Atelier 4 – Introduction à l'Héritage

Introduction

Complètez ce diagramme UML qui correspond à l'énoncé suivant :

Personne	Etudiants	Professeur	Employé
imprimer_adr() est_voisin(qq1)	imprimer_adr() est_voisin(qq1)	imprimer_adr() est_voisin(qq1)	imprimer_adr() est_voisin(qq1)

L'Henallux doit stocker diverses informations au sujet des personnes liées à son établissement, notamment des étudiants, des professeurs et des employés. Pour chacune de ces *personnes*, on retiendra un nom, une adresse (chaine de caractère avec rue et numéro), un code postal et une localité.

Si la personne est un *étudiant*, on retiendra également une année d'études et éventuellement une adresse de kot. Si la personne est un *professeur*, on retiendra sa spécialité. Si la personne est un *employé*, on retiendra sa date d'engagement et sa fonction.

Le programme doit permettre d'imprimer une étiquette à coller sur une enveloppe en cas de courrier papier à envoyer (celle du Kot le cas échéant). De plus, pour inciter au covoiturage, le programme doit permettre d'indiquer (par un booléen) si une personne donnée a le même code postal qu'une autre personne.

Quelle(s) critique(s) pouvez-vous faire de cette solution ?

L'héritage

Vous avez sans doute remarqué qu'il y avait beaucoup de **répétitions** dans la solution proposée à l'exercice précédent. Cette solution impliquerait d'écrire 4 classes fort semblables avec beaucoup de code identique qui se répète.

Pour régler ce problème, nous allons utiliser l'héritage !

C'est quoi l'héritage ?

« En programmation orientée objet, l'héritage est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe. » (wikipédia)

« C'est une technique qui permet de créer une classe à partir d'une autre classe. Elle lui sert de modèle, de base de départ. Cela permet d'éviter d'avoir à réécrire un même code source plusieurs fois » (openclassrooms)

« Afin de réduire la complexité d'un programme, il est intéressant d'appliquer le principe de généralisation, en remplaçant plusieurs entités qui partagent des fonctions similaires par une construction unique. Autrement dit, il est intéressant de créer une classe qui encapsule les fonctionnalités partagées par plusieurs classes et de construire de nouvelles classes à partir de cette classe mère en ajoutant des fonctionnalités spécifiques. » (syllabus java IG1)

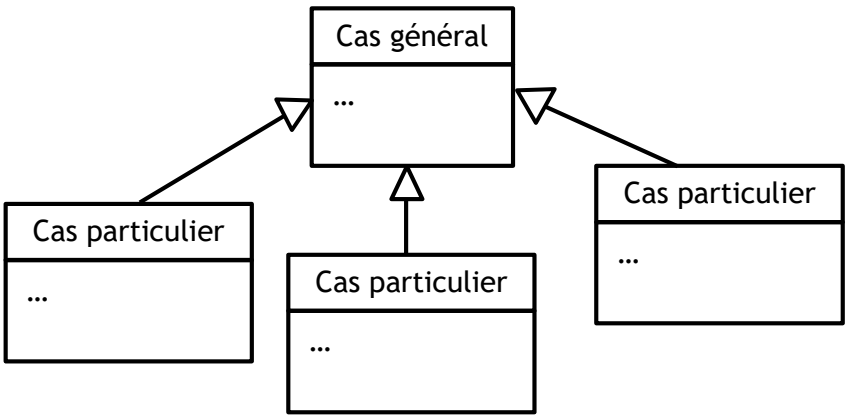
Regardez si, dans la solution, il n'est pas possible de généraliser certaines choses. Parmi les 4 classes, la quelle peut être identifiée comme un **cas général** des autres. En d'autres mots, les quelles sont des **cas particuliers** d'une autre ?

Personne	Étudiant	Professeur	Employé
nom adresse code_postal localité	nom adresse code_postal localité année adresse_kot [0..1]	nom adresse code_postal localité spécialisation	nom adresse code_postal localité date_embauche fonction
imprimer_adr() est_voisin(X)	imprimer_adr() est_voisin(X)	imprimer_adr() est_voisin(X)	imprimer_adr() est_voisin(X)

Aide : entourez dans chaque classe ce qui est en commun. Voyez quelle classe reprend tous les éléments. Il s'agit du cas général !

Représentation UML

Voici comment représenter l'héritage en UML :



La notation est une flèche à bout triangulaire (vide). **Les cas particuliers héritent automatiquement des attributs et des méthodes du cas général**. Il n'est donc pas nécessaire de les réécrire dans les cas particuliers.

Signification :

« Cas général » est la **superclasse / classe-mère / classe-parent** de « Cas particulier ».

« Cas particulier » est une **sous-classe / classe-enfant / classe-fille / classe dérivée** de « Cas général ».

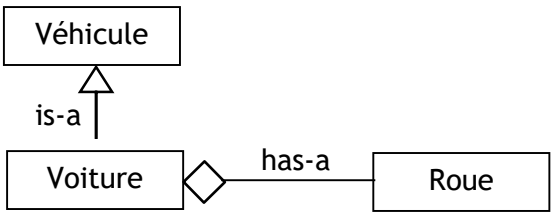
Les classes « Cas particulier » **héritent de** / **spécialisent / étendent** la classe « Cas général ».

➔ À partir de maintenant, nous utiliserons principalement les termes **superclasse** et **sous-classe.**

Corrigez votre schéma pour y introduire l'héritage.
Attention à ne pas recopier les informations héritées de la super classe.

Distinguer les « is-a » et les « has-a »

L'héritage correspond à un lien « **is-a** » (« est-un ») contrairement à l'agrégation (et donc la composition) qui est un lien « **has-a** » (« a-un »). Il ne faut pas confondre les deux !



Exercices

Comment reconnaître un héritage ?

Il est parfois compliqué de reconnaitre l'héritage. Il ne faut pas non plus en voir partout.

Pour savoir s'il y a de l'héritage entre deux classes, suivez cette règle très simple :
***A hérite de B** si on peut dire : « **A est un B** ».*

Voici quelques exemples d'héritage :

- une voiture *est un* véhicule (Voiture *hérite* de Vehicule) ;
- un bus *est un* véhicule (Bus *hérite* de Vehicule) ;
- un moineau *est un* oiseau (Moineau *hérite* d'Oiseau) ;
- un corbeau *est un* oiseau (Corbeau *hérite* d'Oiseau) ;
- un chirurgien *est un* docteur (Chirurgien *hérite* de Docteur) ;
- un diplodocus *est un* dinosaure (Diplodocus *hérite* de Dinosaure) ;

Par contre, vous ne pouvez pas dire « Un dinosaure est un diplodocus », ou encore « Un bus est un oiseau ». Donc, dans ces cas-là, pas d'héritage … ça n'aurait aucun sens.

Si vous respectez cette règle, vous ne devriez pas rencontrer de soucis ;)

Faites les diagrammes UML des énoncés suivants**. Pensez à l'héritage.**

a. Pour chaque bâtiment, on connait son adresse, le nom du propriétaire et sa superficie. Lorsqu'il s'agit de bureaux, on retient également le nom de la société. Dans le cas d'immeuble à appartements, on veut connaitre le nombre d'étages et savoir combien d'appartements il existe dans l'immeuble en question.

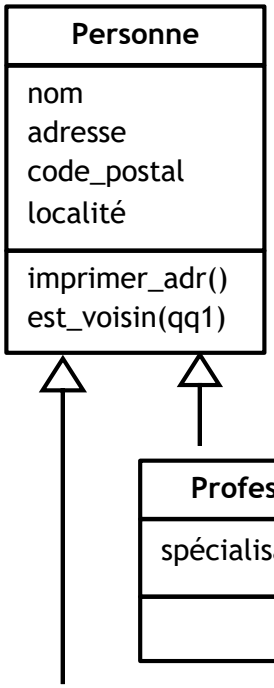
b. Un chien a un nom, une date de naissance, un numéro de puce, une taille et est un animal de concours ou non mais il est aussi capable d'aboyer, de dormir, de manger. Un chat, quant à lui, possède les mêmes attributs et peut miauler, dormir, manger et ronronner. Un lapin possède également un nom, une date de naissance, un numéro de puce, une taille et peut être un animal de concours mais on souhaite aussi connaitre la taille de ses oreilles. Il peut dormir, manger et faire des bonds.[1]

c. Dans le cadre d'un jeu de rôle, chaque personnage a un nom, de la force, de la vie, et de l'armure. Tous les personnages peuvent attaquer quelqu'un, prendre des dégâts ou se faire soigner. Il y plusieurs types de personnages : les mages, les guerriers, les chasseurs et les animaux. Un mage est un personnage qui possède, en plus, du mana. Il peut soigner et faire une attaque magique. Un Guerrier possède quant à lui de la rage et peut faire une attaque enragée. Il redéfinira la méthode qui prend des dégâts car il ne les prend pas de la même manière que les autres. Les chasseurs ont un animal (de compagnie) et leur attaque est différente de celle des autres personnages. Les animaux sont des personnages avec une armure qui est toujours à 0.

[1] Il arrivera que la superclasse ne soit pas décrite dans l'énoncé. Mais si plusieurs classes ont visiblement quelque chose en commun, il peut être intéressant de faire une superclasse dont elles hériteront toutes. Tout en respectant la règle « A est un B »

Notions avancée

Héritage des attributs et méthodes, substitution, redéfinition



La sous-classe **hérite** des attributs et des méthodes de la superclasse.

→ Chaque instance de Professeur a un nom, une adresse, un codePostal et une localité.

→ Chaque instance de Professeur a une méthode imprime_adr() et une méthode est_voisin(qq1).

Principe de **substitution** :

On peut utiliser un objet de la sous-classe partout où on attend un objet de la superclasse.

Une classe peut évidemment ajouter des attributs et des méthodes à ce dont elle hérite.

Redéfinition :

Une classe peut aussi redéfinir/écraser les méthodes dont elle hérite.

La classe Professeur utilise le même code que Personne pour imprimer une étiquette d'adresse.

Par contre, la classe Étudiant redéfinit la méthode imprime_adr() pour tenir compte du kot. C'est cette version spécialisée qui sera utilisée si on appelle imprime_adr() pour une instance de la classe Étudiant.

Petit résumé :

➔ La sous-classe **hérite** des attributs et méthodes de ses ancêtres.

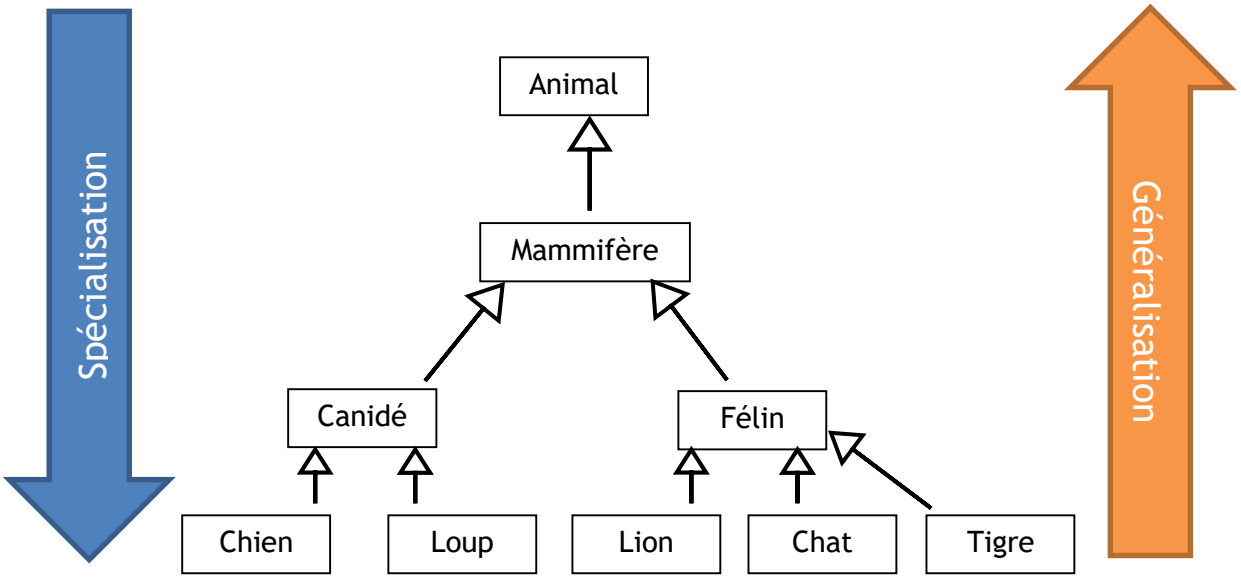
➔ Elle peut en **faire plus** (ajouter de nouvelles méthodes / de nouveaux attributs) ;

➔ Elle peut **faire différemment** (redéfinir des méthodes) ;

➔ Mais elle ne peut **pas en faire moins** !
Car il faut respecter le principe de substitution.
On peut utiliser une instance de la sous-classe partout où on attend une instance de la superclasse, donc les instances de la sous-classe doivent être capables de faire tout ce que les instances de la superclasse savent faire.

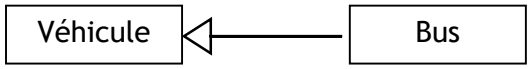
Arborescence

Il est également possible d'enchainer l'héritage ce qui crée une **arborescence**.
Par exemple :



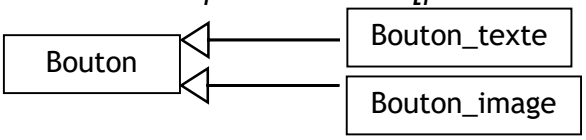
Il y a deux visions de l'arborescence :

Spécialisation (du haut vers le bas)
→ *Réutiliser ce qu'on a déjà fait, affiner pour des cas particulier.*



On réutilise les méthodes (rouler, faire le plein…) de Véhicule pour Bus…

Généralisation (du bas vers le haut)
→ *Centraliser ce qui est commun [point de modification unique].*

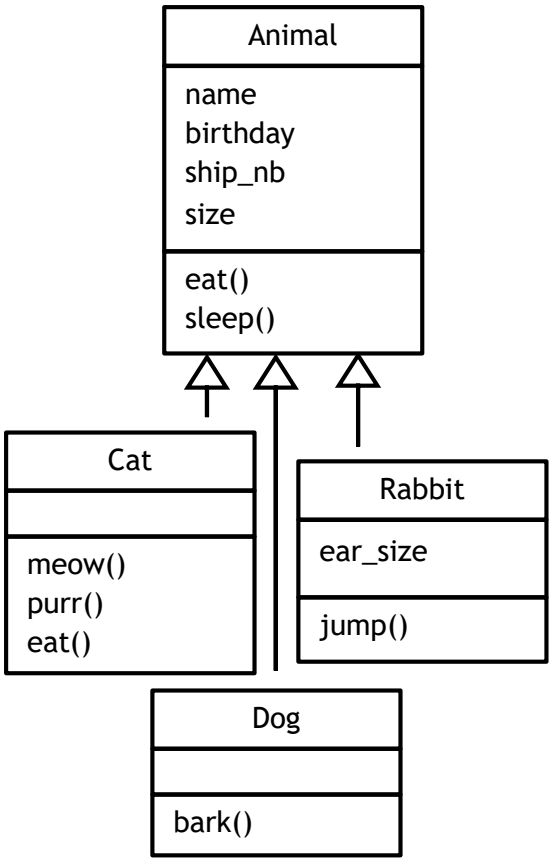


Je centralise le code commun (réagir à un clic…) à tous les types de boutons…

Et l'encapsulation dans tout ça ?

Dans une sous-classe, l'accès aux caractéristiques **privées** de la classe mère ne peut se faire qu'avec un sélecteur !

L'héritage en Python



La superclasse, s'écrit comme une classe normale :

```
class Animal :

    def __init__(self, name, bd, ship_nb, size):
        self.name = name
        self.birthday = bd
        self.ship_nb = ship_nb
        self.size = size

    def sleep(self):
        print(self.name, ": ZzzzzzZzzzzz...")

    def eat(self):
        print(self.name, ": *Crounch*")
```

Pour cet exemple, nous avons volontairement omis l'encapsulation pour ne pas surcharger mais les principes restent les mêmes

Rabbit (Lapin)

Pour les sous-classes, il faut préciser la super classe entre parenthèses.
Voici l'exemple pour un lapin :

```
class Rabbit (Animal):

    def __init__(self, name, bd, ship_nb, size, ear_size):
        self.name = name
        self.birthday = bd
        self.ship_nb = ship_nb
        self.size = size
        self.ear_size = ear_size

    def jump(self):
        print(self.name, ": *Bwing*")
```

Notez qu'un des buts de l'héritage est de ne pas réécrire du code déjà écrit.

C'est le cas des méthodes `eat` et `sleep` que `Rabbit` hérite de `Animal`. Il n'est pas nécessaire de les réécrire dans `Rabbit`. La méthode `sleep` ou `eat` pourra cependant être appelée sur un lapin.

Recopier ces deux classes dans un fichier suivit du bloc main suivant et exécuter votre script. Vous verrez que Pong, votre lapin de 10 ans, peut dormir.

```
if __name__ == "__main__" :
    r = Rabbit("Pong", "11/02/2010", 25874136, 20,15)
    r.sleep()
```

Une minute !

Dans le diagramme UML, toujours dans l'optique de ne pas recopier du code, les attributs des sous-classes n'étaient **pas réécrits** dans celles-ci puisqu'ils sont automatiquement hérités de la superclasse.

Vous remarquerez cependant qu'en python tous les attributs ont été réécrits dans le constructeur…

➔ Il est possible de ne pas réécrire tous les attributs en utilisant le constructeur de la superclasse !

Comment ? **En l'appelant explicitement.**

Je m'explique. De base, si vous faites, par exemple, `self.sleep()` dans la classe `Rabbit`, Python va voir s'il existe la méthode `sleep` dans la classe courante (à savoir, `Rabbit` dans notre cas). Si elle n'existe pas, il va voir de lui-même (et donc **implicitement**) dans la superclasse de `Rabbit` (à savoir, `Animal` dans notre cas). Python remontera ainsi de superclasse en superclasse jusqu'à trouver la méthode `sleep`.

Mais il y a des cas où vous voudriez appeler **explicitement** une méthode de la superclasse sans que Python ne regarde si cette méthode n'existe dans la classe courante. Pour cela, utilisez la méthode `super()`. Celle-ci désigne la superclasse directe de la classe courante.

Pour éviter la duplication inutile de code, modifiez le constructeur de `Rabbit` pour utiliser celui de `Animal` pour les attributs hérités. Comme ceci :

```
class Rabbit (Animal):

    def __init__(self, name, bd, ship_nb, size, ear_size):
        super().__init__(name, bd, ship_nb, size)
        self.ear_size = ear_size
    …
```

→ De cette manière, le constructeur de `Animal` est appelé **explicitement** avec les paramètres correspondant.

Dog (Chien)

Voici **une** implémentation de la classe `Dog` :

```
class Dog (Animal) :

    def __init__(self, name, bd, ship_nb, size):
        super().__init__(name, bd, ship_nb, size)

    def bark(self):
            print(self.name, ": WOOOUF !")
```

> Si le constructeur de votre classe ne contient rien de plus que le constructeur de la superclasse, vous n'êtes pas obligé de le mettre.
>
> En effet, lorsque vous allez construire un objet de type `Dog`, Python va chercher dans la classe `Dog` le constructeur. S'il ne le trouve pas, que va-t-il faire ?
>
> *→ Le chercher dans la superclasse*

Implémentez la classe `Dog` avec simplement une méthode `bark` (aboyer)… et donc sans constructeur.

Cat (Chat)

Implémentez la classe `Cat`. `Cat` hérite de `Animal` et possède une méthode `meow` (miauler) et une méthode `purr` (ronronner).

Un chat va aussi manger mais de manière différente des autres animaux. **Redéfinissez** la méthode `eat` pour un chat. Pour manger, le chat ronronne d'abord puis seulement mange (comme les autres).

Test

Dans le bloc main, définissez un lapin, un chat et un chien. Par exemple :

```
if __name__ == "__main__" :
    c = Cat("Tesla", "1/04/2017", 12568743, 35)
    d = Dog("Pixel", "16/10/2019", 58963217, 70)
    r = Rabbit("Pong", "11/02/2010", 25874136, 20,15)
```

Affichez ensuite leurs informations respectives. Par exemple avec `__dict__` pour voir leur état.

Faites dormir et manger les trois. Tesla miaule-t-elle bien avant de faire *Crounch* quand elle mange ?

Faites aboyer Pixel, miauler Tesla et sauter Pong.

LA superclasse object

En Python (comme la plupart des langages OO), toutes les classes héritent automatiquement de **LA superclasse `object`.** Cette superclasse ne possède **aucun attribut** mais possède **toutes les méthodes spéciales** de Python.

Dans un terminal Python, taper une à une les commandes suivantes et voyez ce qu'il se passe :

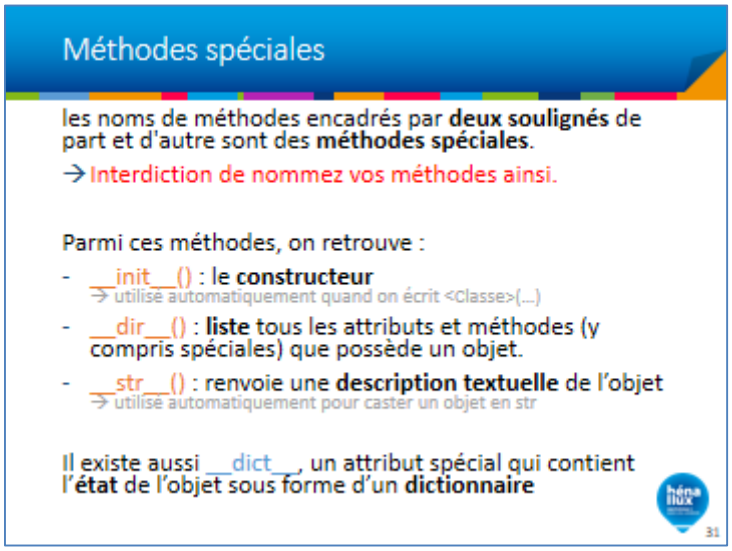
```
o = object()
o
o.__dir__()
o.__dict__
```

Plusieurs petites observations :

- La superclasse `object` est la seule classe qui commence par une lettre minuscule.
- La méthode `.__dir__()` sur un objet liste les méthodes et attributs que l'objet possède.
- La classe `object` n'a aucun attribut ! Même pas `__dict__` - qui ne figure d'ailleurs par dans la liste générée par `__dir__()`.

Tout s'explique

Maintenant que vous êtes un peu plus familiarisés avec l'héritage en Python, certains phénomènes vus précédemment peuvent enfin s'expliquer.



Souvenez-vous des **méthodes spéciales**. Ce sont des méthodes que Python générait automatiquement et que vous pouviez éventuellement réécrire si vous le désiriez.

Il n'y a, en fait, rien de magique à cela !

Ce sont simplement des méthodes définies dans la superclasse `object`. Et comme toutes les classes héritent automatiquement de `object`, elles héritent également de ses méthodes. Et, de même que pour n'importe quelle méthode héritée, vous pouvez les redéfinir.