

Conception et Programmation Objet

UE Informatique – Inf-TC2

Stéphane Derrode, Bât. E6, 2ième étage, stephane.derrode@ec-lyon.fr

S5 • INF-TC2 - S. Derrode

- « Conception et programmation objet »
- Volume : 8h de cours, 17h de TD et 5h d'autonomie
- Travail à rendre : Séances BE #3 et #5

- **INF-TC1 - R. Vuillemot**

- « Algorithmes et structures de données »

- **INF-TC3 - D. Muller / R. Chalon**

- « Projet d'application WEB »

- **Évaluation de l'UE informatique**

- Examen papier : 50 %
- Moyenne des 2 BEs (binôme) : 50%



- **Séances #1 : Python orienté objet**
 1. Paradigme « objet »
 2. Les classes/objets
 3. Association et composition
- **Séances #2 : Python orienté objet (suite)**
 4. L'héritage et le polymorphisme
 5. Les exceptions
- **Séance #3 :**
 - Interfaces graphiques (T. Raillaç)
- **Séance #4 :**
 6. Autres points (surcharge des opérateurs, et attributs de classes)
 7. Les tests unitaires
 8. Introduction au GL, aux méthodes agiles
 9. *Versionning* avec Git



Quiz Wooclap

- [Wooclap](#) est un dispositif permettant de d'interagir avec l'audience (sous forme de quizz) dans le cadre d'un cours (ou d'une conférence).
- Wooclap sera utilisé tout au long du cours de INF-TC2 pour vérifier si les points clés sont acquis en temps réels, et éventuellement redonner des explications si les résultats du quizz montrent des lacunes de l'assistance.
- Wooclap n'est utilisable qu'en mode synchrone. Pour les étudiants qui suivent le cours en mode asynchrone, j'ai copié les questions posées à l'audience dans ces slides (mais vous n'aurez pas accès aux bonnes réponses !).
- Quizz sous forme de QCM, nuage de mots, textes à trous, sondage... avec parfois un temps limité pour répondre.

Remarque: la participation à Wooclap est nominative (vous êtes identifié et donc pistable!). Evitez les réponses indésirables...



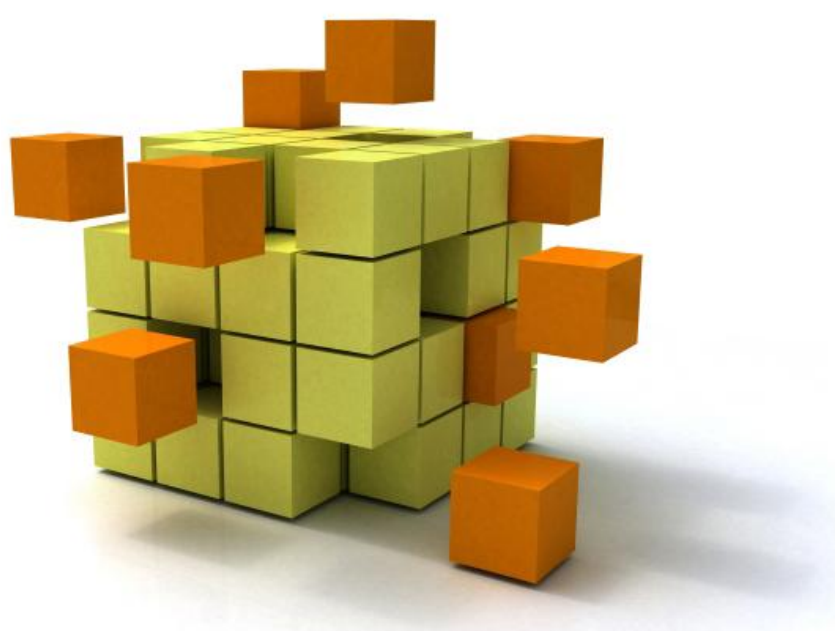
Quizz Wooclap

Quizz 1 (suivi asynchrone du cours):

Quels mots caractérisent le mieux la discipline
Informatique selon vous ?

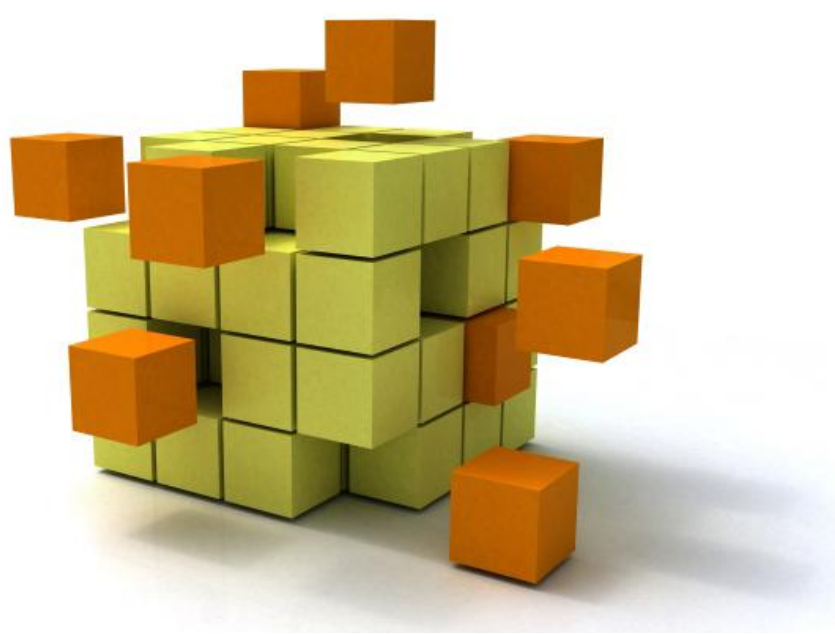
Réponse graphique : un nuage des mots les plus cités
par l'audience.





Séances #1 et #2

1. Paradigme « objet »
2. Les classes/objets
3. Association et composition
4. L'héritage et le polymorphisme
5. Les exceptions
6. Autres points
7. Les tests unitaires



#1- Paradigme « objet »

Les objets, ... c'est pas nouveau!

```
# Different types of objects can possess  
# different methods  
  
string = "hello world"  
string.capitalize() # use the string-method 'capitalize'  
  
import numpy as np  
array = np.array([[0, 1, 2], [3, 4, 5]])  
array.sum() # use the array-method 'sum'
```

```
# accessing an object's attributes  
>>> array.ndim  
2  
>>> array.shape  
(2, 3)
```



La plupart de ces librairies sont programmées à l'aide de classes → **programmation orientée objet !**

Exemple: *Data sciences*

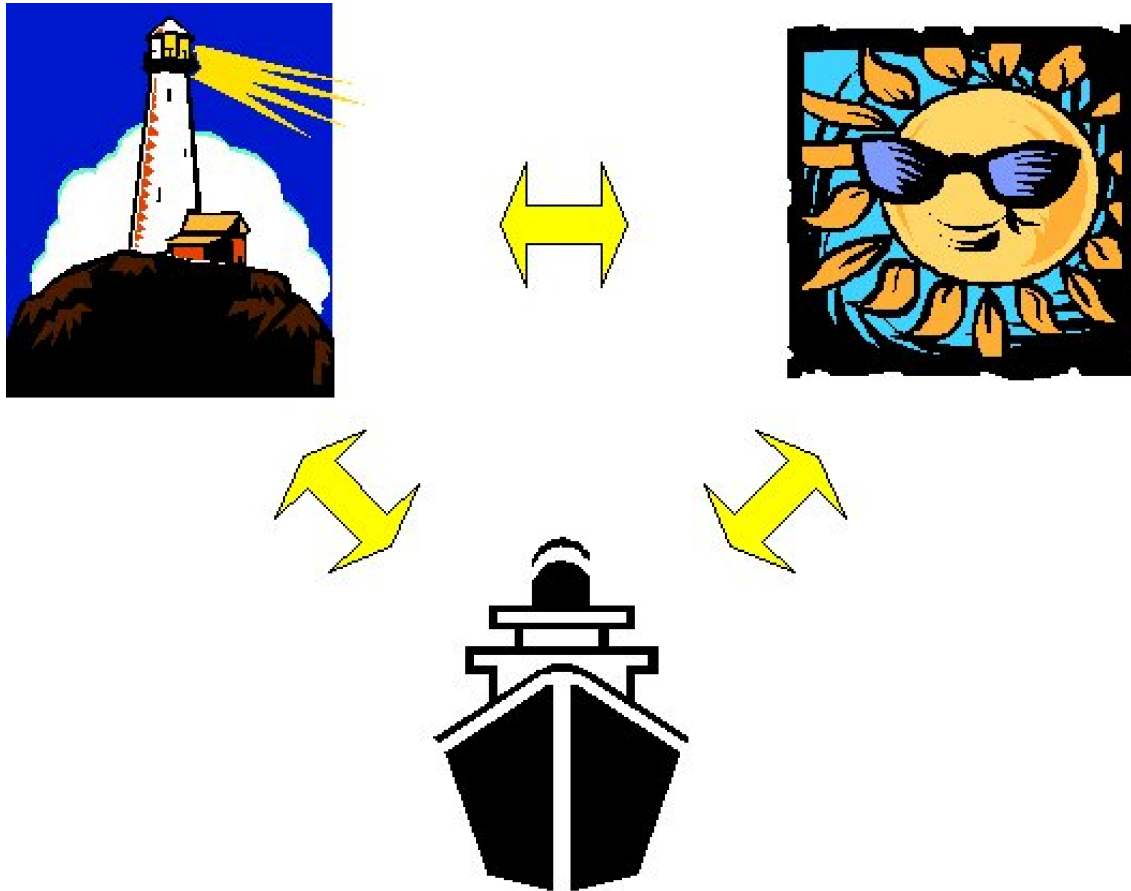
- *Data visualization* :
 - [Matplotlib gallery](#),
 - [Seaborn gallery](#)...
- [Librairie Plotly](#) (*dashboard*) :
 - [Word embedding visualization](#),
 - [Image processing App](#),
 - [Hand digit recognition](#).
- *Machine learning* :
 - *Deep learning*: [Keras](#) ([tensorflow](#))...
- Applications graphiques :
 - PyQT (wrapper Python pour la librairie C++ QT): [gallery](#)



La plupart de ces librairies sont programmées à l'aide de classes → **programmation orientée objet!**

Paradigme objet : application

Le monde réel est composé d'objets (réels ou conceptuels) autonomes qui sont en relation (communication) et coopèrent.

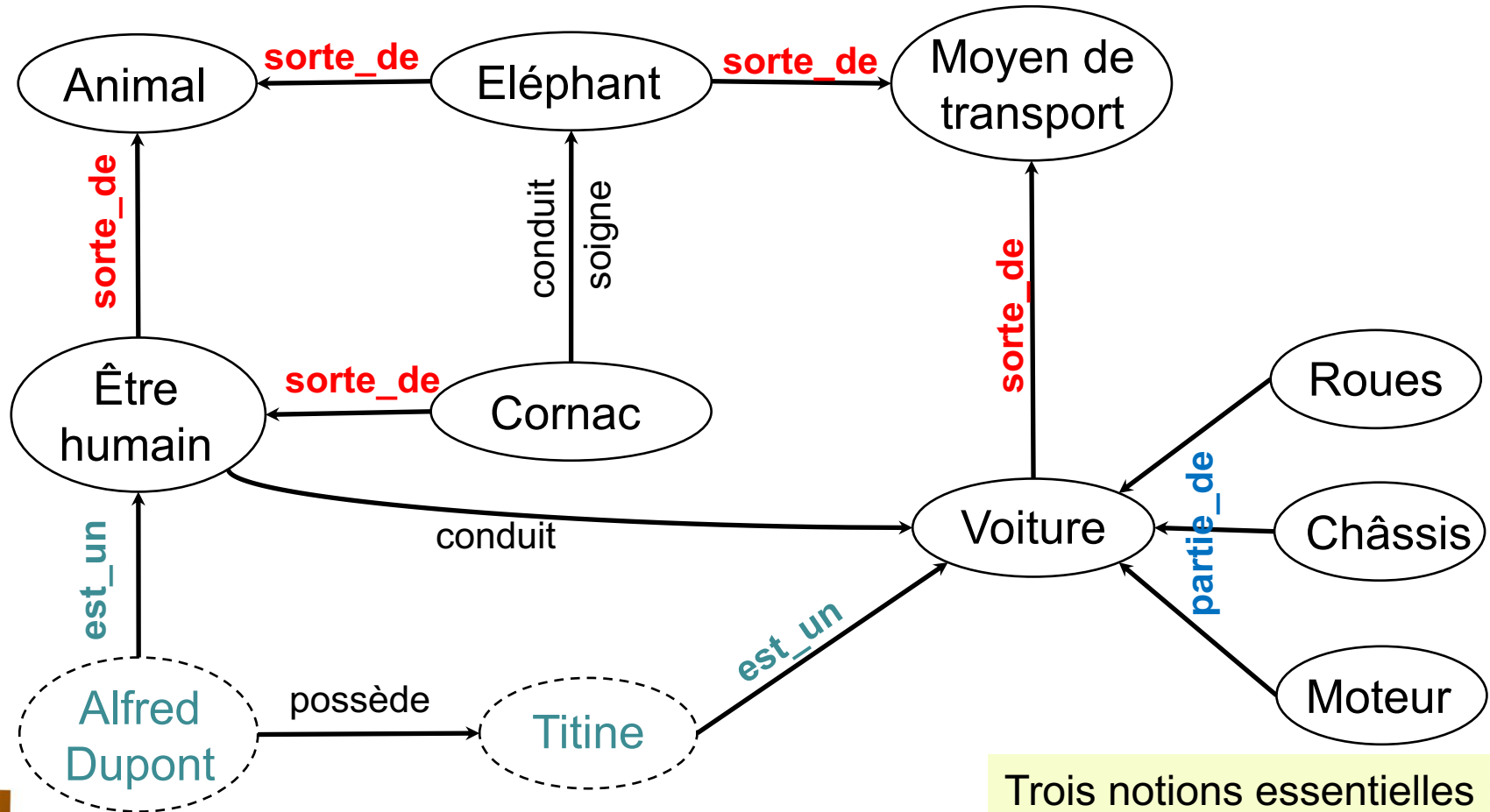


Paradigme : représentation, vision du monde, modèle, courant de pensée, point de vue.



Paradigme objet : modélisation

Exemple : Réseau sémantique

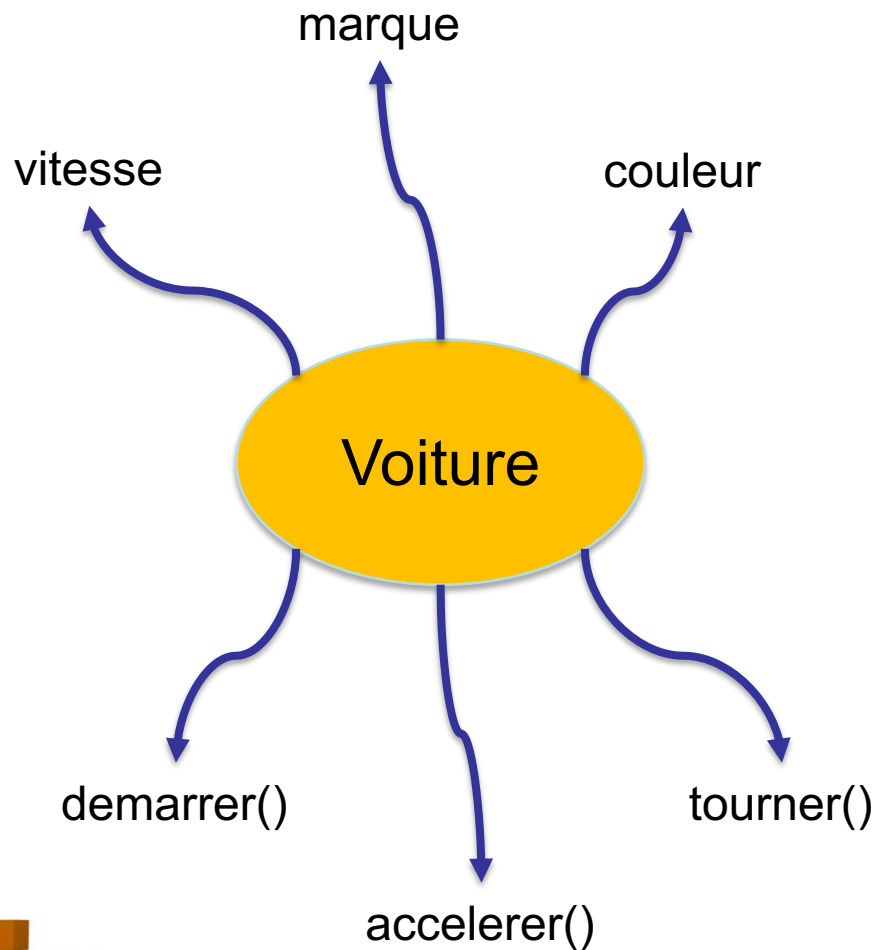


Cornac : Personne chargée des soins et de la conduite d'un éléphant.

Trois notions essentielles :

- Classe/objet
- Héritage
- Composition

Paradigme objet: classe



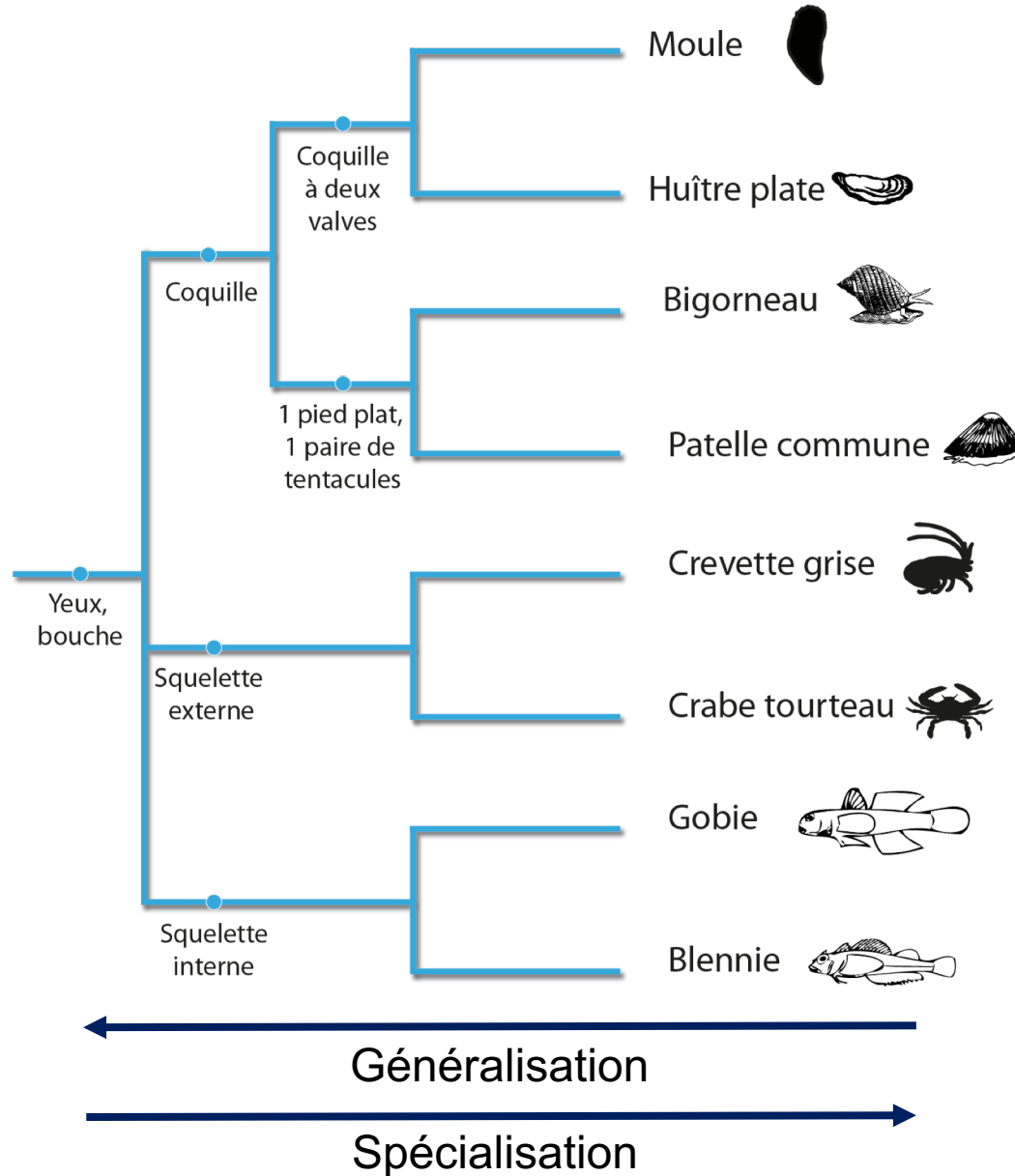
Attributs qui définissent son état.

Comportement : ensemble de **méthodes** qui modifie généralement son état.



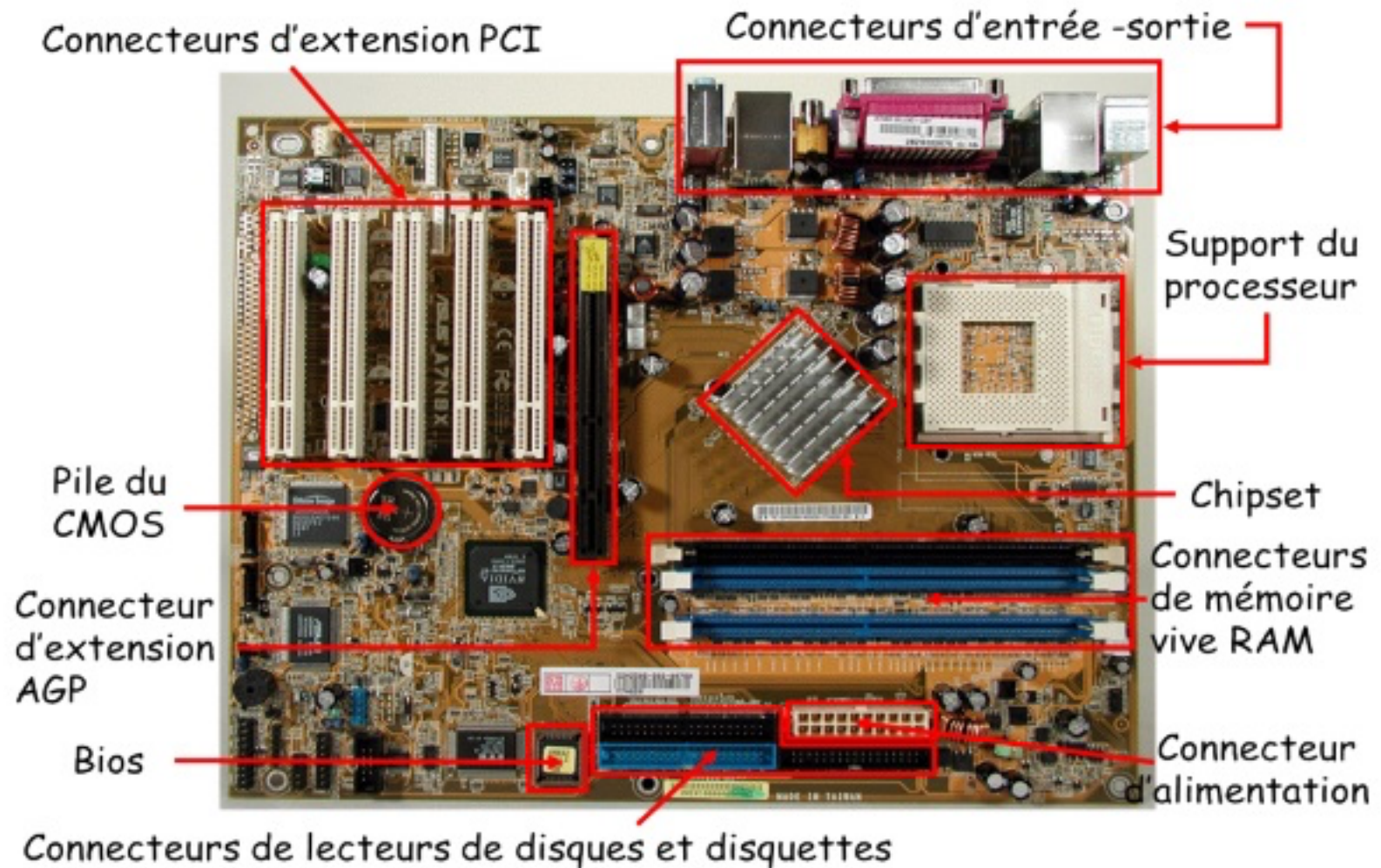
Paradigme objet: héritage

La biologie des gastéropodes

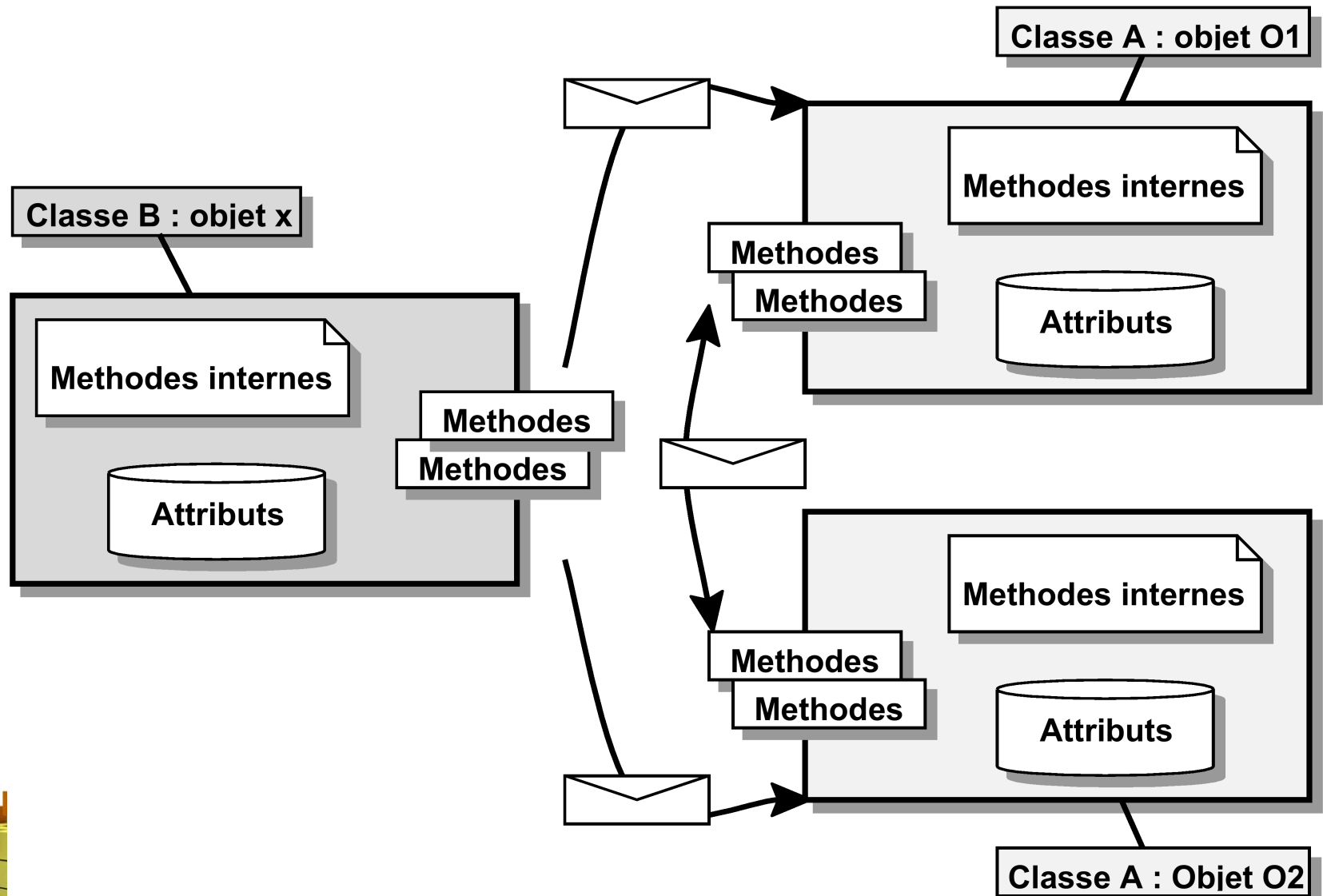


Paradigme objet: composition

La carte mère



Classes et objets



Quizz Wooclap

Quizz 2 (suivi asynchrone du cours):

Pour les 3 questions ci-dessous, répondez par

- Héritage (sorte de)
- Composition (partie de)
- Objet (est un)
- Aucun

Questions: Quelle relation existe t'il entre ...

1. ... un ordinateur (computer) et une carte mère (mother board) ?
2. ... un Monument et la Tour Eiffel ?
3. ... une étoile et un soleil ?



Quizz Wooclap

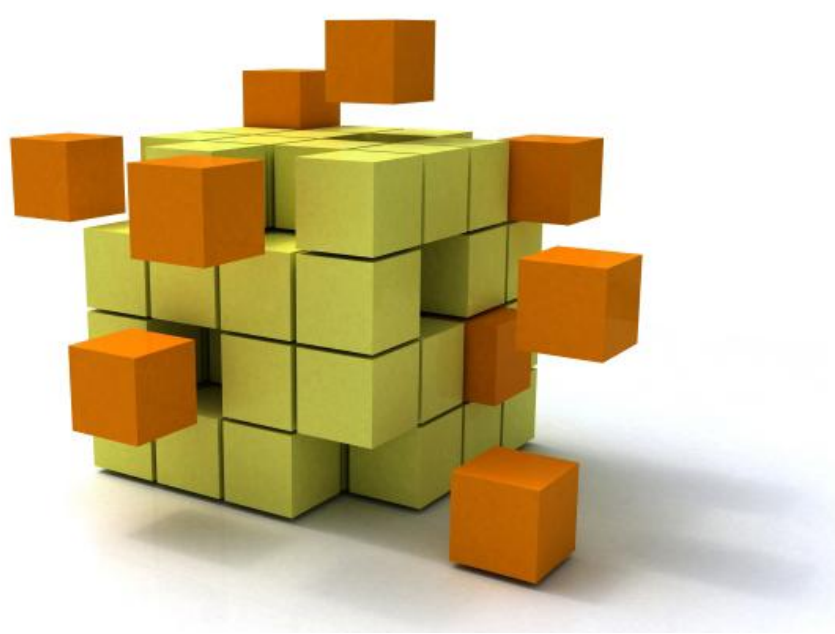
Quizz 3 (suivi asynchrone du cours):

Remplacer le texte à trou par le mot ATTRIBUT ou par le mot METHODE.

« Mon voilier est amarré à Saint-Tropez. Il possède 3 mats ([?]) et mesure 30 mètres de long ([?]). Pour changer de cap ([?]), il faut savoir manipuler ([?]) les cordages associés aux différentes voiles. Un moteur de 300 chevaux ([?]) permet de le manœuvrer ([?]) dans les ports. »

English translation: My sailboat is docked in Saint-Tropez. It has 3 masts and is 30 meters long. To change its course, it is necessary to know how to handle the ropes of the different sails. A 300 horsepower engine allows to maneuver in the harbors.



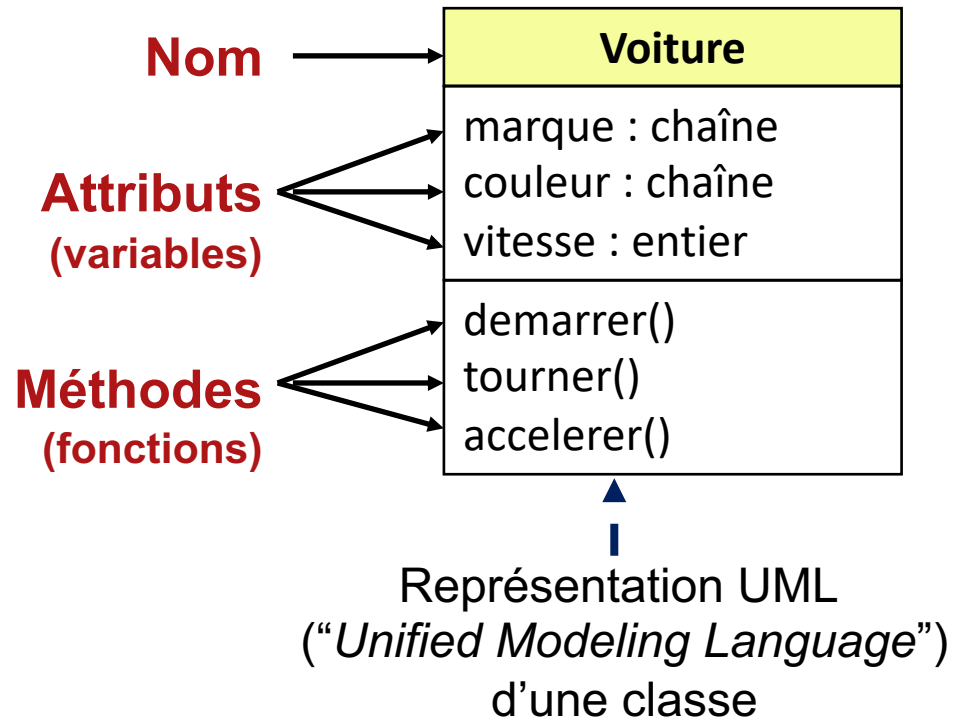
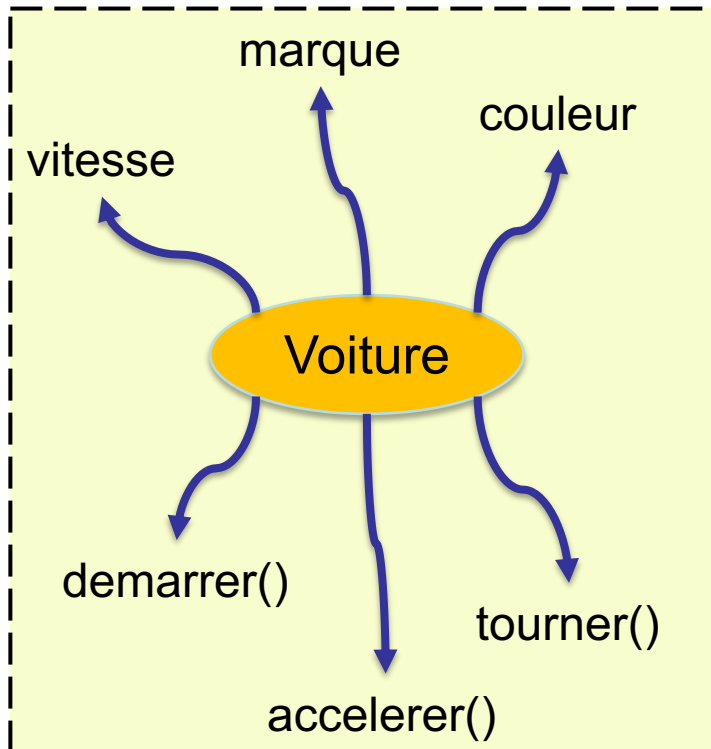


#2- Classes et objets

constructeur
encapsulation

Classes et objets

- **Une classe** : regrouper, en une seule entité informatique (la classe), les données et les traitements sur ces données.

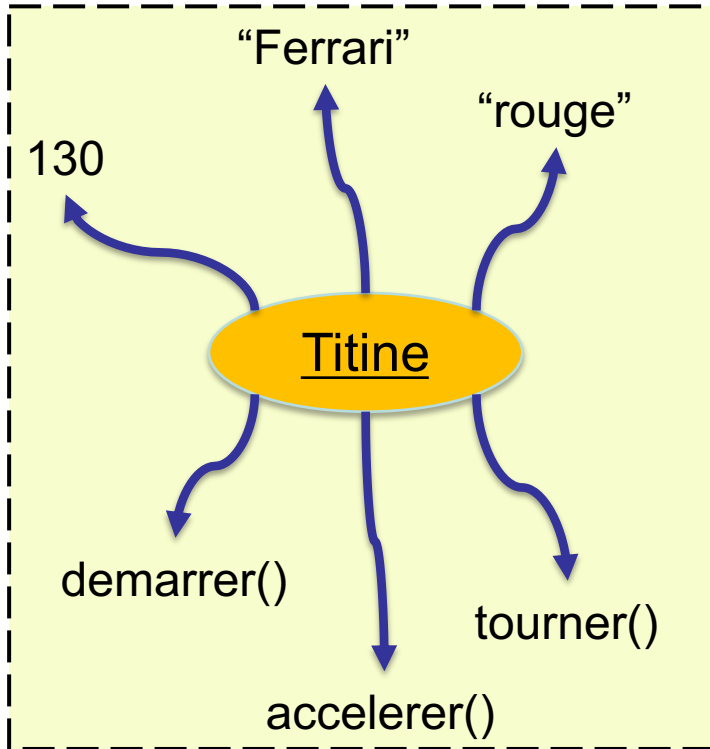


UML dessin vectoriel : <https://app.diagrams.net>.

Tutoriel : <https://drawio-app.com/uml-class-diagrams-in-draw-io/>. 20

Classes et objets : instance

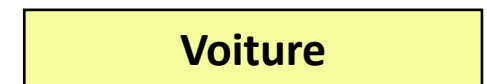
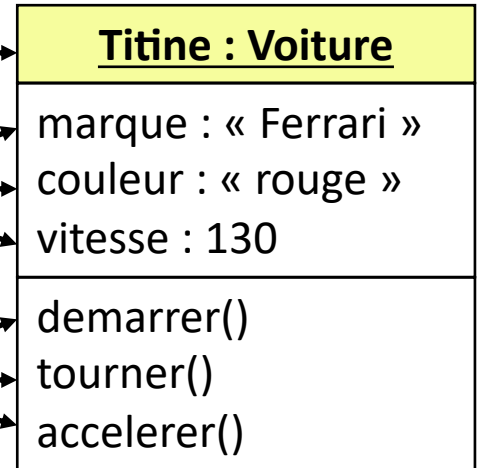
- Un objet (instance d'une classe)



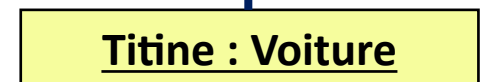
Identité : Type

Attributs

Méthodes



"Titine est un objet de la classe Voiture"
"Titine est une instance de la classe Voiture"



Classes et objets : python

```
class Voiture:
    def demarrer(self):
        print('Je démarre ma voiture')

    def tourner(self):
        print('Je tourne le volant')

    def accélérer(self):
        print('J\'accélère jusqu\'à 50km')

if __name__ == '__main__':
    titine = Voiture()
    titine.demarrer()
    titine.tourner()
    titine.accélérer()
```

Une classe

“moi-même”!

Signature de la méthode

Appel au constructeur par défaut

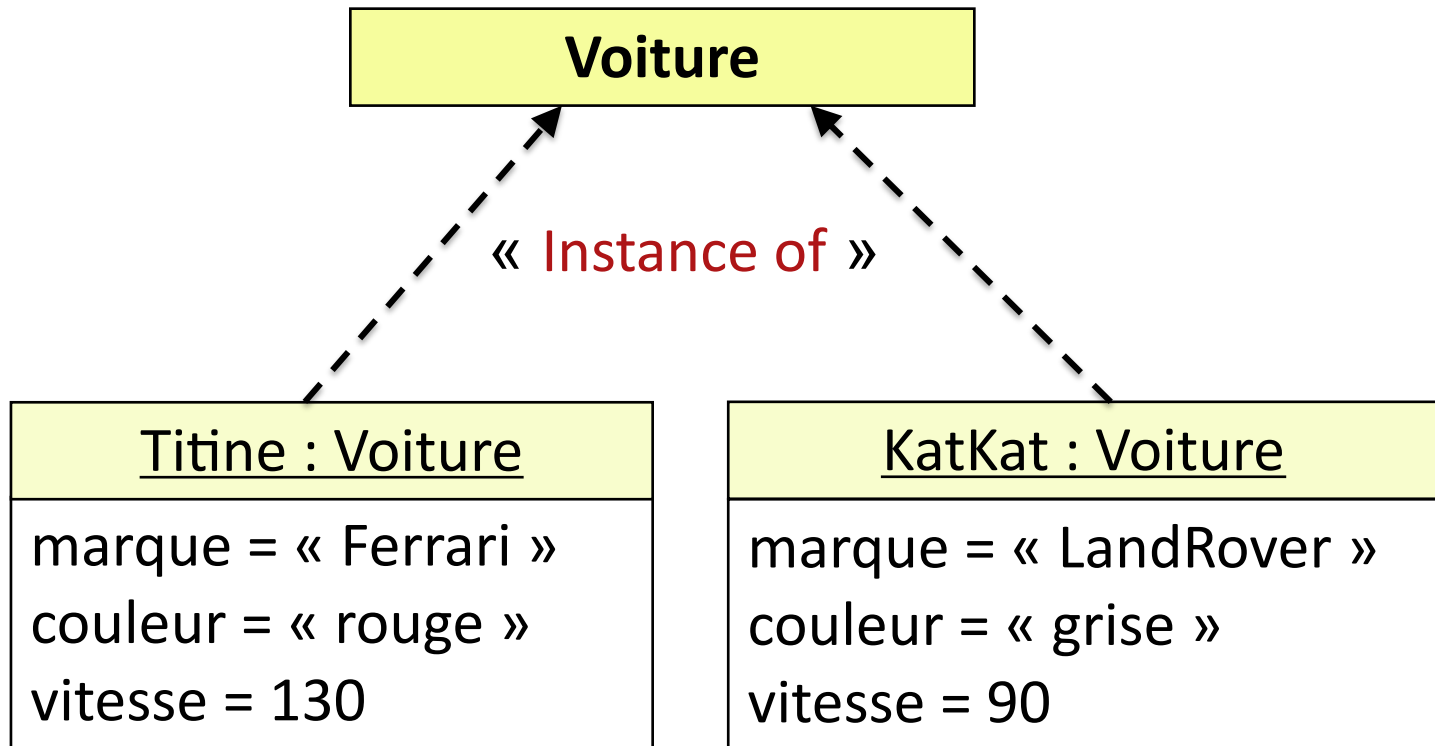
Transmission de messages à l'objet à l'aide des méthodes (publiques)

Programme principal



Classes et objets : python

- Chaque objet qui est une instance de la classe **Voiture** possède ses propres valeurs d'attributs



Classes et objets : python

```
class Voiture:
```

```
    def __init__(self, m, v, c):
```

← “constructeur”

```
        self.marque=m
```

```
        self.vitesse=v
```

← Attributs

```
        self.couleur=c
```

```
    def demarrer(self):
```

```
        print('Je démarre ma', self.marque)
```

```
    def accélérer(self):
```

```
        print('J\'accélère jusqu\'a', self.vitesse)
```

```
if __name__ == '__main__':
```

```
    titine = Voiture("Ferrari", 130, "rouge")
```

```
    titine.démarrer()
```

```
    titine.accélérer()
```

← Création d'un objet

← par appel au constructeur

```
    katkat=Voiture("LandRover", 90, "gris")
```

```
    katkat.démarrer()
```

```
    katkat.accélérer()
```



Classes et objets : adresse mémoire

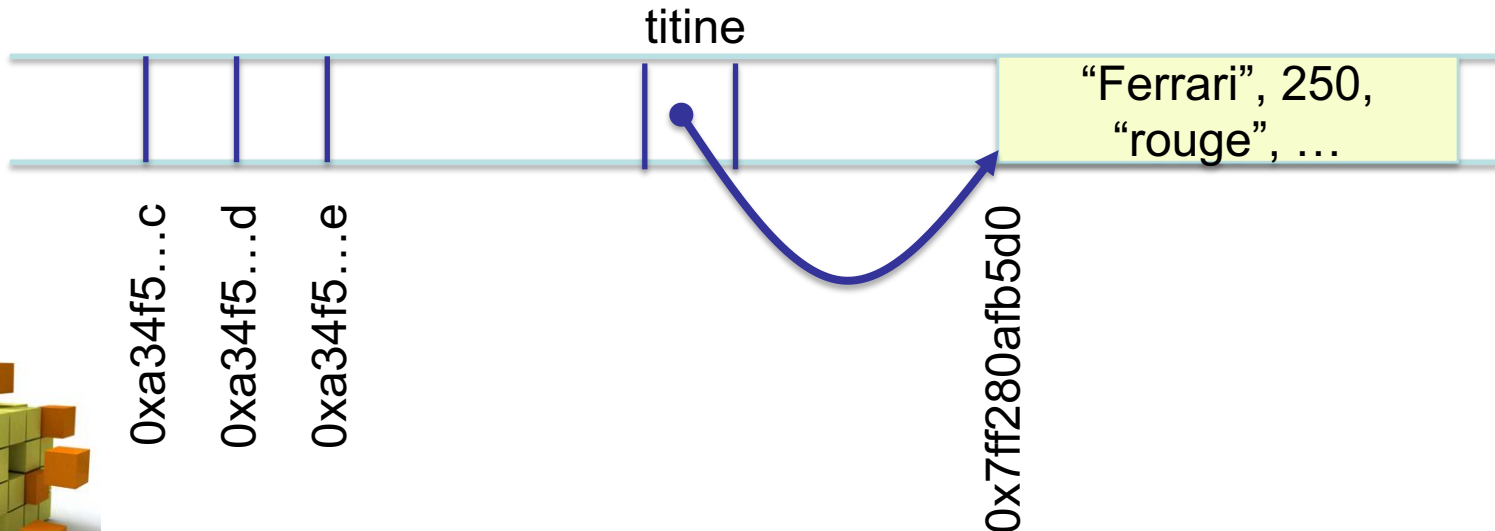
```
from Voiture2 import Voiture

if __name__ == '__main__':
    titine = Voiture("Ferrari", 250, "rouge")
    print(titine)
```

<Voiture2.Voiture object at 0x7ff280afb5d0>

Adresse de la mémoire
où est stocké l'objet (en
base hexadécimale)

Schématisation de la mémoire



Classes et objets : méthode `__str__`

```
class Voiture:
    def __init__(self, m, v, c):
        self.marque=m
        self.vitesse=v
        self.couleur=c

    def __str__(self):
        S = '  Marque   : ' + self.marque + '\n'
        S += '  Couleur  : ' + self.couleur + '\n'
        S += '  Vitesse  : ' + str(self.vitesse)
        return S

if __name__ == '__main__':
    titine = Voiture('Ferrari', 250, 'rouge')
    print('Etat:\n', titine) ← Équiv. print(titine.__str__())
    print('Adresse:', hex(id(titine)))
```

Etat:

Marque : Ferrari

Couleur : rouge

Vitesse : 250

Adresse: 0x7fdbccb36710



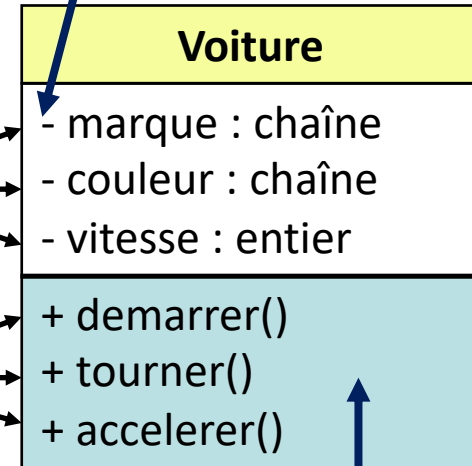
Classes et objets : encapsulation (1)

- Masquer les détails de l'implémentation en définissant une interface.
- L'interface définit les services accessibles par les autres objets : méthodes publiques de l'objet.
- L'encapsulation facilite l'évolution car on peut changer l'implémentation de l'objet sans changer l'interface.
- Les attributs et les méthodes **privées** sont protégées (accessibles seulement par les méthodes de la classe).

Symboles '-', '+'
attributs / méthodes
privés ou publics

Attributs
(variables)

Méthodes
(fonctions)



Interface de la classe =
ensemble des méthodes
publiques



Classes et objets : encapsulation (2)

```
class Voiture:

    def __init__(self, m, v, c):
        self.__marque=m
        self.__vitesse=v
        self.__couleur=c

    def demarrer(self):
        print('Je démarre ma', self.__marque)

if __name__ == '__main__':
    titine = Voiture('Ferrari', 130, 'rouge')
    titine.demarrer()
    print("Vitesse :", titine.__vitesse)
```

Voiture

- marque : chaîne
- couleur : chaîne
- vitesse : entier

- + constructeur(...)
- + demarrer()

Je démarre ma Ferrari

Traceback (most recent call last):

File "Voiture4.py", line 18, in <module>

print("Vitesse Max :", titine.__vitesse)

AttributeError: 'Voiture' object has no attribute '__vitesse'



Classes et objets : encapsulation (3)

```
class Cercle:
    def __init__(self, r, x, y):
        self.__ray = 1
        self.setr(r)
        self.pos = x,y      # tuple
    def __str__(self):
        S = 'rayon='+str(self.__ray)+' , pos='
        S += str(self.pos[0])+' , '+str(self.pos[1])
        return S
    def getr(self):          # -->getter sur le rayon
        return self.__ray
    def setr(self, newr):    # -->setter sur le rayon
        if newr>0.: # protection de l'attribut
            self.__ray = newr

if __name__ == '__main__':
    C1 = Cercle(10, 5, -3)
    print('C1:', C1)
    C1.pos= -1,-1
    print('Nouvelle position:', C1.pos)
    #C1.__ray = -10          # <-- code interdit
    C1.setr(-10) # sans erreur (mais sans effet)
    print('rayon', C1.getr())
```

Cercle

- ray : réel
+ x, y : réels

+ constructeur(...)
+ affiche()
+ getr() : réel
+ setr(newr:réel)

Classes et objets : Liste d'objets

```
from Voiture3 import Voiture

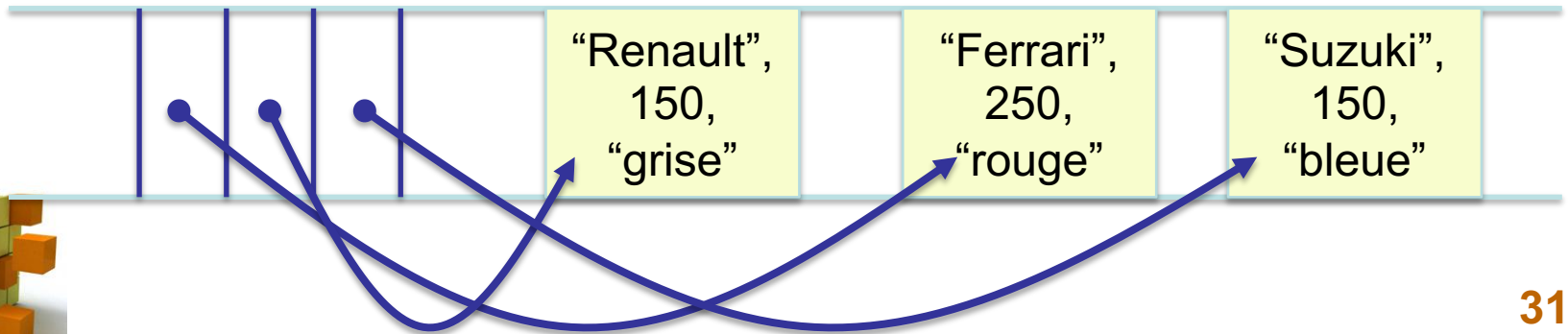
if __name__ == '__main__':

    lMarque = ['Ferrari', 'Renault', 'Suzuki']
    Lv      = [250,      150,      150]
    lCouleur= ['rouge',   'grise',   'bleue']

    lVoitures = []
    for m, v, c in zip(lMarque, Lv, lCouleur):
        lVoitures.append(Voiture(m, v, c))

    print('lVoitures=', lVoitures)
```

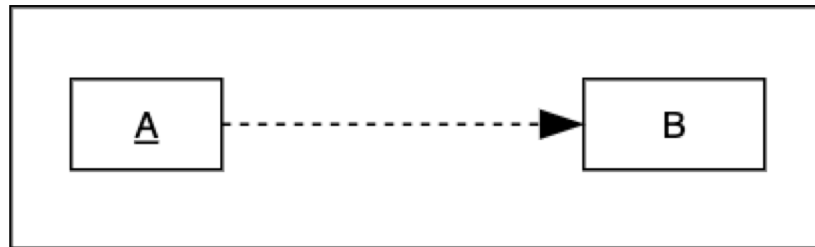
lVoitures= [
 Marque : Ferrari
 Couleur : rouge
 Vitesse : 250,
 Marque : Renault
 Couleur : grise
 Vitesse : 150,
 Marque : Suzuki
 Couleur : bleue
 Vitesse : 150]



Quizz Wooclap

Quizz 4 (suivi asynchrone du cours):

Que représente ce lien ?



Réponses possibles:

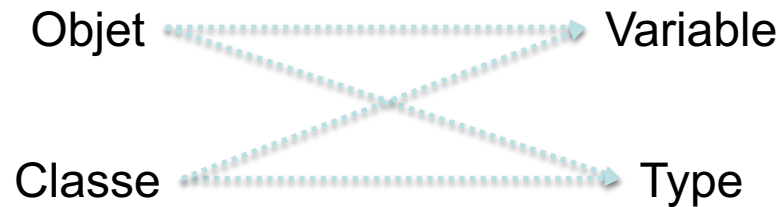
1. B est une méthode de A
2. B est une instance de A
3. A est un attribut de B
4. A est une instance de B



Quizz Wooclap

Quizz 5 (suivi asynchrone du cours):

Liez les notions similaires. À gauche des notions de programmation objet, à droite des notions de programmation structurée.

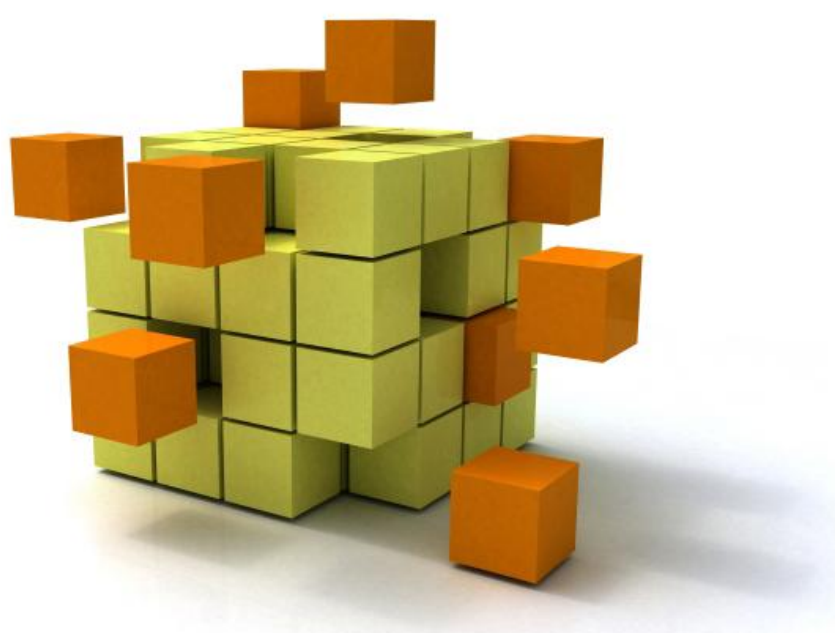


Quizz 6 (suivi asynchrone du cours):

Trouvez une correspondance pour chaque variable.

self.__A	Variable locale
self.Z	Attribut public
self.R__	Variable locale
__R	Attribut public
P_self	Attribut privé



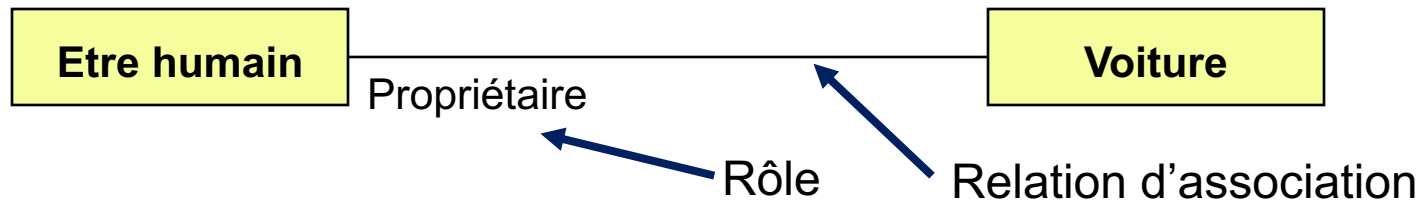


#3- Association et composition

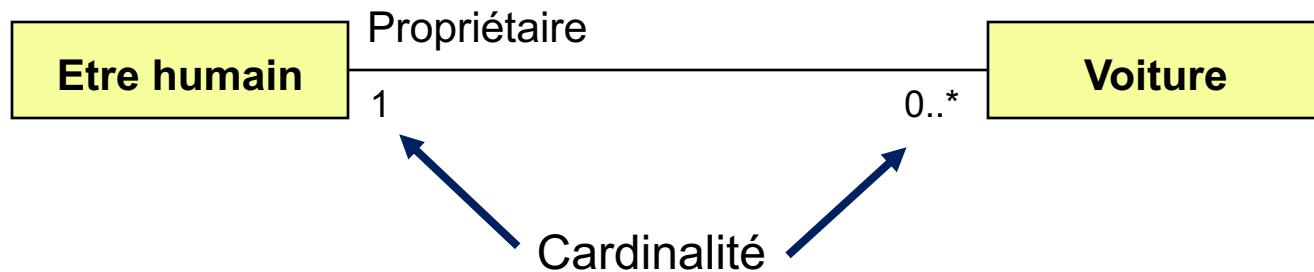
Association:

- **Association :**

- relation signifiant que les instances de classes ont certaines liaisons entre elles (lien sémantique)



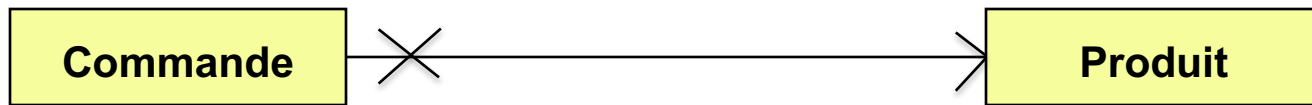
- la cardinalité de la relation exprime le nombre d'instances pouvant être associées ; exemple :
 - un "être humain" peut ne pas avoir de voiture ou être propriétaire d'une ou plusieurs "voitures" (cardinalité 0..*)
 - une "voiture" est liée à un propriétaire et un seul (cardinalité 1)



Association:

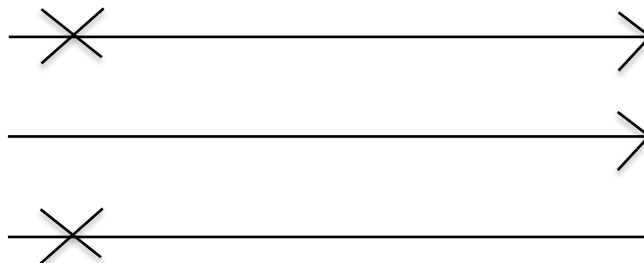
- **Association : navigabilité**

- Par défaut, une association est bidirectionnelle



Chaque commande contient une liste de produits.

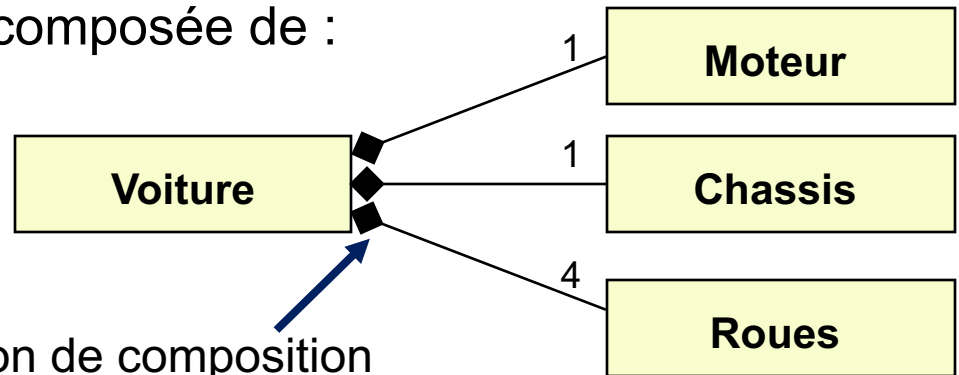
- La navigabilité contraint le parcours du graphe de classes.
- Lorsque l'association est contrainte pour devenir unidirectionnelle, le sens de navigation qui reste possible est spécifié par une flèche.
- Pour être encore plus explicite, UML autorise d'alerter sur le sens de navigation interdit en dessinant une croix, en plus de la flèche.



Association:

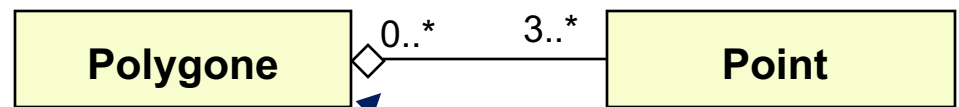
- **Composition :**

- relation entre classes signifiant que les instances d'une classe sont les composants d'une autre classe
- la composition permet d'assembler des objets pour en créer de plus complexes
- exemple : une voiture est composée de :
 - 1 moteur,
 - 1 châssis
 - et 4 roues

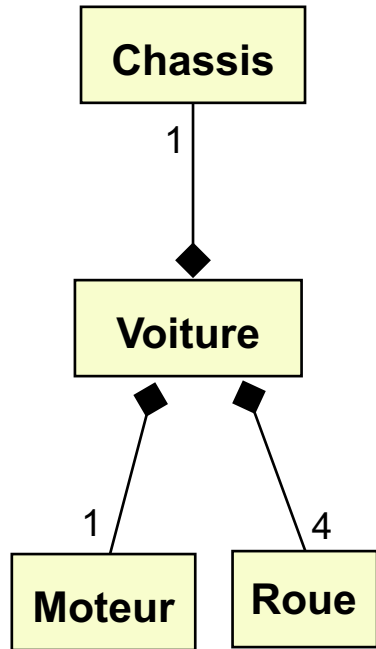


- **Agrégation :**

- composition faible : les objets agrégés ont une durée de vie indépendante de l'agrégat
- exemple :



Association: composition



```
class Chassis:
    def __init__(self, ...):
        ...

class Moteur:
    def __init__(self, ...):
        ...

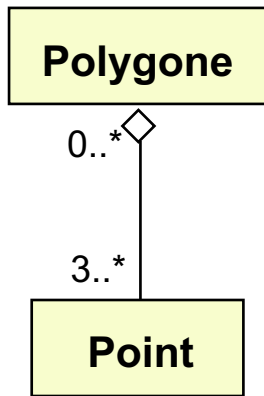
class Roue:
    def __init__(self, ...):
        ...

class Voiture:
    def __init__(self, ...):
        self.__M = Moteur(...)
        self.__C = Chassis(...)
        self.__lRoue = [Roue(...), \
                        Roue(...), Roue(...), Roue(...)]

if __name__ == '__main__':
    titine = Voiture(...)
```



Association: agrégation



```
class Point:
    def __init__(self, x, y):
        self.__x, self.__y = x, y

class Polygone:
    def __init__(self, lP):
        self.__lPoints = lP
        ...

if __name__ == '__main__':

    P1, P2 = Point(0,0), Point(1,1)
    P3, P4 = Point(1,0), Point(0,1)
    Toto = [P1, P2, P3]
    Pol1 = Polygone(Toto)
    Pol2 = Polygone([P1, P4, P3, P2])
```

Association

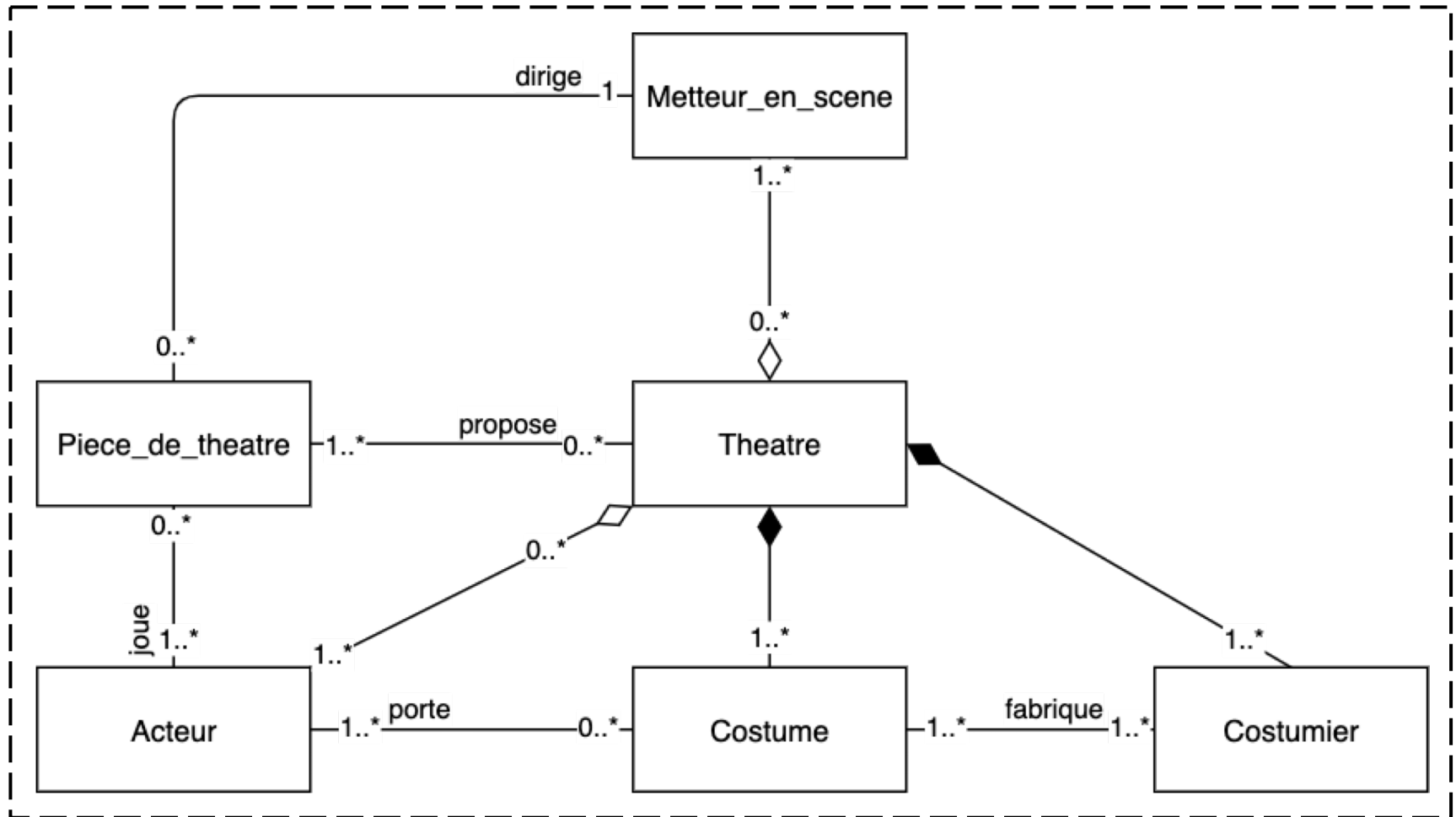


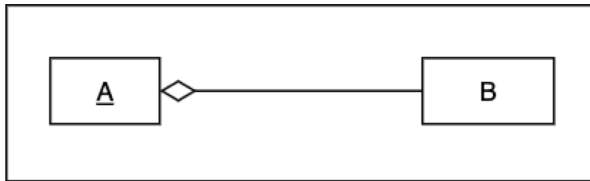
Diagramme réalisé avec l'appli draw.io



Quizz Wooclap

Quizz 7 (suivi asynchrone du cours):

Liez chaque diagramme (colonne de gauche) à sa description (colonne de droite)



Agrégation



Composition



Association



Quizz Wooclap

Quizz 8 (suivi asynchrone du cours):

Une compétition est organisée autours de 8 équipes.

Résultats possibles

- Agrégation
- Composition

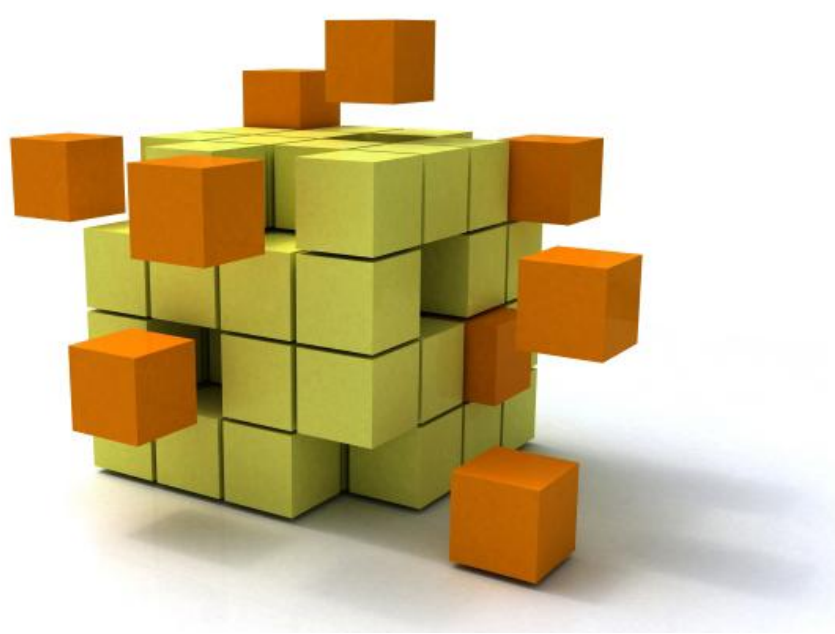
Quizz 9 (suivi asynchrone du cours):

Un projet informatique est constituée de développeurs.

Résultats possibles

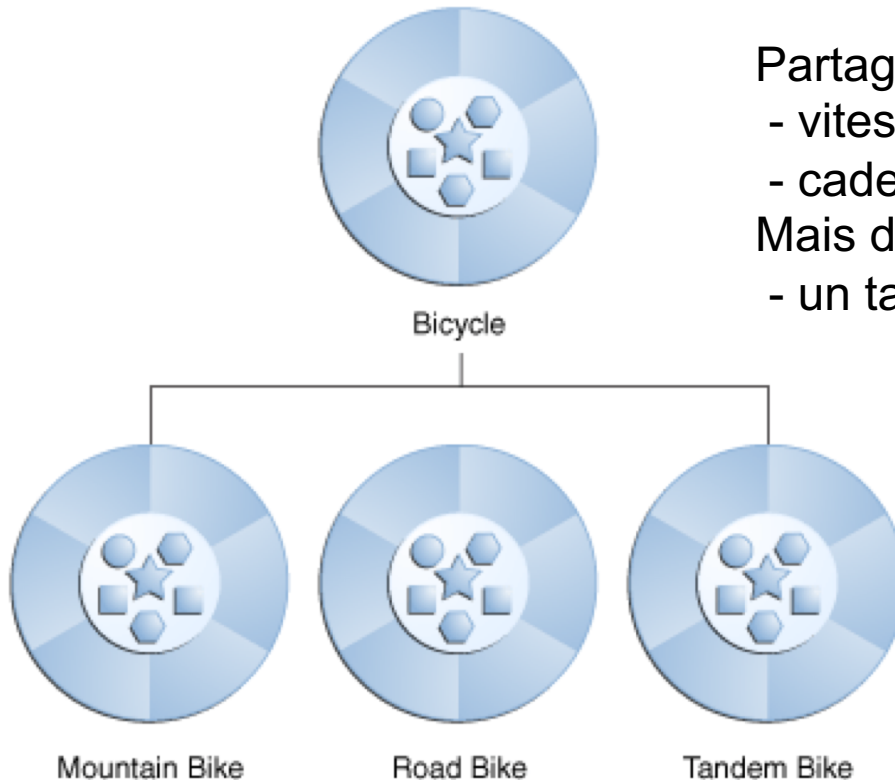
- Agrégation
- Composition





#4- L'héritage et le polymorphisme

Héritage



Partage de caractéristiques communes

- vitesse courant
- cadence des pédales...

Mais des différences aussi:

- un tandem à 2 selles...

```
class Bicycle:
```

```
...
```

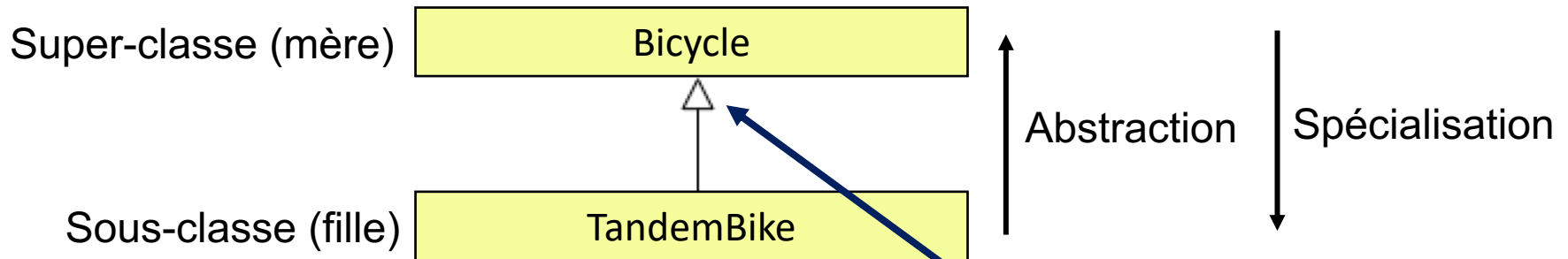
```
from Bicycle import Bicycle
```

```
class TandemBike(Bicycle):
```

```
...
```

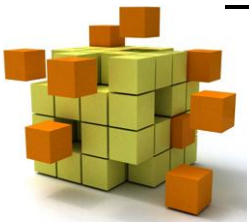
Héritage : représentation UML

- La généralisation exprime une relation « **une sorte de** » entre une classe et sa super-classe.

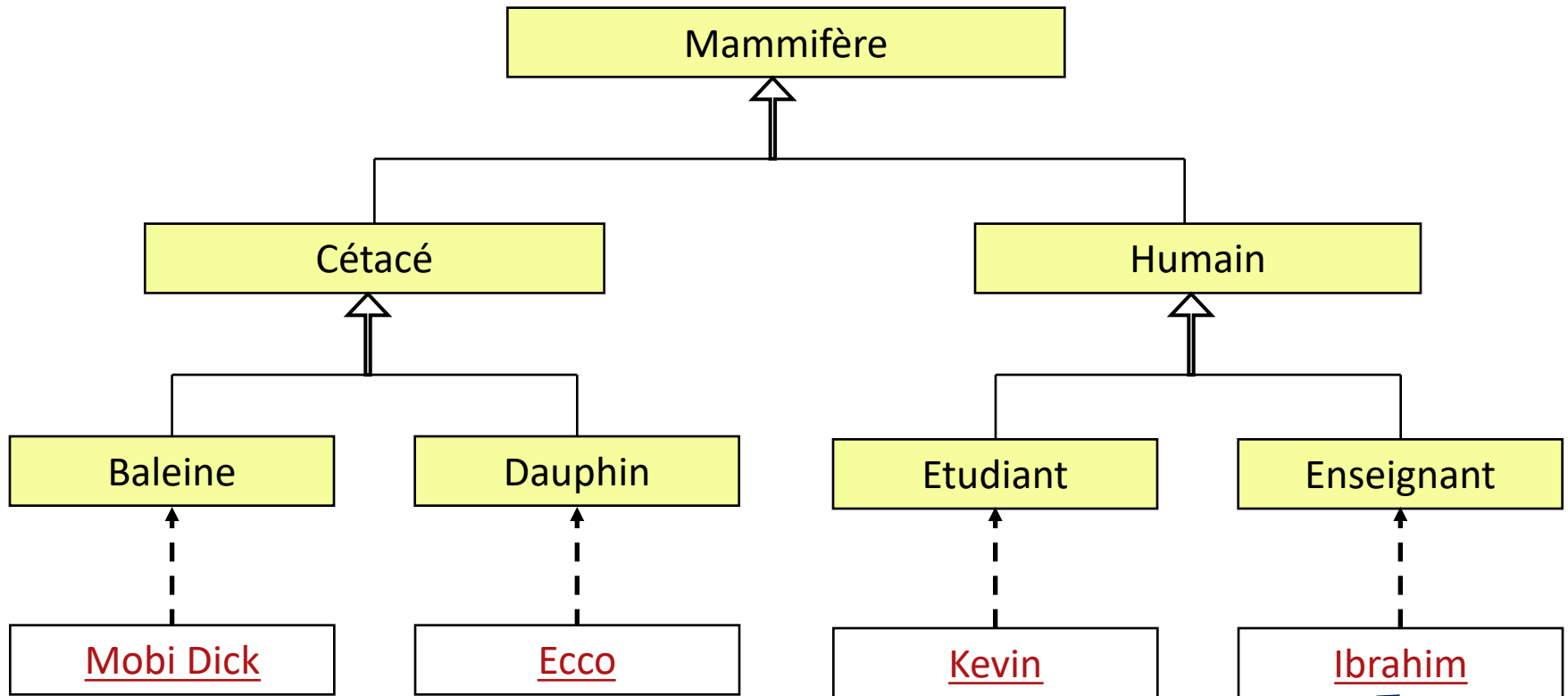


- L' héritage permet
 - de **généraliser** dans le sens abstraction
 - de **spécialiser** dans le sens raffinement

A noter!
représentation
UML de l'héritage



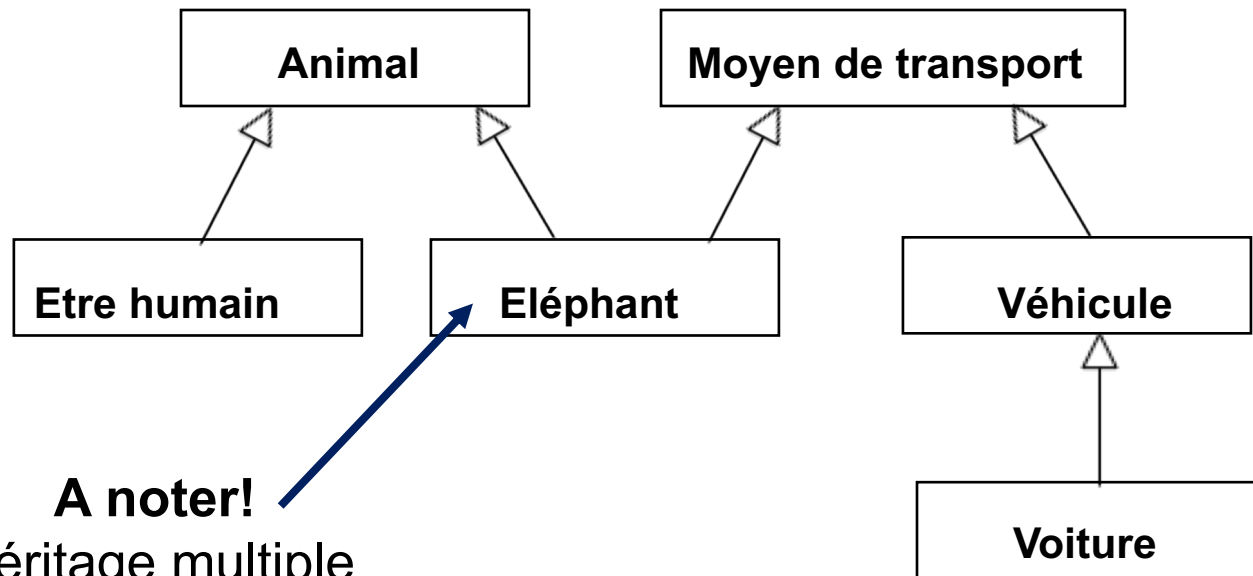
Héritage



A noter!
représentation
UML d'une
instance

Héritage

- Héritage (spécialisation et généralisation) :
 - transmission des propriétés d'une classe vers une sous-classe
 - la spécialisation permet d'ajouter des caractéristiques ou d'en adapter certaines.
 - la généralisation permet de factoriser des classes en regroupant des propriétés qui leur sont communes.
 - l'héritage évite la duplication et favorise la réutilisation.



Héritage

A noter!
représentation
UML d'un
attribut protégé

Compteur
cpt : entier
+ construteur() + inc() + affiche()

En Python
statut "protected"
(simple underscore)

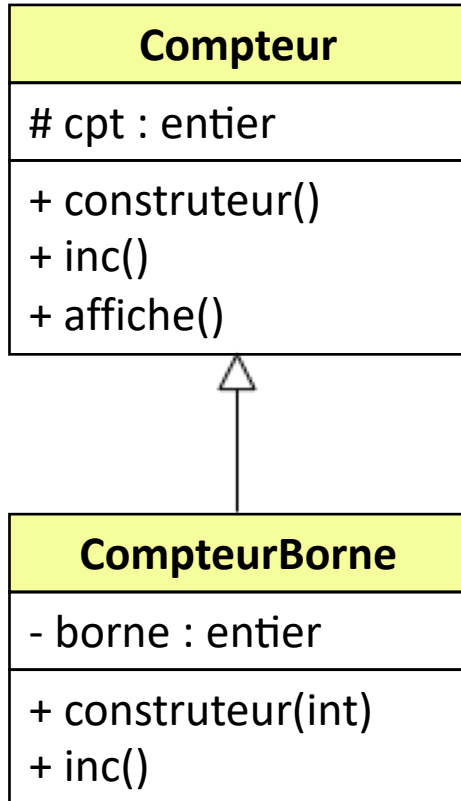
```
class Compteur:
    def __init__(self):
        self._cpt = 0
    def inc(self):
        self._cpt += 1
    def __str__(self):
        return 'Value = ' + str(self._cpt)

if __name__ == '__main__':
    chrono = Compteur()
    for i in range(10):
        chrono.inc()
    print(chrono)
```

Attribut *protected*: attribut public pour la classe et ses classes filles, mais privé par ailleurs.



Héritage



```
class Compteur:
    ...
class CompteurBorne(Compteur):
    def __init__(self, borne):
        Compteur.__init__(self)
        self.__borne = borne
    def inc(self):
        if self._cpt < self.__borne:
            Compteur.inc(self)

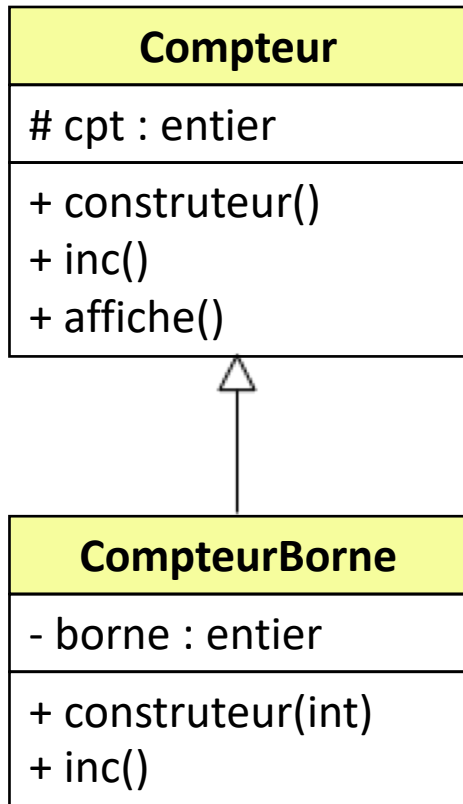
if __name__ == '__main__':
    chrono5 = CompteurBorne(5)
    for i in range(10):
        chrono5.inc()
    print(chrono5)
```

A noter!
Appel au
constructeur
de la classe
mère

A noter!
redéfinition



Héritage :



CompteurBorne.py

```
class Compteur:
    ...

class CompteurBorne(Compteur):
    ...

if __name__ == '__main__':
    chrono5 = CompteurBorne(5)
    print(isinstance(chrono5, CompteurBorne))
    print(isinstance(chrono5, Compteur))
    print(isinstance(chrono5, int))

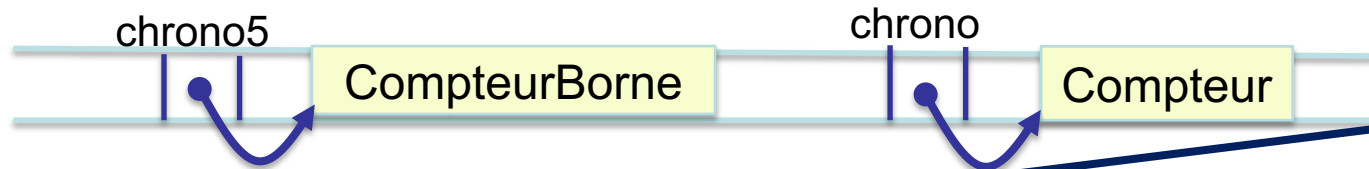
    print(issubclass(CompteurBorne, Compteur))
    print(issubclass(Compteur, str))
```

True, True, False, True, False



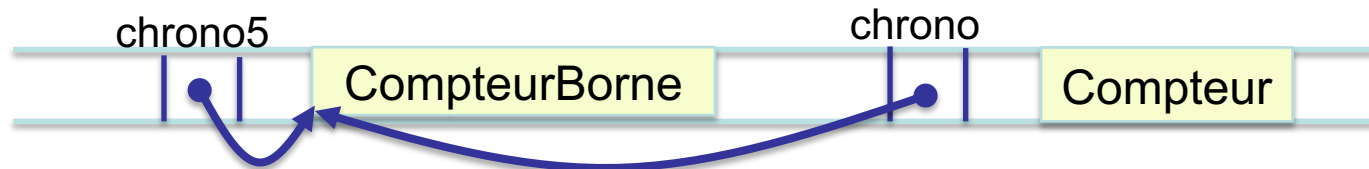
Héritage : polymorphisme et typage dyn.

```
from CompteurBorne import Compteur, CompteurBorne
if __name__ == '__main__':
    chrono5 = CompteurBorne(5)
    chrono = Compteur()
```

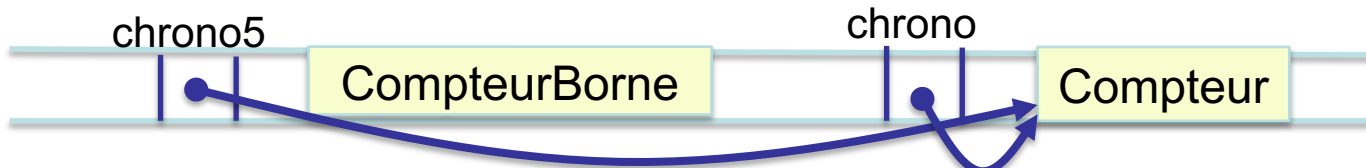


A noter!
Copie superficielle!

```
chrono = chrono5
print(isinstance(chrono, CompteurBorne)) # --> True
```



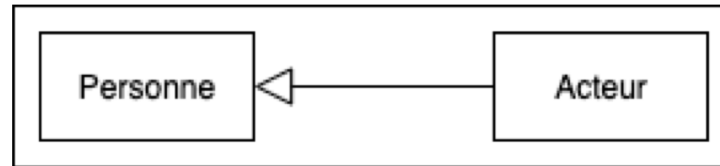
```
chrono5 = chrono
print(isinstance(chrono5, CompteurBorne)) # --> False
print(isinstance(chrono5, Compteur))     # --> True
```



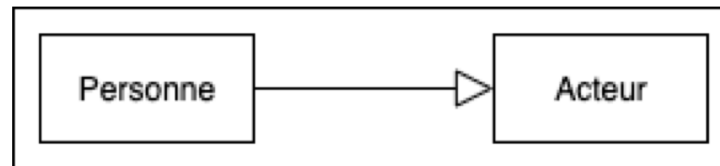
Quizz Wooclap

Quizz 10 (suivi asynchrone du cours):

Dans quel sens mettre la flèche?



OU



Quizz Wooclap

Quizz 11 (suivi asynchrone du cours):

```
import string
allowed = '123456789-'

class Personne:
    def __init__(self, nom):
        self.nom = nom
    def __str__(self):
        return "Personne : " + self.nom

class AgentSpecial(Personne):
    def __init__(self, nom, matricule):
        Personne.__init__(self, nom)
        self.__matricule = self.__check_naive(matricule)
    def __check_naive(self, mat):
        if all(c in allowed for c in mat) == True:
            return mat
        else:
            return '11111-111'
    def __str__(self):
        return "Agent : " + self.nom + ", Matricule : " \
            + self.__matricule
```



Quizz Wooclap

Quizz 11 suite (suivi asynchrone du cours):

```
if __name__ == '__main__':  
    agent = AgentSpecial("Fisher", "12345-788")
```

Question 1: que donne l'instruction `print(agent.nom)` ? Choix :

- 'Martin',
- 'Fisher',
- une erreur

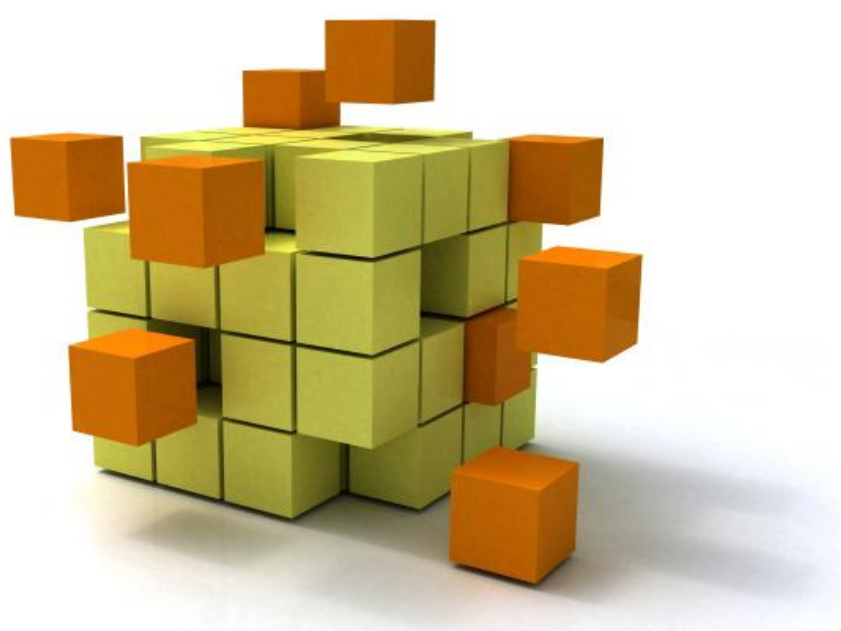
Question 2: que donne l'instruction `print(agent)` ? Choix :

- 'Agent Fisher, matricule 12345-788',
- 'Agent Fisher, matricule 11111-111',
- une erreur,
- '<__main__.AgentSpecial object at 0x7fe455b3af10>'

Question 2: que donne l'instruction `print(agent.matricule)` ? Choix :

- '12345-788',
- '11111-111',
- une erreur





#5- Les exceptions

Exceptions :

- Problème posé :
 - Des erreurs peuvent se produire au cours de l'exécution du programme (division par zéro, dépiler une pile vide, traitement d'une variable d'un type inapproprié, ...)
 - Sans gestion de ces erreurs, le programme s'arrêtera brutalement.
- Solution offerte par les exceptions :
 - Surveillance d'instructions susceptibles de provoquer des erreurs.
 - Possibilité d'"attraper" ces erreurs pour réaliser un traitement adapté, puis de poursuivre l'exécution du programme.
- Exception : objet qui indique que le programme ne peut pas continuer de s'exécuter normalement
 - La nature de l'objet indique le type d'erreur rencontré.
 - En python, les objets exception sont des instances de classes dérivées de la classe "Exception".



Exceptions :

Sans capture d'exception

```
def division(x, y):  
    return x / y  
  
if __name__ == '__main__':  
    v1 = division(4, 2)  
    print(v1)  
    v2 = division(5, 0)  
    print(v2)  
  
print("Le programme se poursuit")
```

2.0

Traceback (most recent call last):

File "Exception1.py", line 7, in <module>

v2 = division(5,0)

File "Exception1.py", line 2, in division

return x / y

ZeroDivisionError: division by zero



Exceptions :

Avec capture d'exception

```
def division(x,y):  
    return x / y  
  
if __name__ == '__main__':  
    try :  
        v1 = division(4,2)  
        print(v1)  
        v2 = division(5,0)  
        print(v2)  
    except :  
        print("Une erreur s'est produite")  
  
    print("Le programme se poursuit")
```

A noter!

Capture d'exception

Toutes exceptions
(indifférenciées)

2.0

Une erreur s'est produite

Le programme se poursuit



Exceptions :

- Exemples d'exceptions prédéfinies:
 - **AssertionError** : levée lorsqu'une évaluation avec *assert* échoue.
 - **IndexError** : levée lorsque l'indice utilisé pour accéder à un élément d'une séquence n'est pas correct.
 - **OverflowError** : levée lorsque le résultat d'une opération arithmétique est trop grand pour être représenté.
 - **TypeError** : levée lorsqu'un traitement est réalisé sur une donnée n'ayant pas le type approprié.
 - **ValueError** : levée lorsqu'un traitement est réalisé sur une donnée ayant le type approprié, mais pas la valeur.
 - **ZeroDivisionError** : levée lorsque le dénominateur d'une division est 0.
 - ...

- Liste complète :

https://www.tutorialspoint.com/python/standard_exceptions.htm



Exceptions :

```
def division(x, y):  
    return x / y  
  
if __name__ == '__main__':  
  
    try :  
        print(division('a', 2))  
    except TypeError :  
        print("Une exception TypeError a été levée")  
    except ZeroDivisionError :  
        print("Une exception ZeroDivisionError a été levée")  
    except:  
        print("Une autre exception a été levée")  
    else:  
        print("Aucune exception n'a été levée")
```

Une exception TypeError a été levée



Exceptions :

```
def division(x,y):  
    return x / y  
  
if __name__ == '__main__':  
  
    try :  
        print(division(5, 0))  
    except TypeError :  
        print("Une exception TypeError a été levée")  
    except ZeroDivisionError :  
        print("Une exception ZeroDivisionError a été levée")  
    except:  
        print("Une autre exception a été levée")  
    else:  
        print("Aucune exception n'a été levée")
```

Une exception ZeroDivisionError a été levée



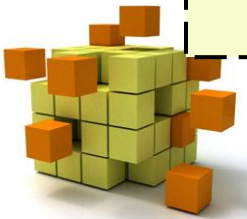
Exceptions : raise

Il est possible de lever une exception avec "raise"

```
def division(x, y):  
    if y == 0:  
        raise ValueError  
    return x / y  
  
if __name__ == '__main__':  
  
    try :  
        print(division(2, 0))  
    except ZeroDivisionError :  
        print("Une exception ZeroDivisionError a été levée")  
    except ValueError :  
        print("Une exception ValueError a été levée")  
    except:  
        print("Une autre exception a été levée")  
    else:  
        print("Aucune exception n'a été levée")
```

A noter!
Lancement
d'une exception

Une exception ValueError a été levée



Exceptions : raise

Associer une information à un déclenchement d'exception

```
def division(x, y):  
    if y == 0:  
        raise ValueError("Problème de division par zero")  
    return x / y  
  
if __name__ == '__main__':  
    try :  
        print(division(4, 0))  
    except ValueError as e:  
        print("Erreur : ", e)
```

Erreur : Problème de division par zero



Exceptions : AssertionError

Utilisation de l'exception standard **AssertionError**

```
# Roots of a quadratic equation
import math
def ShridharAcharya(a, b, c):
    try:
        assert a != 0, "Not a quadratic eq"
        D = (b * b - 4 * a*c)
        assert D >= 0, "Roots are imaginary"
        r1 = (-b + math.sqrt(D))/(2 * a)
        r2 = (-b - math.sqrt(D))/(2 * a)
        print("Roots of the quadratic eq are :", r1, "", r2)
    except AssertionError as msg:
        print(msg)

if __name__ == '__main__':
    ShridharAcharya(-1, 5, -6)
    ShridharAcharya(1, 1, 6)
    ShridharAcharya(0, 12, 18)
```

Roots of the quadratic eq are : 2.0 3.0
Roots are imaginary
Not a quadratic eq



Exceptions : créer sa propre exception

```
class MonException(Exception):
    def __init__(self, num, den, f):
        Exception.__init__(self)
        self.__n, self.__d, self.__f = num, den, f
    def __str__(self):
        return "PB fonction {}: {}/"
            {}.format(self.__f, self.__n, self.__d)

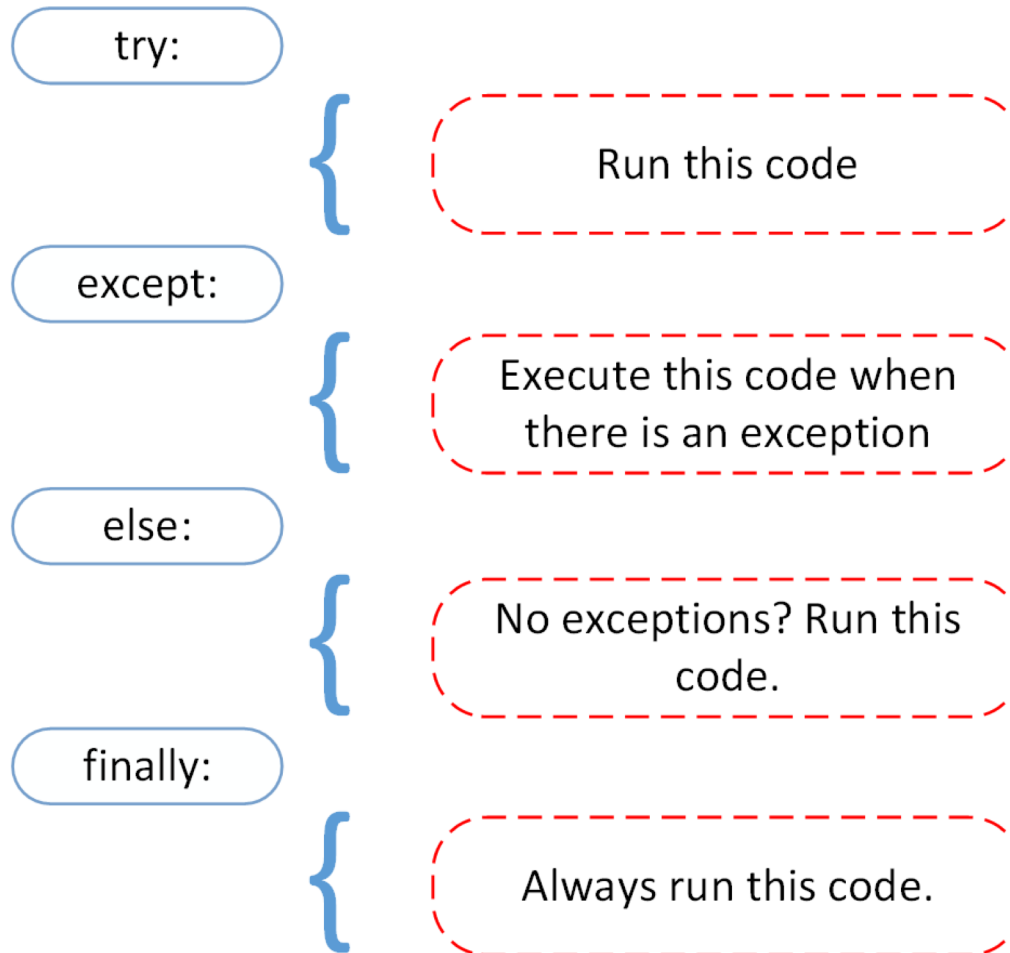
def division(x, y):
    if y == 0:
        raise MonException(x, y, division.__name__)
    return x / y

if __name__ == '__main__':
    try :
        print(division(4, 0))
    except MonException as e:
        print("Erreur :", e)
```

Erreur : PB fonction division: 4 / 0!



Exceptions : forme générale



Quizz Wooclap

Quizz 13 (suivi asynchrone du cours):

```
if __name__ == '__main__':  
    try:  
        a = int(input('Entrez un entier > 0 : '))  
        if a <= 0:  
            raise ValueError("Entier strictement positif!")  
        inv_a = 1./a  
    except TypeError as t:  
        print('TypeError : ', t)  
    except ValueError as v:  
        print('ValueError : ', v)  
    except ZeroDivisionError as z:  
        print('ZeroDivisionError : ', z)  
    else:  
        print("inv_a = ", inv_a, ", tout s'est bien passé!")
```



Quizz Wooclap

Quizz 12 suite (suivi asynchrone du cours):

Quelle exception est lancée pour les entrées suivantes ?

Choix possibles :

- ValueError : Entier strictement positif,
- ValueError : invalid literal for int() with base 10
- ZeroDivisionError : ...
- rien.

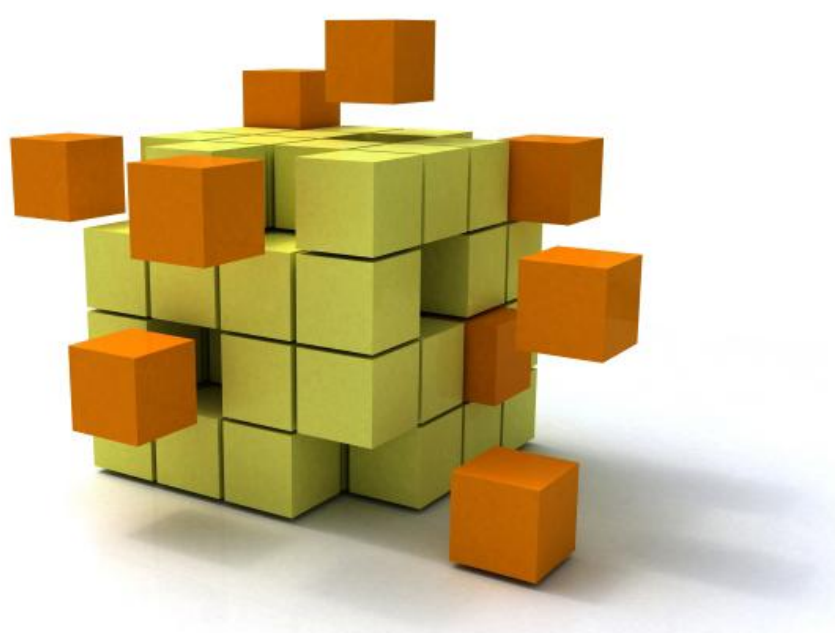
Question 1: si l'entrée est « -3 »

Question 2: si l'entrée est « 0 »

Question 3: si l'entrée est « toto »

Question 4: si l'entrée est « 3.5 »





#6- Autres points

1. Surcharge des opérateurs
2. Attribut de classe

Autres : surcharge des opérateurs

Problème : On considère une classe **Temps** qui représente un horaire (h/m/s). Comment peut-on additionner deux objets **Temps** T1 et T2 ?

- Soit $T3 = T1.\text{ajout}(T2)$ (ou bien $T3 = T2.\text{ajout}(T1)$)
- Soit $T3 = T1 + T2$ (ou bien $T3 = T2 + T1$)

Surcharge de l'opérateur
'+' pour la classe Temps

Temps
+ h : entier + m : entier + s : entier
+ constructeur() + ajout(Temps) : Temps + affiche()

Liste complete :

<https://riptutorial.com/fr/python/example/7334/surcharge-de-l-operateur>



Autres : surcharge des opérateurs

```
class Temps:
    def __init__(self, h=0, m=0, s=0):
        self.h, self.m, self.s = h, m, s

    def __str__(self):
        return str(self.h)+'h' \
            + str(self.m)+'m' + str(self.s)+'s'

    def __add__(self, other):
        if not isinstance(other, Temps):
            print("Erreur: l'arg. n'est pas un Temps")
            return NotImplemented
        s = (self.s+other.s)%60
        m = (self.m+other.m+(self.s+other.s)//60)%60
        h = self.h+other.h+(self.m+other.m)//60
        res = Temps(h, m, s)
        return res

if __name__ == '__main__':
    t1,t2 = Temps(1,24,48), Temps(0,52,32)
    print('t1=', t1, 't2=', t2, 't1+t2=', t1+t2)
```



Autres : surcharge des opérateurs

Addition à droite

```
class Temps:
    ...

    def __radd__(self, h_int):
        if not isinstance(other, int):
            print("Erreur: l'argument n'est pas un entier")
            return NotImplemented
        s = self.s
        m = self.m
        h = self.h + h_int
        return Temps(h, m, s)

if __name__ == '__main__':
    t1 = Temps(1, 24, 48)
    print('1+t1=', 1+t1)
    print('1.5+t1=', 1.5+t1) # Not supported
```



Autres : surcharge des opérateurs

Ajout *inplace*

```
class Temps:
    ...

    def __iadd__(self, other):
        if not isinstance(other, Temps):
            print("Erreur: l'argument n'est pas un Temps")
            return NotImplemented
        self.h = self.h+other.h + (self.m+other.m)//60
        self.m = (self.m+other.m + (self.s+other.s)//60)%60
        self.s = (self.s+other.s)%60
        return self

if __name__ == '__main__':
    t1, t2 = Temps(1,24,48), Temps(0,52,32)
    t1+=t2
    print('t1=', t1)
```



Autres : surcharge des opérateurs

Surcharge de méthodes classiques

```
import math
class Temps:
    ...
    def __floor__(self):
        return Temps(self.h, 0, 0)
    def __ceil__(self):
        return Temps(self.h+1, 0, 0)
    def __int__(self):
        return self.s+(self.m+self.h*60)*60

if __name__ == '__main__':
    T1 = Temps(1,24,48)
    print('floor(t1)=', math.floor(t1))
    print('ceil(t1)=', math.ceil(t1))
    print('int(t1)=', int(t1))
```

1h00m0s

2h00m0s

8240

Liste complete :

<https://riptutorial.com/fr/python/example/7334/surcharge-de-l-operateur>



Autres : surcharge des opérateurs

Exemple : opérateur ==

```
class Temps:
    ...

    def __eq__(self, other):
        if not isinstance(other, Temps):
            print("Erreur: l'argument n'est pas un Temps")
            return NotImplemented
        return (self.s == other.s) and (self.m == other.m)
            and (self.h == other.h)

if __name__ == '__main__':
    t1,t2 = Temps(1,24,48), Temps(0,52,32)
    print("t1 == t2 ? -->", t1 == t2)
```



Autres : surcharge des opérateurs

Opérateur	Méthode	Expression
+ Ajout	<code>__add__(self, other)</code>	<code>a1 + a2</code>
- soustraction	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
* Multiplication	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
/ Division	<code>__truediv__(self, other)</code>	<code>a1 / a2</code> (Python 3)
// Division de plancher	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
% Modulo / reste	<code>__mod__(self, other)</code>	<code>a1 % a2</code>
** puissance	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
< Moins que	<code>__lt__(self, other)</code>	<code>a1 < a2</code>
<= Inférieur ou égal à	<code>__le__(self, other)</code>	<code>a1 <= a2</code>
== égal à	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!= Pas égal à	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
> Supérieur à	<code>__gt__(self, other)</code>	<code>a1 > a2</code>
>= Supérieur ou égal à	<code>__ge__(self, other)</code>	<code>a1 >= a2</code>
[index] Opérateur d'index	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
in opérateur In	<code>__contains__(self, other)</code>	<code>a2 in a1</code>



Liste complete :

<https://riptutorial.com/fr/python/example/7334/surcharge-de-l-operateur>

Autres : attribut de classe

Attributs d'instance, attributs de classe

- **attributs d'instance** dupliqués dans les instances d'une classe
- **attributs de classe** (déclarés en dehors du constructeur)
communs à toutes les instances d'une classe

Cercle
- ray : réel - x, y : réels + cptCercles : entier
+ constructeur() + affiche()

```
class Cercle:
    cptCercles = 0

    def __init__(self, r, x, y):
        self.__ray = r
        self.pos = x,y # tuple
        Cercle.cptCercles+=1

if __name__ == '__main__':
    lCercle = []
    for i in range(1,6):
        lCercle.append(Cercle(i, 0, 0))
    print("Nb cercles : ", Cercle.cptCercles)
    print("Nb cercles : ", lCercle[3].cptCercles)

    del lCercle[2]
    print("Longueur liste : ", len(lCercle))
    print("Nb cercles : ", Cercle.cptCercles)
```

A noter!
Variable de classe!

5
5
4
5!



Autres : attribut de classe

```
class Cercle:
    cptCercles = 0

    def __init__(self, r, x, y):
        self.__ray = r
        self.pos = x,y
        Cercle.cptCercles+=1

    def __del__(self):
        Cercle.cptCercles-=1

if __name__ == '__main__':
    listeCercle = []
    for i in range(1,6):
        listeCercle.append(Cercle(i, 0, 0))
    del listeCercle[2]
    print("Longueur liste : ", len(listeCercle))
    print("Nb cercles : ", Cercle.cptCercles)
```

A noter!

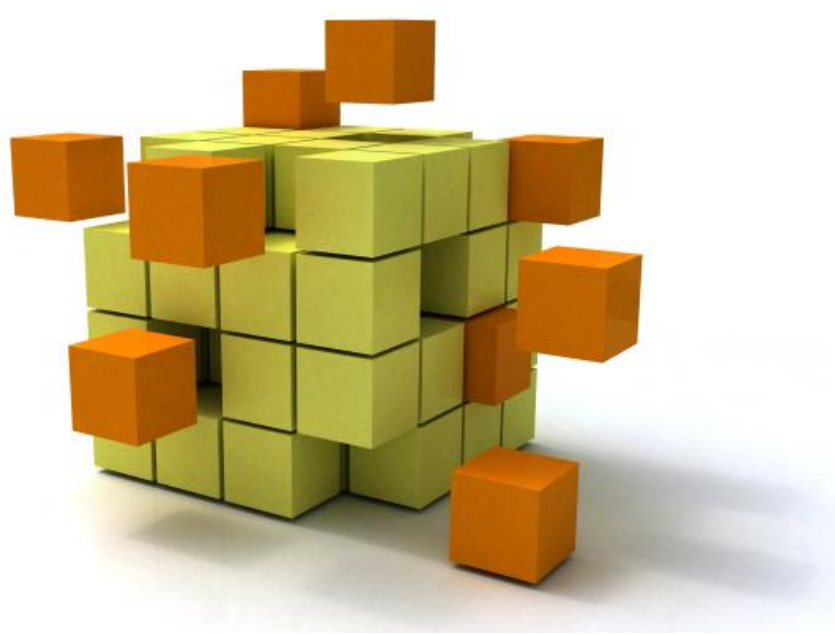
Méthode appelée
au moment de la
destruction d'un
objet

→ 4

→ 4

Ramasse-miette (*garbage collector*): un objet est supprimé de la mémoire dès lors que plus aucune variable ne le référence.





#7- Les tests unitaires

Une petite introduction...

Tests unitaires

- Les **tests unitaires** s'inscrivent dans la philosophie de la [programmation agile](#) (cycle de développement incrémental, court et adaptatif : les *sprints*) et du *Test-Driven Development (TDD)*.
- Il s'agit de vérifier le bon fonctionnement de son programme à l'aide d'un jeu de tests que l'on lance régulièrement.
- Cela permet de vérifier la stabilité d'un programme, notamment après l'ajout de nouvelles fonctionnalités ou des améliorations (qui peuvent introduire des comportements non attendus).
- On prévoit un test par fonctionnalité ou par résultat attendu d'une fonction (ou d'une méthode). Une fonction est donc l'objet de plusieurs tests.

Utilisation de la librairie **pytest** pour mettre en oeuvre les tests unitaires sous python (d'autres packages sont possibles : unittest, doctest...)



Tests unitaires : package *pytest*

Voici la structure que vous rencontrerez le plus souvent :

- Une fonctionnalité est codée grâce à un ensemble de fonctions, de classes, de modules et autres.
- Pour chaque fonctionnalité, un test vérifie que la fonctionnalité fait bien ce qu'on lui demande. Par ex., si une certaine fonction est appelée avec certains paramètres, alors elle retourne telle valeur.

Exemple trivial :

test_1.py

```
def incremente_de_1(x):  
    return x + 2  
  
def test_answer():  
    assert incremente_de_1(3) == 4  
  
def test_answer1():  
    assert incremente_de_1(1) == 2
```

Fonctionnalité à tester
(introduction d'un bug évident)

Un test

Un autre test



Tests unitaires : package *unittest*

- Exemple trivial

```
test_1.py  
  
def incremente_de_1(x):  
    return x + 2  
  
def test_answer():  
    assert incremente_de_1(3) == 4
```

A noter!

La commande *Assert* ne fait rien si le test qui suit est vrai, sinon elle lance une exception. Si celle-ci n'est pas attrapée, alors le prg est stoppé..

Règles :

- Tous les fichiers contenant des tests doivent s'appeler *test_*.py*
- Tous les tests (c'est-à-dire les fonctions de test) doivent s'appeler *test_**() ou **_test()*

Exécution des tests: On lance pytest

- Sur un test: **pytest test_1.py::test_answer**,
- sur un fichier : **pytest test_1.py**,
- sur un répertoire: **pytest ./code**.



Tests unitaires : package *unittest*

- Exemple trivial : `pytest test_1.py`

```
def incremente_de_1(x):  
    return x + 2  
  
def test_answer():  
    assert incremente_de_1(3) == 4
```

```
===== FAILURES =====
```

```
_____ test_answer _____
```

```
def test_answer():  
> assert incremente_de_1(3) == 4  
E      assert 5 == 4  
E      + where 5 = incremente_de_1(3)
```

```
test_1.py:5: AssertionError
```

```
===== short test summary info =====
```

```
FAILED test_1.py::test_answer - assert 5 == 4  
===== 1 failed 0.03s =====
```



Tests unitaires : exceptions

```
import pytest
class MonException(Exception):
    def __init__(self, num, den, f):
        Exception.__init__(self)
        self.__n, self.__d, self.__f = num, den, f

def division(x, y):
    if y == 0:
        raise MonException(x, y, division.__name__)
    return x / y

def test_exception():
    with pytest.raises(MonException):
        division(3, 0)

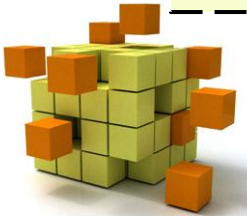
if __name__ == '__main__':
    print(division(4, 3))
```

A noter!

On doit importer
le package

A noter!

Test de l'exception
lancée en cas de
division par 0



Quizz Wooclap

Quizz 13 (suivi asynchrone du cours):

Est-ce que ce code passe le test ?

```
def hello(name):  
    return 'Hello ' + 'name'  
  
def test_hello():  
    assert hello('Celine') == 'Hello Celine'
```



Quizz Wooclap

Quizz 14 (suivi asynchrone du cours):

Etant donné le code suivant :

```
def minim(values):  
    _min = 0  
    for val in values:  
        if val < _min:  
            _min = val  
    return _min
```

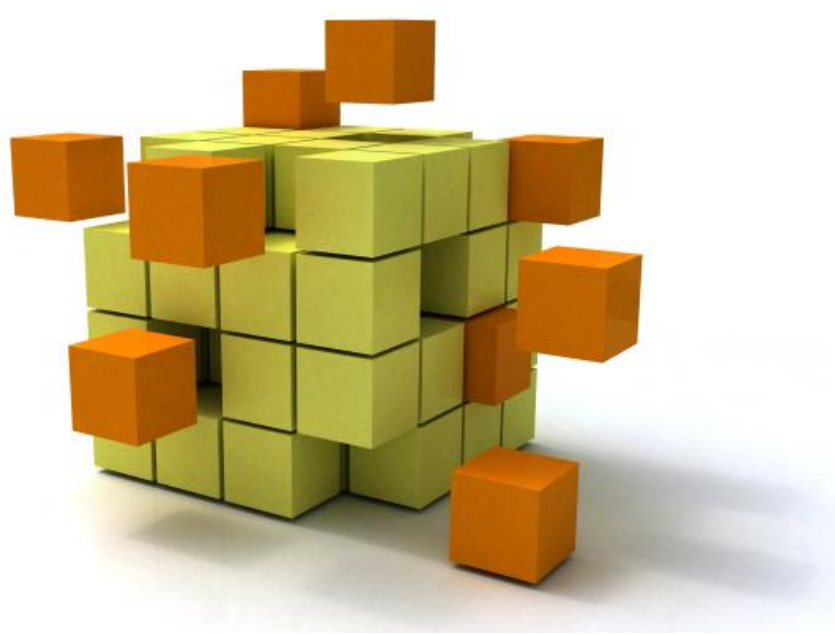
Est-ce que ce code passe le test ?

```
def test_min1():  
    values = (0, -3, 1, -1, 2)  
    assert minim(values) == -3
```

Et celui-ci ?

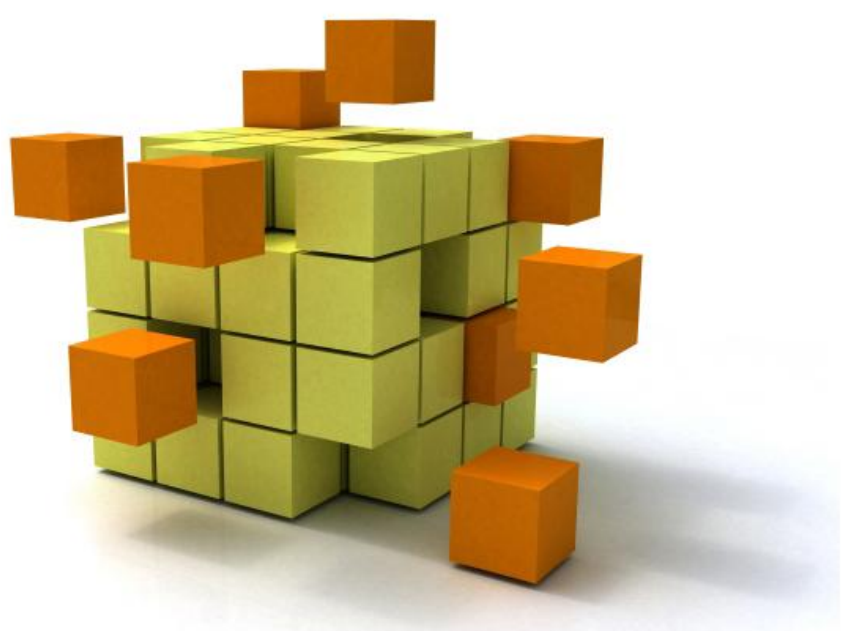
```
def test_min2():  
    values = (2, 3, 1, 4, 6)  
    assert minim(values) == 1
```





#8- Intro au génie logiciel

1. Génie Logiciel : la méthode en V
2. Méthode agile, avec SCRUM



#9- Versionning avec Git

- « Qui a modifié le fichier X, il marchait bien avant et maintenant il provoque des bugs ! » ;
- « Luigi, tu peux m'aider en travaillant sur le fichier X pendant que je travaille sur le fichier Y ? Attention à ne pas toucher au fichier Y car si on travaille dessus en même temps je risque d'écraser tes modifications ! » ;
- « Qui a ajouté cette ligne de code dans ce fichier ? Elle ne sert à rien ! » ;
- « À quoi servent ces nouveaux fichiers et qui les a ajoutés au code du projet ? » ;
- « Quelles modifications avons-nous faites pour résoudre le bug de la page qui se ferme toute seule ? »

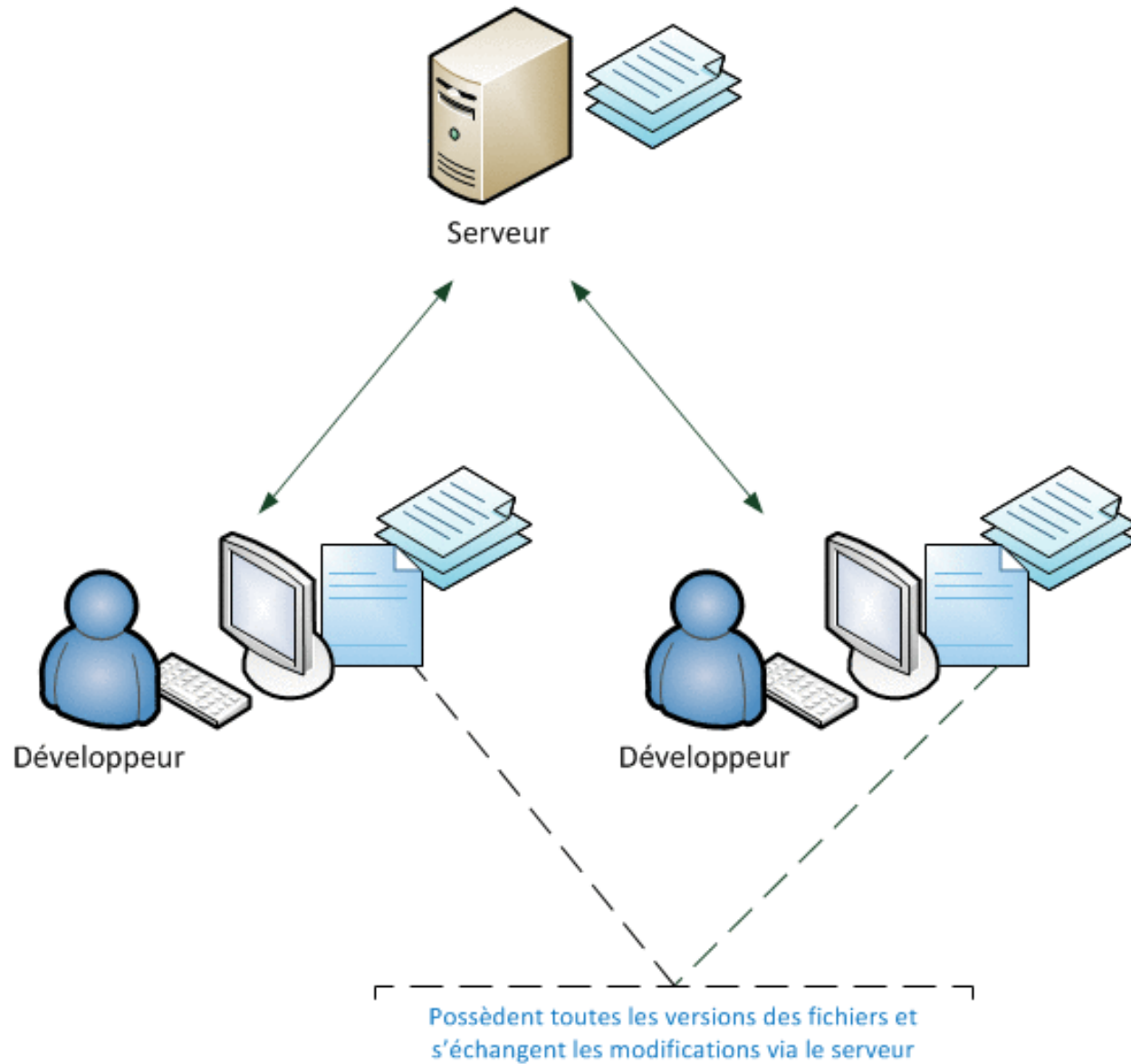


Git : versionning

- Devient indispensable pour un travail
 - **individuel (+)** : sauvegarde incrémentale et donc conservation d'un historique des modifications
 - **collaboratif (++)** : gérer automatiquement la fusion des sources dans un même projet. Sais à tout moment *qui* et *pourquoi* une modification a été faite.
- Logiciels de « gestion de versions » (*versioning*)
 - Cvs, Svn (subversion); mode centralisé
 - Mercurial, Bazaar, Git (prononcez « Guite »); mode distribué (pas de *daemon*)



Git : système distribué



Git : sites WEB

- Sites WEB collaboratifs basés sur Git:

- Github : <https://github.com>
- GitLab: <https://gitlab.com/explore>
- BitBucket : [https://bitbucket.org /](https://bitbucket.org/)

Sorte de réseaux sociaux pour développeurs : on peut regarder les projets évoluer et participer à leur développement. On peut y créer son propre projet, libre ou privé (gratuit ou payant).

- Nous utiliserons une version de gitlab installée en local sur les serveurs de la DSI de Centrale: <https://gitlab.ec-lyon.fr>

- Git est installé en natif sur Linux et Mac, il faut l'installer soi-même pour Windows: <https://git-scm.com/download/win>
- Ce logiciel se pilote par des instructions en ligne de commandes (par le Terminal) mais il existe de nombreux soft pour faciliter la tâche (<https://git-scm.com/downloads/guis>). De notre côté, nous utiliserons <https://desktop.github.com>.



Git : tuto gitlab

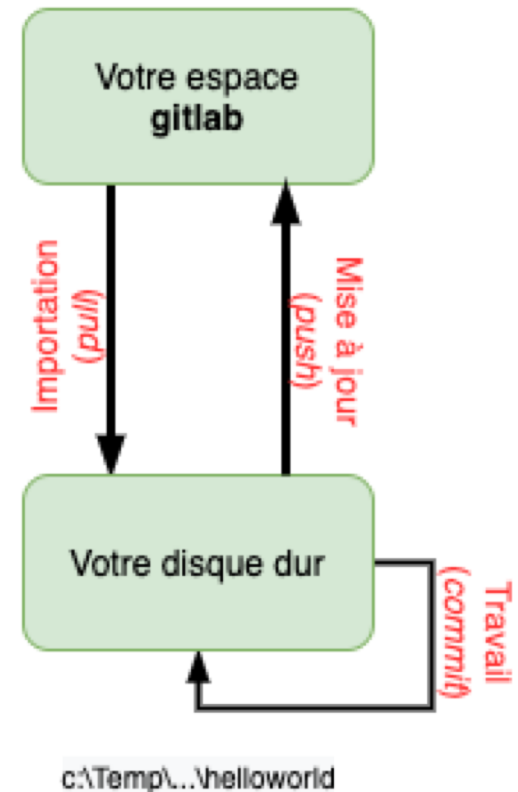
Objectif du tuto :

- Savoir travailler en mode standalone
- Créer son propre projet git / gitlab
- Savoir importer et publier ses modifications
- Savoir détruire un projet
- Scénario du tuto disponible à l'adresse <https://gitlab.ec-lyon.fr/sderrode/INF-TC2>

Liens utilisés dans le tuto :

- Github Desktop: <https://desktop.github.com/>
- Gitlab de l'École Centrale : <https://gitlab.ec-lyon.fr>
- Lien pour le fichier `.gitignore` : <https://github.com/github/gitignore/blob/master/Python.gitignore>

<https://gitlab.ec-lyon.fr/xyyyyy/helloworld>





Fin du cours

UE Informatique – Inf-TC2

