

Qu'est-ce que l'encapsulation ?

- L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels que le C++, le Java ou le PHP, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, **mon_objet.mon_attribut**.
- Mais c'est stupide ! Comment fait-on pour accéder aux attributs ?

- On va définir des méthodes un peu particulières, appelées des **accesseurs** et **mutateurs**.
- Les **accesseurs** donnent accès à l'attribut. Les **mutateurs** permettent de le modifier.
- Pour accéder à l'attribut, au lieu d'écrire **mon_objet.mon_attribut**, vous allez écrire **mon_objet.get_mon_attribut()**.
- De la même manière, pour modifier l'attribut écrivez **mon_objet.set_mon_attribut(valeur)** et non pas **mon_objet.mon_attribut = valeur**.
- C'est bien tordu tout cela ! Pourquoi ne peut-on pas accéder aux attributs directement, comme on l'a déjà fait ?

- Mais d'abord, je n'ai pas dit que vous ne pouviez pas. Vous pouvez très bien accéder aux attributs d'un objet directement, comme on l'a déjà fait. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.
- Mais pour répondre à la question, il peut être très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un attribut de notre objet ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre attribut est modifié. Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

- L'inconvénient de devoir écrire des **accesseurs** et **mutateurs**, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe.
- D'abord, c'est assez lourd. Ensuite, nos méthodes se ressemblent plutôt. Certains **environnements** de développement proposent, il est vrai, de créer ces **accesseurs** et **mutateurs** pour nous, automatiquement. Mais cela ne résout pas vraiment le problème.
- Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait précédemment. On peut y accéder et les modifier en écrivant simplement **mon_objet.mon_attribut**. Et pour certains, on va créer des **propriétés**.

Les propriétés à la casserole

- Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est **privé** ou **public**. Pour schématiser, 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**.
- En Python, il **n'y a pas** d'attribut **privé**. **Tout** est **public**. Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez. Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus bas mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

- Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : Quand un utilisateur souhaite modifier cet attribut, fais cela . De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.
- 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 que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.
- Mais ces propriétés, c'est quoi ?

Les propriétés en action

- Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe. J'ai dit qu'il s'agissait d'une classe, son nom est **property**. Elle attend quatre paramètres, tous optionnels :
  - la méthode donnant accès à l'attribut ;
  - la méthode modifiant l'attribut ;
  - la méthode appelée quand on souhaite supprimer l'attribut ;
  - la méthode appelée quand on demande de l'aide sur l'attribut.
- En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit nos **accesseur** et **mutateur** d'objet.
- Mais j'imagine que ce n'est pas très clair dans votre esprit. Considérez le code suivant :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom ;
    - son prénom ;
    - son âge ;
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris" # Notez le souligné _
            devant le nom
```

```
15      def _get_lieu_residence(self):
16      """Méthode qui sera appelée quand on souhaitera accéder en
            lecture
17          à l'attribut 'lieu_residence'"""
18
19
20          print("On accède à l'attribut lieu_residence !")
21          return self._lieu_residence
22      def _set_lieu_residence(self, nouvelle_residence):
23          """Méthode appelée quand on souhaite modifier le lieu
                de résidence"""
24          print("Attention, il semble que {} déménage à {}.".
                format( \
25                      self.prenom, nouvelle_residence))
26          self._lieu_residence = nouvelle_residence
27      # On va dire à Python que notre attribut lieu_residence
            pointe vers une
28      # propriété
29      lieu_residence = property(_get_lieu_residence,
            _set_lieu_residence)
```

```
>>> jean = Personne("Micado", "Jean")
>>> jean.nom
'Micado'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Paris'
>>> jean.lieu_residence = "Berlin"
Attention, il semble que Jean déménage à Berlin.
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Berlin'
>>>
```

- Notre **accesseur** et notre **mutateur** se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attribut **lieu_residence**.
- Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.
- Il est aussi possible de définir, en troisième position du constructeur **property**, une méthode qui sera appelée quand on fera **del objet.lieu_residence** et, en quatrième position, une méthode qui sera appelée quand on fera **help(objet.lieu_residence)**.

En résumé

- Les propriétés permettent de contrôler l'accès à certains attributs d'une instance.
- Elles se définissent dans le corps de la classe en suivant cette syntaxe : **nom_propriete = proprety (methode_accesseur, methode_mutateur, methode_suppression, methode_aide)**.
- On y fait appel ensuite en écrivant **objet.nom_propriete** comme pour n'importe quel attribut.
- Si l'on souhaite juste **lire** l'attribut, c'est la méthode définie comme **accesseur** qui est appelée.
- Si l'on souhaite **modifier** l'attribut, c'est la méthode **mutateur**, si elle est définie, qui est appelée.
- Chacun des paramètres à passer à **property** est optionnel.