Héritage et polymorphisme

Développeur Python

SUPINFO

Sommaire

1. Le concept d’héritage.
2. L’héritage en Python.
3. Héritage versus composition.
4. Polymorphisme.

1. Le concept d’héritage.

1. Le concept d’héritage.

Héritage : définition

- Relation de spécialisation/généralisation entre deux classes.

- Elle indique qu’une classe dite classe fille spécialise une autre classe dite classe mère, *i.e.* qu’elle possède les attributs et les méthodes de la classe mère plus d’autres qui lui sont propres.

1. Le concept d’héritage.

Héritage : exemple

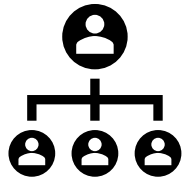
- Une classe “étudiant SUPINFO“ qui spécialise une classe “personne“ :

1. Le concept d’héritage.

Héritage : deux intérêts majeurs

1. Construction d’une hiérarchie de classes : on évite ainsi des répétitions dans le code, en encourageant la réutilisation de classes déjà existantes.

2. Cela simplifie également la conception de la modélisation.

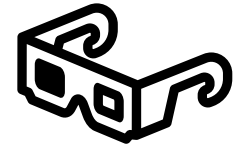


1. Le concept d’héritage.

Héritage : deux visions pour un même concept

1. Une vision ascendante en procédant par généralisation : on décèle des attributs et des méthodes communs à des classes différentes, l’héritage permet alors de les factoriser afin de faciliter la conception et la maintenance du code.

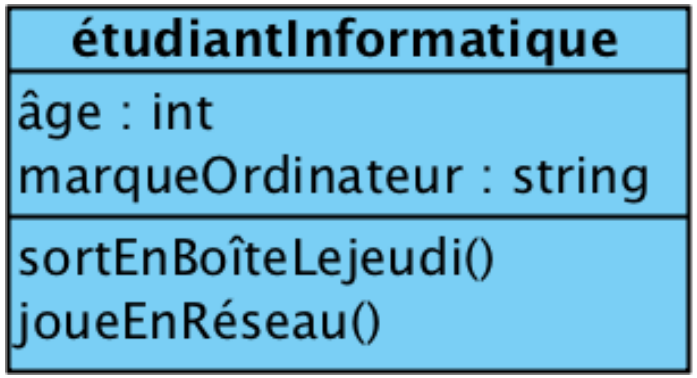
2. Une vision descendante, en procédant par spécialisation : on crée des classes spécialisées à partir d’une classe de base. Le niveau de spécialisation dépend du niveau d’abstraction que l’on souhaite. On procède souvent ainsi quand on veut réutiliser des classes déjà existantes.

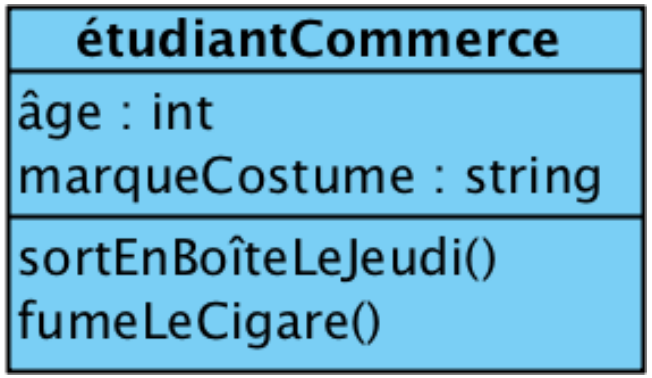


1. Le concept d'héritage.

Vision ascendante de l'héritage : exemple

- On considère les classes "étudiant en informatique" et "étudiant en commerce" :

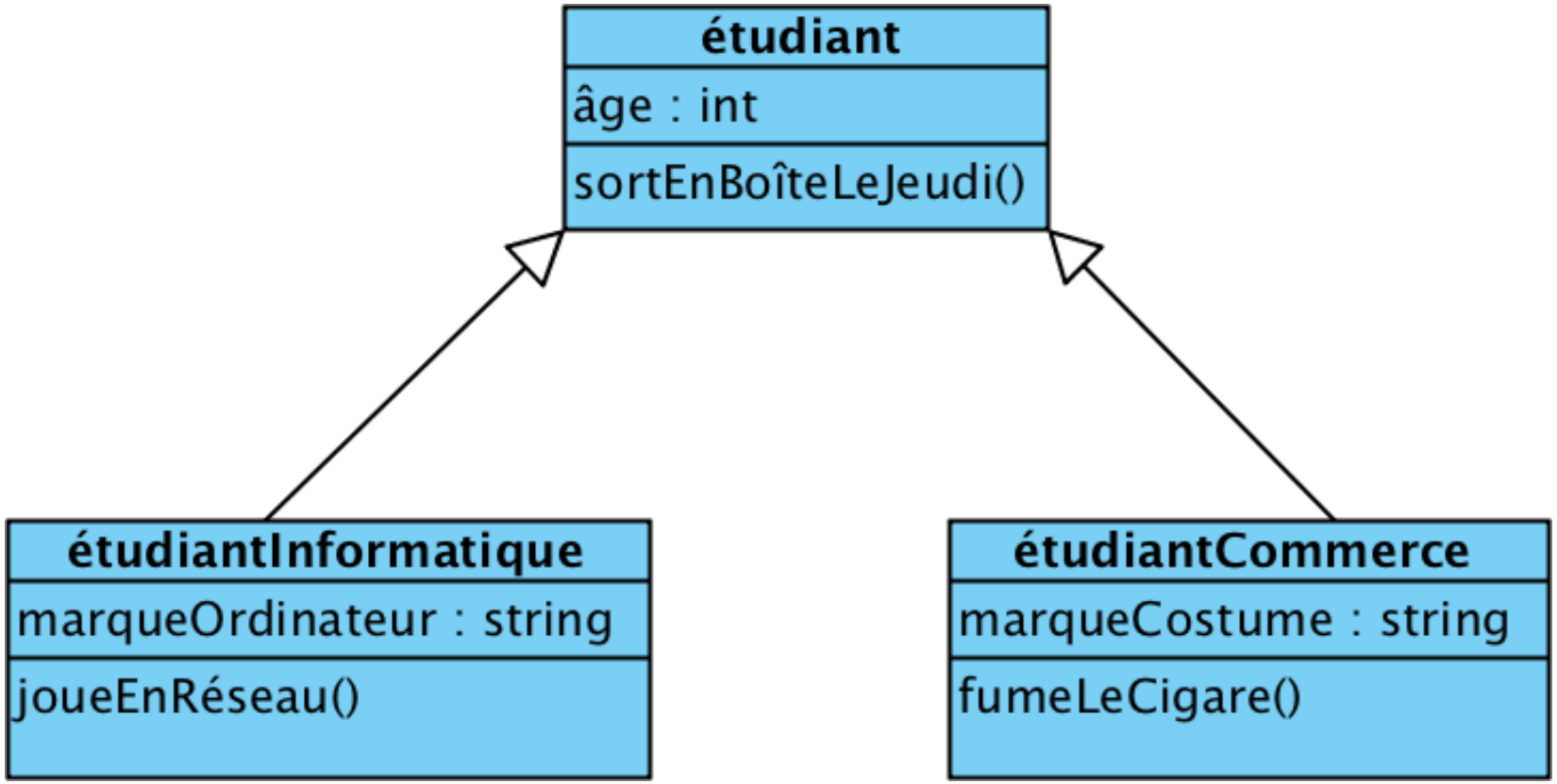




- On décèle un attribut et une méthode en commun dans ces deux classes, on les factorise donc dans une classe "étudiant".

1. Le concept d’héritage.

Vision ascendante de l’héritage : exemple (suite)

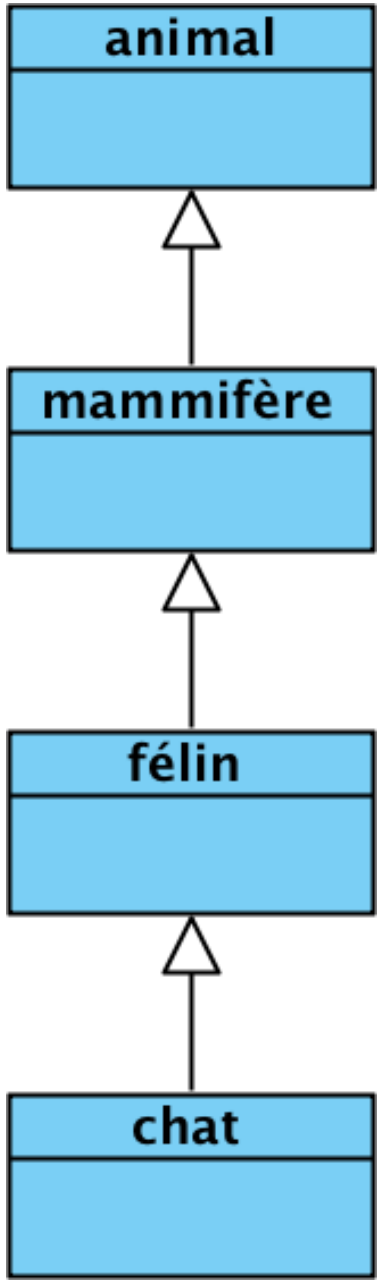


1. Le concept d’héritage.

Vision descendante de l’héritage : exemple

- A partir de la classe "animal" on se spécialise de plus en plus jusqu’à la classe “chat“ :

  1. animal
  2. mammifère
  3. félin
  4. chat



1. Le concept d’héritage.

Transmission des attributs dans une relation d’héritage

- La déclaration privée d’un attribut dans une classe mère interdit son accès direct par les classes filles.

- Deux solutions :

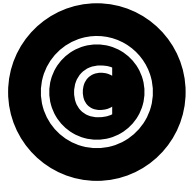
  - Utiliser des getters et setters dans les classes filles.

  - Déclarer de façon protégée les attributs dans la classe mère.

1. Le concept d’héritage.

Visibilité protégée

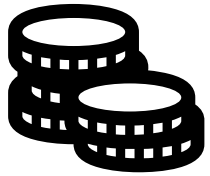
- Un attribut ou une méthode sont dits protégés si leur utilisation est limitée à la classe et ses descendantes.



1. Le concept d’héritage.

Héritage multiple : principe

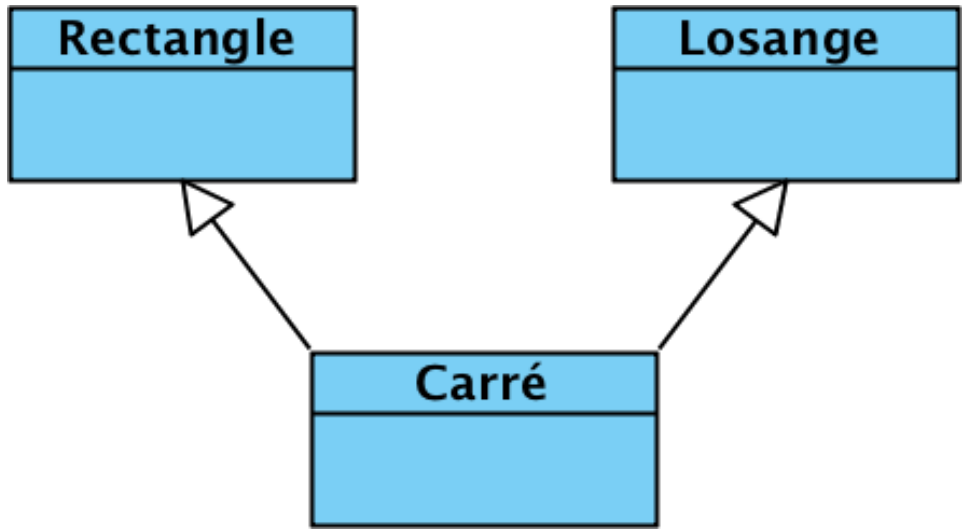
- Possibilité pour une classe de posséder plusieurs classes mères.
- Comme pour un héritage simple, la classe fille possède les attributs et les méthodes de ses classes mères plus d’autres qui lui sont propres.
- À noter que des difficultés peuvent apparaître quand les classes mères possèdent des méthodes de même nom.



1. Le concept d’héritage.

Héritage multiple : exemple

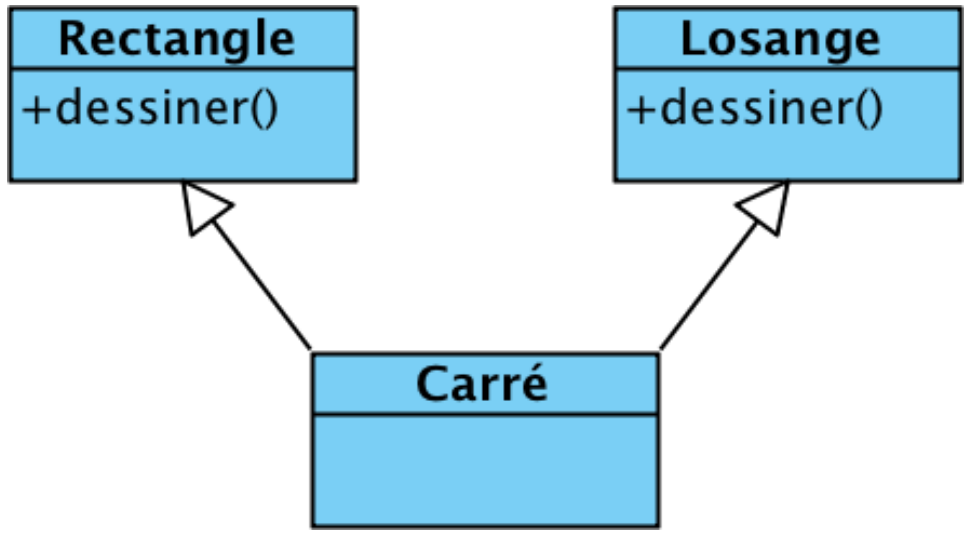
- Un carré est à la fois un rectangle et un losange :



1. Le concept d’héritage.

Héritage multiple : remarque

- Risque d’ambiguïté quand les classes mères possèdent des méthodes de même nom :



1. Le concept d’héritage.

2. L’héritage en Python.

2. L’héritage en Python.

Principe général

- La déclaration de la classe fille comporte une référence explicite à la classe mère :

```
class childClass(parentClass):
    ...
```

- Le constructeur de la classe fille doit faire un appel explicite au(x) constructeur(s) de(s) la classe(s) mère(s) afin d’initialiser les attributs hérités de celle(s)-ci.

2. L’héritage en Python.

Héritage simple : première syntaxe

- Utilisation du mot clé “super“ :

```
class parentClass:
    def __init__(self, para1, para2, ...):
        ...

class childClass(parentClass):
    def __init__(self, para1, para2, ...):
        super().__init__(...)
        ...
```

2. L’héritage en Python.

Héritage simple : seconde syntaxe

- Utilisation du nom de la classe mère :

```
class parentClass:
    def __init__(self, para1, para2, ...):
        ...

class childClass(parentClass):
    def __init__(self, para1, para2, ...):
        parentClass.__init__(...)
        ...
```

2. L’héritage en Python.

Héritage multiple : syntaxe

```
class parentClass1:
    def __init__(self, para1, para2, ...):
        ...

class parentClass2:
    def __init__(self, para1, para2, ...):
        ...

class childClass(parentClass1, parentClass2, …):
    def __init__(self, para1, para2, ...):
        parentClass1.__init__(self, ...)
        parentClass2.__init__(self, ...)
        ...
```

2. L’héritage en Python.

Héritage multiple : remarque

- Si sur un objet de la classe fille on appelle une méthode qui n’appartient pas à cette classe, la recherche s’effectuera de gauche à droite dans la liste des classes mères.

- Cela signifiera que l’on regardera d’abord si la classe “parentClass1“ possède la méthode en question :

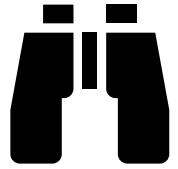
    - Si oui, on applique cette méthode et la recherche est terminée.
    - Si non, on considère la classe “parentClass2“. Etc.

2. L’héritage en Python.

Héritage multiple : remarque (suite)

- Si la recherche est infructueuse dans les classes “parentClass1”, “parentClass2”, etc., on la continue dans les classes parentes de celles-ci.

- On procède alors de la même façon que précédemment, en parcourant la liste de la gauche vers la droite.



2. L’héritage en Python.

Visibilité protégée

- Pour déclarer un attribut protégé on fait précéder son nom d’un simple underscore.

```
def __init__(self, para1, para2, ...):
    self._myAttribute1 = para1
    self._myAttribute2 = para2
    ...
```

2. L’héritage en Python.

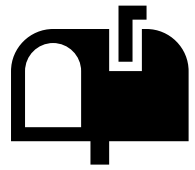
3. Héritage versus composition.

3. Héritage versus composition.

Relation de composition : définition

- Elle modélise une relation d’inclusion entre les instances de deux classes.

- Les objets de la classe conteneur possèdent donc un attribut qui est un objet de la classe contenue.



3. Héritage versus composition.

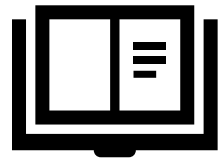
Relation de composition : utilisation

- On peut utiliser une relation de composition entre classes si on peut établir un lien sémantique du type “possède” ou “a un”.

- Exemples :

    - Une voiture a une plaque d’immatriculation.

    - Un livre possède des pages.



3. Héritage versus composition.

Relation de composition : une alternative à l’héritage

- Dans certains cas il est tout à fait possible techniquement d’utiliser une relation de composition à la place d’une relation d’héritage.

- Au lieu d’avoir une classe “B“ qui hérite d’une classe “A“, on déclare dans “B“ un attribut qui sera une instance de la classe “A“.

3. Héritage versus composition.

Comment choisir entre héritage et composition ?

- On choisit l'héritage quand :
    - La relation entre classes est bien de la forme "est un", ou pour les anglicistes "is a".
- On choisit la composition quand :
    - La relation entre classes est bien de la forme "a un", ou pour les anglicistes "has a".

3. Héritage versus composition.

Comment choisir entre héritage et composition ?

- On peut aussi préférer une relation de composition afin de respecter *stricto sensu* le principe d'encapsulation.

- La relation de composition est également plus simple à maintenir en cas de modifications du code.

3. Héritage versus composition.

4. Polymorphisme.

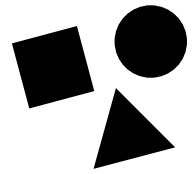
4. Polymorphisme.

Polymorphisme : définition

- Littéralement c’est la faculté de prendre plusieurs formes.

- On distinguera trois cas de figure :

    1. Redéfinition de méthodes.
    2. Surcharge des opérateurs.
    3. Duck Typing.



4. Polymorphisme.

Redéfinition de méthodes : principe

- Mécanisme qui permet à une classe fille de redéfinir une méthode dont elle a hérité de sa classe mère.

- Cela permet d'adapter le traitement qu'effectue la méthode aux spécificités de la classe fille.

4. Polymorphisme.

Redéfinition de méthodes : remarques

- En Python, il n'y a aucune difficulté à mettre en œuvre ce type de polymorphisme, il suffit d'implémenter la méthode en question dans la classe mère et la classe fille.

- Le choix de la "bonne" méthode à utiliser se fera alors automatiquement selon la nature de l'objet.

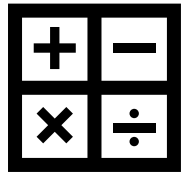
- Il n'en sera pas de même en C++, où ce polymorphisme n'est pas natif.

4. Polymorphisme.

Surcharge des opérateurs : principe

- Adaptation possible des opérateurs usuels natifs du langage aux classes implémentées par le développeur.

- Cela permet d'avoir des codes génériques et lisibles, puisque l'on pourra utiliser les symboles ($+, -, <, =$ etc.) habituels avec les objets que l'on instancie.



4. Polymorphisme.

Surcharge des opérateurs : mise en place

- On va considérer les opérateurs que l'on souhaite surcharger comme des méthodes de la classe.

- Le nom de la méthode correspondante à un opérateur sera de la forme

```
__Nom Opérateur__
```

4. Polymorphisme.

Surcharge des opérateurs : mise en place

- Si l'opérateur en question est unaire, la méthode n'aura que l'objet courant en paramètre :

```
def __Nom Opérateur__(self):
    ...
```

- Si l'opérateur en question est binaire, la méthode aura l'objet courant en paramètre ainsi qu'un autre objet :

```
def __Nom Opérateur__(self, other):
    ...
```

4. Polymorphisme.

Surcharge des opérateurs : nom des opérateurs arithmétiques

Opération	*Nom de l'opérateur associé*
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__div__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

4. Polymorphisme.

Surcharge des opérateurs : nom des opérateurs arithmétiques (suite)

Opération	*Nom de l'opérateur associé*
+=	__iadd__(self, other)
-=	__isub__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)

4. Polymorphisme.

Surcharge des opérateurs : nom de quelques opérateurs unaires

Opération	*Nom de l'opérateur associé*
-	__neg__(self)
abs()	__abs__(self)
bin	__bin__(self)
oct	__oct__(self)
hex	__hex__(self)

4. Polymorphisme.

Surcharge des opérateurs : nom des opérateurs logiques

Opération	*Nom de l'opérateur associé*
<	__lt__(self, other)
<=	__le__(self, other)
>	__gt__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

4. Polymorphisme.

Surcharge des opérateurs : le cas particulier de l'affichage

- Le but ici est de pouvoir afficher des informations relatives à un objet en utilisant la commande "print".

- Pour cela on doit surcharger dans notre classe un opérateur unaire nommé "__str__".

```
def __str__(self):
    ...
```

- À noter que cette méthode devra nécessairement retourner une chaîne de caractères.

4. Polymorphisme.

Duck Typing : principe

- En Python on ne précise pas les types attendus des paramètres des fonctions (*resp*. méthodes).

- Cela implique que l'on peut utiliser une fonction (*resp*. méthode) avec des paramètres de n'importe quel type, à la condition que les opérations de la fonction (*resp*. méthode) soient compatibles avec les types des paramètres.

4. Polymorphisme.

Duck Typing : moralité

- En Python, on s'intéresse donc plus au comportement d'un objet qu'à son type réel.

- Si les méthodes d'un objet sont compatibles avec les opérations requises par une fonction, alors la fonction en question peut s'appliquer sur cet objet.

4. Polymorphisme.