



# - Module 2 - Objets et classes en Python

*1re Technologie de l'informatique*  
*1re Sécurité des systèmes*



# 1.1. Rappels Python

# LES LANGAGES DE L'INFORMATIQUE

## LANGAGES DE PROGRAMMATION

# Python

Python est un langage à typage

- Implicite
- Dynamique
- Fort

Séparation d'instructions → passage à la ligne

Différenciation des blocs d'instruction → indentation

# Python : syntaxe de base

Affectation → `var = val`

## Opérateurs

- Arithmétiques → `+ - * / // % **`
- Booléens → `and or not is`  
`== != < <= > >=`

Affichage (écran) → `print("Hello !")`

Lecture (clavier) → `var = input("entrez qqch :")`

# Python : syntaxe de base

## Conditionnelle

```
if <cond> :  
    <action1>  
elif <cond> :  
    <action2>  
else :  
    <action3>
```

## Boucle

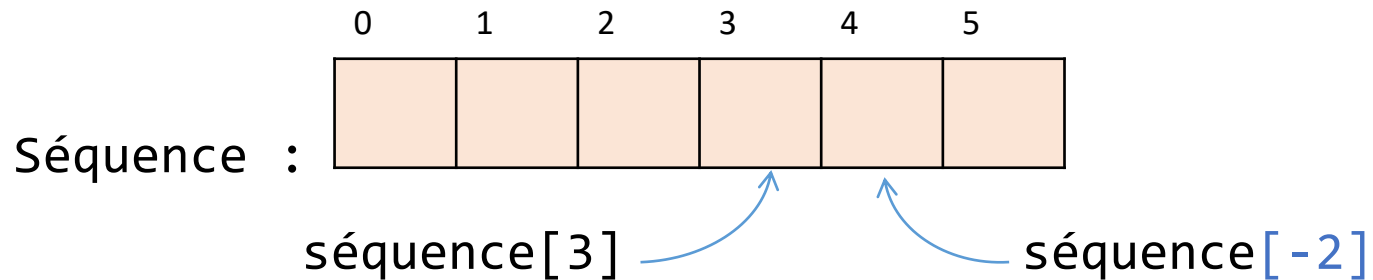
```
While <cond> :  
    <action>
```

# Python : muable et immuable

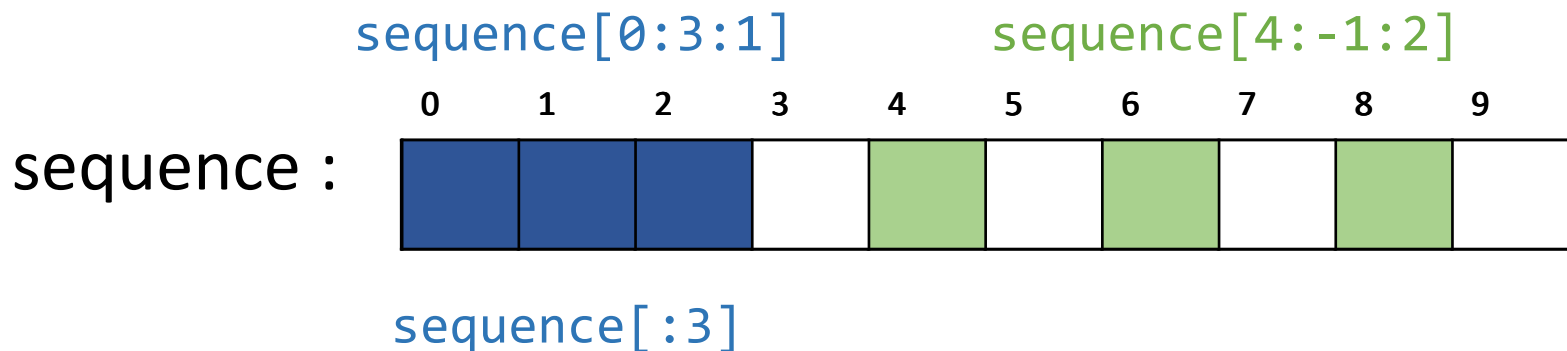
- Types **immuables** : leur valeur **ne peut pas** changer au cours du script ; leur modification renvoie une **nouvelle référence** associée au même nom de variable
  - Les types “primitifs” (int, float, bool...) sont immuables
  - Certaines sequences (tuple, str, range...)
- Types **muables** : leur valeur **peut** changer au cours du script
  - Les listes et les dictionnaires
  - Les **objets**

# Python : séquences

Séquence = **collection** d'éléments **ordonnés**



**Slicing** = récupérer un morceau de séquence





# Python : parcours de séquences

- Boucle WHILE

```
i=0
while i < len(séquence) :
    print (séquence[i])
    i+=1
```

- Boucle FOR

```
for élément in séquence :
    print(élément)
```

# Python : opérateurs de séquences

Opération	Résultat
<code>x in s</code>	True si un élément de <code>s</code> est égal à <code>x</code> , sinon False
<code>x not in s</code>	False si un élément de <code>s</code> est égal à <code>x</code> , sinon True
<code>s + t</code>	la concaténation de <code>s</code> et <code>t</code>
<code>s * n</code> OU <code>n * s</code>	équivalent à ajouter <code>s</code> <code>n</code> fois à lui même
<code>s[i]</code>	$i^{\text{e}}$ élément de <code>s</code> en commençant par 0
<code>s[i:j]</code>	tranche ( <i>slice</i> ) de <code>s</code> de <code>i</code> à <code>j</code>
<code>s[i:j:k]</code>	tranche ( <i>slice</i> ) de <code>s</code> de <code>i</code> à <code>j</code> avec un pas de <code>k</code>
<code>len(s)</code>	longueur de <code>s</code>
<code>min(s)</code>	plus petit élément de <code>s</code>
<code>max(s)</code>	plus grand élément de <code>s</code>
<code>s.index(x[, i[, j]])</code>	indice de la première occurrence de <code>x</code> dans <code>s</code> (à ou après l'indice <code>i</code> et avant indice <code>j</code> )
<code>s.count(x)</code>	nombre total d'occurrences de <code>x</code> dans <code>s</code>

# Python : types de séquences

- Listes : `liste = [var1, var2, var3...]`
  - Muable
  - Hétérogène

---
- Tuples : `tuple = (var1, var2, var3...)`
  - Immuable
  - Hétérogène
  - Taille relativement réduite

---
- Chaines de caractère : `str = "char1char2char3..."`  
ou `str = 'char1char2char3...'`
  - Immuable
  - Homogène (caractères)

---
- Range : `range = range(start, stop, step)`
  - Immuable
  - Homogène

# Python : opérateurs de séquences MUABLES

Opération	Résultat
<code>s[i] = x</code>	élément <i>i</i> de <i>s</i> est remplacé par <i>x</i>
<code>s[i:j] = t</code>	tranche de <i>s</i> de <i>i</i> à <i>j</i> est remplacée par le contenu de l'itérable <i>t</i>
<code>del s[i:j]</code>	identique à <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	les éléments de <code>s[i:j:k]</code> sont remplacés par ceux de <i>t</i>
<code>del s[i:j:k]</code>	supprime les éléments de <code>s[i:j:k]</code> de la liste
<code>s.append(x)</code>	ajoute <i>x</i> à la fin de la séquence (identique à <code>s[len(s):len(s)] = [x]</code> )
<code>s.clear()</code>	retire tous les éléments de <i>s</i> (identique à <code>del s[:]</code> )
<code>s.copy()</code>	Crée une copie de <i>s</i> (identique à <code>s[:]</code> )
<code>s.extend(t)</code> OU <code>s += t</code>	étend <i>s</i> avec le contenu de <i>t</i> (proche de <code>s[len(s):len(s)] = t</code> )
<code>s *= n</code>	met à jour <i>s</i> avec son contenu répété <i>n</i> fois
<code>s.insert(i, x)</code>	insère <i>x</i> dans <i>s</i> à l'index donné par <i>i</i> (identique à <code>s[i:i] = [x]</code> )
<code>s.pop([i])</code>	récupère l'élément à <i>i</i> et le supprime de <i>s</i>
<code>s.remove(x)</code>	supprime le premier élément de <i>s</i> pour qui <code>s[i] == x</code>
<code>s.reverse()</code>	inverse sur place les éléments de <i>s</i>

# Python : Fonction

Définition de la fonction :

```
def ma_somme(nb_1, nb_2) :  
    somme = nb_1 + nb_2  
    return somme
```

Appel de la fonction :

```
total = ma_somme(5, 8)
```

# Python : Dictionnaires

Ensemble de paires clés-valeurs  
(clés uniques et immuables)

```
dico = { cle1 : val1, cle2 : val2, ... }
```

Ajouter/modifier → `dico[cle] = val`

Récupérer 1 elmt → `dico[cle]`

Récupérer les clés → `dico.keys()`

Récupérer les valeurs → `dico.values()`

Récupérer les clés-valeurs → `dico.items()`

## 1.2. Les classes en Python

# Récapitulatif de ce qu'il faut dans une classe

= **Moule** de l'objet ou description générale de l'objet

- Description des attributs et des méthodes de toute une famille d'objets

Chaque classe **définit ses propres méthodes**  
(...)

- On distingue 4 grands types de méthodes "standards" qu'on retrouve dans la majorité des classes :
  - 2 types de méthodes liés à la **création/destruction** d'objets ;
  - 2 types de méthodes liés à l'**aspect privé** des valeurs des attributs.
- À côté de ces méthodes "standards", on trouve toutes les méthodes propres au fonctionnement de la classe.



# Récapitulatif de ce qu'il faut dans une classe

On doit retrouver :

- Description des **attributs**
  - Description des **méthodes**
    - Constructeur
    - Destructeur → Géré par le langage
    - Getters
    - Setters
    - Autres méthodes
- Encapsulation (module3)

# Classe Humain

## Human

name

age

gender

```
class Human :
```

```
def __init__ (self, name) :
```

```
    self.name = name
```

```
    self.age = 0
```

```
    self.gender = random.choice('MF')
```

Constructeur

Pour construire un objet : `<nom_de_la_classe> (<paramètres>)`  
Où `<paramètres>` sont les paramètres du constructeurs (sans le self)

```
>>> a = Human("Alice")
```

```
>>> a.name
```

```
'Alice'
```

```
>>> a.age
```

```
0
```

```
>>> a.gender
```

```
'F'
```

```
>>> a.age +=1
```

```
>>> a.age
```

```
1
```

```
>>> a.age +=1
```

```
>>> a.age
```

```
2
```

# Méthodes (d'instance)

Human
name
age
gender
say(msg)
birthday()

```
class Human :  
    ...  
    def say(self, message) :  
        print(self.name, ":", message)  
  
    def birthday(self) :  
        self.age += 1  
        print("Happy Birthday", self.name)
```

Méthodes d'instance  
(prennent toujours **self** en 1<sup>er</sup>)

**say** et **birthday** sont des **méthodes d'instance** car elles s'appellent sur une **instance** (un objet) de la classe.

Pour appeler une méthode d'instance :

**<objet>.<méthode>(<paramètres>)**

Où <paramètres> sont les paramètres de la méthode (sans le self)

# Méthodes (d'instance)

## Human

name

age

gender

say(msg)

birthday()

Pour appeler une méthode d'instance :  
<objet>.<méthode>(<paramètres>)

```
>>> a = Human("Alice")
>>> a.say("Hello")
Alice : Hello
```

L'appel d'une méthode sur un objet (par exemple a.say(...)) implique que l'objet en question est automatiquement donné comme premier paramètre (le **self**)

```
>>> a.birthday()
Happy Birthday Alice
>>> a.birthday()
Happy Birthday Alice
>>> a.say("I am "+a.age)
I am 2
```

# Attributs de classe

**Attributs d'instance :**  
attribut qui aura des  
valeurs potentiellement  
**différentes** pour les  
objets d'une même classe

**VS**

**Attributs de classe :**  
attribut qui aura la  
**même valeur** pour tous  
les objets de la classe

```
class Human :  
    species = "Homo sapiens"  
    nb_humans = 0  
  
    def __init__(self, name) :  
        self.name = name  
        self.age = 0  
        self.gender = random.choice('MF')  
        Human.nb_humans += 1  
    ...
```

Attributs de classe

Pour utiliser un attribut de classe :  
<classe>.attribut

```
>>> Human.species  
Homo sapiens  
>>> Human.nb_humans  
0
```

# Attributs de classe

```
class Human :  
    species = "Homo sapiens"  
    nb_humans = 0  
  
    def __init__(self, name) :  
        self.name = name  
        self.age = 0  
        self.gender = random.choice('MF')  
        Human.nb_humans += 1  
    ...
```

## Attributs de classe

Pour utiliser un attribut de classe :  
<classe>.attribut

```
>>> Human.species  
'Homo sapiens'  
>>> Human.nb_humans  
0
```

```
>>> a = Human("Alice")  
>>> b = Human("Bob")  
>>> Human.nb_humans  
2
```

Note : on peut aussi faire <objet>.attribut  
Si python ne trouve pas l'attribut dans les attributs  
d'instance, il ira voir les attributs de classe

```
>>> a.species  
'Homo sapiens'
```

# Méthodes de classe et statique

**Méthode d'instance :**  
méthode qui aura un  
comportement  
potentiellement **différent**  
pour les objets d'une même  
classe

**VS**

**Méthode de classe ou  
statique :**  
méthode qui aura le **même  
comportement** pour **tous** les  
objets de la classe

**Méthode de classe :**  
dépend d'attributs de classe

**VS**

**Méthode statique :**  
indépendante des attributs

## Pour résumer :

Méthode **d'instance**

modifie/utilise

des attributs **d'instance**

Méthode de **classe**

modifie/utilise

des attributs de **classe**

Méthode **statique**

modifie/utilise

**rien !**

# Méthodes de classe et statique

```
class Human :  
    species = "Homo sapiens"  
  
    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)  
  
    @classmethod  
    def get_species(cls):  
        return cls.species  
  
    @staticmethod  
    def sneeze():  
        print("Achoo !")
```

Attribut de classe

Appeler méthode de classe:

**<classe>.<méthode>(<paramètres>)**

Où <paramètres> sont les paramètres de la méthode (sans le cls)

```
>>> Human.get_species()  
'Homo sapiens'
```

L'appel d'une méthode sur une classe implique que la classe est automatiquement donnée comme premier paramètre (le **cls**)

Méthode de classe  
(prend **cls** en premier)

Méthode statique



# Méthodes de classe et statique

```
class Human :  
    species = "Homo sapiens"  
  
    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)  
  
    @classmethod  
    def get_species(cls):  
        return cls.species  
  
    @staticmethod  
    def sneeze():  
        print("Achoo !")
```

Appeler méthode de classe:

**<classe>.<méthode>(<paramètres>)**

Où <paramètres> sont les paramètres de la méthode (sans le cls)

```
>>> Human.get_species()  
'Homo sapiens'
```

L'appel d'une méthode sur une classe implique que la classe est automatiquement donnée comme premier paramètre (le **cls**)

Appeler une méthode statique :

**<classe>.<méthode>(<paramètres>)**

```
>>> Human.sneeze()  
Achoo !
```

# Classe Humain : résumé

```
class Human :  
    species = "Homo sapiens"  
  
    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)  
  
    @classmethod  
    def get_species(cls):  
        return cls.species  
  
    @staticmethod  
    def sneeze():  
        print("Achoo !")
```

Classe

Attribut de classe

Constructeur

Attributs  
(d'instance)

Méthodes (d'instance)

→ Prennent self en 1<sup>er</sup> argument

Méthode de classe

Méthode Statique

Human

name

age

gender

say(msg)

birthday()

# Classe Humain : résumé

```
class Human :
    species = "Homo sapiens"

    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)

    @classmethod
    def get_species(cls):
        return cls.species

    @staticmethod
    def sneeze():
        print("Achoo !")
```

```
>>> a = Human("Alice")
>>> a.say("Hello")
Alice : Hello
>>> a.birthday()
Happy Birthday Alice
>>> a.birthday()
Happy Birthday Alice
>>> a.say("I am "+a.age)
I am 2
```

```
>>> Human.get_species()
'Homo sapiens'
>>> Human.sneeze()
Achoo !
```

Human

name

age

gender

say(msg)

birthday()

## 1.3. Méthodes spéciales \_ \_

# Méthodes spéciales

les noms de méthodes encadrés par **deux soulignés** de part et d'autre sont des **méthodes spéciales**.

→ **Interdiction de nommez vos méthodes ainsi.**

Parmi ces méthodes, on retrouve :

- **\_\_init\_\_()** : le **constructeur**  
→ utilisé automatiquement quand on écrit <Classe>(...)
- **\_\_dir\_\_()** : **liste** tous les attributs et méthodes (y compris spéciales) que possède un objet.
- **\_\_str\_\_()** : renvoie une **description textuelle** de l'objet  
→ utilisé automatiquement pour caster un objet en str

Il existe aussi **\_\_dict\_\_**, un attribut spécial qui contient l'**état** de l'objet sous forme d'un **dictionnaire**

# Méthodes spéciales

Human
name
age
gender
say(msg)
birthday()

```
>>> a = Human("Alice")
>>> a.__dict__
{'name': 'Alice', 'age': 0, 'gender': 'F'}
>>> a.__dir__()
['_class_', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'say', 'birthday', 'name', 'age', 'gender']
```

# Réécriture des méthodes spéciales

Ces méthodes peuvent être réécrites !

De base

```
>>> a = Human("Alice")
>>> a.__str__()
'<__main__.Human object at
0x000001BFD37340F0>'
>>> print(a)
<__main__.Human object at
0x000001BFD37340F0>
```

```
class Human :
    ...
    def __init__(self, name) :
        self.name = name
        self.age = 0
        self.gender = random.choice('MF')
    ...
    def __str__(self):
        return "Human "
            + self.name + " "
            + "(" + self.gender + ") : "
            + str(self.age) + "year(s) old"
```

```
>>> a = Human("Alice")
>>> a.__str__()
'Human Alice (F) : 0year(s) old'
>>> print(a)
Human Alice (F) : 0year(s) old
```

/!\ casting

# 1.4. Traduire un diagramme UML en Python



# Traduire un schéma UML

Etapes :

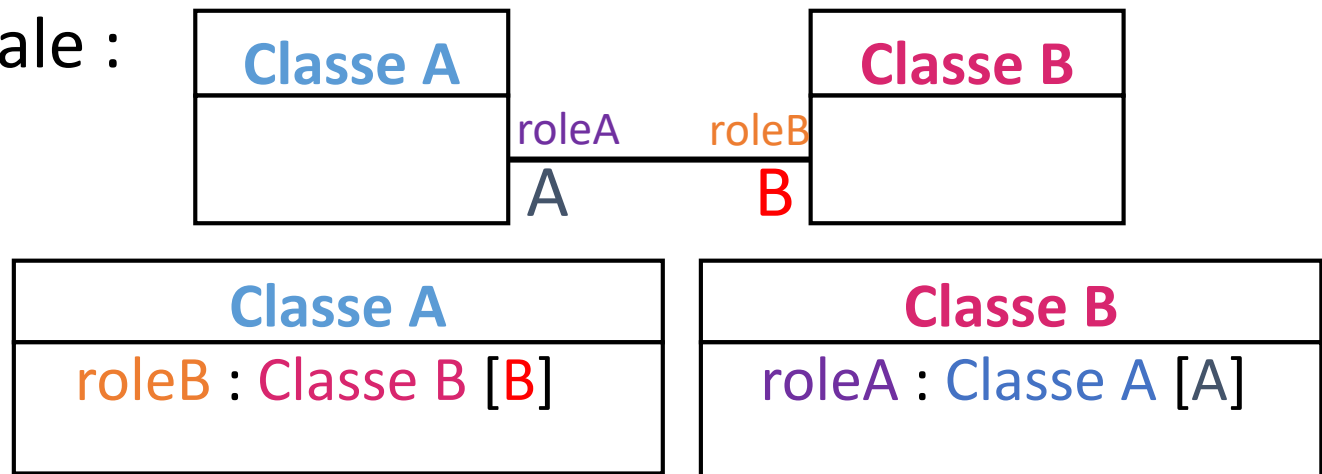
- 1) Faire disparaître les associations
- 2) Traduire chaque classe une à une

# Faire disparaître les associations

## Association entre deux classes

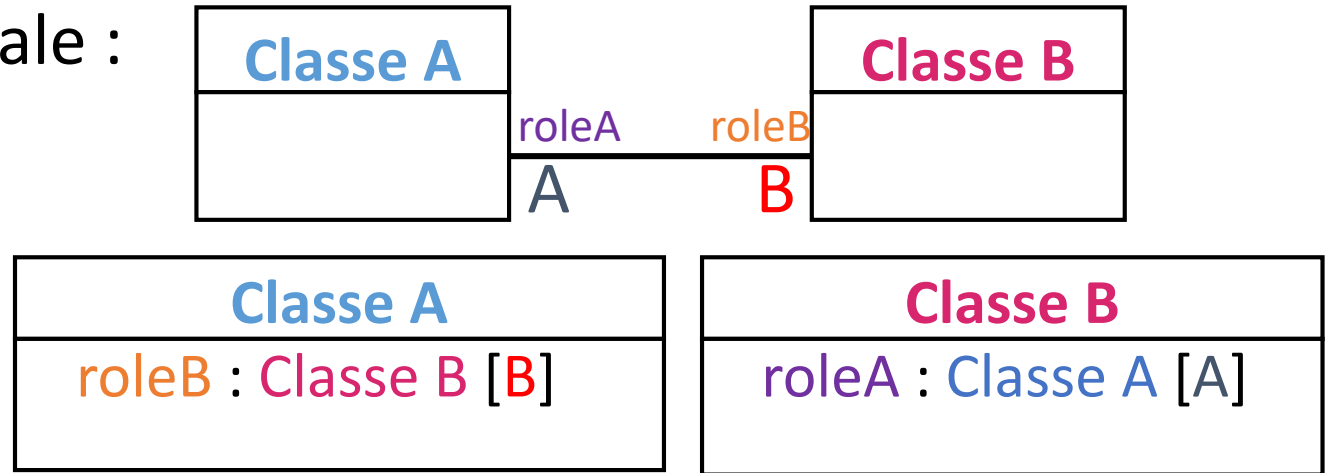
- (au moins) une des classes a un attribut instance de l'autre

Règle générale :

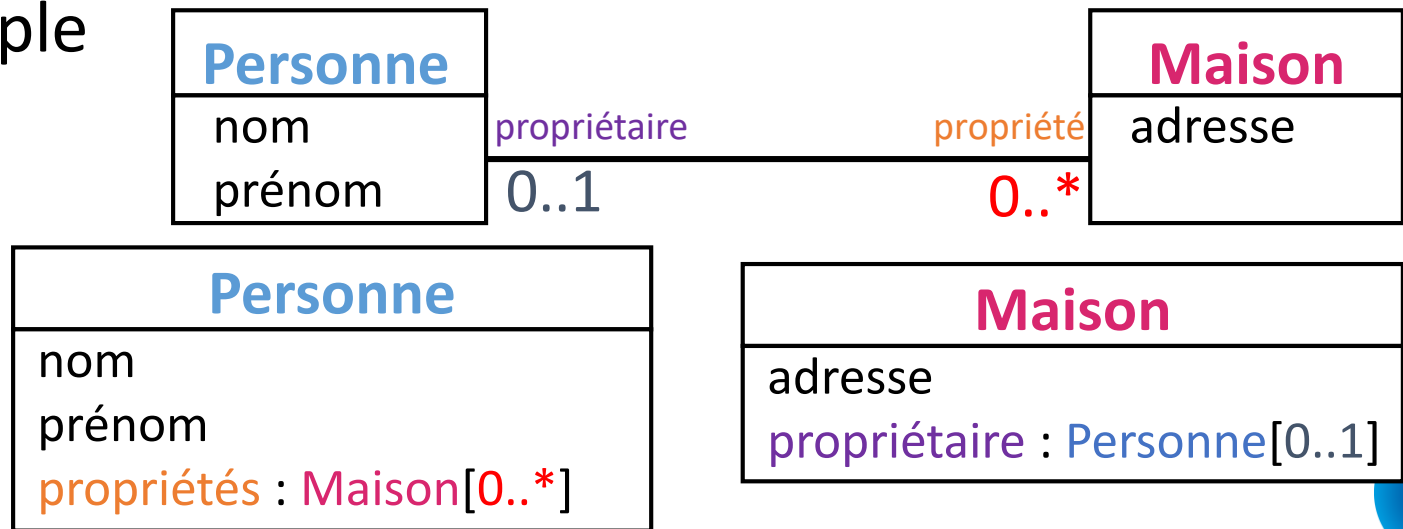


# Faire disparaître les associations

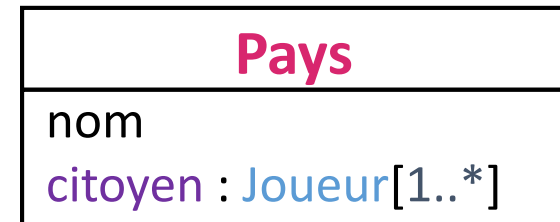
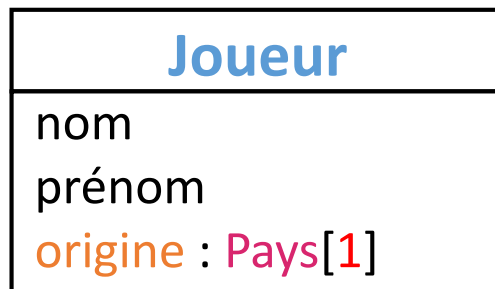
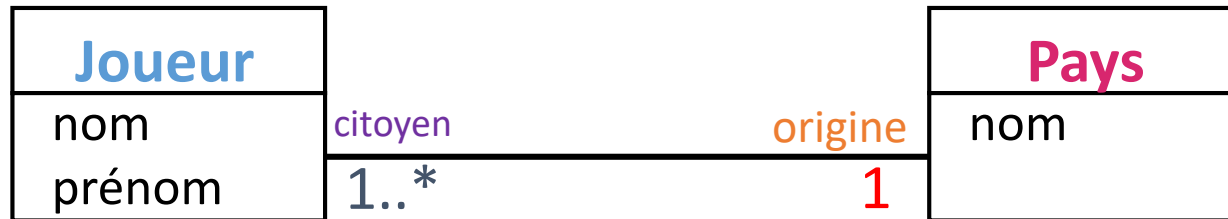
Règle générale :



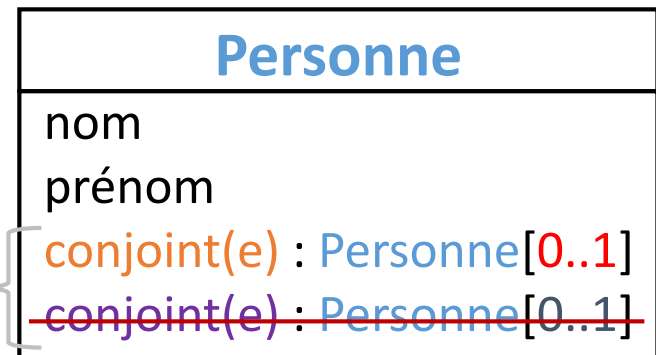
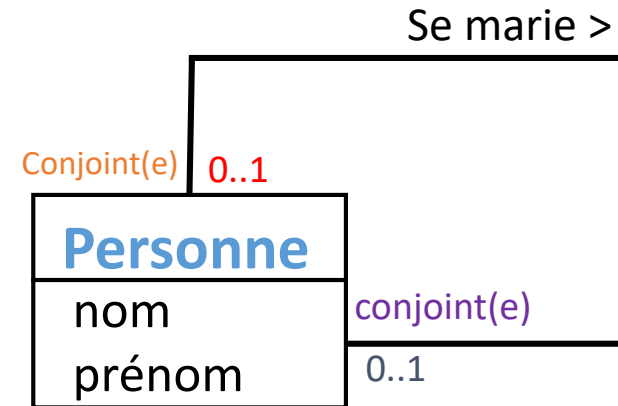
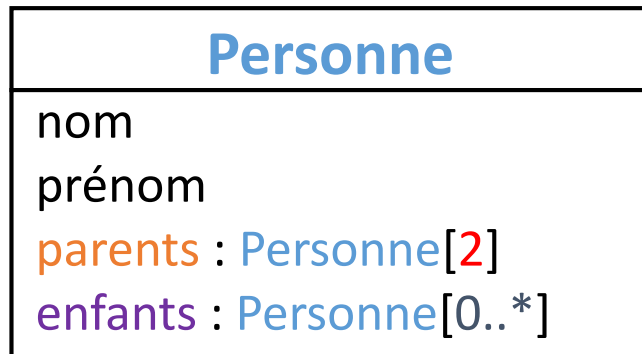
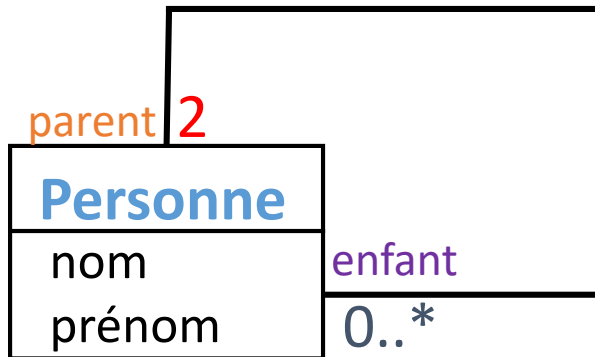
Exemple



# Exemples



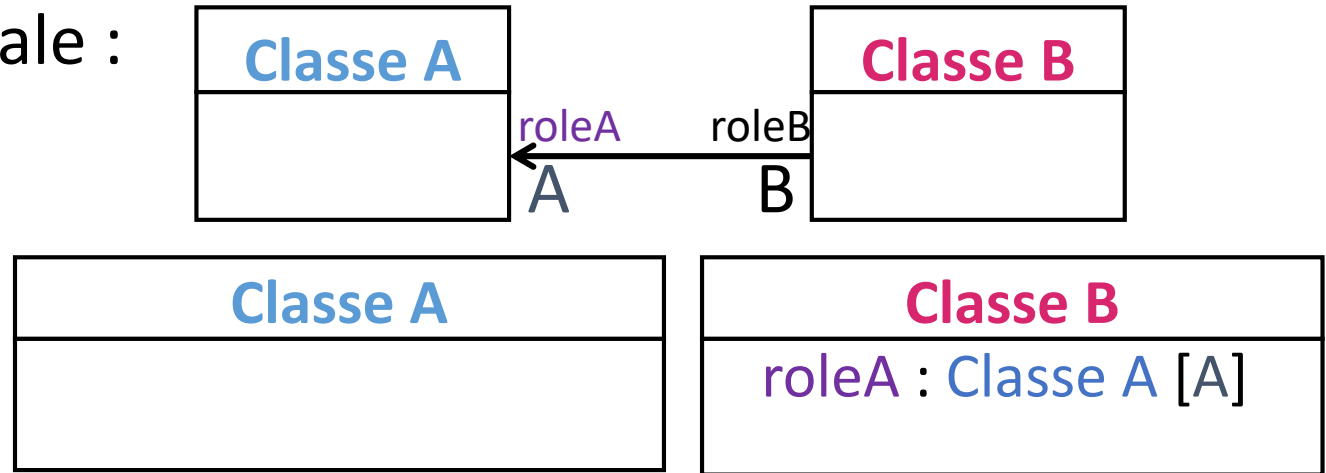
# Exemples



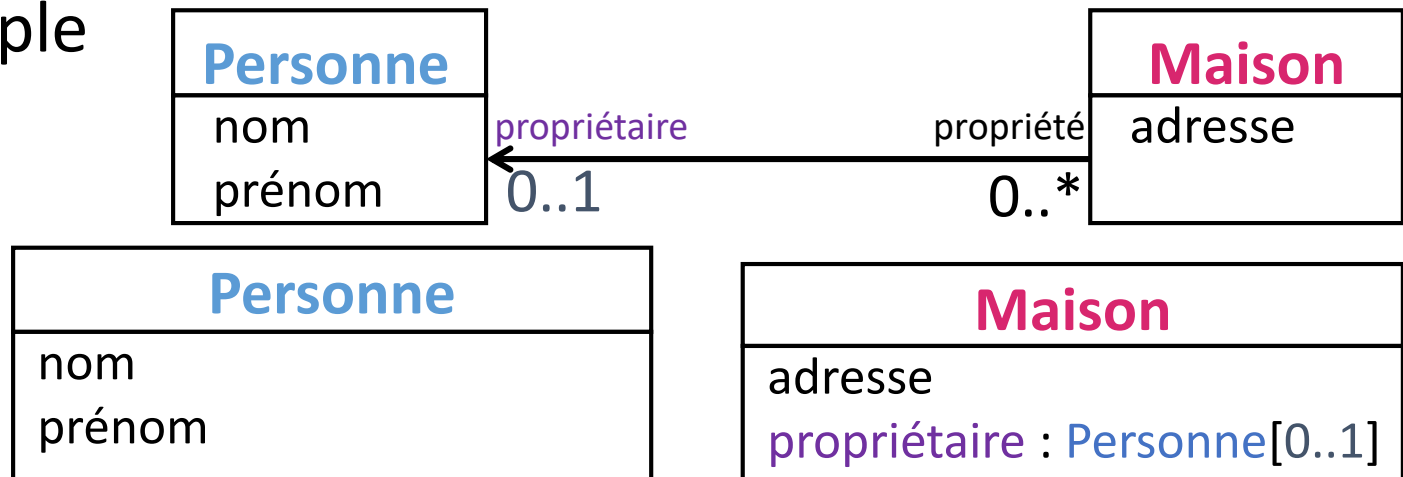
Ce sera le même

# Cas particulier : association unidirectionnelle

Règle générale :



Exemple



# Cas particulier ? agrégation ou composition

Ce sont des associations

➔ **Principe reste le même**

Dans le cas de la **composition** :

"Si le composite disparaît, les composants aussi"

se traduira dans le logiciel :

➔ quand on supprimera un objet composite, il faudra supprimer ses composants aussi

# Traduire : attributs facultatifs ou composites

## Maison

adresse  
propriétaire : Personne[0..1]

Attribut facultatif

→ paramètre facultatif dans le constructeur

```
class House :  
  
    def __init__(self, address, owner=None):  
        self.address = address  
        self.owner = owner
```

```
>>> h1 = House("Rue Calozet")  
>>> h1.__dict__  
{'address': 'Rue Calozet', 'owner': None}  
>>> h2 = House("Rue du Rèlis")  
>>> h2.__dict__  
{'address': 'Rue du Rèlis', 'owner': None}
```

## Personne

nom  
prénom  
propriétés : Maison[1..\*]

Attribut composite  
→ liste

```
class Person :  
  
    def __init__(self, name, fname, props:list):  
        self.name = name  
        self.firstname = fname  
        self.properties = props
```

```
>>> b = Person("Rasowski", "Bob", [h1, h2])  
>>> b.__dict__  
{'name': 'Rasowski', 'firstname': 'Bob',  
 'property': [<__main__.House at 0x1bfd3828780>,  
 <__main__.House at 0x1bfd3828b38>]}
```



# Traduire : attributs facultatifs et composites

## Personne

nom  
prénom  
propriétés : Maison[0..\*]

Attribut facultatif  
→ paramètre facultatif dans le constructeur

Attribut composite  
→ liste

```
class Person :  
  
    def __init__(self, name, fname, props=None):  
        self.name = name  
        self.firstname = fname  
        self.properties = props or []
```

```
>>> h1 = House("Rue Calozet")  
>>> h2 = House("Rue du Rèlis")
```

```
>>> a = Person("Dubois", "Alice")  
>>> a.__dict__  
{'name': 'Dubois', 'firstname': 'Alice', 'property': []}  
>>> b = Person("Rasowski", "Bob", [h1, h2])  
>>> b.__dict__  
{'name': 'Rasowski', 'firstname': 'Bob',  
 'property': [<__main__.House at 0x1bfd3828780>,  
 <__main__.House at 0x1bfd3828b38>]}
```

# Traduire chaque classe

```
>>> h1 = House("Rue Calozet")
>>> h1.__dict__
{'address': 'Rue Calozet', 'owner': None}
>>> h2 = House("Rue du Rèlis")
>>> h2.__dict__
{'address': 'Rue du Rèlis', 'owner': None}
```

```
>>> a = Person("Dubois", "Alice")
>>> a.__dict__
{'name': 'Dubois', 'firstname': 'Alice', 'property': []}
>>> b = Person("Rasowski", "Bob", [h1, h2])
>>> b.__dict__
{'name': 'Rasowski', 'firstname': 'Bob',
 'property': [<__main__.House at 0x1bfd3828780>,
 <__main__.House at 0x1bfd3828b38>]}
```

```
>>> h1.__dict__
{'address': 'Rue Calozet', 'owner': None}
>>> h2.__dict__
{'address': 'Rue du Rèlis', 'owner': None}
```

```
>>> h1.owner = b
>>> h1.__dict__
{'address': 'Rue Calozet', 'owner': <__main__.Person at 0x1bfd3813b70>}
>>> h2.owner = b
>>> h2.__dict__
{'address': 'Rue du Rèlis', 'owner': <__main__.Person at 0x1bfd3813b70>}
```

# Traduire : attributs dérivables

## Personne

nom  
prénom  
conjoint : Personne[0..1]  
/est\_marié : bool

Attribut dérivable  
→ méthode

```
class Person :  
  
    def __init__(self, name, fname, spouse=None):  
        self.name = name  
        self.firstname = fname  
        self.spouse = spouse  
  
    def is_married(self):  
        return self.spouse != None
```

```
>>> a = Person("Dubois","Alice")  
>>> a.is_married()  
False  
>>> b = Person("Dupond","Bob",a)  
>>> a.spouse = b  
>>> b.is_married()  
True  
>>> a.is_married()  
True
```

# Problème du min 1 – min 1

## Joueur

nom  
prénom  
origine : Pays

```
class Player:

    def __init__(self, name, fname, origin):
        self.name = name
        self.firstname = fname
        self.origin = origin
```

## Pays

nom  
citoyen : Joueur[1..\*]

```
class Country:

    def __init__(self, name, citizens):
        self.name = name
        self.citizens = citizens
```



**Qu'est ce que je crée en premier?**

Pour créer un objet joueur → je dois lui donner un pays d'origine ...

... Donc, je vais commencer par un pays ...

Mais pour créer un objet Pays → je dois lui donner une liste de citoyens ...

... Donc il me faut des joueurs ...


# Problème du min 1 – min 1

## 2 solutions :

- Changer les multiplicités

Joueur
nom
prénom
origine : Pays

Pays
nom
citoyen : Joueur[0..*]



- "Frauder"
  - Créer un joueur momentanément sans pays
  - Créer un pays avec ce joueur
  - Changer directement l'origine du joueur

# Problème du min 1 – min 1

- Créer un joueur momentanément sans pays
- Créer un pays avec ce joueur
- Changer directement l'origine du joueur

## Joueur

nom  
prénom  
origine : Pays

```
class Player:
```

```
    def __init__(self, name, fname, origin):  
        self.name = name  
        self.firstname = fname  
        self.origin = origin
```

## Pays

nom  
citoyens:Joueurs[1..\*]

```
class Country:
```

```
    def __init__(self, name, citizens):  
        self.name = name  
        self.citizens = citizens
```

```
>>> p1 = Player("Dubois", "Alice", None)  
>>> p1.__dict__  
{'name': 'Dubois', 'firstname': 'Alice', 'origin': None}  
>>> usa = Country("USA", [p1])  
>>> p1.origin = usa  
>>> p1.__dict__  
{'name': 'Dubois', 'firstname': 'Alice', 'origin':  
<__main__.Country at 0x1bfd3813748>}
```

```
>>> p2 = Player("Dupond", "Bob", usa)  
>>> p2.__dict__  
{'name': 'Dupond', 'firstname': 'Bob', 'origin':  
<__main__.Country at 0x1bfd3813748>}
```

```
>>> usa.citizens.append(p2)  
>>> usa.__dict__  
{'name': 'USA', 'citizens': [<__main__.Player at  
0x1bfd3828e80>, <__main__.Player at 0x1bfd3813630>]}
```

A vous de jouer...