CHAPITRE 11 PROGRAMMATION ORIENTÉE OBJET (POO)

Programmation en Python

1 Introduction

Programmation objet

- La **programmation objet** est un paradigme de programmation.
- L'idée générale consiste à concevoir des concepts, idées ou toutes entités du monde physique comme des « objets ». Les objets possèdent des structures internes, des comportements et peuvent interagir avec leur pairs.

Classe

- Intuitivement, une **classe** est une sorte d'usine, de boîte à outil ou de moule à partir de laquelle il est possible de créer des **objets.**
- Une **classe** peut également être vue comme un concept ou une idée (au sens platonicien).
- **Exemples:** la classe des Humains; la classe des Voitures; la classe des Entiers ('int' en python).
- Plus formellement, une **classe** est un modèles de données qui partagent des caractéristiques communes: **attributs, méthodes,…**

Objet

- Intuitivement, un **objet** est un élément spécifique d'une classe, i.e., un élément créé, fabriqué ou issu d'une classe.
- **Exemples:** Adam est un objet la classe des Humains; une Rolls-Royce et un objet de la classe des Voitures; 3 est un objet la classe des Entiers.
- Plus formellement, une **objet** est une instance d'une classe (un conteneur symbolique et autonome qui regroupe des informations et des mécanismes).

Attributs

- Les **attributs** sont des caractéristiques ou propriétés qui définissent les objets issus d'une classe.
- **Exemples:** les objets la classe des Humains peuvent être définis par les attributs prénom, nom, génôme, etc. Les objets de la classe des Voitures peuvent être définis par les attributs marque, année de construction, puissance, etc.
- Plus formellement, les **attributs** sont un ensemble de variables partagées par tous les objets d'une classe.

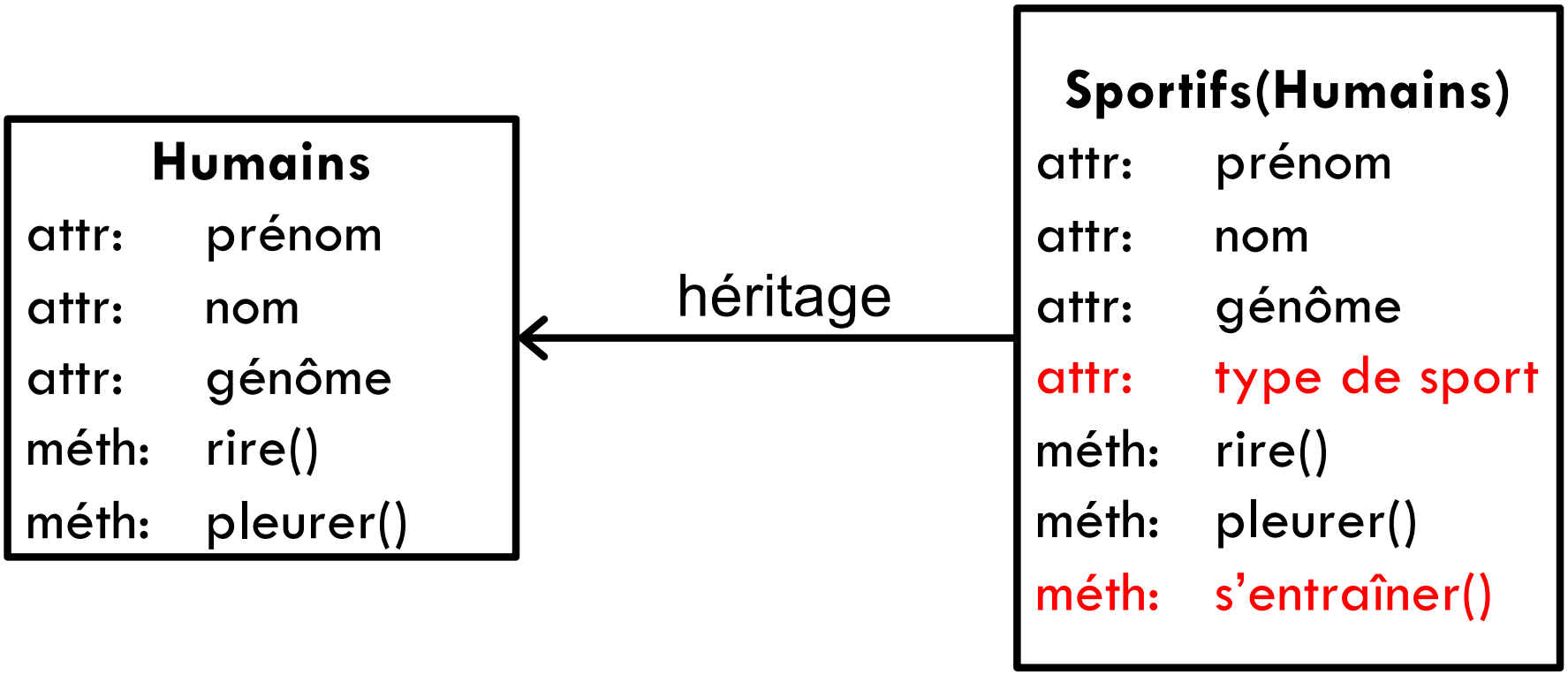
Méthodes

- Intuitivement, les **méthodes** sont des mécanismes qui agissent sur les objets d'une classe.
- **Exemples:** les objets la classe des Humains peuvent posséder les méthodes rire, pleurer. etc. Les objets de la classe des Voitures peuvent posséder les méthodes avancer, reculer, tomber en panne, etc.
- Plus formellement, les **méthodes** sont des fonctions qui agissent sur les objets d'une classe (et transforment potentiellement leur attributs).

Héritage

- L'**héritage** est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe.
- Une classe-fille ou sous-classe **hérite** des attributs et méthodes d'une classe-mère ou super-classe.
- **Exemple:** la classe des Humains possède les attributs prénom, nom, génôme et les méthodes rire, pleurer. On pourrait définir la sous-classe des Sportifs – des Humains particuliers – qui aurait un attribut supplémentaire type de sport et une méthode supplémentaire s'entraîner.

Héritage


Humains
attr: prénom
attr: nom
attr: génôme
méth: rire()
méth: pleurer()
héritage
Sportifs(Humains)
attr: prénom
attr: nom
attr: génôme
attr: type de sport
méth: rire()
méth: pleurer()
méth: s'entraîner()

2

Classes et objets

Classe

- L'instruction pour définir une **classe** est :

```
class NomClasse:
    """
    description facultative
    """
```

Exemple


```
class Humain:
    """
    classe de tous les humains
    """
    pass
```

Constructeur

- Le **constructeur** est une méthode spécifique qui est automatiquement exécutée lors de l'instanciation de tout objet de la classe.
- C'est la fonction qui crée l'objet à partir de la classe. Sa syntaxe est :

```
class NomClasse:
        def __init__(self):
                ...
```

- Le paramètre **`self`** signifie que la méthode **`__init_()`** s'applique à l'objet lui-même (to itself)

Objet

- Une fois le constructeur défini, on peut instancier (i.e., créer) des **objets** de la classe grâce à la syntaxe suivante :

```
obj1 = NomClasse()
obj2 = NomClasse()
```

...

- Les objets crées sont alors stockés dans les variables appelées `obj1`, `obj2`.

Exemple


```
class Humain:
    """
    classe de tous les humains
    """
    # constructeur
    def __init__(self):
        print("création d'un objet humain...")
```

Exemple


```
# création d'un objet de la classe Humain
# l'objet est stocké dans une variable h1
# le constructeur est exécuté lors de la création
h1 = Humain()
```

```
création d'un objet humain...
```

3

Attributs

Attribut d'instance

- Les **attributs (d'instances)** sont des variables partagées par tous les objets de la classe. Ils sont définis dans le constructeur :

```
class NomClasse:
    def __init__(self):
        self.attribut1 = valeur1
        self.attribut2 = valeur2
        ...
```

- Le paramètre **`self`** signifie que les attributs sont liés à l'objet lui-même (to itself)

Exemple


```
class Humain:
    # constructeur
    def __init__(self):
        print("création d'un objet humain...")
        # attributs d'instances
        self.prenom = "Adam" # attribut prenom
        self.age = 33        # attribut age
```

Attribut d'instance

- On accède aux valeurs des **attributs** d'un objet `objet` via la syntaxe :

```
objet.attribut1
objet.attribut2
...
```

Exemple


```
# création d'un objet de la classe Humain
h1 = Humain()
création d'un objet humain...

# affichage des attributs
print("prenom:", h1.prenom)
prenom: Adam
print("age:", h1.age)
age: 33
```

Exemple


```
# on peut changer les valeurs des attributs!
h1.prenom = "Eve" # réaffectation attribut
h1.age = 27       # réaffectation attribut

# affichage des nouvelles valeurs des attributs
print("prenom:", h1.prenom)
prenom: Eve
print("age:", h1.age)
age: 27
```

Constructeur avec paramètres

- Le **constructeur** peut prendre des paramètres. En général, ces paramètres servent à définir les attributs de l'objet lors de sa création :

```
class NomClasse:
    def __init__(self, par1, par2,...):
        self.attribut1 = par1
        self.attribut2 = par2
        ...
```

- Dans ce cas, il faudra spécifier la valeurs de ces paramètres lors de l'instanciation de tout objet.

Exemple


```
class Humain:
    # constructeur avec paramètres
    def __init__(self, prenom_cstr, age_cstr):
        self.prenom = prenom_cstr
    # attribut prenom: valeur du param. prenom_cstr
        self.age = age_cstr
    # attribut age: valeur du param. age_cstr
```

Exemple


```
# création d'objets de la classe Humain
# on donne des valeurs aux paramètres à l'instanciation
h1 = Humain("Eve", 35)
h2 = Humain("Adam", 33)
# affichage des attributs
print("prenom:", h1.prenom)
prenom: Eve
print("age:", h1.age)
age: 35
print("prenom:", h2.prenom)
prenom: Adam
print("age:", h2.age)
age: 33
```

Exemple


```
class Humain:
    # constructeur avec param. et valeurs par défaut
    def __init__(self,
                 prenom_cstr = "Adam",
                 age_cstr = 0):
        self.prenom = prenom_cstr # attribut prenom
        self.age = age_cstr       # attribut age
```

Exemple


```
# création d'objets de la classe Humain
h1 = Humain(age_cstr = 33) # prenom_cstr = "Adam" par déf.
h2 = Humain(prenom_cstr = "Eve") # age_cstr = 0 par déf.
# affichage des attributs
print("prenom:", h1.prenom)
prenom: Adam
print("age:", h1.age)
age: 33
print("prenom:", h2.prenom)
prenom: Eve
print("age:", h2.age)
age: 0
```

Exemple


```
# on peut les modifier les attributs
h1.prenom = "Noe" # réaffectation
h1.age = 100      # réaffectation

# affichage des nouvelles valeurs des attributs
print("prenom:", h1.prenom)
prenom: Noe
print("age:", h1.age)
age: 100
```

Attribut de classe

- Les **attributs de classe** sont des variables associées directement à la classe et non à ses objets. Ils sont définis hors du constructeur (pas de mot-clé **`self`**) :

```
class NomClasse:
    attribut_de_classe = valeur
    def __init__(self, par1, par2,...):
        self.attribut1 = par1
        self.attribut2 = par2
        ...
```

Attribut de classe

- On accède aux valeurs des **attributs de classe** via la syntaxe :

```
NomClasse.attribut1
NomClasse.attribut2
...
```

Exemple


```
class Humain:
    nb_humains = 0 # attribut de classe
    # constructeur
    def __init__(self,
                 prenom_cstr = "Adam"
                 age_cstr = 0):
        self.prenom = prenom_cstr
        self.age = age_cstr
        # incrémentation de l'attribut de classe
        # à chaque appel du constructeur
        Humain.nb_humains += 1
```

Exemple


```
# création d'objets de la classe Humain
h1 = Humain("Alice", 31)
print("Nombre d'objets crées:", Humain.nb_humains)
Nombre d'objets crées: 1
h2 = Humain("Bob", 32)
print("Nombre d'objets crées:",Humain.nb_humains)
Nombre d'objets crées: 2
h3 = Humain("Chloe", 33)
print("Nombre d'objets crées:", Humain.nb_humains)
Nombre d'objets crées: 3
```

Exemple


```
# on peut définir une multitude d'objets
# et les stocker dans une liste p.ex.
ma_liste = []
for i in range(1000):
    ma_liste.append(Humain("Adam_" + str(i), 33))

# affichage des attributs du 104ème humain
print("prenom:", liste[104].prenom)
prenom: Adam_104
print("age:", liste[104].age)
age: 33
```

4 Méthodes

Méthode d'instance

- Une **méthode (d'instance)** est une fonction définie à l'intérieur d'une classe et qui agit sur les objets de la classe :
- Pour indiquer que la méthode agit sur les objets, on donne Le paramètre **`self`**.

```
class NomClasse:
    def __init__(self):
        ...
    def methode(self,par1,par2,...):
        ...
```

Méthode d'instance

- Pour appliquer une méthode (d'instance) `methode()` à un objet `objet`, on utilise la syntaxe :

```
objet.methode(par1,par2,...)
```

Exemple


```
class Humain:

    lieu_habitation = "Terre" # attribut de classe

    def __init__(self, nom, age):
        self.nom = nom # attribut (d'instance)
        self.age = age # attribut (d'instance)

    def parler(self, message): # méthode (d'instance)
        print(self.nom + " dit: " + message)
```

Exemple


```
# création d'un objet
h1 = Humain("Dan", 23)
# application d'une méthode (d'instance)
h1.parler("Salut!")
Dan dit: Salut!
# application d'une méthode (d'instance)
h1.parler("Au revoir...")
Dan dit: Au revoir...
```

Méthode de classe

- Une **méthode de classe** est une fonction définie à l'intérieur d'une classe, mais qui agit sur la classe totale (ses attributs), au lieu d'agir sur ses objets.
- Pour indiquer que la méthode agit sur la classe, on donne le paramètre **`cls`** ainsi qu'une instruction supplémentaire **`classmethod()`:**

```
class NomClasse:
    def __init__(self):
        ...
    def methode_c(cls,par1,par2,...):
        ...
    methode_c = classmethode(methode_c)
```

Méthode de classe

- Pour appliquer une méthode de classe `methode_c()` à une classe `classe`, on utilise la syntaxe :

```
classe.methode_c(par1,par2,...)
```

Exemple


```
class Humain:

    lieu_habitation = "Terre" # attribut de classe

    def __init__(self, nom, age):
        self.nom = nom # attribut (d'instance)
        self.age = age # attribut (d'instance)

    # méthode de classe
    def changer_planete(cls, nouvelle_planete):
        Humain.lieu_habitation = nouvelle_planete

    changer_planete = classmethod(changer_planete)
```

Exemple


```
# application d'une méthode de classe
print(Humain.lieu_habitation)
Terre
Humain.changer_planete("Mars")
print(Humain.lieu_habitation)
Mars
```

Méthode statique (plus rare)

- Une **méthode statique** est une fonction définie à l'intérieur d'une classe, mais qui n'agit ni sur les objets, ni sur les attributs de classe.
- Pour indiquer que la méthode est statique, on donne une instruction supplémentaire **`staticmethod()`:**

```
class NomClasse:
    def __init__(self):
        ...
    def methode_s(par1,par2,...):
        ...
    methode_s = staticmethode(methode_s)
```

Méthode statique (plus rare)

- Pour appliquer une méthode statique `methode_s()` qui est définie dans la classe `classe`, on utilise la syntaxe :

```
classe.methode_s(par1,par2,...)
```

Exemple


```
class Humain:

    lieu_habitation = "Terre" # attribut de classe

    def __init__(self, nom, age):
        self.nom = nom # attribut (d'instance)
        self.age = age # attribut (d'instance)

    # méthode statique
    def message(): # méthode statique
        print("Les humains ne sont pas les seuls \
      êtres vivants à éprouver de la sensibilité.")

    message = staticmethod(message)
```

Exemple


```
# application d'une méthode statique
# la méthode s'applique à la classe, mais ne modifie
# rien – ou n'agit sur rien – dans cette classe
Humain.message()
```

Les humains ne sont pas les seuls êtres vivants à éprouver de la sensibilité.

5

Propriétés

Propriétés

- En Python, les **propriétés** permettent de contrôler l'accès à certains attributs d'instance.
- Une propriété permet de contrôler :
  - l'accession à la valeur d'un attribut : **accessor**
  - la modification de la valeur d'un attribut : **mutator**
  - la suppression d'un attribut : **deleter**
  - La documentation d'un attribut : **helper**

Propriétés

- En python, une **propriété** est une fonction prédéfinie qui retourne un objet `property` qui prend en paramètre (au plus) *4* méthodes.
- Pour contrôler un attribut `attribut` via une propriété, on utilise les syntaxes :

```
self._attribut = attribut                    (constructeur)
attribut =                                   (corps de la classe)
property(getter, setter, deleter, helper)
```

où `getter`, `setter`, `deleter`, `helper`, sont *4* méthodes définies dans le corps de la classe.

Propriétés

- La méthode `getter` est appelée lors de l'accession à la valeur de attribut (`objet.attribut`) : **accessor**
- La méthode `setter` est appelée lors de la modification de la valeur de attribut : **mutator** (`objet.attribut = nouvelle_valeur`)
- La méthode `deleter` est appelée lorsqu'on désire supprimer un attribut : **deleter**
- La méthode `helper` permet d'en registrer une documentation de l'attribut **helper**

Exemple


```
class Humain:

    def __init__(self, nom, age):
        self.nom = nom # attribut (d'instance)
        self.age = age # attribut (d'instance)
```

Exemple


```
# création d'un objet
h1 = Humain("Adam", 30)
print(h1.nom, h1.age)
Adam 30
# modification des attributs
h1.prenom = "Noe" # réaffectation
h1.age = 100      # réaffectation
# affichage des nouvelles valeurs des attributs
print(h1.nom, h1.age)
Noe 100
```

Exemple


```
class Humain:

    def __init__(self, nom, age):
        self.nom = nom  # attribut
        self._age = age # attribut _age -> propriété age

    def _get_age(self):
        """accessor"""
        if not isinstance(self._age, int):
            print("Type de l'âge non conforme")
            print("Entrez un entier pour l'âge")
        return self._age
```

Exemple


```
def _set_age(self, nouvel_age):
    """mutator"""
    if nouvel_age < 0:
        print("âge négatif, je le remplace par 0")
        self._age = 0
    else:
        self._age = nouvel_age

def _del_age(self):
    """deleter"""
    del self._age
```

Exemple


```
# la propriété age encapsule la valeur
# de l'attribut _age (cf. constructeur)
age = property(_get_age, _set_age,
               _del_age, "age de l'humain")
```

Exemple


```
# création d'un humain
h1 = Humain("Adam", 33)
print(h1.age) # appel la propriété _get_age
33
# création d'un humain
h2 = Humain("Eve", 2.3)
print(h2.age) # appel la propriété _get_age
Type de l'âge non conforme
Entrez un entier pour l'âge
2.3
```

Exemple


```
# création d'un humain
h1 = Humain("Adam", 33)
print(h1.age) # appel la propriété _get_age
33
h1.age = 22    # appel la propriété _set_age
               # pour modifier l'attribut _age
print(h1.age) # appel la propriété _get_age
22
h1.age = -2    # appel la propriété _set_age
               # pour modifier l'attribut _age
âge négatif, je le remplace par 0
print(h1.age) # appel la propriété _getage
0
```

Exemple


```
# création d'un humain
h1 = Humain("Adam", 33)
print(h1.age) # appel la propriété _get_age
33
del h1.age    # appel la propriété _del_age
print(h1.age) # appel la propriété _get_age
---------------------------------------------------------------
AttributeError                Traceback (most recent call last)
<ipython-input-263-30d3143f7859> in <module>()
      3 print(h1.age) # appel la propriété _get_age
      4 del h1.age    # appel la propriété _del_age
----> 5 print(h1.age) # appel la propriété _get_age
...
```

Exemple


```
# création d'un humain
h1 = Humain("Adam", 33)
help(Humain.age) # appel la propriété <helper> (dernier argument de age)
```

```
Help on property:
    age de l'humain
```

6

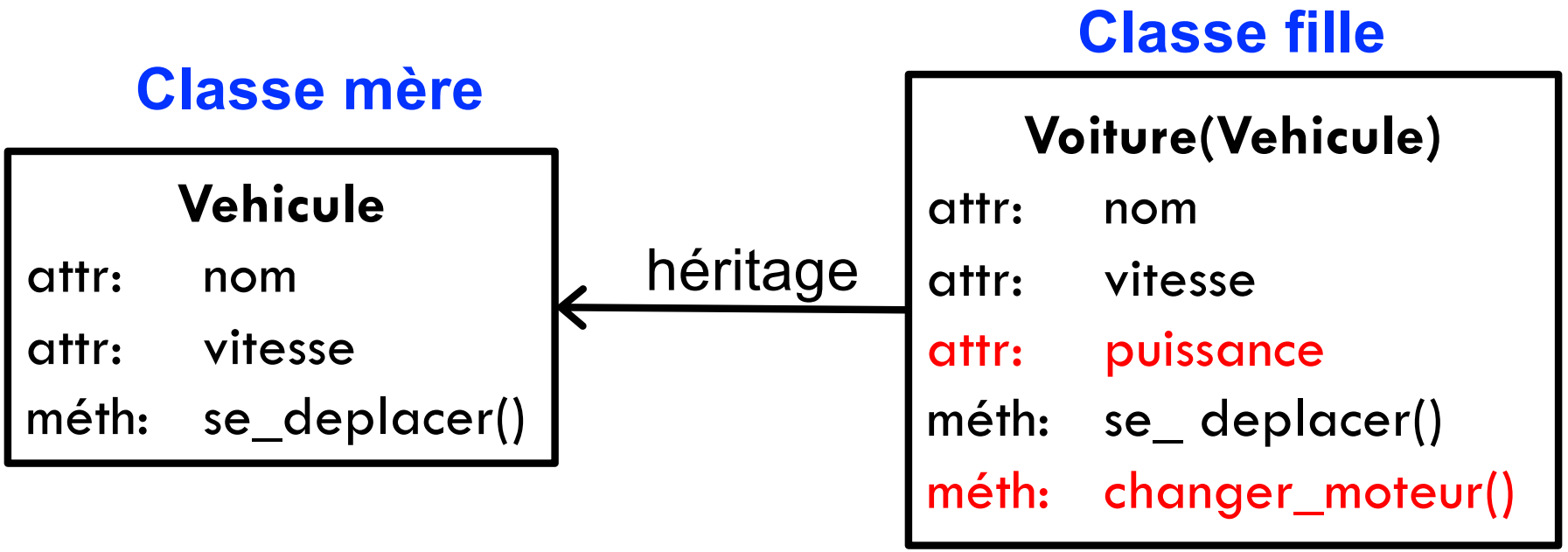
Héritage

Héritage

- l'**héritage** est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe.
- L'héritage établit une relation de généralisation / spécialisation entre une **classe mère** (**classe parent, super-classe**) et une **class fille** (**classe enfant, sous-classe**).
- La classe fille **hérite** des caractéristiques, i.e., propriétés, attributs et méthodes, de la classe mère.

Héritage

- La classe mère généralise la classe fille.
- La classe fille spécialise la classe mère.



Héritage

- La syntaxe pour définir une classe fille qui hérite d'une classe mère est la suivante :

```
class ClasseFille(ClasseMere):
    def __init__(self,attr1,...,attr1',...)
        ClasseMere.__init__(self,attr1,...)
        self.attr1' = attr1'
        ...
    def nouvelle_methode(self,...):
        ...
```

Exemple


```
# classe mère
class Vehicule:

    # constructeur
    def __init__(self, nom, vitesse):
        self.nom = nom          # attribut
        self.vitesse = vitesse # attribut

    # methode
    def se_deplacer(self, x):
        print("le véhicule bouge de {} km.".format(x))
```

Exemple


```
v1 = Vehicule("Avion A380", 903)
print(v1.nom, "-" , v1.vitesse, "km/h")
Avion A380 - 903 km/h
v1.se_deplacer(3000)
le véhicule bouge de 3000 km.

v2 = Vehicule("BMW M4", 200)
v2.se_deplacer(5)
le véhicule bouge de 5 km.
```

Exemple


```
# classe fille (ou sous-class de Vehicule)
# Voiture hérite des attributs et méthodes de Vehicule
class Voiture(Vehicule):
    # constructeur
    def __init__(self, nom, vitesse, puissance):
        # héritage du constructeur de Vehicule
        # 2 attributs et 1 méthode hérités
        Vehicule.__init__(self, nom, vitesse)
        self.puissance = puissance # nouvel attribut

    # nouvelle méthode
    def changer_moteur(self, nouvelle_puissance):
        self.puissance = nouvelle_puissance
        print("nouvelle puissance:", self.puissance, "cv")
```

Exemple


```
voiture1 = Voiture("Porsche Carrera 911", 293 , 385)
print(voiture1.nom)          # attributs hérité
Porsche Carrera 911
print(voiture1.vitesse)      # attribut hérité
293
print(voiture1.puissance) # nouvel attribut
385
voiture1.se_deplacer(125)     # methode héritée
le véhicule bouge de 125 km.
voiture1.changer_moteur(300) # nouvelle méthode
nouvelle puissance: 300 cv
```

Exemple


```
# autre classe fille (ou sous-class de Vehicule)
# Avion hérite des attributs et méthodes de Vehicule
class Avion(Vehicule):

    # constructeur
    def __init__(self, nom, vitesse, annee):
        # héritage du constructeur de Vehicule
        # 2 attributs et 1 méthode
        Vehicule.__init__(self, nom, vitesse)
        self.annee = annee # 1 nouvel attribut

    # nouvelle methode (écrase celle du même nom)
    def se_deplacer(self, x):
        print("l'avion bouge de {} km.".format(x))
```

Exemple


```
avion1 = Avion("Boeing 777", 1098 , 2007)
print(avion1.nom)          # attributs hérité
Boeing 77
print(avion1.vitesse)      # attribut hérité
1098
print(avion1.annee)        # nouvel attribut
2007
avion1.se_deplacer(6000) # nouvelle methode (même nom)
l'avion bouge de 6000 km.
```

Héritage

- On peut définir une classe fille qui hérite de plusieurs classes mères. Dans ce cas, one parle d'**héritage multiple** :

```
class ClasseFille(ClasseMere1,ClasseMere2,...):
                ...
                ...
```

- Par exemple, la classe bus pourrait hériter à la fois de la classe des véhicules et de celle des moyens de transports.

Quelques fonctions

- On peut tester si un objet est une instance d'une certaine classe ou non (retourne un Booléen) :

```
isinstance(<objet>, <classe>)
```

- On peut tester si une classe est une sous-classe d'une autre ou non (retourne un Booléen) :

```
issubclass(<classe1>, < classe2>)
```

Exemple


```
print(isinstance(avion1, Vehicule))
True
print(isinstance(avion1, Avion))
True
print(isinstance(avion1, Voiture))
False
print(issubclass(Avion, Vehicule))
True
```