



Qu'est-ce que la
programmation objet ?





La programmation orienté objet (OOP)

- Un principe de programmation. On regroupe les données et les fonctions de façon à ce qu'il soit facile à réutiliser et à modifier selon les besoins.
- On crée des Classes qui sont des schémas pour créer des objets.
- Dans les classes on écrit les propriétés qui vont décrire les objets futurs et les méthodes qui décrivent les actions que pourront faire ces objets.
- La OOP permet plus facilement à l'être humain de conceptualiser les programmes et leurs utilisations.



- Les Classes sont des schémas qui nous permettrons ensuite de créer des objets.
- Elles ont des propriétés et des méthodes
- Cette chaise aurait plusieurs propriétés dont:
 - "largeur", "profondeur", "hauteur"
- Elle aurait aussi des méthodes, par ex :
 - "ajuster_hauteur()"



Classes



```
class Chaise:
    def __init__(self, largeur, profondeur, hauteur):
        self.largeur = largeur
        self.profondueur = profondeur
        self.hauteur = hauteur
    def ajuster_hauteur(self, nouvelle_hauteur):
        self.hauteur = nouvelle_hauteur
```



Le mot-clef **class** signifie que nous allons faire une classe (nécessaire)

Les noms des classes commencent toutes par une majuscule par convention

La méthode "magic" ou dunder **__init__** est toujours appelée lorsqu'on crée un nouvel objet. Il s'agit du constructeur de la **classe**

```
class Chaise:  
    ...def __init__(self, largeur, profondeur, hauteur):  
    ...    self.largeur = largeur  
    ...    self.profondueur = profondeur  
    ...    self.hauteur = hauteur  
    ...  
    ...def ajuster_hauteur(self, nouvelle_hauteur):  
    ...    self.hauteur = nouvelle_hauteur
```

Ici, toutes nos méthodes commencent par **self**, qui référence l'objet qui est créé.





Instanciation d'objets à partir d'une classe

- › On créer de nouveaux objets à partir d'une **classe** en appelant la **classe**, et en lui donnant les valeurs dont son constructeur a besoin.

```
chaise_1 = Chaise(40, 40, 110)
chaise_2 = Chaise(40, 40, 110)

chaise_2.ajuster_hauteur(80)
```

(largeur, profondeur, hauteur) -> None

- › Chaque objet créé ainsi est appelé une **instance** de la **classe**.



Utiliser les propriétés et méthodes d'un objet

```
chaise_1 = Chaise(40, 40, 110)
chaise_2 = Chaise(40, 40, 110)

print("hauteur 1: ", chaise_1.hauteur)
print("hauteur 2: ", chaise_2.hauteur)

chaise_2.ajuster_hauteur(80)

print("hauteur 1: ", chaise_1.hauteur)
print("hauteur 2: ", chaise_2.hauteur)
```

PROBLÈMES	SORTIE	CONSOLE DE DÉBOGAGE	<u>TERMINAL</u>
	hauteur 1: 110		
	hauteur 2: 110		
	hauteur 1: 110		
	hauteur 2: 80		



Création d'objets avec des valeurs par défauts

```
class Chaise:
    ...def __init__(self, largeur = 40, profondeur = 40, hauteur = 110):
    ...    ...self.largeur = largeur
    ...    ...self.profondueur = profondeur
    ...    ...self.hauteur = hauteur
```

```
chaise_3 = Chaise()
chaise_4 = Chaise(35, 45, 95)

print("chaise 3 :")
print(chaise_3.largeur, chaise_3.profondueur, chaise_3.hauteur)

print("chaise 4 :")
print(chaise_4.largeur, chaise_4.profondueur, chaise_4.hauteur)
```

PROBLÈMES	SORTIE	<u>TERMINAL</u>
	chaise 3 :	
	40 40 110	
	chaise 4 :	
	35 45 95	

Modélisation UML



- > Le "Unified Modeling Language"(UML) est utilisé pour faire les schémas des classes.





Variables de Classes et héritage

Pseudo-code,
héritage de méthodes,
et méthodes de classes



Variables de classes

- Ici une classe Employe possédant uniquement trois attributs, "nom", "prenom" et "ID"

```
class Employe:....  
    def __init__(self,nom,prenom) -> None:  
        self.nom = nom  
        self.prenom = prenom  
        self.ID = random.randint(1000,9999)
```

Variables de classes



```
class Employe:
    ....liste_employe = []
    ....next_ID = 1000
    ....
    ....def __init__(self,nom,prenom) -> None:
    ....    ....self.nom = nom
    ....    ....self.prenom = prenom
    ....    ....self.ID = Employe.next_ID
    ....
    ....    Employe.next_ID += 1
    ....    Employe.liste_employe.append(self)
```

Variables appartenant à la classe Employe

Variables appartenant à l'objet créé à partir de la classe Employe

Ici on modifie les variables de classes. Toutes les instantiations accéderont aux même variables de classe



Variables de classes

- > Une variable de classe est accessible directement à partir de la classe ou bien à partir d'une instantiation.
- > Il s'agit de la même variable ayant la même valeur.

```
employe1 = Employe("Anna", "Tremblay")  
employe2 = Employe("Bartholemy", "Duchamp")
```

```
for emp in Employe.liste_employe:·  
    ····print(f'{emp.prenom} {emp.nom} {emp.ID}')
```



```
for emp in employe1.liste_employe:·  
    ····print(f'{emp.prenom} {emp.nom} {emp.ID}')
```

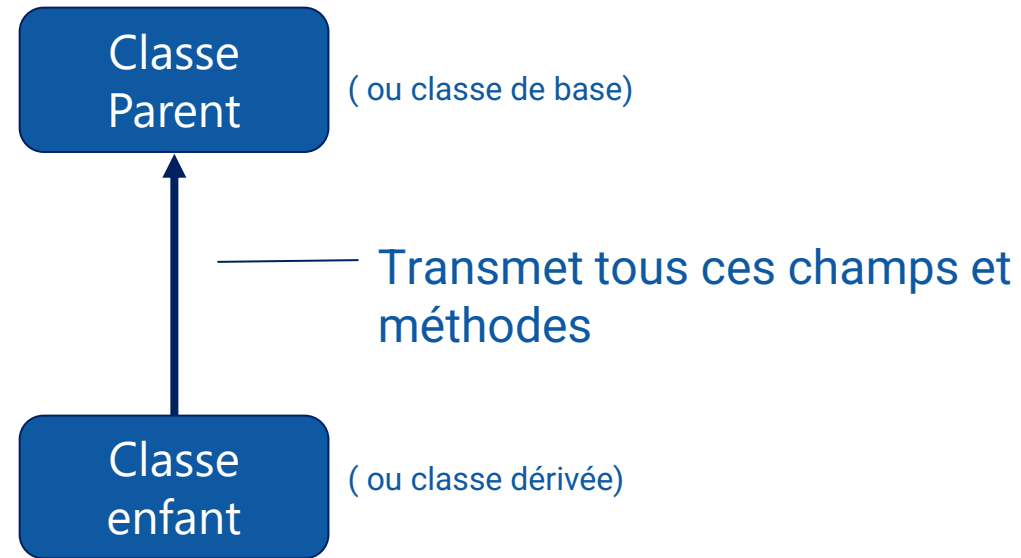
PROBLÈMES 1 SORTIE TERMINAL ...

```
Tremblay Anna 1000  
Duchamp Bartholemy 1001  
Tremblay Anna 1000  
Duchamp Bartholemy 1001
```

L'héritage des classes



- Permet de définir une classe à partir d'une classe déjà existante




- Permet d'hériter des champs et des méthodes de la classe parent.

Héritage



```
class Programmeur(Employe):  
    ...def __init__(self, nom, prenom, language_favori) -> None:  
    ...    super().__init__(nom, prenom)  
    ...    self.language_favori = language_favori
```

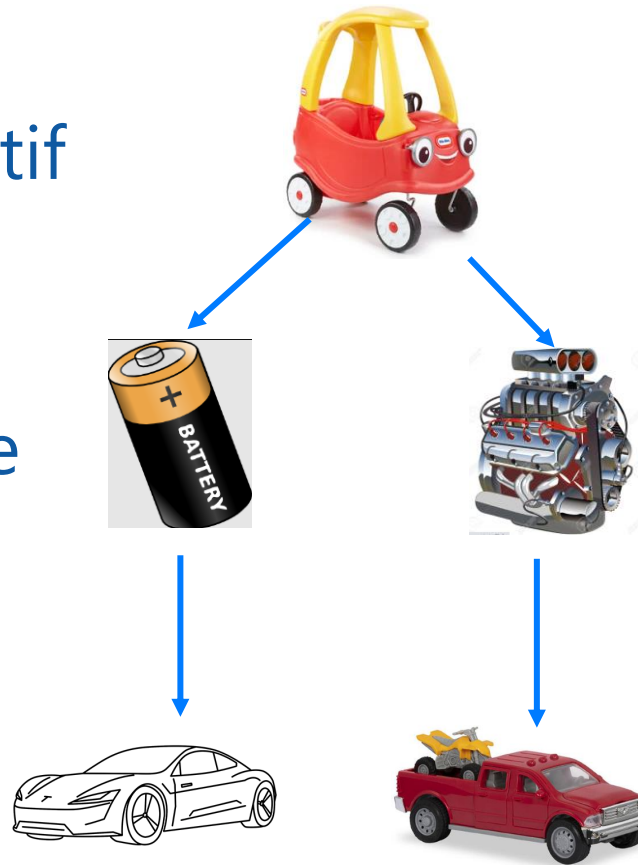


super() fait référence au parent
Employe. On utilise sa méthode `__init__()`
pour lui passer les valeurs de ses propriétés.

Héritage transitif



- > L'héritage est transitif en Python.
- > Une classe hérite de tous les attributs et méthodes de tous ses ancêtres.

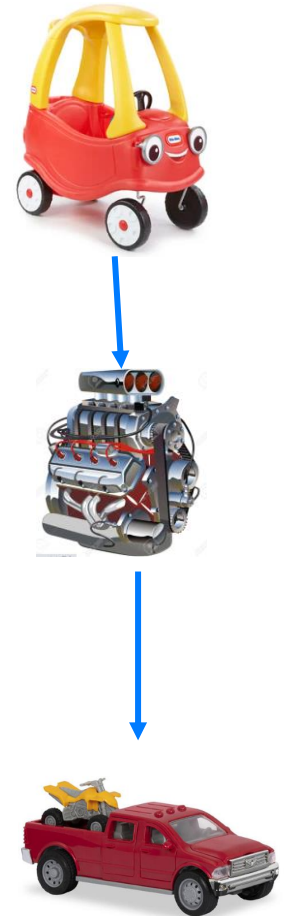


```
class Voiture:  
    ...def __init__(self,marque) -> None:  
    ...self.marque = marque
```

```
class Voiture_moteur(Voiture):  
    ...def __init__(self, marque,reservoir):  
    ...super().__init__(marque)  
    ...self.reservoir = reservoir
```

```
class Pickup(Voiture_moteur):  
    ...def __init__(self, marque, reservoir,puissance):  
    ...super().__init__(marque, reservoir)  
    ...self.puissance = puissance
```


Héritage transitif



```
class Voiture:  
    ...def __init__(self, marque) -> None:  
    ...self.marque = marque
```

```
class Voiture_moteur(Voiture):  
    ...def __init__(self, marque, reservoir):  
    ...super().__init__(marque)  
    ...self.reservoir = reservoir
```

```
class Pickup(Voiture_moteur):  
    ...def __init__(self, marque, reservoir, puissance):  
    ...super().__init__(marque, reservoir)  
    ...self.puissance = puissance
```

➤ Un Pickup hérite des attributs de tous ses ancêtres

```
16   remorque = Pickup("Ford", "60L", "1200hp")  
17  
18   print(f"J'aime mon pick {remorque.marque} avec  
      {remorque.puissance} et et un réservoir de  
      {remorque.reservoir}")
```

PROBLÈMES

1

TERMINAL

...

Python

+

✓

□

🗑

^

×

J'aime mon pick Ford avec 1200hp et et un réservoir de 60L

Héritage des méthodes



```
1 class Voiture:
2     ...def __init__(self,marque) -> None:
3         ...self.marque = marque
4         ...
5     ...def klaxon(self):
6         ...print("honk!")
7
```

```
15 class Pickup(Voiture_moteur):
16     ...def __init__(self, marque, reservoir,puissance):
17         ...super().__init__(marque, reservoir)
18         ...self.puissance = puissance
19
20 remorque = Pickup("Ford","60L","1200hp")
21 remorque.klaxon()
```

> La classe Pickup hérite de la méthode klaxon et les objets de la classe Pickup peuvent donc utiliser cette méthode

PROBLÈMES 1 TERMINAL ...

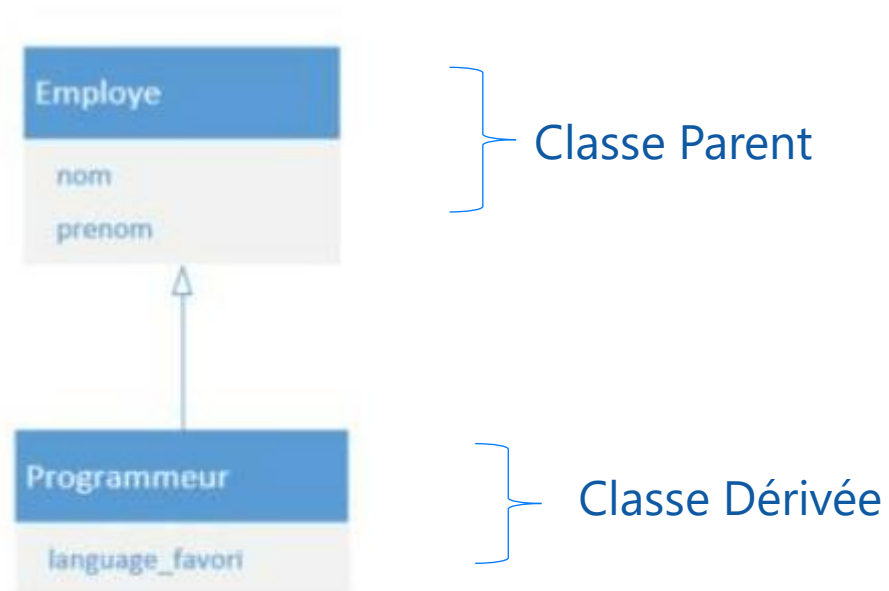
Python + - □ □ ^ ×

honk!

Modélisation UML



- › L'héritage est illustré ainsi dans UML



Un Programmeur EST un Employe



Pseudocode

Pseudo-code,
héritage de méthodes,
et méthodes de classes



- L'UML permet de mettre les objets en relation avant de commencer un projet
- Avant de coder, on veut comprendre la logique de ce qu'on va faire.
 - Plus facile de le faire sans se préoccuper de la syntaxe du code
 - D'où l'utilité du pseudo code



Pseudocode (Kecéça?)

- > Le pseudocode est une façon d'écrire du code sans se préoccuper des particularités d'un langage spécifique.
- > On écrit ce qui sera fait à chacune des étapes du code dans un langage naturel.
- > Permet de planifier facilement ce qu'on veut faire, de le communiquer avec d'autres programmeurs et même avec des non programmeurs.



Exemple simple de pseudo-code

> On veut calculer la somme de deux nombres:

Début

Demander à l'utilisateur d'entrer les deux nombres

Demander "Entrez le premier nombre : " et le stocker dans la variable a

Demander "Entrez le deuxième nombre : " et le stocker dans la variable b

Calculer la somme des deux nombres

somme = a + b

Afficher le résultat

Afficher "La somme de ", a, " et ", b, " est égale à ", somme

Fin



Exemple simple de pseudo-code

- Dans cet exemple, le pseudocode utilise des instructions simples comme "Demander", "Afficher" et des opérateurs mathématiques comme "+" pour représenter les étapes nécessaires pour calculer la somme de deux nombres.
- Ce pseudocode est très simple, mais il illustre comment le pseudocode peut être utilisé pour décrire un algorithme de manière plus claire et compréhensible que le code réel.

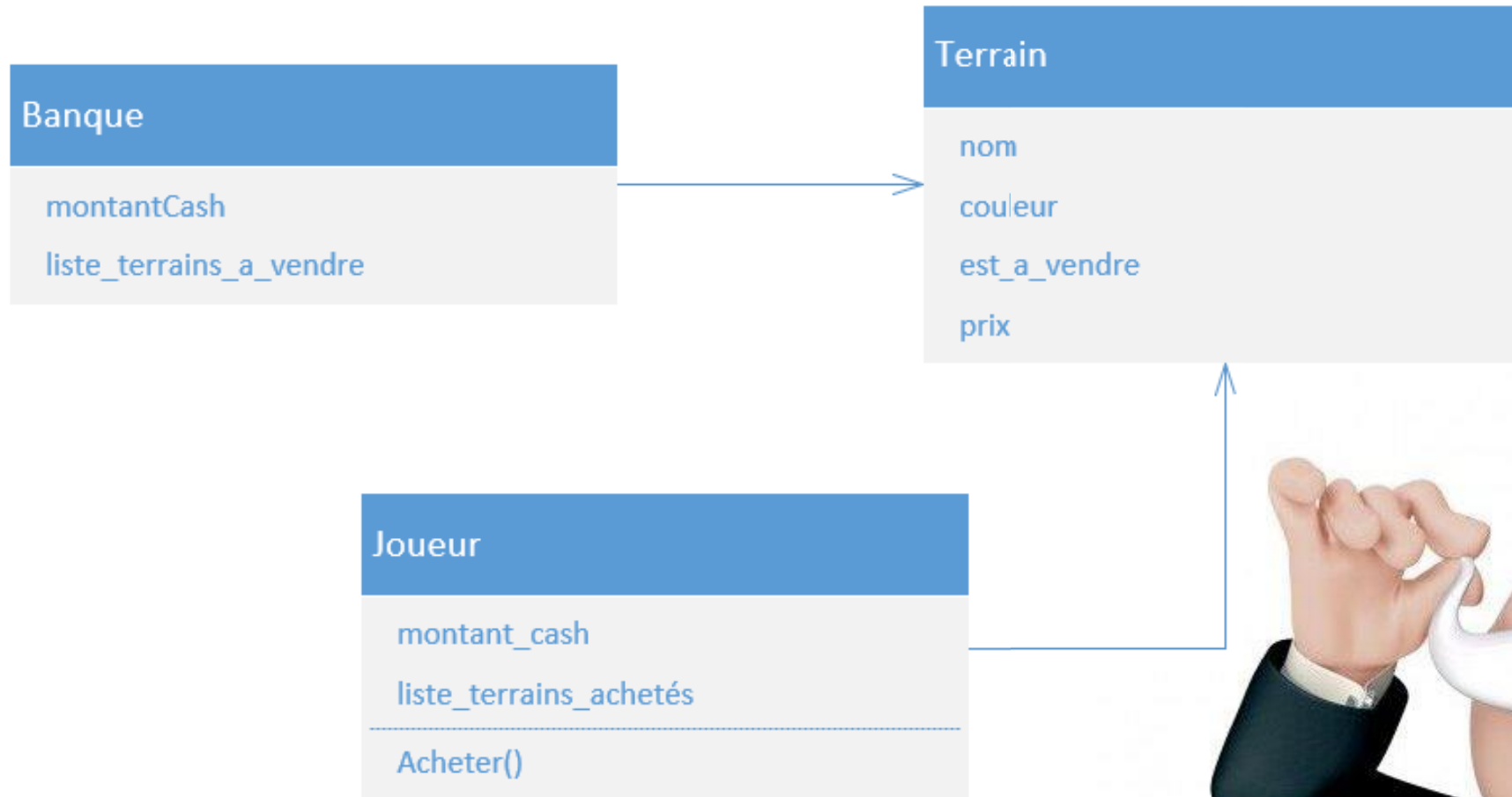


Exemple de Monopoly

- Supposons qu'on développe un jeu de Monopoly
- On veut développer une fonction qui va permettre au joueur d'acheter des terrains de la banque.
- Avant de s'y lancer. On conceptualise ce que nous allons faire.



Exemple de Monopoly - UML

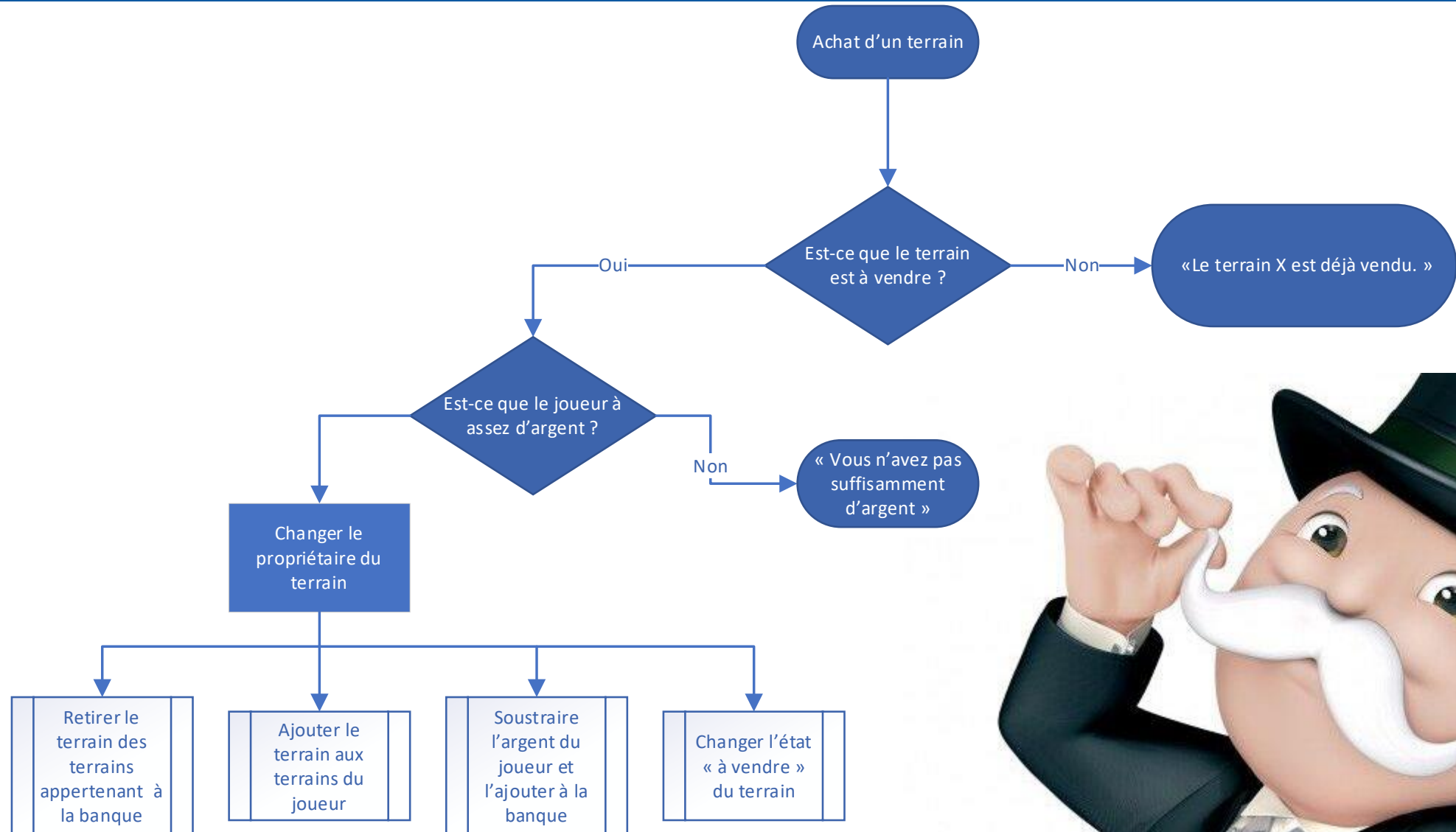


Ex : Monopoly - Pseudocode

- > Vérifier si le « propriétaire » a bel et bien le terrain qu'on veut acheter
 - > Si oui
 - > Vérifier si on a assez d'argent
 - > Si oui
 - > Retirer le terrain de la liste des terrains du propriétaire
 - > Retirer le cash du joueur
 - > Ajouter le cash dans le montant_cash du propriétaire
 - > Ajouter le terrain de la liste des terrains du joueur
 - > Si non
 - > Écrire un message « Vous n'avez pas assez d'argent pour acheter le terrain X »
 - > Si non
 - > Écrire un message « Désolé, je ne suis pas propriétaire de ce terrain »



Exemple de Monopoly – Diagramme de flux





Méthodes de classe

Héritage des méthodes



```
1 class Voiture:
2     ...def __init__(self,marque) -> None:
3     ...self.marque = marque
4     ...
5     ...def klaxon(self):
6     ...print("honk!")
7
```

```
15 class Pickup(Voiture_moteur):
16     ...def __init__(self, marque, reservoir,puissance):
17     ...super().__init__(marque, reservoir)
18     ...self.puissance = puissance
19
20 remorque = Pickup("Ford","60L","1200hp")
21 remorque.klaxon()
```

> La classe Pickup hérite de la méthode klaxon et les objets de la classe Pickup peuvent donc utiliser cette méthode

PROBLÈMES 1 TERMINAL ...

Python + - □ □ ^ ×

honk!

Surcharge de méthodes



```
15 class Pickup(Voiture_moteur):
16     ...def __init__(self, marque, reservoir,puissance):
17     ...     super().__init__(marque, reservoir)
18     ...     self.puissance = puissance
19     ...
20     ...def klaxon(self):
21     ...     print("HOOONKKK!")
22
23 remorque = Pickup("Ford","60L","1200hp")
24 remorque.klaxon()
```

PROBLÈMES 1 SORTIE TERMINAL ...

Python + v I

HOOONKKK!

```
28
29 class Voiture_de_luxe(Voiture_moteur):
30     ...def __init__(self, marque, reservoir,prix):
31     ...     super().__init__(marque, reservoir)
32     ...     self.prix = prix
33
34 fancy_car = Voiture_de_luxe("Mercedes","60L","120000")
35 fancy_car.klaxon()
```

PROBLÈMES 1 SORTIE TERMINAL ...

Python + v I X

honk!

Méthodes de classes



```
class Employe:
    nb_employes = 0
    base_augmentation = 1.04

    def __init__(self, prenom, nom, salaire):
        self.prenom = prenom
        self.nom = nom
        self.salaire = salaire
        self.courriel = prenom + '.' + nom + '@gmail.com'
        Employe.nb_employes += 1

    def nom_complet(self):
        return '{} {}'.format(self.prenom, self.nom)

    def donner_augmentation(self):
        self.salaire = int(self.salaire * self.base_augmentation)
        # nous utilisons self car l'augmentation de base pourrait varier selon l'employé instancié

    @classmethod
    def from_string(cls, emp_str):
        """Constructeur pour créer un employé à partir d'une chaîne séparée avec un '-'
        prenom, nom, salaire = emp_str.split('-')
        return cls(prenom, nom, salaire)
```

Decorator

Méthode
de classe

Fait référence à la classe

Méthodes de classes



```
@classmethod
def from_string(cls, emp_str):
    "Constructeur pour créer un employé à partir d'une chaîne séparée avec un '-'
    prenom, nom, salaire = emp_str.split('-')
    return cls(prenom, nom, salaire)
```

- On va rarement instancier tous nos employés à la main. Cette méthode permettrait d'instancier des employés à partir d'une seule ligne de texte qui nous proviendrait de la lecture d'un fichier, csv ou autre.

```
emp_1 = Employe('Marc', 'Tremblay', 50000)
```

```
emp_2 = Employe.from_string("Joanna-Tremblay-52000")
```



- On fait des méthodes de classe quand la méthode ne fait pas référence aux objets instanciés
- Cette méthode sera la même pour tous les objets instanciés.
- On doit utilisé le decorator **@classmethod** pour identifier que c'est une méthode de classe et pour pouvoir appeler la classe en utilisant **cls**

Priorité des noms



```
1  class Employe:
2      ....id_1 = 1
3      ....id_2 = 1
4      ....id_3 = 1
5      ....
6      ....def __init__(self,nom,prenom) -> None:
7          ....self.nom = nom
8          ....self.prenom = prenom
9          ....self.id_1 = 2
10         ....self.id_2 = 2
11
12     ....def retourne_id(self):
13         ....id_1 = 3
14         ....print(id_1)
15
16 exemple = Employe("a","b")
17
18 exemple.retourne_id()
19 print(exemple.id_2)
20 print(exemple.id_3)
```

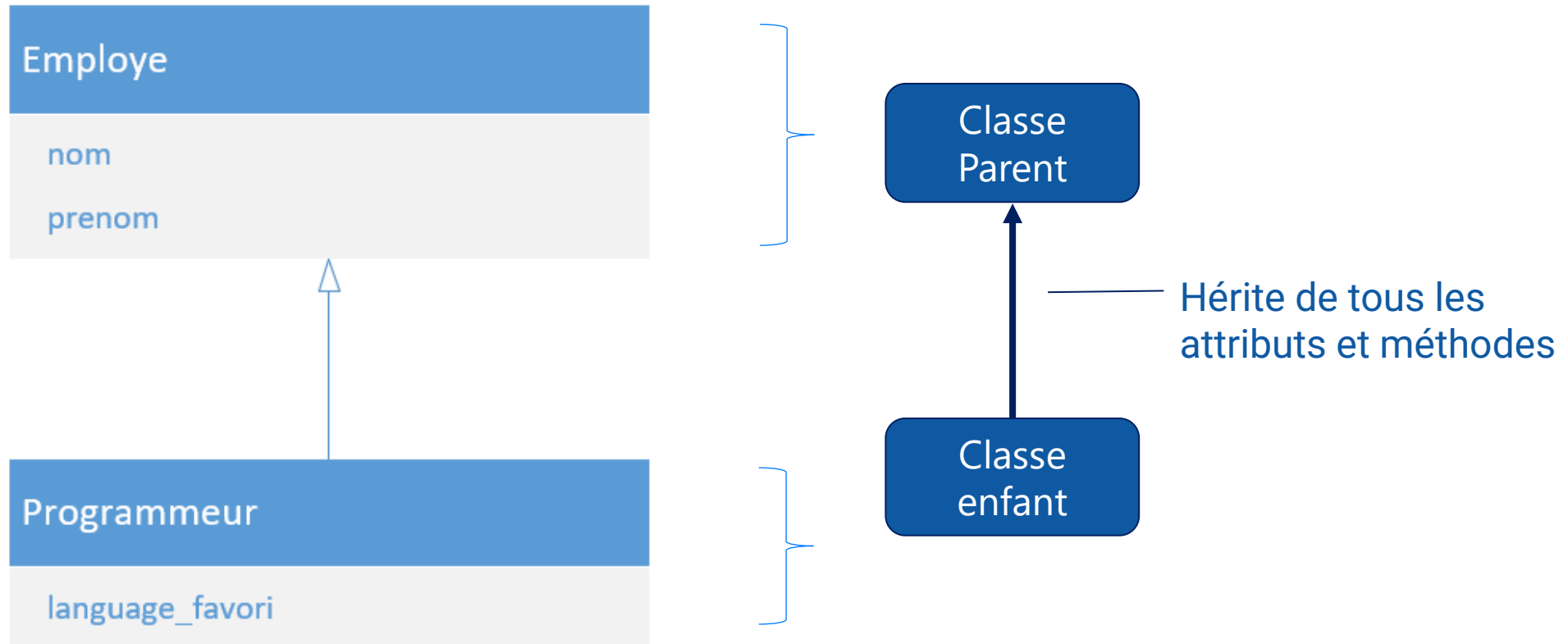


Les relations entre les classes. (et la nomenclature)

- Héritage
- Composition
- Agrégation
- Cardinalité

Héritage (en UML)

- > L'héritage est illustré ainsi dans UML



Un Programmeur EST un Employe

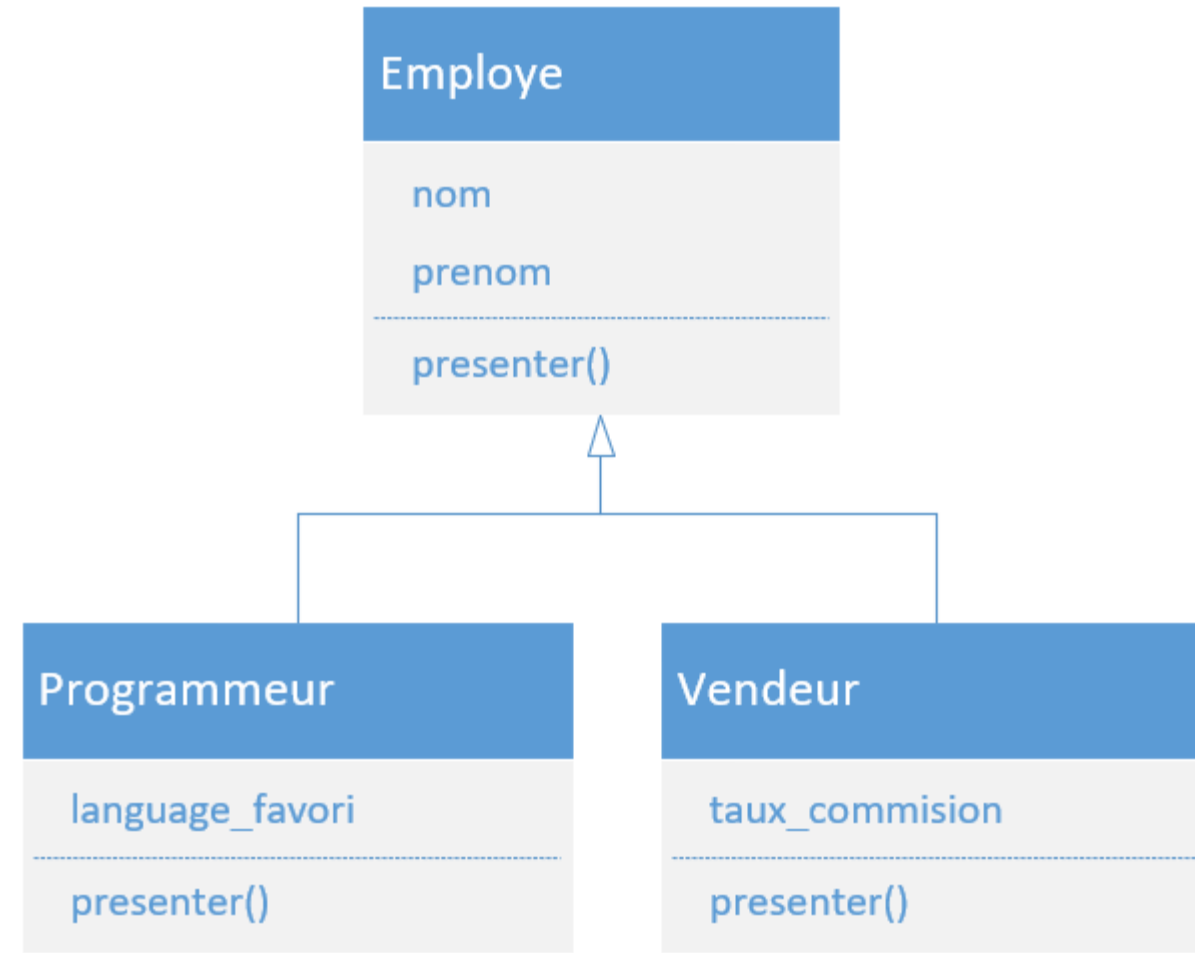
Héritage et polymorphisme



plusieurs

formes

- > Le **Polymorphisme** :
 - > En programmation objet : la capacité d'une variable, d'une fonction, ou d'un objet à prendre plusieurs formes
- > Programmeur et vendeur peuvent tous deux être utilisés dans toutes les situations où la classe Employe peut être utilisée mais la méthode "presenter()" est redéfinie et agira différemment.



Héritage et polymorphisme



```
1  class Employe:
2  ... def __init__(self, nom, prenom) -> None:
3  ...     self.nom = nom
4  ...     self.prenom = prenom
5  ... def __str__(self):
6  ...     return f"Je m'appelle {self.prenom} {self.nom} et je  
    fait partie de l'équipe du CEM."
```

```
8  class Programmeur(Employe):
9  ... def __init__(self, nom, prenom, langage) -> None:
10 ...     super().__init__(nom, prenom)
11 ...     self.langage_favori = langage
12 ... def __str__(self):
13 ...     return (super().__str__() + "\n" +
14 ...             f"Je travaille en tant que programmeur en {self.  
                langage_favori}")
15
16 class Vendeur(Employe):
17 ... def __init__(self, nom, prenom, taux) -> None:
18 ...     super().__init__(nom, prenom)
19 ...     self.taux_commission = taux
20 ... def __str__(self):
21 ...     return (super().__str__() + "\n" +
22 ...             "Je travaille en tant que vendeur avec  
                commission" )
23
```

Héritage et polymorphisme



```
27
28 emp = Employe("Gallant","Pierre")
29 print(emp)
30 emp_prog = Programmeur("Tremblay","Ana","Python")
31 print(emp_prog)
32 emp_vendeur = Vendeur("Borne","Margaut",2)
33 print(emp_vendeur)
```

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET INTERACTIVE

```
Je m'appelle Pierre Gallant et je fait partie de l'équipe du CEM.
Je m'appelle Ana Tremblay et je fait partie de l'équipe du CEM.
Je travaille en tant que programmeur en Python
Je m'appelle Margaut Borne et je fait partie de l'équipe du CEM.
Je travaille en tant que vendeur avec commission
```


Signature



- > Lorsqu'on implémente une nouvelle classe ou un nouveau groupe de classes interreliées : on commence par écrire leur signature
- > La signature consiste est le « *squelette* » d'une classe.
- > Elle consiste en le nom des attributs et méthodes, ainsi que les paramètres que les méthodes prennent

```
1 class Employe:
2     ...def __init__(self, nom, prenom) -> None:
3     ...|...pass
4     ...def __str__(self):
5     ...|...return
6
7 class Programmeur(Employe):
8     ...def __init__(self, nom, prenom, langage) -> None:
9     ...|...pass
10    ...def __str__(self):
11    ...|...return
12
13 class Vendeur(Employe):
14    ...def __init__(self, nom, prenom, taux) -> None:
15    ...|...pass
16    ...def __str__(self):
17    ...|...return
18
```



- La cardinalité permet d'indiquer le sens d'une relation entre des classes différentes ainsi que le nombre d'entités en relation.

1 à 0..1	Une entité à aucune ou une instance
1 à 1	Une entité à une instance exactement
1 à 0..N ou 1 à N	Une entité à aucune ou plusieurs instances
1 à 1..N	Une entité à une instance ou plusieurs (au moins une)



- Dans les relations un à plusieurs, il y a deux concepts importants :
 - **Composition** : Un objet est fait de 1 ou plusieurs autres objets
 - **Agrégation** : Un objet possède ou regroupe d'autre(s) objet(s)

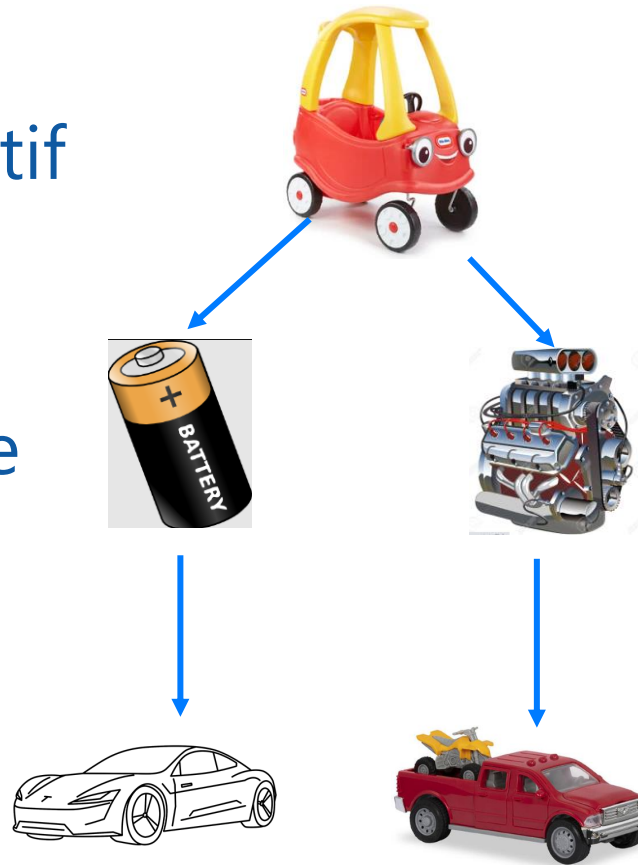


- Concept en programmation objet.
- Un objet dit "composite" est un objet qui "*possède*" d'autre objets de classes différentes.
- Lorsqu'on supprime un objet "composite", on va normalement aussi supprimer les objets que le compose.

Voir exemple

Ex : Voitures par héritage transitif

- > L'héritage est transitif en Python.
- > Une classe hérite de tous les attributs et méthodes de tous ses ancêtres.



```
class Voiture:  
    ...def __init__(self,marque) -> None:  
    ...self.marque = marque
```

```
class Voiture_moteur(Voiture):  
    ...def __init__(self, marque,reservoir):  
    ...super().__init__(marque)  
    ...self.reservoir = reservoir
```

```
class Pickup(Voiture_moteur):  
    ...def __init__(self, marque, reservoir,puissance):  
    ...super().__init__(marque, reservoir)  
    ...self.puissance = puissance
```

Ex : Voitures par composition



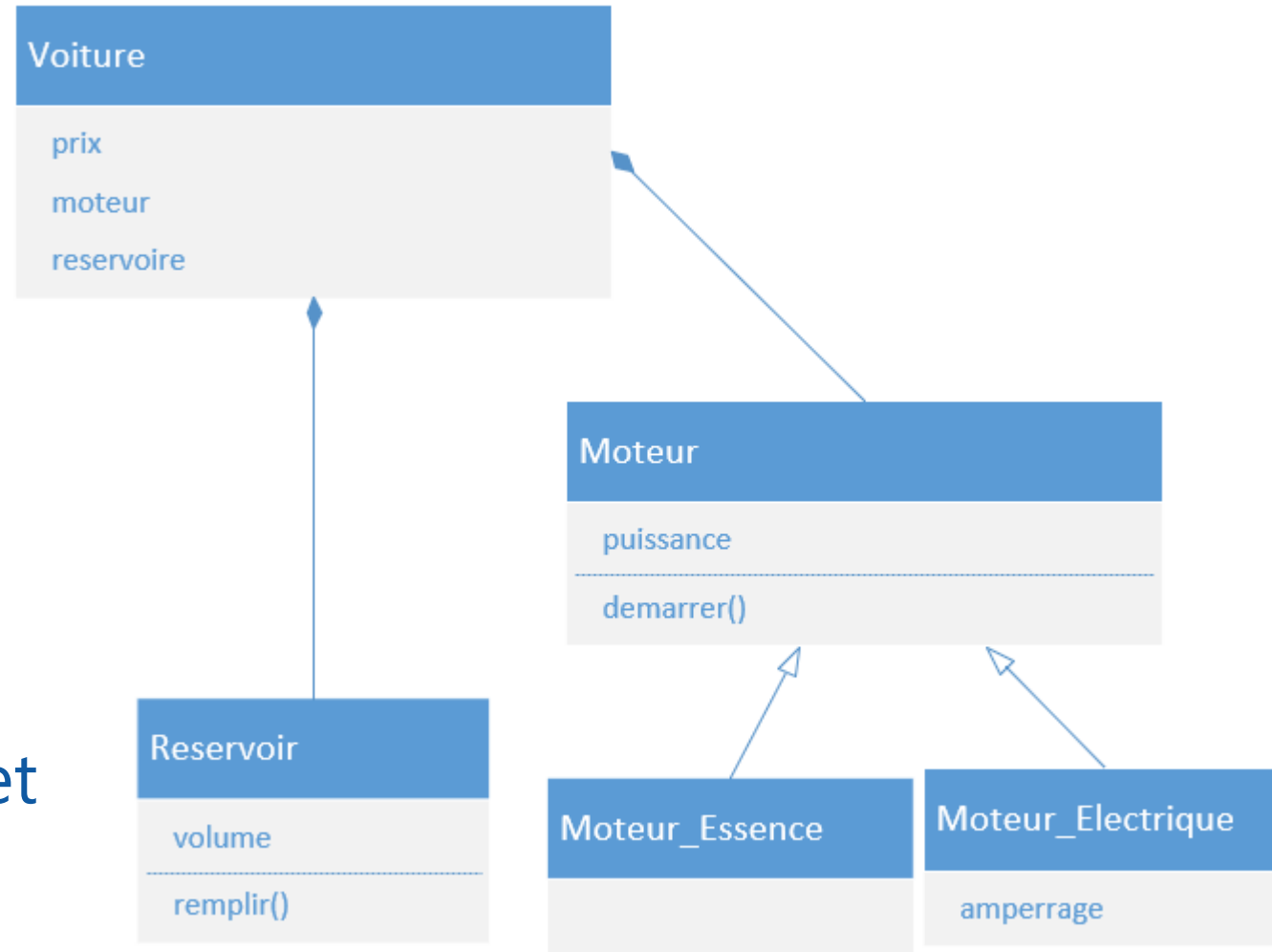
- > La classe Moteur contient les propriétés des objets Moteur
- > La sous-classe Moteur_Electrique hérite de la classe Moteur
- > La classe Voiture contient un objet Moteur et un objet Reservoir.

```
1  class Moteur:
2  |  ...def __init__(self) -> None:
3  |  |  ...pass
4
5  class Moteur_Electrique(Moteur):
6  |  ...def __init__(self) -> None:
7  |  |  ...super().__init__()
8
9  class Reservoir:
10 |  ...def __init__(self) -> None:
11 |  |  ...pass
12 |  ....
13
14 class Voiture:
15 |  ...def __init__(self, moteur, reservoir, prix):
16 |  |  ...pass
```

Ex : Voitures par composition (UML)



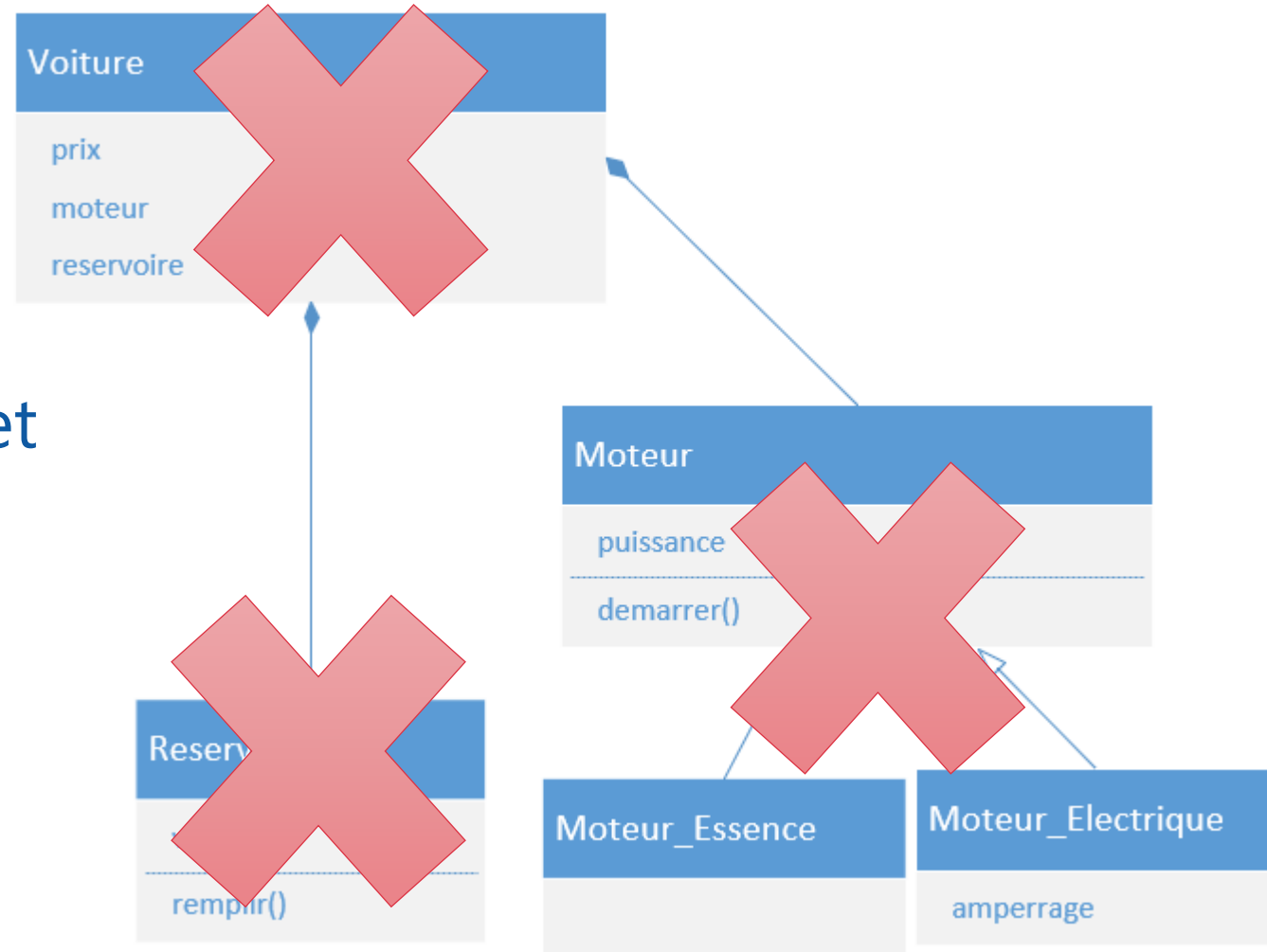
- > La classe Moteur contient les propriétés des objets Moteur
- > La sous-classe Moteur_Electrique hérite de la classe Moteur
- > La classe Voiture contient un objet Moteur et un objet Reservoir.



Ex : Voitures par composition (UML)



- > La classe Voiture contient un objet Moteur et un objet Reservoir.
- > Si une voiture est supprimé, son moteur et son réservoir le sont aussi.



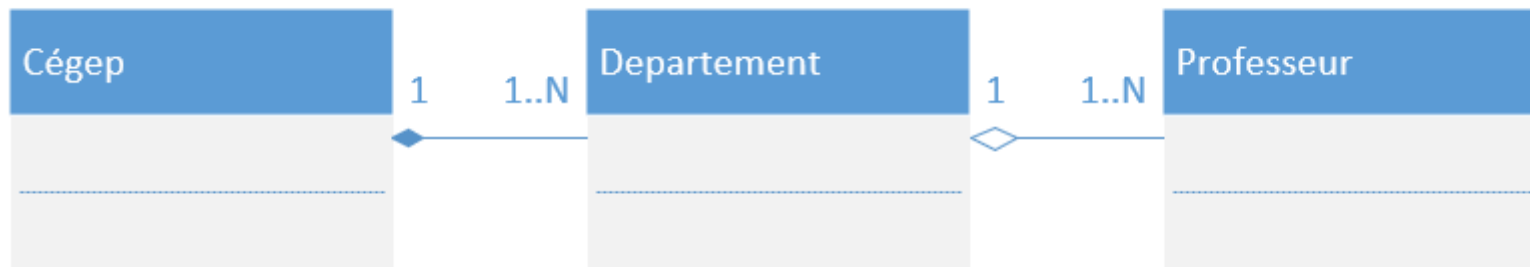


- On parle d'agrégation lorsqu'un objet va faire référence à un ou plusieurs objets sans pour autant en être le propriétaire.
- Contrairement à la composition, lorsqu'un "agrégat" est supprimé, les objets auxquels il fait référence ne le sont pas.

Agrégation



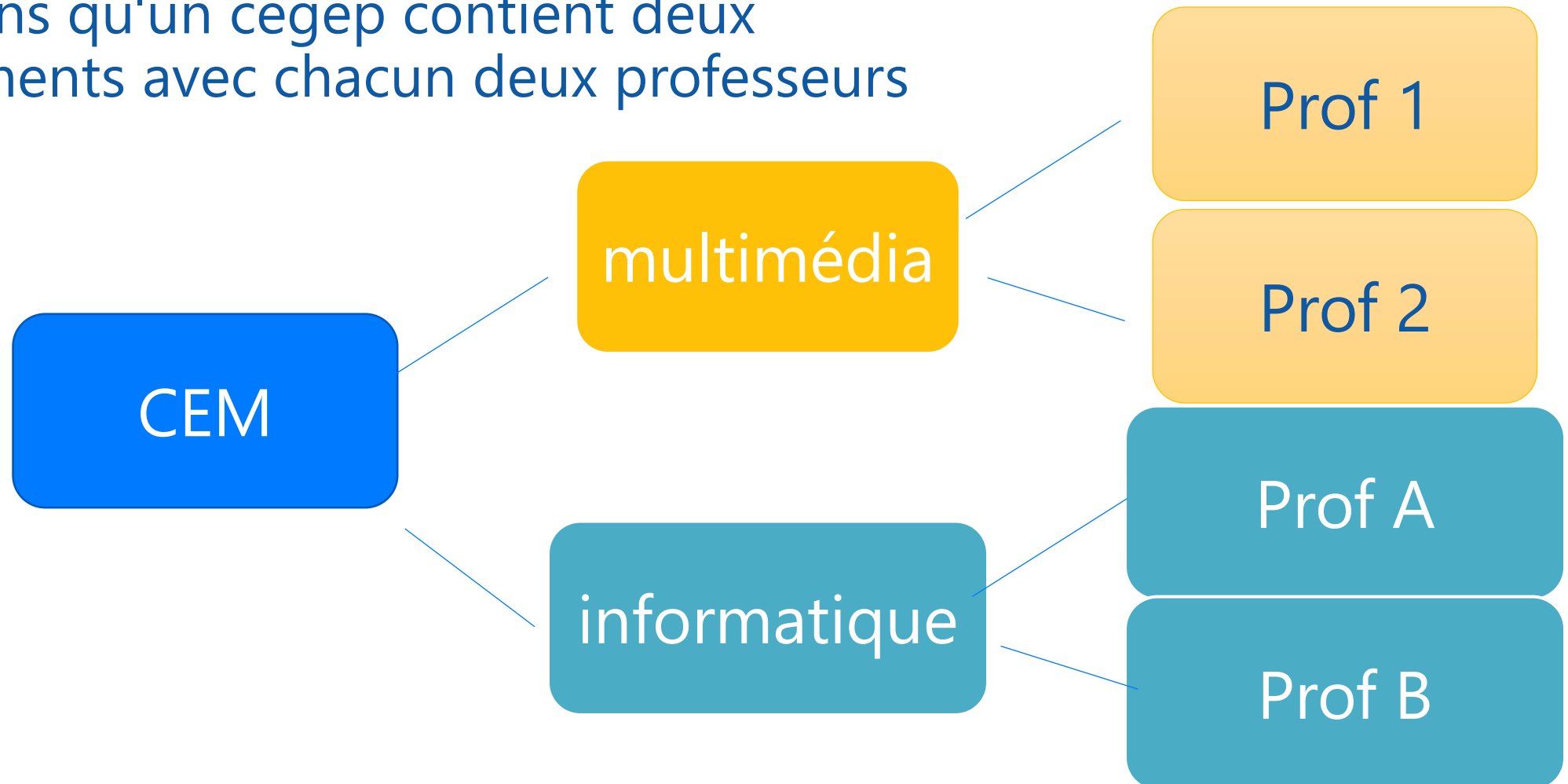
- Le cégep contient plusieurs département (composition) qui engagent chacun plusieurs professeurs (agrégation)



Ex : Agrégation



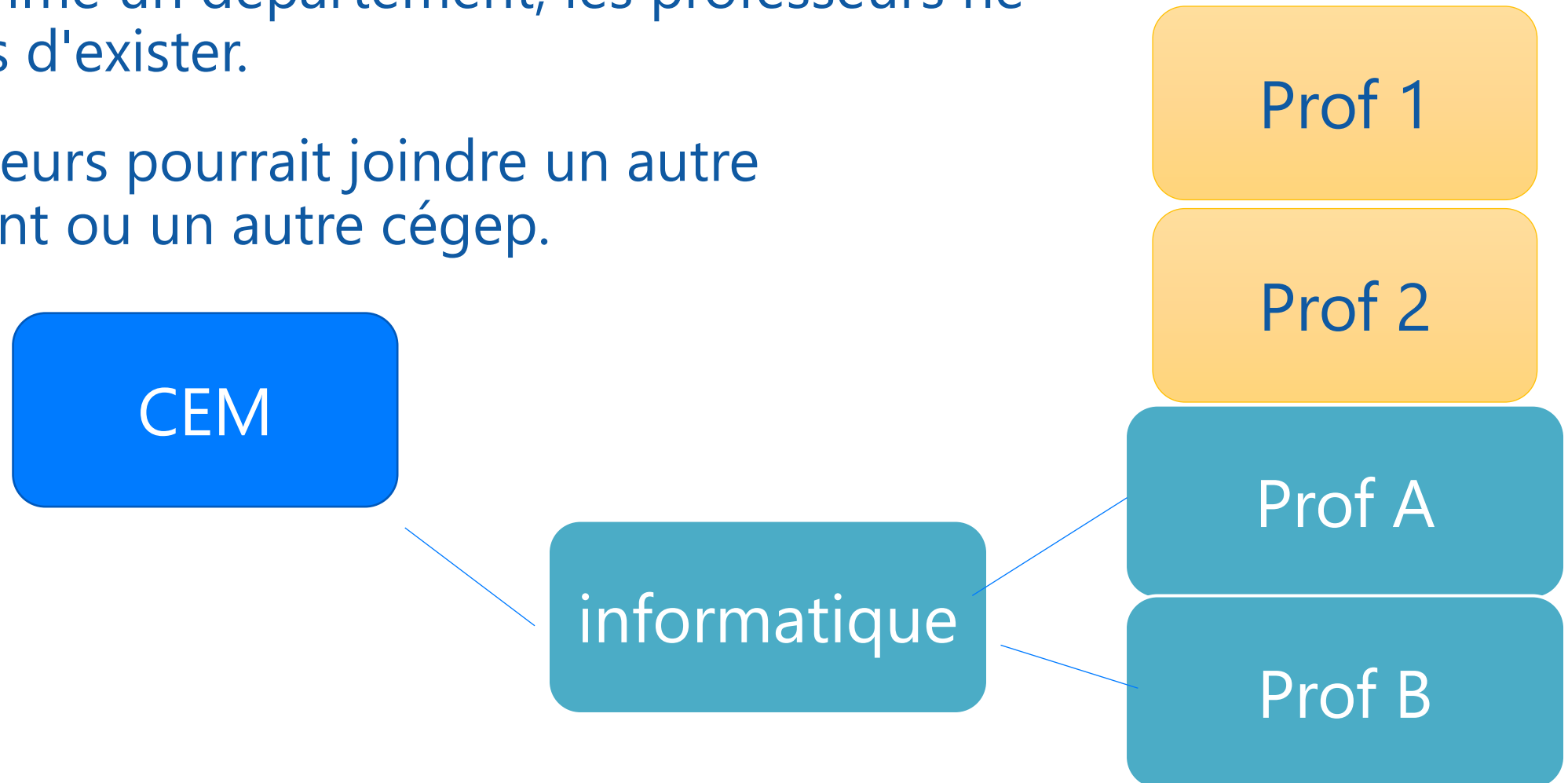
- Supposons qu'un cégep contient deux départements avec chacun deux professeurs





Ex : Agrégation

- > Si on supprime un département, les professeurs ne cessent pas d'exister.
- > Les professeurs pourraient joindre un autre département ou un autre cégep.





Ex : Agrégation vs Composition

- Par contraste, si on supprime le cégep, les départements cesseront d'exister (il s'agit alors de composition)

Prof 1

Prof 2

Prof A

Prof B



Nouvelles méthodes et classes abstraites

Méthodes statiques

Méthodes privées et abstraites

Classes abstraites



Sommaire des différentes méthodes

- > Méthodes d'instances : Méthodes appartenant à une instance d'une classe.
- > Méthodes de classe : Méthodes appartenant à la classe elle-même et non à chaque instance.
- > Méthodes statiques : Méthodes qui sont contenues dans la classe mais fonctionnent indépendamment. Elles ne font pas références et n'affectent pas la classe ou une instance.
- > Méthodes privées : Méthodes qui ne sont pas appelées hors de la classe.
- > Méthodes abstraites : Méthodes qui DOIVENT être redéfinis dans les sous-classes.



Résumer différentes méthodes (EXEMPLE)

```
class Employe:
...liste_employe = []
...next_ID = 1000
...def __init__(self,nom,prenom):
...|...pass
...
...def retourner_nom_complet(self):
...|...return f"{self.prenom} {self.nom}"
...
...@classmethod
...def afficher_liste_employe(cls):
...|...print(json.dumps(cls.liste_employe, indent = 4))
...
...@staticmethod
...def info_retraite():
...|...return "Il faut avoir 65 et plus ou avoir 35 ans d'ancienneté pour se qualifier."
```

méthode d'instance

méthode de classe

méthode statique



Méthodes statiques

- Méthodes qui appartiennent à la classe mais qui ne font pas référence à une instance ou bien à la classe elle-même.
- On utilise le décorateur "@staticmethod"

```
....@staticmethod
....def info_retraite():
....|....return "Il faut avoir 65 et plus ou avoir 35 ans d'ancienneté pour se qualifier."
```

- La méthode `info_retraite()` est une méthode statique. Elle appartient à la classe `Employe` mais n'utilise pas d'attributs ou méthodes de la classe ou de l'instance.



Méthodes de classe vs méthodes statique

```
1 class Employe:
2     ....taux = 1.09
3     ....def __init__(self) -> None:
4     ....    ....pass
5
6     ....def changer_taux_1(nvx_taux):
7     ....    ....Employe.taux = nvx_taux
8
9     ....def changer_taux_2(Employe,nvx_taux):
10    ....    ....Employe.taux = nvx_taux
11
12    ....@classmethod
13    ....def changer_taux_3(cls,nvx_taux):
14    ....    ....cls.taux = nvx_taux
15    ....
16    ....@staticmethod
17    ....def changer_taux_4(nvx_taux):
18    ....    ....Employe.taux = nvx_taux
19
```

> Toutes ces méthodes donnent le même résultat.

MAIS

> Une seule est conforme aux standards de programmation

> On DOIT respecter les standards pour que notre code soit lisible par d'autre programmeurs et par nous-mêmes



Méthodes privées

- > Les méthodes privées sont des méthodes qui ne sont pas utilisées hors de la classe.
- > Seule la classe peut appeler ces méthodes.
- > On indique qu'une méthode est privée en mettant un "__" (double "underscore")

```
.....def __methode_prive():  
.....|.....print("Cette méthode est privée.")
```



Méthodes privées

- Seule la classe peut appeler ces méthodes.
- Appeler une méthode privée hors de la classe génère une erreur.

```
28     ...def __methode_prive():
29     ...     print("Cette méthode est privée.")
30
31     Employe.__methode_prive()
32
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTERACTIVE JUPYTER

```
Employe.__methode_prive()
AttributeError: type object 'Employe' has no attribute 'methode prive'
```



Méthodes privées

- Toutes les types de méthodes peuvent être privés.

```
... def __methode_instance_prive(self):  
...     pass  
... @classmethod  
... def __methode_classe_prive(cls):  
...     pass  
... @staticmethod  
... def __methode_static_prive():  
...     pass
```



- On utilise les méthodes privées pour les mêmes raisons qu'on utilise des fonctions dans un script :
 - Faire des blocs de code réutilisable.
 - Séparer les tâches en sous-tâches plus simples.
 - Rendre le code lisible en donnant des noms significatifs aux différentes tâches.
 - Rendre le code facile à maintenir.
- DE PLUS, les méthodes privées permettent d'encapsuler des opérations qu'on ne veut pas qu'elles soient accessibles par d'autres classes ou objets.



- Supposons que nous avons une classe Employe avec des sous-classes Programmeur et Vendeur
- MAIS
- On ne veut pas avoir d'instances de la classe Employe. Parce que tous nos employés ont un rôle ou un poste.
 - On transforme Employe en une classe abstraite. Une classe abstraite est une classe qui ne peut pas être instanciée mais à partir de laquelle on peut créer des sous-classes.



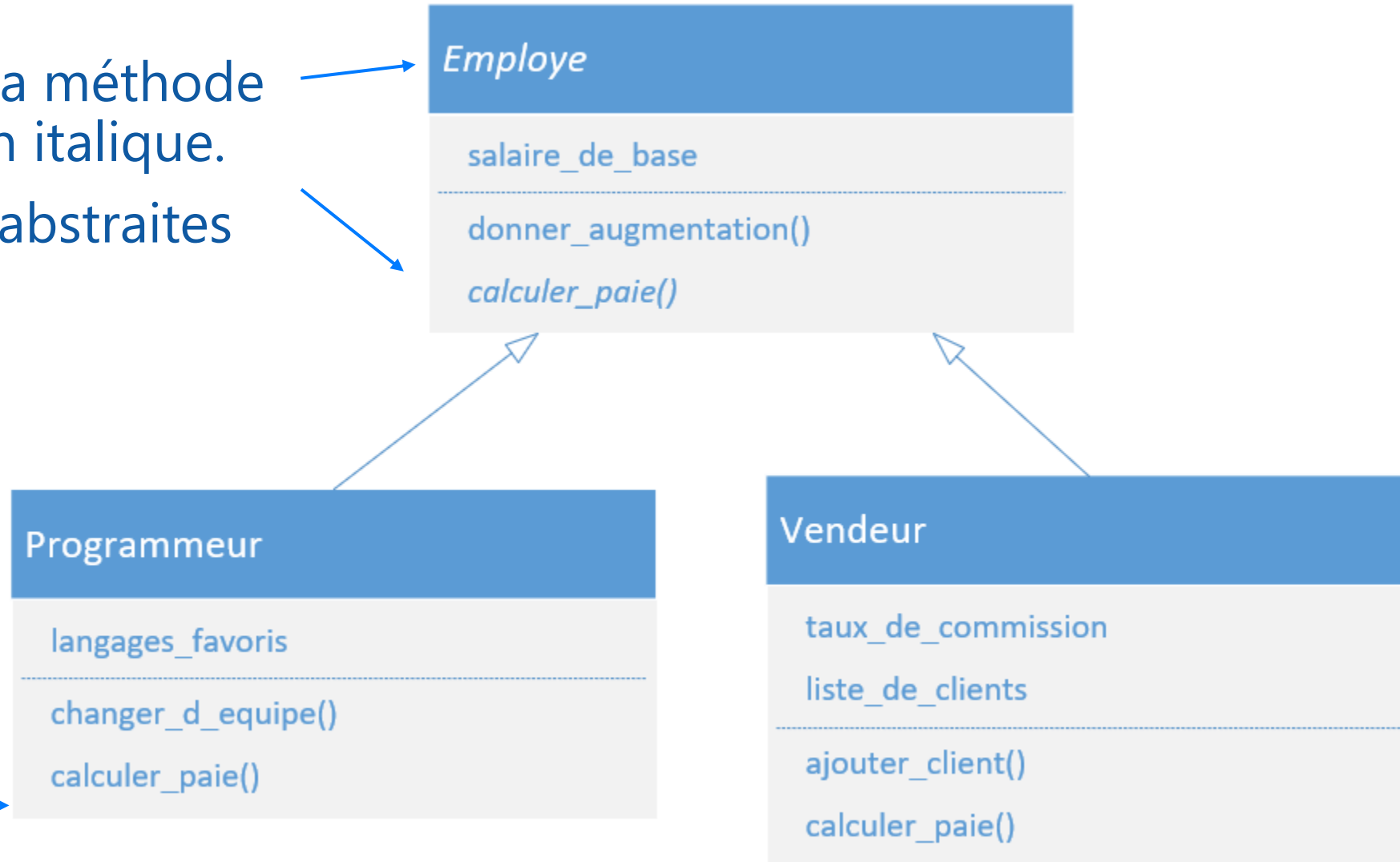
- Cette classe abstraite a des méthodes dont au moins une qui sera abstraite.
- Lorsqu'on fait une sous-classe à partir de cette classe, on devra redéfinir les méthodes qui étaient abstraites dans la classe parent.

Méthodes et classes abstraites (UML)



La classe *Employe* et la méthode *calculer_paie()* sont en italique. Indique qu'elles sont abstraites

La méthode *calculer_paie()* réapparaît. Cette fois elle n'est pas en italique. Indique que la méthode a été redéfinie.





Création de méthodes et classes abstraites

- > Nécessite l'utilisation du module abc (Abstract Base Classe)

- > Permet de créer une classe abstraite simplement en dérivant cette classe de la classe ABC

- > Permet de créer des méthodes abstraites à l'aide du décorateur @abstractmethod

```
/* employe.py > ...  
1  from abc import ABC, abstractmethod  
2  
3  class Employe(ABC):  
4      ....liste_employe = []  
5      ....next_ID = 1000  
6      ....def __init__(self, nom, prenom):  
7          ....pass  
8      ....  
9      ....@abstractmethod  
10     ....def calculer_paie(self):  
11     ....pass
```



Attributs privés et gestion d'erreur

- Attributs privés,
- Propriétés,
- Setters et getters
- Try...except
- raiseError

Encapsulation



- > Principe en programmation.
- > Concept fondamental en programmation objet.
- > Consiste en le regroupement de données, valeurs, et fonctions dans un bloc pour permettre la lecture et manipulation de ces données.
- > En programmation objet, on fait de l'encapsulation en utilisant des classes et des instances.

Attributs privés



- On peut vouloir restreindre l'accès à certaines valeurs hors de la classe.
- En Python, la désignation d'un attribut comme étant privé est fait simplement en commençant son nom par un "_" (un seul underscore)

```
class Employe:
    ... def __init__(self, nom, prenom, salaire):
    ...     self.nom = nom
    ...     self.prenom = prenom
    ...     self._salaire = salaire
```

Attributs publiques

Attribut privé

Attributs privés



- Par standard, on n'appelle jamais les attributs commençant par un "_" hors de la classe.

```
class Employe:
    ...def __init__(self,nom,prenom,salaire):
    ...    self.nom = nom
    ...    self.prenom = prenom
    ...    self._salaire = salaire

chimiste = Employe("Belatekallim","Tapputi","45k")

print(chimiste.nom)
print(chimiste.prenom)
print(chimiste.salaire)
```

PROBLÈMES

1

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET

Belatekallim

Tapputi

Traceback (most recent call last):

File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpe

*Dans la plupart des langages on peut indiquer qu'un attribut est privé avec un mot-clé.
Cet attribut n'est alors aucunement accessible hors de la classe.



Propriétés (getters)

- Si on veut quand même avoir accès au salaire de l'employé, on va devoir utiliser un décorateur pour créer une propriété.
- Une propriété se comporte généralement comme un attribut mais est définie à l'aide d'une méthode.

```
....@property
....def salaire(self):
....    ....return self._salaire
```

```
print(chimiste.nom)
print(chimiste.prenom)
print(chimiste.salaire)
```

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NE

```
Belatekallim
Tapputi
45k
```



Propriétés (getters)

- > Dans cet exemple, la différence entre l'attribut et la propriété est:
- > nom et prénom sont des attributs et on peut changer leur valeur.
- > salaire est une propriété et sa valeur ne peut pas être changée.

```
chimiste.nom = "Tapi"  
print(chimiste.nom)
```

```
chimiste.prenom = "Bela"  
print(chimiste.prenom)
```

PROBLÈMES	SORTIE	CONSOLE DE DÉBOGAGE	<u>TERMINAL</u>
			Tapi Bela

```
chimiste.salaire = "50k"  
print(chimiste.salaire)
```

PROBLÈMES	SORTIE	CONSOLE DE DÉBOGAGE	TERMINAL
			Traceback (most recent call last): File "c:\Users\pierre-paul.gallant\Cégep Édouard (Pilote réseau)\R21\exemple.py", line 22, in <module> chimiste.salaire = "50k"



setters

- > Les setters nous permettent de changer les valeurs des propriétés.
- > Il faut encore utiliser un décorateur et le comportement des setters sera défini par une méthode.

```
....@salaire.setter
....def salaire(self,nvx_salaire):
....|....if nvx_salaire > self._salaire:
....|....|....self._salaire = nvx_salaire
```

- > **N.B.** la syntaxe de ce décorateur est légèrement différente.
@nom_de_la_propriété.setter



setters

- > Ce setter contrôle la façon dont on change la valeur du salaire.
- > Si la nouvelle valeur est inférieure à l'ancienne, la valeur du salaire n'est pas changée.

```
....@salaire.setter  
....def salaire(self,nvx_salaire):  
....|....if nvx_salaire > self._salaire:  
....|....|....self._salaire = nvx_salaire
```

```
chimiste = Employe("Belatekallim","Tapputi",45000)
```

```
chimiste.salaire = 30000  
print(chimiste.salaire)  
chimiste.salaire = 60000  
print(chimiste.salaire)
```

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET I

45000

60000



setters

- > Ce setter contrôle la façon dont on change la valeur du salaire.
- > Si la nouvelle valeur est inférieure à l'ancienne, la valeur du salaire n'est pas changée.

```
....@salaire.setter  
....def salaire(self,nvx_salaire):  
....|....if nvx_salaire > self._salaire:  
....|....|....self._salaire = nvx_salaire
```

```
chimiste = Employe("Belatekallim","Tapputi",45000)
```

```
chimiste.salaire = 30000  
print(chimiste.salaire)  
chimiste.salaire = 60000  
print(chimiste.salaire)
```

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET I

45000

60000



proprieties delete

- > Les propriétés ne peuvent pas être supprimées normalement.
- > Le décorateur `@nom_de_la_propriété.delete` est nécessaire pour supprimer une propriété

```
...@salaire.delete
...def salaire(self):
...|...del self._salaire
```



proprieties deleter

> Sans deleter :

```
30
31 del chimiste.salaire
32 print(chimiste.salaire)
```

```
PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE  TERMINAL

in <module>
  del chimiste.salaire
AttributeError: can't delete attribute 'salaire'
```

> Ne peut pas exécuter la ligne 31, ne peut pas supprimer l'attribut "salaire"

> avec deleter :

```
23     ....@salaire.deleter
24     ....def salaire(self):
25     ....|....del self._salaire
26
27 chimiste = Employe("Belatekallim","Tapputi",45000)
28
29 del chimiste.salaire
30 print(chimiste.salaire)
31
```

```
Traceback (most recent call last):
  File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_Intro
in <module>
    print(chimiste.salaire)
  File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_Intro
in salaire
    return self._salaire
AttributeError: 'Employe' object has no attribute '_salaire'. Did you me
```

> Ne peut pas faire l'impression, l'attribut a été supprimé.



Property deleters

- > Puisque le comportement des deleters est déterminé par une fonction, on décide ce qui se passe lorsqu'on supprime un attribut.

```
....@salaire.delete  
....def salaire(self):  
....    self._salaire = 0
```

```
chimiste = Employe("Belatekallim", "Tapputi", 45000)
```

```
del chimiste.salaire  
print(chimiste.salaire)
```

- > Maintenant, supprimer la valeur "salaire" ne retire pas l'attribut, mais plutôt il réinitialise le salaire à 0.

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET INTERACTIVE

0



raise Error & Try ... except

- > Reprenons l'exemple de salaire.setter
- > La valeur du salaire change seulement lorsqu'on rentre un chiffre supérieur à l'ancien salaire.
- > Dans cet exemple il n'y a pas d'indications que le salaire n'a pas été modifié lorsqu'on passe un chiffre inférieur au salaire original.

```
....@salaire.setter  
....def salaire(self,nvx_salaire):  
....|....if nvx_salaire > self._salaire:  
....|....|....self._salaire = nvx_salaire
```

```
chimiste = Employe("Belatekallim","Tapputi")
```

```
chimiste.salaire = 30000  
print(chimiste.salaire)  
chimiste.salaire = 60000  
print(chimiste.salaire)
```

PROBLÈMES	SORTIE	CONSOLE DE DÉBOGAGE	TERMINA
	45000		
	60000		



raise Error

- > Le mot-clefs **raise** nous permet de soulever une erreur dans les situations où le code fonctionne mais une erreur de logique a lieu.
- > L'erreur interrompt l'exécution du code et nous affiche un message de notre choix.

```
....@salaire.setter
....def salaire(self,nvx_salaire):
....    if nvx_salaire > self._salaire_de_base:
....        self._salaire_de_base = nvx_salaire
....    else :
....        raise ValueError("Le nouveau salaire doit être plus grand.")
```

```
chimiste = Employe("Belatekallim","Tapputi",45000)
```

```
chimiste.salaire = 3000
```

PROBLÈMES

SORTIE

CONSOLE DE DÉBOGAGE

TERMINAL

.NET INTERACTIVE

JUPYTER

AZURE

Traceback (most recent call last):

```
File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_Informatique
chimiste.salaire = 3000
```

```
File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_Informatique
raise ValueError("Le nouveau salaire doit être plus grand.")
```

```
ValueError: Le nouveau salaire doit être plus grand.
```


Try ... except



- > On peut aussi vouloir le comportement opposé.
- > Si on anticipe une erreur possible, on peut l'attraper à l'aide des mots-clefs **try** et **except** pour éviter d'interrompre l'exécution du programme lorsqu'une erreur survient.

```
....@salaire.setter
....def salaire(self,nvx_salaire):
....    try:
....        if nvx_salaire > self._salaire:
....            self._salaire = nvx_salaire
....    except TypeError:
....        print("Vous devez entrer un montant en chiffres.")
```

Try ... except

- Maintenant, le programme n'est pas interrompu lorsqu'on entre le mauvais type de données.

```
....@salaire.setter
....def salaire(self,nvx_salaire):
....    try:
....        if nvx_salaire > self._salaire:
....            self._salaire = nvx_salaire
....    except TypeError:
....        print("Vous devez entrer un montant en chiffres.")
```

```
chimiste = Employe("Belatekallim","Tapputi",45000)
```

```
chimiste.salaire = "15k"
print("Le programme n'est pas interrompu")
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTERACTIVE

```
Vous devez entrer un montant en chiffres.
Le programme n'est pas interrompu
```

2N6 Programmation 2



GUI

Graphical User Interface.

Utilisation du module customtkinter.

Introductions aux widgets.



L'interface graphique (GUI)

- Le *Graphical User Interface* (GUI) permet à un utilisateur d'interagir avec un programme à l'aide d'éléments visuels.
- Inclut icônes, boutons, fenêtres, zones de texte et plus.
- Permet de communiquer plus d'informations rapidement.
- Plus agréable à utiliser pour l'utilisateur novice.



Les options de GUI

- > Selon que vous voulez faire quelque chose de basique ou développer une application complexe qui sera commercialisée, vous utiliserez différents modules pour développer votre interface GUI.
- > Quelques modules, du plus simple au plus complexe:
 - > Tkinter → inclut dans librairie standard
 - > Custom tkinter ... permet d'obtenir des gui stylisé rapidement à partir de Tkinter.
 - > Qt5 framework
 - > PySimpleGUI
 - > PyQt5.



- Fait partie de la librairie standard.
- Fournit des outils pour créer des éléments d'interface graphique pour des applications de bureau.
- Placement des éléments dans une grille.
- Peut être un peu compliqué mais offre beaucoup de possibilités.

Module **customtkinter**



- Module ne faisant pas partie de la librairie standard.
- Doit être installé avec pip

```
pip install customtkinter
```

- Utilise les éléments de tkinter.
- Contient déjà des styles associés à chacun des contrôles.
- Facilite la création d'interfaces décentes rapidement.
- Pas recommandé pour les plus gros projets.



Créer une fenêtre dans **customtkinter**



```
1 import customtkinter
2
3 customtkinter.set_appearance_mode("dark")..# Style de la fenêtre"
4 customtkinter.set_default_color_theme("blue")..# couleurs des éléments
5
6 app = customtkinter.CTk()
7 # Paramètres de l'app tel que la taille de la fenêtre.
8
9 # Tous les widgets qui formeront notre interface graphique.
10
11 app.mainloop() # Lance notre application..
```



Frame

- On peut découper une fenêtre en différentes parties, appelées Frame.





- > Un widget est un élément d'interface graphique interactif.
- > L'assemblage de plusieurs widgets va former notre interface graphique.
- > Aujourd'hui, nous allons voir :
 - > Un **label** pour afficher de l'information.
 - > Une **zone de texte** pour entrer de l'information.
 - > Une **liste déroulante** pour faire des choix parmi une liste.
 - > Des **boutons** pour interagir avec l'interface.



- > Petite zone pour afficher du texte.
- > Une "étiquette" en français.
- > Donne des informations sur les différentes sections et/ou widgets.
- > Pas une source de saisie de données (Fait uniquement de l'affichage.)
- > Peut être utilisé pour afficher un message de façon dynamique à l'aide de fonctions.





- > Un bouton !
- > Ne fait rien par lui-même
- > On doit associer le bouton avec une méthode lors de sa création.

```
def appuyer_sur_button():  
    ...label_1.configure(text="Yay un bouton !")  
  
button_1 = customtkinter.CTkButton(master=frame_1, command=appuyer_sur_button, width=200)
```

- > **Notez :** le paramètre *command* prend la fonction elle-même et non son résultat, (sans les parenthèses).


Associer du code



- > Il faut définir des fonctions pour associer du code à des éléments graphiques.

```
def appuyer_sur_button():  
    ...label_1.configure(text="Yay un bouton !")  
  
button_1 = customtkinter.CTkButton(master=frame_1, command=appuyer_sur_button, width=200)
```

Pas de parenthèses



- > Appuyer sur le bouton devient exactement la même chose que d'exécuter la commande.



- > Champs pour insérer du texte qui sera utilisé par le code.
- > Une fois que l'information est écrite :
 - > La méthode `get()` va nous retourner le texte entré par l'utilisateur.
 - > La méthode `delete()` va supprimer les caractères entre deux index donnés.

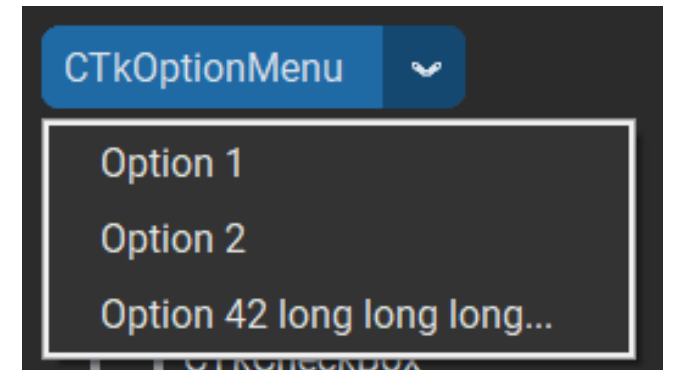
```
#Champs où écrire de l'information
entry_1 = customtkinter.CTkEntry(master=frame_1, placeholder_text="CTkEntry")
entry_1.grid(column=1,row=0,padx=30,pady=20,sticky="w")
```

```
def appuyer_sur_button():
    ...print(entry_1.get())
    ...entry_1.delete(0,len(entry_1.get()))
```

CTkcombobox



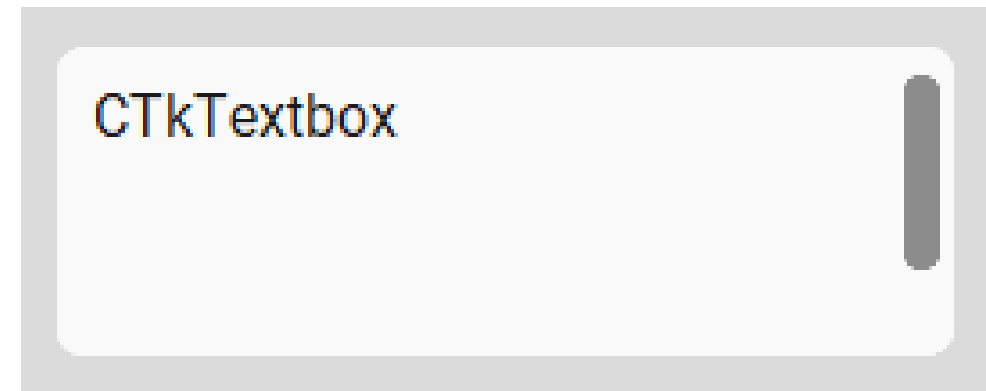
- > Listes de choix que l'utilisateur peut sélectionner.
- > On obtient l'élément choisi avec la méthode `get()`



```
combobox_1 = customtkinter.CTkComboBox(frame_1, values=["Option 1", "Option 2", "Option 42 long long long..."])  
combobox_1.set("CTkComboBox") # donne une valeur par défaut
```



- Peut être utilisé pour entrer plus d'informations que le widget CtkEntry
- Peut aussi être utilisé pour afficher de l'information.



```
text_1 = customtkinter.CTkTextbox(master=frame_1, width=200, height=70)
text_1.insert("0.0", "CTkTextbox\n\n\n\n")
```




- On peut choisir si on veut que l'utilisateur puisse entrer du texte en changeant la valeur de l'attribut "state"

```
text_1 = customtkinter.CTkTextbox(master=frame_1, width=200, height=70, state="disabled")

text_1.configure(state="normal")
text_1.insert("0.0", "CTkTextbox\n\n\n\n")
text_1.configure(state="disabled")
```

- state="normal" veut dire que l'utilisateur peut entrer du texte
- state="disabled" veut dire que le texte est affiché uniquement



GUI



Suite

Autres widgets



- > On a vu les:
 - > Les **Frames** pour structurer l'interface.
 - > Cinq widgets :
 - > **Label** pour afficher de l'information.
 - > **Entry** pour saisir une courte chaîne de caractères.
 - > **TextBox** pour entrer/afficher plus d'information.
 - > **Combobox** (liste déroulante) pour permettre de faire des choix parmi une liste.
 - > **Button** pour interagir avec l'interface.

Aujourd'hui :



- > On va voir trois nouveaux widgets :
 - > **CheckBox** pour faire des choix True / False
 - > **RadioButton** pour sélectionner une option parmi plusieurs.
 - > **Image** pour communiquer plus d'informations et rendre le GUI plus beau.



CheckBox (case à cocher)

- > Les cases à cocher permettent de faire une sélection booléenne, vrai ou faux

```
self.chk_1 = ctk.CTkCheckBox(master=frm_experience,  
.....text="Programmation Python",  
.....onvalue=1, offvalue=0)
```

- > onvalue : lorsque la case est cochée,
- > offvalue : lorsque la case n'est pas cochée
- > On obtient la valeur (onvalue ou offvalue) avec la méthode .get()

```
valeur_checkbox = self.chk_1.get()
```

<input type="checkbox"/>	Programmation Python
<input type="checkbox"/>	Programmation PowerShell
<input type="checkbox"/>	Maintenance des serveurs
<input type="checkbox"/>	Maintien des postes de travail
<input type="checkbox"/>	Maintien du réseau câblé et sans fil
<input type="checkbox"/>	Maintien des autres périphériques
<input type="checkbox"/>	Gestion des licences
<input type="checkbox"/>	Formation usagers
<input type="checkbox"/>	Gestion des accès usagers
<input type="checkbox"/>	Gestion des anti-virus
<input type="checkbox"/>	Gestion des télécommunications
<input type="checkbox"/>	Gestion des routeurs



RadioButton (boutons radio)

- Permettent de choisir entre plusieurs options.
- Les groupes de boutons radio sont définis par la variable à laquelle ils réfèrent.

```
choix_pizza = tk.StringVar(value="nature")
```

```
pizza_nature = ttk.Radiobutton(frm_pizza_choix, text='Nature', variable=choix_pizza, value='nature')  
pizza_vege = ttk.Radiobutton(frm_pizza_choix, text='Végétarienne', variable=choix_pizza, value='végétarienne')  
pizza_garnie = ttk.Radiobutton(frm_pizza_choix, text='Toute garnie', variable=choix_pizza, value='toute garnie')
```

- On obtient la valeur du groupe de boutons radios à partir de la variable à laquelle ils réfèrent grâce à la méthode .get()

```
choix_pizza_val = choix_pizza.get()
```

RadioButton (boutons radio)



```
choix_pizza = tk.StringVar(value="nature")
```

Classe importée de tkinter. Elle est similaire à la classe "str" mais fonctionne dans les widgets et possède la méthode .get()

```
pizza_nature = ttk.Radiobutton(frm_pizza_choix, text='Nature', variable=choix_pizza, value='nature')  
pizza_vege = ttk.Radiobutton(frm_pizza_choix, text='Végétarienne', variable=choix_pizza, value='végétarienne')  
pizza_garnie = ttk.Radiobutton(frm_pizza_choix, text='Toute garnie', variable=choix_pizza, value='toute garnie')
```

La même variable → ces boutons font partie du même groupe.
Un seul RadioButton peut être sélectionné à la fois.

```
choix_pizza_val = choix_pizza.get()
```

La méthode retourne la valeur sous forme de "str" qui peut ensuite être utilisé.

Images



- > Besoin d'un autre module :

```
> pip install pillow
```

```
from PIL import ImageTk, Image
```

- > Permet d'ajouter des images dans des labels ou dans des boutons.
- > Par standard, les images sont situées dans un répertoire "images" dans le même emplacement que notre script.

```
R25_Ex2_MyOrder
├── R25_YourOrder.py
└── images
    ├── Logo.jpg
    ├── pizza.jpg
    ├── poutine.jpg
    └── sousmarin.jpg
```


Images



- Par standard, les images sont situées dans un répertoire "images".
- On va chercher dans l'emplacement de l'image avec le module os.

```
R25_Ex2_MyOrder
├── R25_YourOrder.py
└── images
    ├── Logo.jpg
    ├── pizza.jpg
    ├── poutine.jpg
    └── sousmarin.jpg
```

```
self.image_path = os.path.join(os.path.dirname(os.path.realpath(__file__)), "images")
```

```
self.logo_image = ImageTk.PhotoImage(Image.open(os.path.join(self.image_path, "logomatissoft.jpg")))
```

- On peut ensuite récupérer l'image et la mettre dans un objet. Qu'on utilisera dans un label ou un bouton.

```
# lbl pour le logo de l'entreprise
```

```
self.lbl_logo = tk.Label(master=frm_container, image=self.logo_image)
```



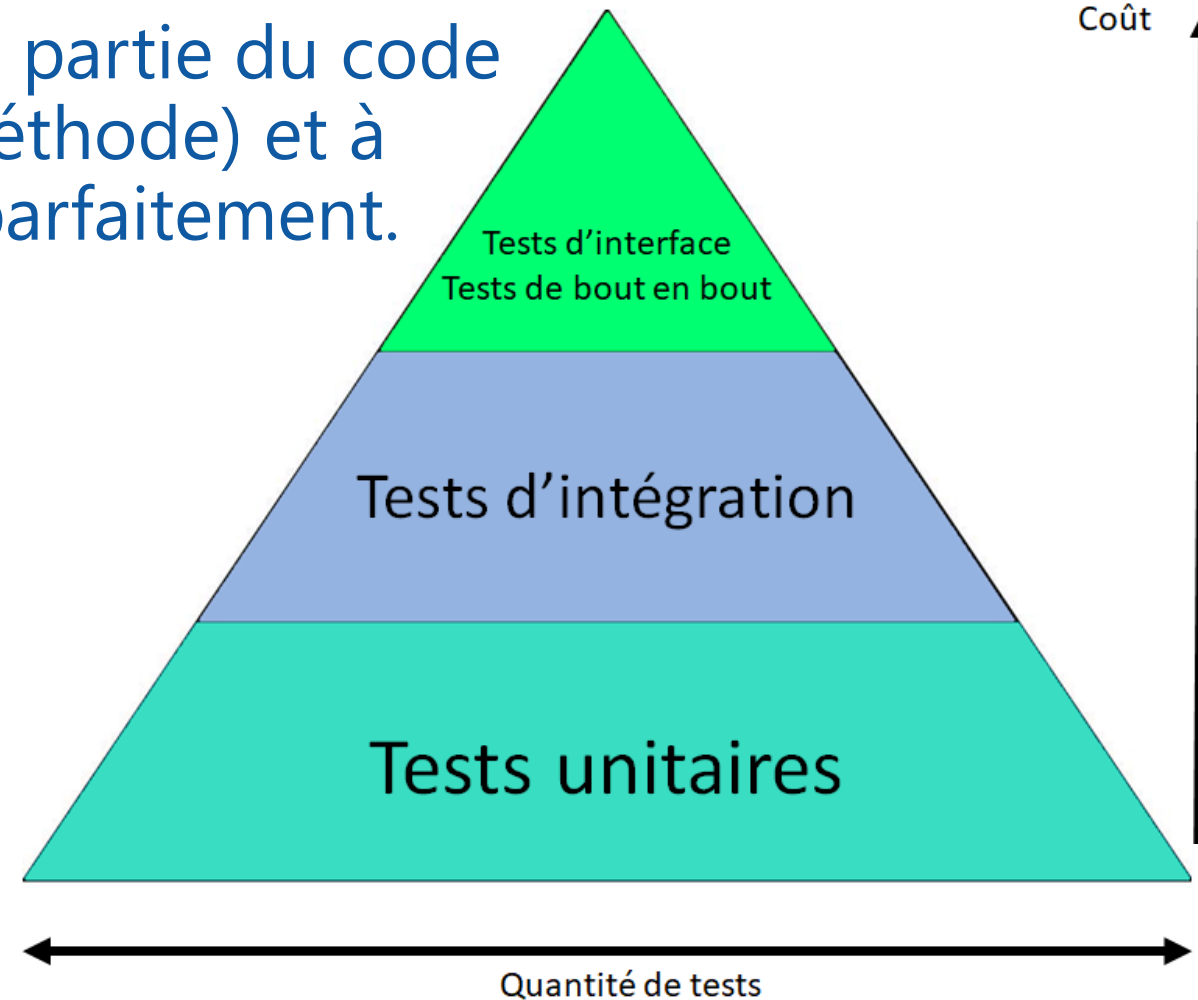
avec pythonTM

Tests Unitaires

Les tests unitaires



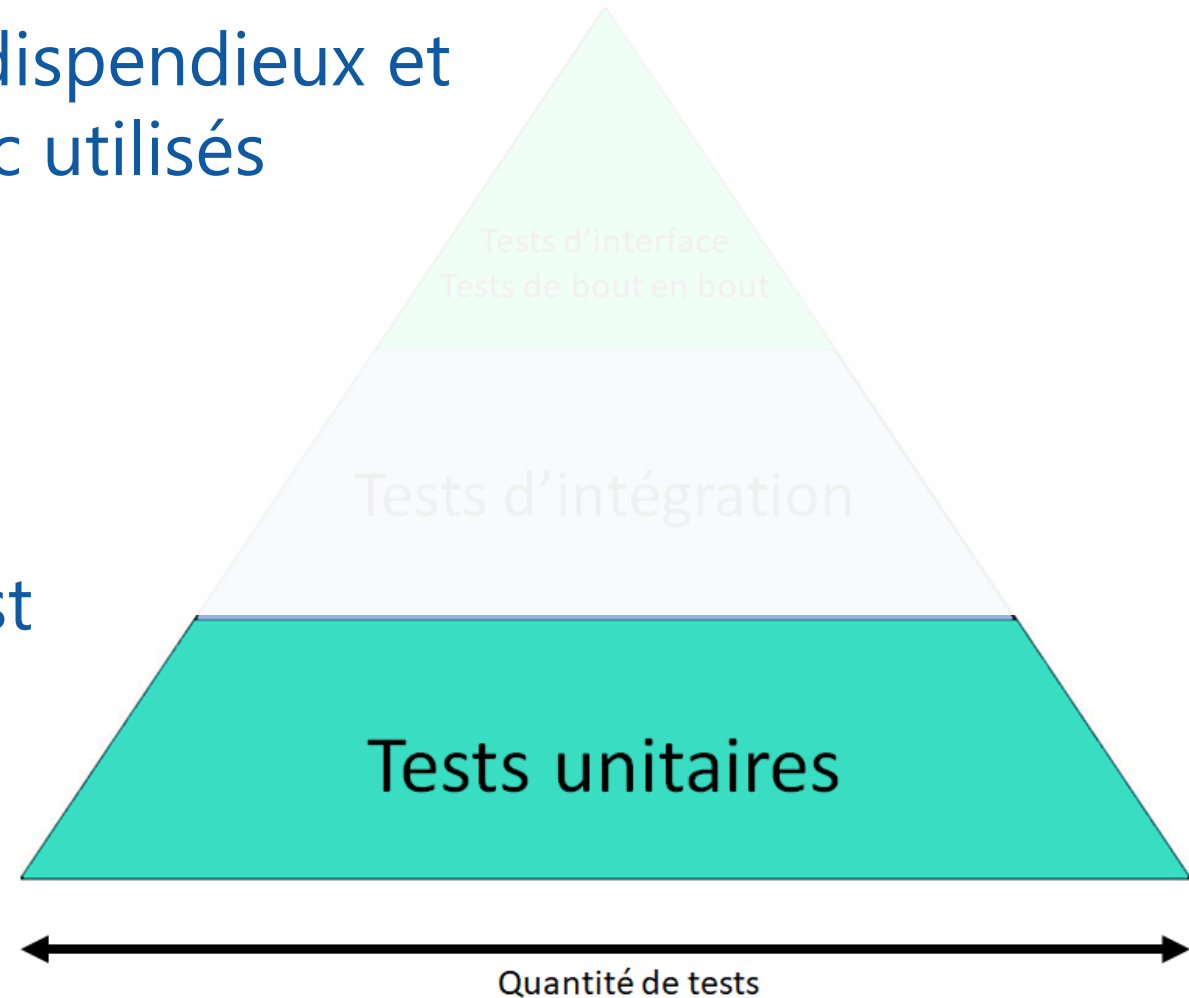
- Tests consistant à isoler une petite partie du code (généralement une fonction ou méthode) et à s'assurer que le code fonctionne parfaitement.
- UN test unitaire examine UNE seule chose à la fois
- Très utilisé lors du développement de logiciels



Les tests unitaires



- Les tests unitaires sont les moins dispendieux et les plus faciles à faire. Ils sont donc utilisés abondamment.
- Il y a 3 étapes à un test unitaire :
 - Préparer les données pour le test
 - Déclencher l'action à tester
 - Effectuer un **assert** pour vérifier le résultat de notre action.





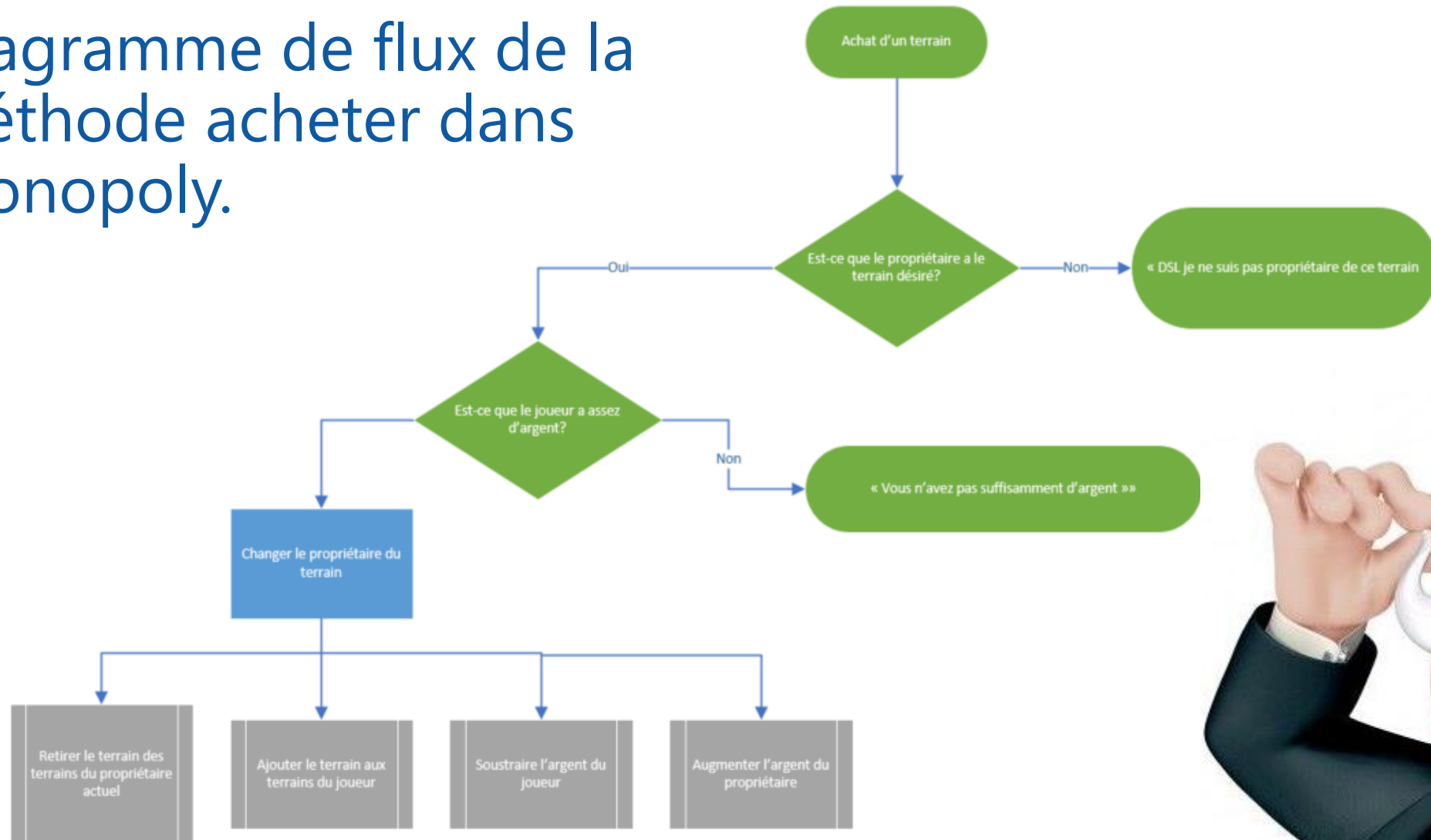
Tests unitaires - exemple

- Revenons à notre exemple d'un jeu de Monopoly.
- La classe joueur a une méthode qui lui permet d'acheter un terrain.

```
class Joueur:  
    ...def __init__(self, montant_cash, ls_terrains):  
    ...  
    ...pass  
  
    ...def acheter(self, proprietaire_actuel, terrain):  
    ...  
    ...pass
```

Tests unitaires - exemple

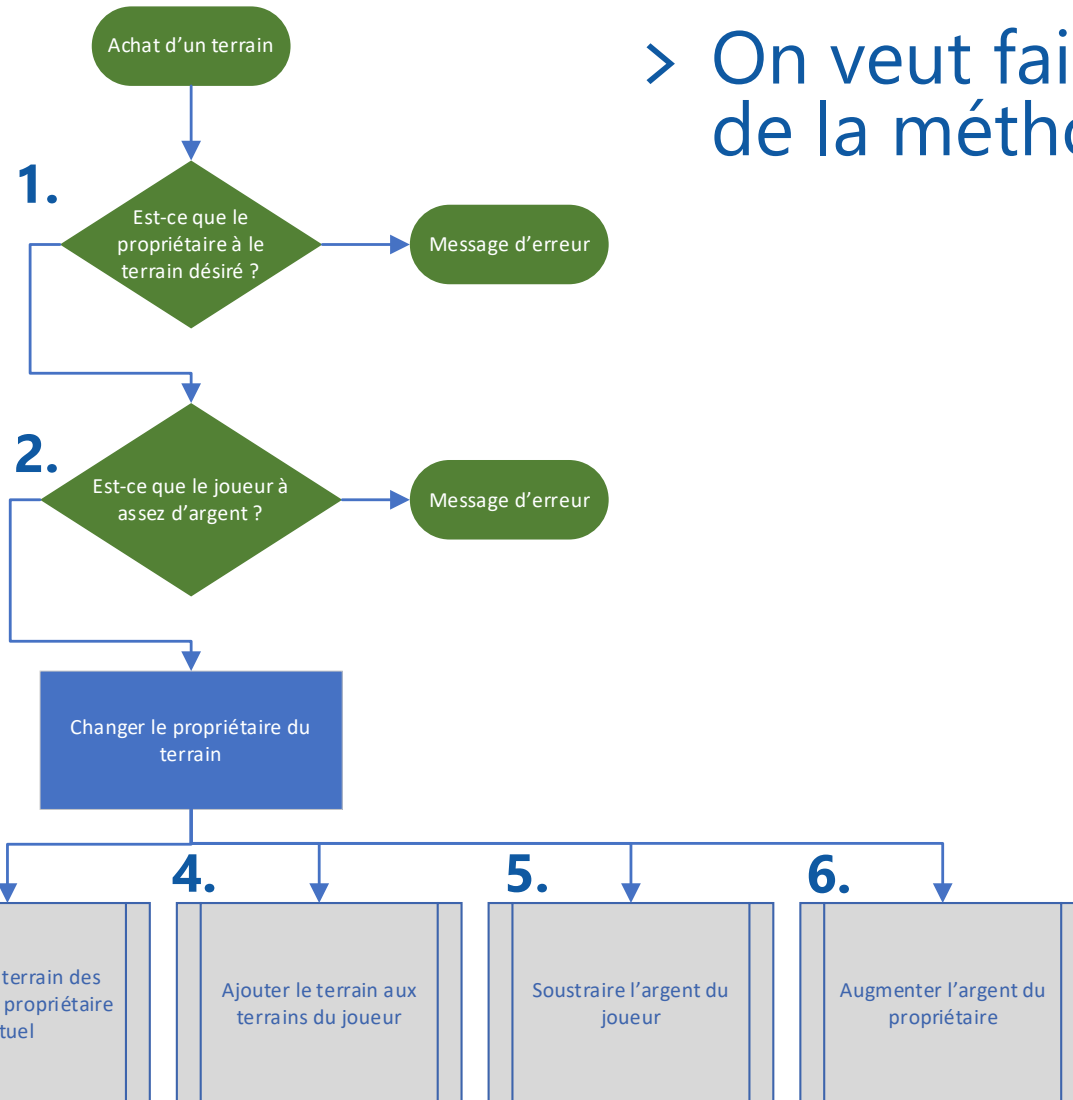
Diagramme de flux de la méthode acheter dans Monopoly.



Tests unitaires - exemple



> On veut faire un test pour chacune des portions de la méthode.

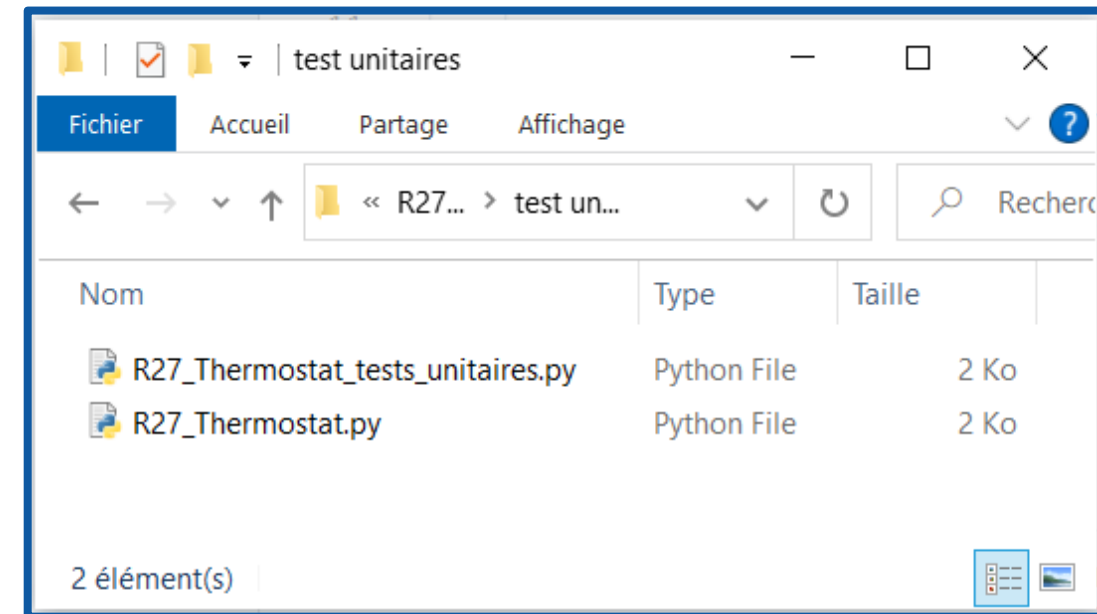


1. Est-ce qu'on vérifie bien que le propriétaire initial a le terrain ?
2. Est-ce qu'on vérifie bien que le joueur a assez d'argent ?
3. Est-ce qu'on retire bien le terrain du propriétaire actuel ?
4. Est-ce qu'on ajoute bien le terrain aux terrains du joueur qui achète ?
5. Est-ce qu'on soustrait bien l'argent de ce joueur ?
6. Est-ce qu'on donne bien cet argent au propriétaire initial ?

Module unittest



- Le module **unittest** fait partie de la librairie standard.
- Permet de créer et d'exécuter facilement des tests unitaires.
- Un nouveau fichier .py est créé pour les tests.
- Ce fichier va importer le module **unittest** ainsi que le script que nous voulons tester.





Structure de tests unitaires

```
1 import unittest
2
3 class test_methodes_string(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('FOO', 'foo'.upper())
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('Foo'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # Vérifie que le split échoue lorsqu'on
16        # ne sépare pas avec un caractère
17        with self.assertRaises(TypeError):
18            s.split(2)
19
20 if __name__ == '__main__':
21     unittest.main(verbosity=2)
```

- > Importation du module unittest
- > Création d'une classe pour faire une série de tests unitaires.
- > Création de méthodes, chacune testant UNE fonctionnalité.
- > Appel de la fonction .main() de unittest.
 - > Exécute tous les tests et fournis un rapport

Exécution des tests unitaires



```
24
25 ✓ if __name__ == '__main__':
26     |...unittest.main(verbosity=2)
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTER

```
test_isupper (__main__.test_methodes_string) ... ok
test_split (__main__.test_methodes_string) ... ok
test_upper (__main__.test_methodes_string) ... ok
```

Ran 3 tests in 0.001s

OK

- > Nombre de tests effectués
- > Temps d'exécution
- > Résumé (ok indiquant que tous les tests ont réussi)

- > Après avoir déclaré nos classes et méthodes test, on appelle la fonction `unittest.main()`
- > La fonction `unittest.main()` va exécuter chacune des fonctions de chaque classe et nous fournir un rapport.
- > Un test est réussi si tous les **asserts** contenus réussissent.

Exécution des tests unitaires



```
24
25 if __name__ == '__main__':
26     ...unittest.main(verbosity=2)
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL .NET INTERACTIVE JUPYTER

```
=====
FAIL: test_split (__main__.test_methodes_string)
-----
```

```
Traceback (most recent call last):
```

```
File "c:\Users\pierre-paul.gallant\Cégep Édouard-Montpetit\CMT-420_I
2 (Pilote réseau)\R26-27 - tests unitaires - tp 3\demo\test_unitaire_d
    with self.assertRaises(TypeError):
AssertionError: TypeError not raised
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (failures=1)
```

- > Si un test échoue, nous obtenons un message d'erreur.
- > Les autres tests qui suivent sont quand même exécutés
- > Une phrase résume l'**assert** qui a échoué.



Types de vérifications (assert)

```
1 import unittest
2
3 class test_methodes_string(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('FOO', 'foo'.upper())
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('Foo'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # Vérifie que le split échoue lorsqu'on
16        # ne sépare pas avec un caractère
17        with self.assertRaises(TypeError):
18            s.split(2)
19
20 if __name__ == '__main__':
21     unittest.main(verbosity=2)
```

- > **assertTrue** : Vérifie si le paramètre qu'on lui passe est résolu à une valeur **True**
- > **assertFalse** : Vérifie que le paramètre passé est résolu à une valeur **False**
- > **assertEqual** : Compare deux valeurs et vérifie si elles sont égales.
 - > Par standard : On place la valeur attendue en premier, puis la valeur évaluée.
 - > Ex: **assertEqual**(5, 2+2+1)
- > with **assertRaises** : Vérifie que le code contenu dans cette section soulève bien l'erreur ou l'exception attendue. (le test réussi uniquement si l'erreur est soulevée)



Les différentes méthodes **assert**

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

- > Il y a plusieurs autres types de **assert** selon ce qu'on désire évaluer.
- > Dans les exercices d'aujourd'hui, on se limitera aux plus fréquents (true, false, equals, raises



Sauté des tests

- > S'il y a des tests qu'on ne veut pas effectuer, on peut les sauter à l'aide de décorateurs **@unittest**
 - > **.skip** va sauter le test automatiquement
 - > **.skipif** va sauter le test uniquement si une condition est évaluée à vrai
 - > **.skipunless** va sauter le test à **moins** qu'une condition soit évaluée à vrai.
- > La méthode **.skipTest()** va permettre de sauter un test après qu'il est été commencé.

```
1  import unittest
2  import sys
3
4  version_python = sys.version
5
6  class MyTestCase(unittest.TestCase):
7
8      @unittest.skip("skip à l'aide de décorateurs")
9      def test_nothing(self):
10         self.fail("ne devrait pas se rendre ici")
11
12         @unittest.skipIf(version_python < "3.10",)
13         def test_format(self):
14             pass
15
16         @unittest.skipUnless(sys.platform.startswith("win"))
17         def test_windows_support(self):
18             pass
19
20         def test_maybe_skipped(self):
21             if "situation plus complexe que prévu":
22                 self.skipTest("Hors du cadre de ce test")
23             pass
```



Les constantes dans python

- > Les constantes ne font pas partie de python contrairement à d'autres langages de programmation.
- > Les constantes fonctionnent selon un standard.
De façon similaire aux attributs privés, il s'agit d'une entente entre programmeurs quant à leur utilisation.
- > Les constantes sont placées dans un fichier constant.py
- > Leurs noms sont entièrement en majuscule, ex : MIN_TEMPERATURE
- > On ne modifie jamais la valeur d'une constante.

Automatisation des tests



- Les tests unitaires permettent entre autres l'automatisation des tests.
- Fréquent suivant chaque commit ou avant de merge dans une branche.
- Peut-être effectué à des intervalles réguliers si on interagit beaucoup avec d'autres services (services web, api, etc.)
- Vous en verrez un peu plus dans les prochains cours, yay !



Le Développement Dirigé par Tests

- Le TDD (Test Driven Development) consiste à faire les tests en premier.
- Essentiellement :
 1. Identifier un besoin.
 2. Créer un test juste assez grand pour qu'il échoue.
 3. Créer juste assez de code/méthodes pour que le test soit un succès.
 4. Répéter jusqu'à ce que le projet soit terminé ou que weekend arrive.
- D'autres types de tests sont très utilisés dans l'industrie. Telle que les tests fonctionnels et les tests d'intégrations... vous en verrez quelques-uns dans les prochains cours.