LINGE1225 : Programmation en économie et gestion

Point Théorique 7

Classes: Objets, attributs, méthodes, héritage

François Fouss & Marco Saerens

Année académique 2020-2021

1

Livre de référence

- Chapitre 11 : Classes, objets, attributs
- Chapitre 12 : Classes, méthodes, héritage

2

Plan

1. Classes, objets et méthodes
2. Les chaînes sont des objets
3. Les listes sont des objets
4. Héritage
5. Application

3

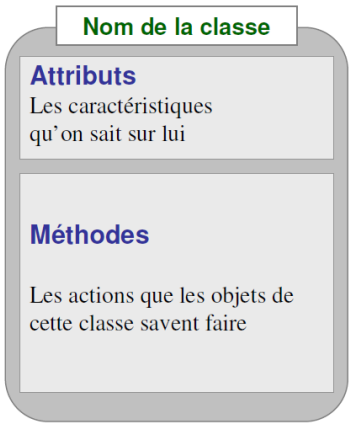
Classes, objets et méthodes

4

Certains programmes sont orientés objet (Python, Java…)
D'autres pas

La programmation orientée-objet

- Permet la modélisation des objets réels et des concepts
- Une **classe** est composée de :
  - **Attributs** : les variables pour stocker des informations d'état d'un objet.
  - **Méthodes** : les fonctions pour décrire les actions que les objets peuvent avoir.
- Des **objets** ou seront crées à partir de la classe.
  - Ce sont des **instances** de la classe.



5

ex: Voitures sur Mario Kart

Attribut —> Caractéristiques de la voiture:
- Puissance
- Maniabilité
- …

méthode: dicte le comportement de la voiture —> Les fonctionnalités de la voiture:
- Accélérer
- Freiner
- …

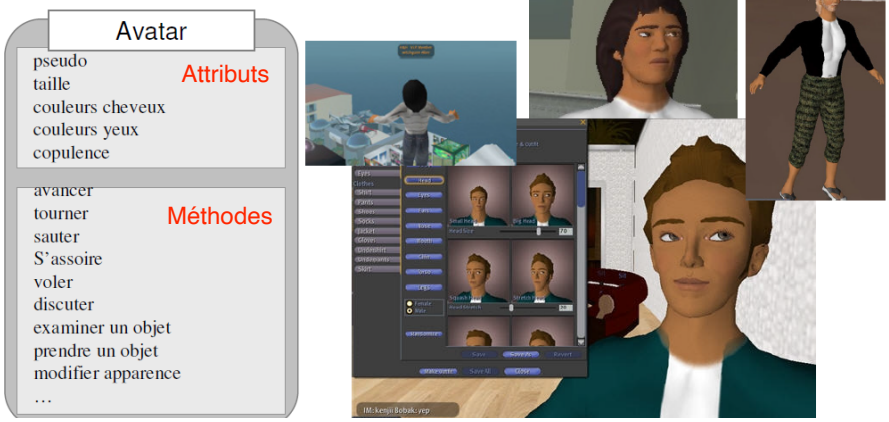
La programmation orientée-objet

- **Classe** = le patron (template) indiquant à la machine virtuelle comment construite un objet d'un type de données.
- **Objet** = une des réalisations concrètes possible de la classe. Chaque objet a ses caractéristiques propres.
- **Attribut** = les informations et propriétés qui caractérisent la classe (attribut de classe) ou bien spécifiques à chaque objet de la classe (attribut d'objet ou d'instance).
- **Méthode de classe** = les actions que tous les objets de cette classe peuvent exécuter.
- Pour créer des objets (instances de classe) à partir d'une classe, on doit appeler une méthode spéciale portant le nom de **constructeur**.

6

Exemple : personnage virtuel

Classe **Objets**



7

Exemple de définition de classe

➢ Création d'une classe Point() qui reprend les coordonnées d'un point géométrique:

Mot réservé `classe`

```
#classes
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

Méthode d'initialisation (le constructeur)

Attributs/ variables d'objet ou d'instance

Méthode constructeur

—> méthode est automatiquement exécutée

ex: si x=2 et y=3 alors va directement être mis comme les caractéristiques du point

On appelle le constructeur: __init__ (2 _ de chaque côté)
self = objet sur lequel est exécuté la méthode

8

La méthode constructeur

- Méthode qui permet que les attributs/variables d'instances soient prédéfinis à l'intérieur des classes avec pour chacune d'elles une valeur initiale "par défaut".

- La méthode constructeur est exécutée automatiquement lorsqu'on instancie (crée) un nouvel objet à partir de la classe. On peut donc y placer tout ce qui semble nécessaire pour **initialiser** automatiquement l'objet que l'on crée.

A chaque fois qu'on crée un nouvel objet, le constructeur est « repassé en revue »

9

La méthode constructeur

- Afin qu'elle soit reconnue comme tel par Python, la méthode constructeur devra obligatoirement s'appeler __init__

- Prenons maintenant le cas de la création d'une horloge

```
class Time:
    "Une nouvelle classe temporelle"
    def __init__(self):
        self.heure=12
        self.minute=0
        self.seconde=0
    def affiche_heure(self):
        print("{0}:{1}:{2}".format(self.heure, self.minute, self.seconde))
```

initialisation des variables d'instance

Méthode qui permet d'afficher l'heure (les h, min, sec sont séparées par :)

Méthode qui permet d'afficher l'heure

10

On donne une instruction de formatage où doivent se placer les arguments h,min,sec

La méthode constructeur

Exemple de création d'un objet :

Appel de classe vide — Appel de méthode

```
tstart = Time()
tstart.affiche_heure()
12:0:0
```

On a exécuté la classe « Time »

L'objet "tstart" s'est vu attribuer automatiquement les trois attributs heure, minute et seconde par la méthode constructeur avec 12 et 0 comme valeurs par défaut. Dès lors qu'un objet de cette classe existe, on peut donc tout de suite demander l'affichage de ses attributs.

11

Attribut d'objet et attribut de classe

- Jusqu'à présent, nous nous sommes intéressés aux attributs associés aux objets ou instances.
- Nous pouvons également définir des attributs associés à la classe.
  - Par exemple, nous introduisons une variable de classe qui va comptabiliser le nombre d'objets `Point` créés. Celle-ci est définie au niveau de la classe.
- On accède alors à cette variable par `Point.nPoints` (on utilise le nom de la classe et non pas le nom de l'objet)

ex Mario: Attribut de classe est spécifique aux voitures (ttes) et non à chaque voiture !

Dès qu'un nouveau point est créé, le compteur va être augmenté

```
class Point:
    nPoints = 0
    def __init__(self,x,y):
        self.x = x
        self.y = y
        nPoints = nPoints + 1
```

Attribut de classe

On incrémente la variable à chaque nouveau point créé

12

Getter (Va juste demander à l'utilisateur de donner l'attribut: x ou y)

- La méthode getter permet de retourner la valeur de l'attribut demandé

```
#classes

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

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

Constructeur de la classe

Méthode getter

```
#appel de la méthode getter

p9.getX()
p9.getY()
```

13

ex: demande la vitesse de Mario à un certain endroit

Setter (Permet de changer les coordonnées d'un point = ses attributs)

- La méthode setter permet de modifier la valeur de l'attribut

```
#classes

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

    def setX(self,abs):
        self.x=abs

    def setY(self,ord):
        self.y=ord
```

Constructeur de la classe

Setter

```
#appel de la méthode setter

p9.setX(6)
p9.setY(1)
```

La coordonnée X de p9 va devenir égale à 6

14

!!! TJ () pour les méthodes >< Pas de () pour les attributs

p01 = Point(2,5)
p02 = Point(4,-2)
p03 = Point(2,-6)
Point nPoints —> donne le nbr de points créés
p01.x et p01.y —> donne les coord du point 1
p01.getX() et p01.getY() —> révèle les coord du point1
p01.setCoord(5,5) —> change les coord du point 1 par (5,5)

Les chaînes sont des objets

Les chaînes sont des objets

- Il est possible d'agir sur un objet à l'aide de méthodes (c'est-à-dire des fonctions associées à cet objet).
- On utilise `nomObjet.nomMethode(arguments)`
- Les chaînes de caractères sont des objets.
- Il est possible d'effectuer de nombreux traitements sur les chaînes en utilisant les méthodes appropriées.

Exemple :

- **`split()`** : convertit un chaîne en une liste de sous-chaînes. On peut choisir le caractère séparateur en le fournissant comme argument, sinon c'est un espace par défaut.
- **`join(liste)`** : rassemble une liste de chaînes en une seule.
- **`find(sch)`** : cherche la position d'une sous-chaîne *sch* dans la chaîne.
- **`count(sch)`** : compte le nombre de sous-chaînes *sch* dans la chaîne.
- **`lower()/upper()`** : convertit une chaîne en minuscules/majuscules.

Les listes sont des objets

Les listes sont des objets

- Sous Python, les listes sont des objets à part entières.
- Il est donc possible de leur appliquer un certain nombre de méthodes particulières.

Exemple :

```
#objet

nombres = [17, 38, 10, 25, 72]
nombres.sort()                    #ajouter à la liste
nombres
[10, 17, 25, 38, 72]

nombres.append(12)              #ajouter un élément à la fin
nombres
[10, 17, 25, 38, 72, 12]
```

D'autres méthodes existent telles que : `nombres.reverse()` qui inverse l'ordre des éléments, `nombres.index()` qui retrouve l'index d'un élément ou encore `nombres.remove()` qui enlève un élément.

Les listes sont des objets

- Notez qu'il existe aussi, pour les listes, des instructions intégrées telle que **del** qui permet d'effacer un ou plusieurs éléments à partir de leur index:

```
#objet  On veut supprimer l'élément d'indice 2

nombres = [17, 38, 10, 25, 72]
del nombres[2]
nombres

[17, 38, 25, 72]
```

Remarque **:**
la méthode `remove()` supprime à partir d'une valeur qui se trouve dans la liste. Tandis que l'instruction `del` travaille avec un index ou une tranche d'index.

19

Héritage

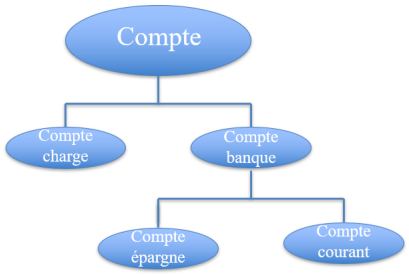
20

L'héritage

➢ L'héritage est une relation entre :

- Une super-classe ou classe parent/mère
- Une ou des sous-classe(s) (ou classes enfant/fille) qui héritent des attributs et méthodes de la super-classe.
  - → De plus, les sous-classes étendent la super-classe, c'est à dire qu'elles reçoivent, en plus de ce qui est hérité, d'autres attributs/méthodes qui leur sont propres et auxquels la super classe ne peut avoir accès.

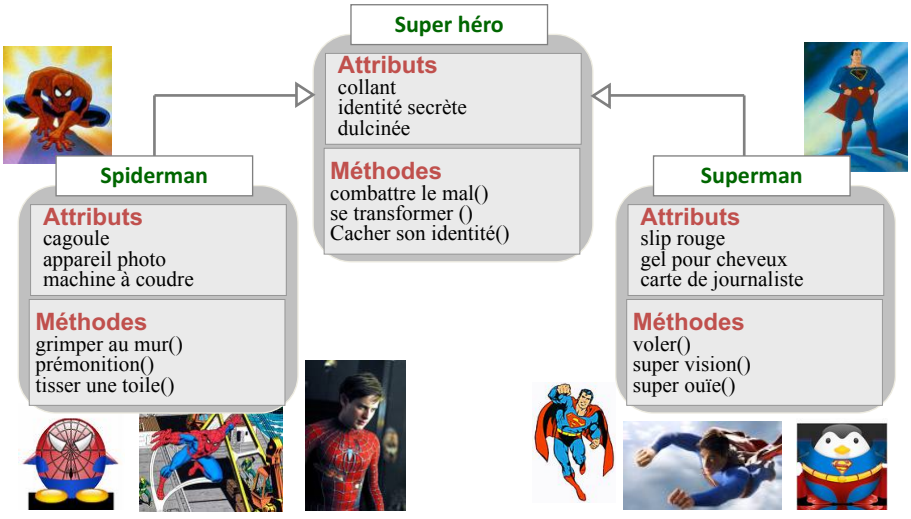
Exemple d'héritage sous forme d'arbre :



21

ex: super-classe = moyen de transport (avec attributs position et vitesse)
sous-classes : voiture, vélo, avion… qui vont tous hériter d'une position et d'une vitesse mais avec des attributs en plus

Exemple



22

On ne doit bien évidemment que coder les attributs supplémentaires spécifique à la classe, pas de nouveau ceux de la super-classe

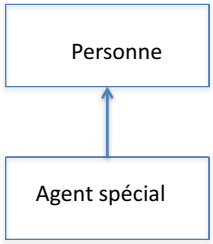
Quand utiliser l'héritage ?

- Utiliser l'héritage quand on doit définir une version plus spécifique d'une classe existante.
- Utiliser l'héritage quand on doit définir plusieurs classes qui partagent des comportements similaires.
- L'héritage n'a pas de sens si une classe ne fait qu'utiliser des morceaux de codes ou des instances (objets) d'une autre classe.

23

Héritage

- La relation d'héritage est souvent représentée par un diagramme avec une flèche de la classe fille vers la classe mère



agent spécial est une sous-classe de pers

- L'héritage devrait créer une "est un" relation, c'est à dire que l'enfant "est une" version plus spécifique des parents.

24

Héritage

- En Python, la syntaxe pour créer une relation d'héritage est la suivante :

```
#héritage                                   ,prénom

class Personne:
    def __init__(self,nom):
        self.nom = nom
        self.prenom = "Martin"
    def __str__(self):                      —> Méthode qui est exécutée qd on imprime l'objet
        return"{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):
    def __init__(self,nom,matricule):
        Personne.__init__(self, nom)
        self.matricule = matricule
    def __str__(self):
        return "Agent{0},
matricule{1}".format(self.nom,self.matricule)
```

Classe définissant un agent spécial. Elle hérite de la classe Personne

Appel explicite du constructeur de Personne

25

Par convention: __str__

On va concaténer le nom et le prénom avec un espace entre

On spécifie la classe mère par l'argument
Si on ne passe rien en argument, la nouvelle classe n'hérite de rien

Deux fonctions très pratiques

- En Python, il existe deux fonctions très pratiques, `issubclass` et `isinstance`
- `issubclass` vérifie si une classe est une sous classe d'une autre classe.
- Elle renvoie True si c'est la cas et False sinon :

```
#issubclass

issubclass(AgentSpecial, Personne)
True

issubclass(AgentSpecial, object)
True

issubclass(Personne, object)
True

issubclass(Personne, AgentSpecial)
False
```

AgentSpecial hérite de Personne

AgentSpecial hérite de la classe générique object

Personne n'hérite pas d'AgentSpecial

26

La classe « objet » est au sommet de tt les hiérarchies
Elle est déjà implémentée dans Python

Deux fonctions très pratiques

- `isinstance` permet de savoir si un objet est issu d'une classe ou de ses classes filles.

```
#isinstance

agent = AgentSpecial("Fisher", "18327-121")

isinstance(agent, AgentSpecial)
True

isinstance(agent, Personne)
True
```

Agent est une instance d'AgentSpecial

Agent est une instance héritée de Personne

27

Héritage Multiple

(Généralement on hérite d'1 seule classe mère donc héritage multiple n'est pas vraiment utilisé)

- Python inclut un mécanisme permettant l'héritage multiple.
- L'idée est simple, au lieu d'hériter d'une seule classe, on peut hériter de plusieurs classes.

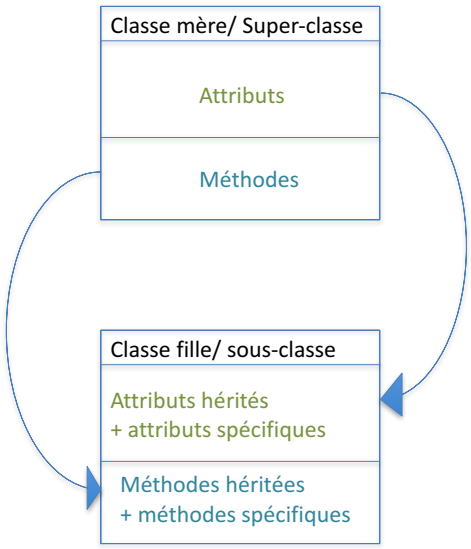
```
#Héritage multiple

class MaClasseHeritee(MaClasseMere1, MaClasseMere2)
```

- *Remarque* : Il est possible de faire hériter votre classe de plus de deux autres classes.

28

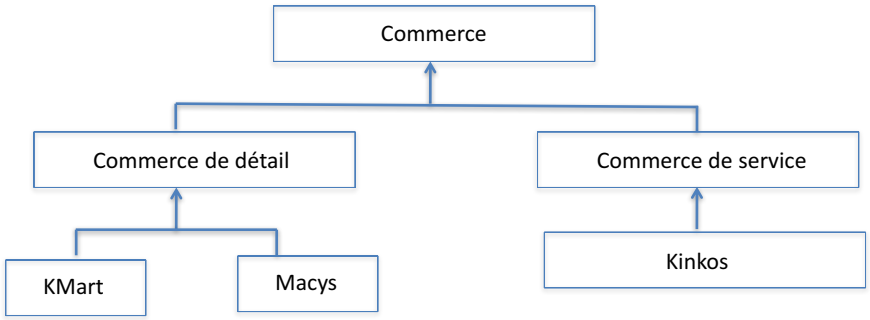
Héritage



29

Héritage

- Une classe enfant d'un parent peut devenir le parent d'une autre classe enfant, formant ainsi une hiérarchie de classe :



30

Héritage

- Une sous-classe:

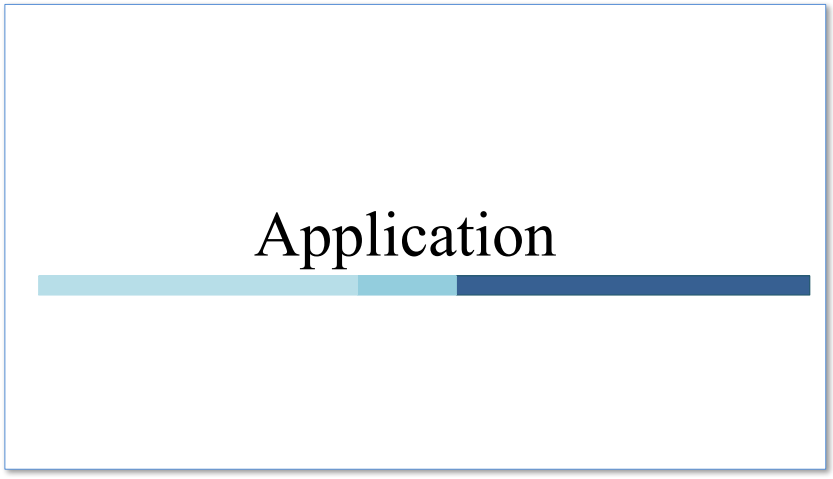
  - Hérite des attributs et méthodes de la super-classe.

  - Ajoute des attributs et méthodes qui lui sont propres.

- Une sous-classe peut également redéfinir les méthodes héritées de la super classe (overriding). La méthode redéfinie doit avoir la même signature que la méthode jumelle dans la super-classe.

Si on réimplémente la même méthode, c'est cette méthode-là qui sera exécutée et pas la méthode des parents dont elle a hérité. C'est la méthode de la nouvelle classe qui a priorité!

31



32

Formes géométriques et leur méthodes (exercice 12.5 du livre)

1) Définissez une classe **Cercle()**. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre **rayon**), vous définirez une méthode **surface()**, qui devra renvoyer la surface du cercle.

2) Définissez ensuite une classe **Cylindre()** dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres **rayon** et **hauteur**. Vous y ajouterez une méthode **volume()** qui devra renvoyer le volume du cylindre (rappel : volume d'un cylindre = surface de section × hauteur).

Exemple d'utilisation de cette classe :

```
cyl = Cylindre(5, 7)
print(cyl.surface())
78.54
print(cyl.volume())
549.78
```

car c'est une méthode

33

Solution :

```
# Classes dérivées - Polymorphisme
class Cercle(object):
    def __init__(self, rayon):
        self.rayon = rayon
    def surface(self):
        return 3.1416 * self.rayon**2

class Cylindre(Cercle):
    def __init__(self, rayon, hauteur):
        Cercle.__init__(self, rayon)
        self.hauteur = hauteur
    def volume(self):
        return self.surface()*self.hauteur
    # la méthode surface() est héritée de la classe parente

class Cone(Cylindre):
    def __init__(self, rayon, hauteur):
        Cylindre.__init__(self, rayon, hauteur)
    def volume(self):
        return Cylindre.volume(self)/3
        # cette nouvelle méthode volume() remplace celle que
        # l'on a héritée de la classe parente (exemple de polymorphisme)

# Programme test :
cyl = Cylindre(5, 7)
print("Surf. de section du cylindre =", cyl.surface())
print("Volume du cylindre =", cyl.volume())
co = Cone(5,7)
print("Surf. de base du cône =", co.surface())
print("Volume du cône =", co.volume())
```

On peut utiliser cette méthode car elle a été définie dans la classe mère

Résultat lorsque l'on exécute ce programme :

```
Surf. de section du cylindre = 78.53999999999999
Volume du cylindre = 549.78
Surf. de base du cône = 78.53999999999999
Volume du cône = 183.26
```

34

self = objet sur lequel on exécute la méthode
Pour cyl.surface() —> dans le code, self sera remplacé par l'objet cyl qui est un cylindre

PAUSE

35