2N6 Programmation 2

avec python™

Programmation orienté objet (OOP)

Classes et instanciations

Début héritage

Qu’est-ce que la programmation objet ?

- Basé sur 4 concepts fondamentaux :
  - Encapsulation
  - Abstraction
  - Héritage
  - Polymorphisme

Principes de programmation

- Procédurale :
  - Série d'appels de fonctions
  - Permets l'utilisation de modules
  - Ne fais pas d'encapsulation.
    - Toutes les variables sont accessibles.
    - Niveau d'abstraction limité.
  - Permet réutilisation de code
- Orienté Objet :
  - Encapsulation des données et fonctions
  - Représentation des parties d'un programme comme des objets différents.
  - Permet réutilisation de code
  - Haut niveau d'abstraction.
    - Pas besoin de connaitre les mécanismes internes d'un objet pour l'utiliser.
  - Héritage et polymorphisme :
    - Création de nouvelles classes à partir de classes existantes.

La programmation orientée objet (OOP)

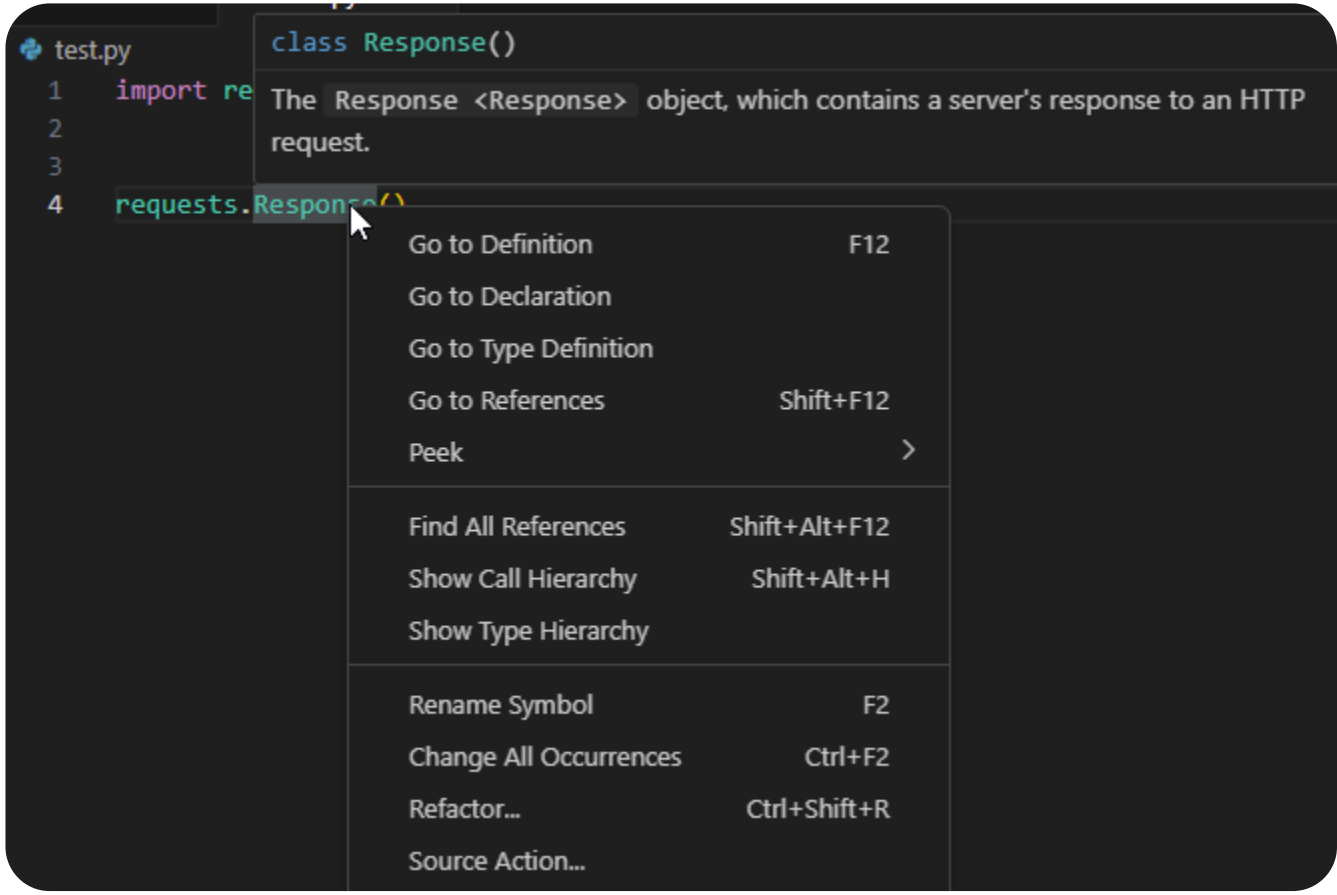
- Un principe de programmation. On regroupe les données et les fonctions de sorte qu'elles soient faciles à réutiliser et à modifier selon les besoins.
- On crée des Classes qui sont des schémas pour créer des objets.
- Dans les classes on écrit les propriétés qui vont décrire les objets futurs et les 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.

La programmation orientée objet (OOP)

- Un principe de programmation.
  - Regroupe les données et les fonctions.
    - Elles deviennent faciles à réutiliser et à modifier selon les besoins.
  - Utilisation de 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.

Exemple d'objet

- Nous avons déjà utilisé des instances de la classe Response
- On peut voir la définition de la classe dans vscode
- Cette classe permet de crée des objets de type <Response> qui nous sont retourné lors de requêtes

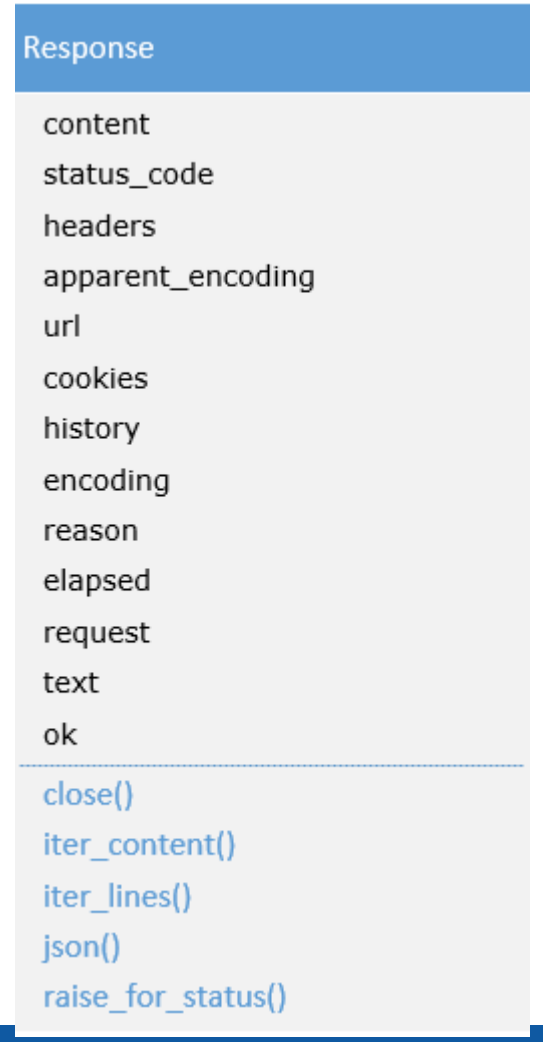


Exemples : w3schools.com/python/module_requests.asp
w3schools.com/python/ref_requests_response.asp

Définition accessible par VScode

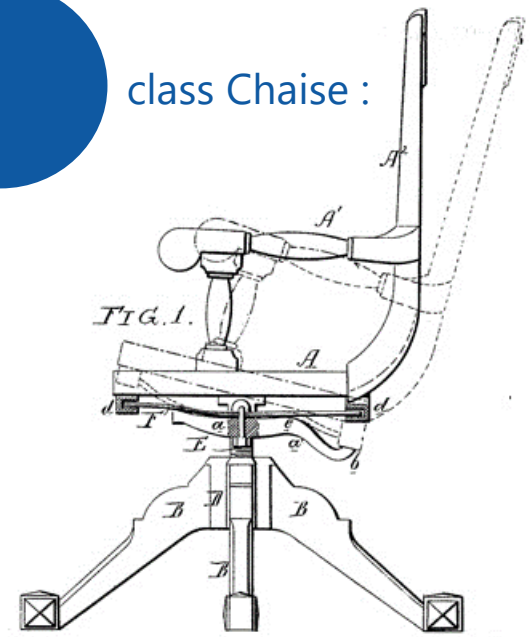
```
640
641 class Response:
642     """The :class:`Response <Response>` object, which contains a
643     server's response to an HTTP request.
644     """
645
646     __attrs__ = [
647         "_content",
648         "status_code",
649         "headers",
650         "url",
651         "history",
652         "encoding",
653         "reason",
654         "cookies",
655         "elapsed",
656         "request",
657     ]
658
659     def __init__(self):
660         self._content = False
661         self._content_consumed = False
662         self._next = None
663
664         #: Integer Code of responded HTTP Status, e.g. 404 or 200.
665         self.status_code = None
666
667         #: Case-insensitive Dictionary of Response Headers.
668         #: For example, ``headers['content-encoding']`` will return the
669         #: value of a ``'Content-Encoding'`` response header.
```

Schéma UML de la classe



Classes

class Chaise :



- Les Classes sont des **schémas** qui nous permettront ensuite de créer des objets.
- Un objet crée à partir d'une classe est une **instance** de cette classe.
- La classe Chaise() est le "plan" qui nous permet de construire plusieurs objets de type <Chaise>

Trois instances : chaise1 chaise2 chaise3

Classes

Le **self** réfère à l'objet qui est en train d'être crée / à l'instance courante de la classe.

Rappel : la classe est le "blueprint" qui permet de crée des objets (instances) qui seront utilisable.

```
class Chaise:
    def __init__(self, largeur, profondeur, hauteur):
        self.largeur = largeur
        self.profondeur = 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 tous par une majuscule par convention

retour

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.profondeur = 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

retour

- 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.

```
(largeur, profondeur, hauteur) -> None
chaise_1 = Chaise(40, 40,110)
chaise_2 = Chaise(40, 40, 110)

chaise_2.ajuster_hauteur(80)
```

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

Utiliser les propriétés et méthodes d'un objet

retour

```
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 TERMINAL

```
hauteur 1:  110
hauteur 2:  110
hauteur 1:  110
hauteur 2:  80
```

Création d'objets avec des valeurs par défauts

retour

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

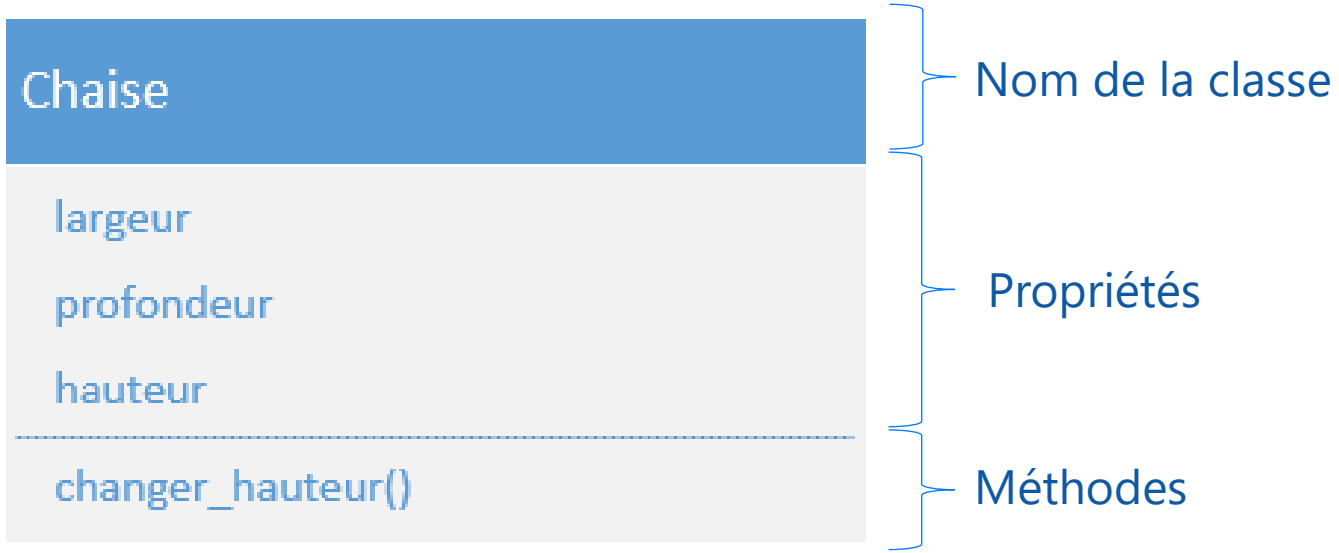
```python
chaise_3 = Chaise()
chaise_4 = Chaise(35, 45, 95)

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

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

```
PROBLÈMES    SORTIE    TERMINAL

chaise 3 :
40 40 110
chaise 4 :
35 45 95
```

Modélisation UML

retour

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



Les variables de classes

- Différentes des variables d'**instances**
- Appartiennent à la **classe** elle-même et non à chaque objet crée (instance)
- Donc : même valeur commune à toutes les instances

Variables de classes

- Ici une classe Employe possédant uniquement trois attributs, "nom", "prenom" et "ID"
- Ce sont des variables d'**instance**
  - Chaque objet de type <Employe> aura ses propres valeurs de nom, prenom, ID

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

Variables de classes

```python
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
Ces variables appartiennent à la **classe** elle-même.

Variables appartenant à l'objet créé à partir de la classe Employe (l'**instance**)

Ici on modifie les variables de classe. **Toutes les instanciations accèderont aux mêmes variables de classe**

Variables de classes

- Une variable de classe est accessible directement à partir de la classe ou bien à partir d'une instanciation.
- 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}')
```

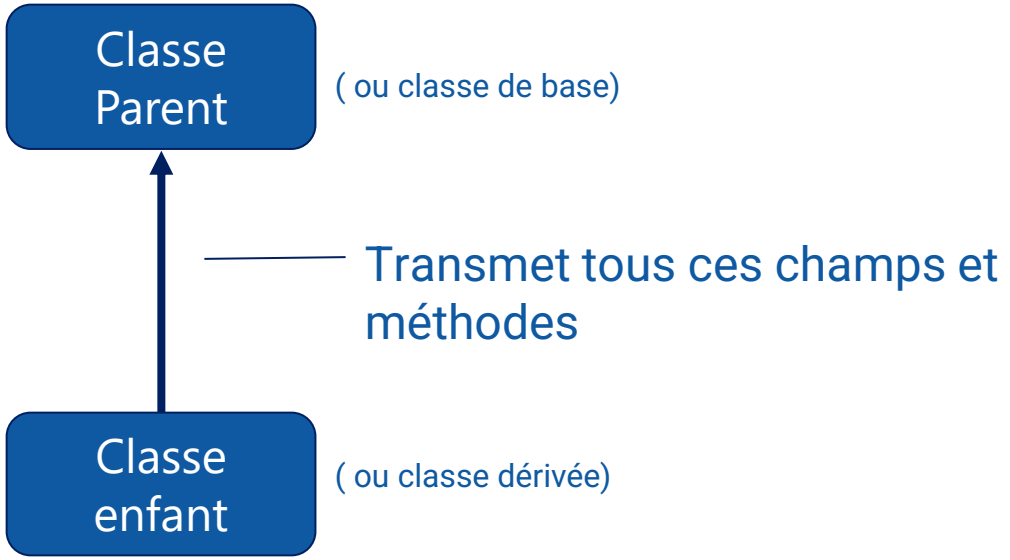
```
PROBLÈMES 1   SORTIE   TERMINAL   ...

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

L'héritage

- Permet de définir une nouvelle classe à partir d'une classe existante
- Permet la réutilisation de code d'une classe à l'autre
- Permet le polymorphisme

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

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

classe parent
ou classe mère
ou classe de base

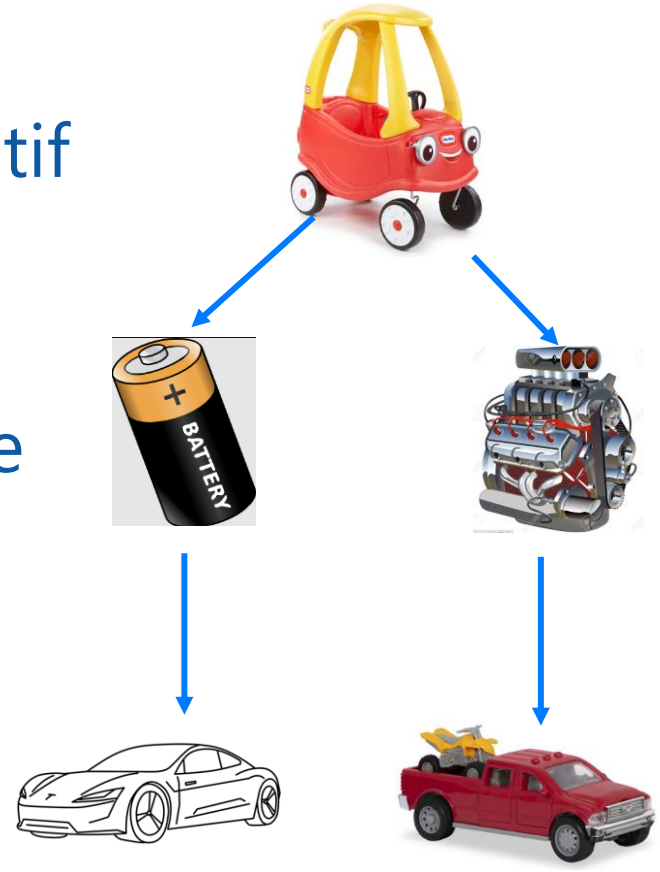
```python
class Programmeur(Employe):
    def __init__(self, nom, prenom,language_favori) -> None:
        super().__init__(nom, prenom)
        self.language_favori = language_favori
```

classe enfant
ou classe fille
ou classe dérivée

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.

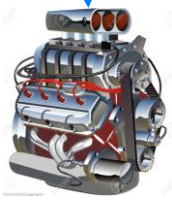



```python
class Voiture:
    def __init__(self,marque) -> None:
        self.marque = marque
```

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

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

Héritage transitif



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

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

```python
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

```python
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

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

    def klaxon(self):
        print("honk!")
```

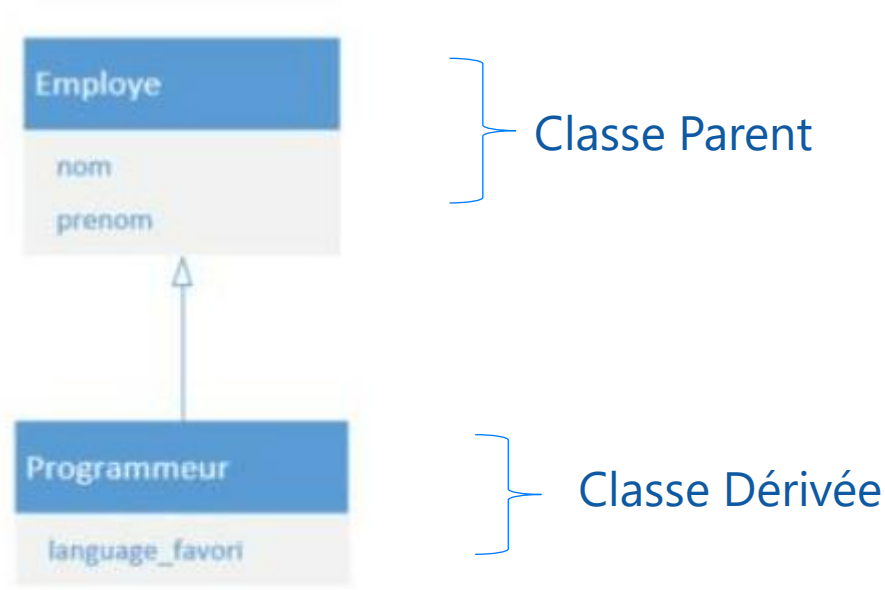
```python
class Pickup(Voiture_moteur):
    def __init__(self, marque, reservoir,puissance):
        super().__init__(marque, reservoir)
        self.puissance = puissance

remorque = Pickup("Ford","60L","1200hp")
remorque.klaxon()
```

```
PROBLÈMES 1   TERMINAL   ···   Python

honk!
```

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

Modélisation UML

- L'héritage est illustré ainsi dans UML



Un Programmeur EST un Employe