LINGE1225 : Programmation en économie et gestion

Cours 7

Classes: Objets, attributs, méthodes, héritage

François Fouss & Marco Saerens

Année académique 2020-2021

Livre de référence

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

2

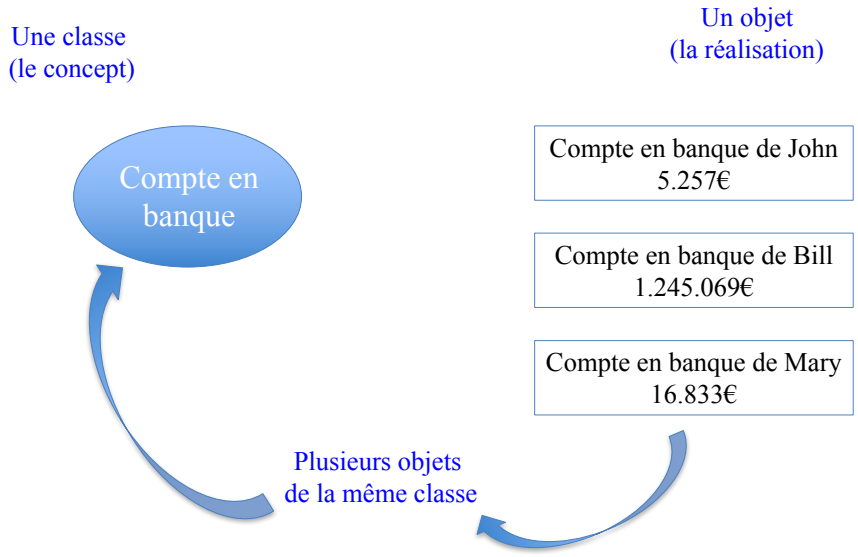
Plan

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

3

Classes, objets et méthodes

Introduction objets



5

Classe, objet, méthode

- Idée de base de la programmation orienté-objet = regrouper dans un même ensemble (objet), un certain nombre de données (attributs d'instances) et les méthodes (fonctions particulières encapsulées dans l'objet) qui permettent d'effectuer des traitements sur ces données.

objet = [attributs + méthodes]

6

Introduction objets

- Un objet représente quelque chose qui nous permet d'interagir avec le programme.
- Un objet fournit une collection de services que nous pouvons lui demander d'exécuter à notre place.
- Les services sont définis par des méthodes dans une classe qui définit les objets.
- Une classe représente un concept, et un objet représente une réalisation/instance de cette classe.
- Une classe peut être utilisée pour créer plusieurs objets.

7

Principe d'encapsulation

- Principe de base de la programmation orienté-objet.
- la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet.
- Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'interface de l'objet.
- Objectif: établir une séparation stricte entre la fonctionnalité d'un objet (telle qu'elle a été déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

8

Définition d'une méthode

- En python, on définit une méthode comme on définirait une fonction en écrivant un bloc d'instructions à la suite du mot réservé `def`, mais avec deux différences :

- La définition d'une méthode est toujours placée à l'intérieur de la définition d'une classe, pour que la relation qui relie la méthode à la classe soit clairement établie

- La définition d'une méthode doit toujours comporter au moins un paramètre, lequel doit être une référence d'instance, et ce paramètre particulier doit toujours être listé en premier.

9

Bien comprendre ce qu'est une classe, un objet, une instance, …

Tous ces concepts sont **très** importants ! Les comprendre vous permettra de coder des codes de plus en plus complexes et utiles. Prenons un exemple de la vie de tous les jours pour comprendre ces concepts :

1) Une classe : cela définit un groupe de données sur lequel s'applique des propriétés qui leur sont propres (des méthodes). Disons que nous avons la classe : **Chat** et une autre classe : **Oiseau**.

2) Une méthode : Nous sommes d'accord que ces deux « types » d'animaux n'ont pas les mêmes propriétés, c'est pourquoi ils ont leurs propres méthodes qui ne s'appliquent qu'à eux ! Pour le chat ce serait par exemple : *courir(), sauter(), marcher()*,… tandis que pour l'oiseau, ce serait : *voler(), sauter(),* … On ne pourrait pas utiliser la méthode *voler()* sur un chat…

10

Bien comprendre ce qu'est une classe, un objet, une instance, …

3) Un objet/une instance : maintenant que nous avons vu ce qu'est une classe et ses méthodes, qu'est-ce qu'une instance d'une classe ? Imaginons que vous avez 2 chats chez vous (Garfield et Grumpy). Premièrement, nous savons que les chats sont différents des oiseaux et qu'ils avaient chacun leur propre classe. Deuxièmement, au sein d'une même classe, il y aussi des différences : Garfield est différent de Grumpy ! Ils ont chacun leurs propres propriétés (attribut) : âge, poids, taille, … on dit que ce sont deux objets/instances différents de la même classe.
Donc une instance d'une classe est une entité qui accède à toutes les méthodes de la classe et qui a ses propres attributs.

→ On ne peut pas appeler une méthode de la classe oiseau sur une instance de la classe chat.

Exemple d'objet que vous avez déjà utilisé : les strings, les listes, les dictionnaires, …
→ La fonction len() sur un string ne fonctionne pas sur un integer car la class « integer » n'a pas cette méthode.

11

Utilités des classes

- Les classes sont les principaux outils de la programmation orienté-objet.
- Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux, et avec le monde extérieur.

12

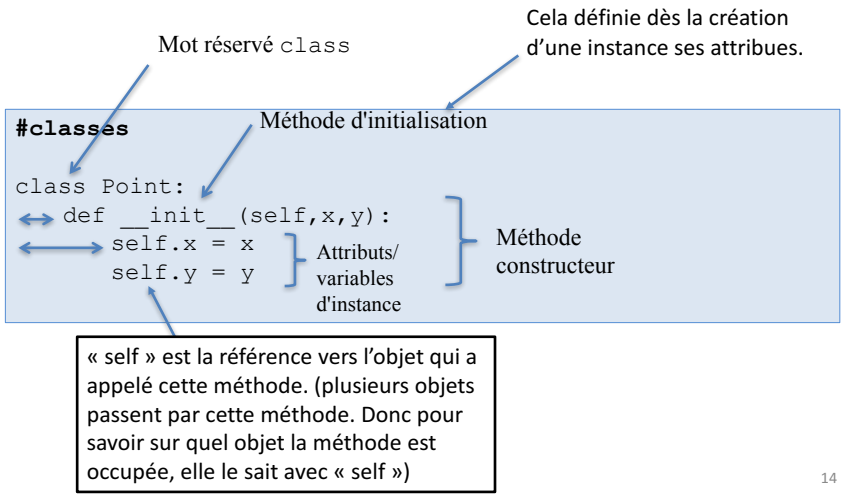
Définition classes

- Définition de classe grâce à l'instruction `class`.
- Les classes créent un nouveau type de donnée .
- Ce type de donnée composite sera différent des types de données intégrés dans le langage lui-même.
- Les définitions de classes se situent généralement au début du script .

13

Exemple de définition de classe

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



14

Création d'objet de classe

- Instanciation = création d'un objet de classe

```
#objet de classe

p9 = Point(3,4)
```

variable objet

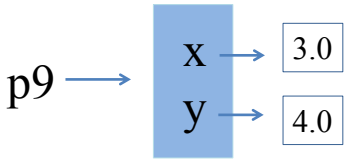
- La variable p9 est un point de coordonné (x=3,y=4)

Remarque : Objets de classe = instances de classe.

15

Attributs d'instance

- Diagramme d'état



La variable p9 contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs x et y. Ceux-ci contiennent les références des valeurs 3.0 et 4.0 mémorisées ailleurs.

16

La méthode constructeur

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

- La méthode constructrice est exécutée automatiquement lorsqu'on instancie 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.

17

La méthode constructeur

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

- Prenons le cas de 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

18

La méthode constructeur

Exemple :

Appel de classe vide

Appel de méthode

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

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

19

Getter

- 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() #return 3.0
p9.getY() #return 4.0
```

20

Setter

- La méthode setter permet de définir 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

21

Similitude et unicité

- Egalité entre deux objets :

```
#objet

p1 = Point(3,4)

p2 = Point(3,4)

print(p1 == p2)

False
```

- Ces deux instructions créent deux objets qui restent distincts même s'ils font partie d'une même classe et ont des contenus similaires.

→ Exemple : 2 chats peuvent se ressembler à 100% et pourtant, ce ne sont pas les mêmes.

- Ceci est dû au fait qu'ils ne sont pas stockés au même endroit sur le disque

Similitude et unicité

Pourquoi les deux objets ne sont-ils pas égaux?

```
#objet

print(p1)
print(p2)

<__main__.Point object at 0x10b282278>
<__main__.Point object at 0x10b2c3198>
```

Zone dans la mémoire où sont stocké ces instances

- Les deux variables p1 et p2 référencent bien des objets différents, car ils sont mémorisés à des emplacements différents dans la mémoire de l'ordinateur.

Similitude et unicité

Essayons :

```
#objet

p2 = p1
print(p1 == p2)

True
```

Assigne le contenu de p1 à p2, ces deux variables référencent le même objet

- L'expression entre parenthèse est vraie : p1 et p2 désignent bien toutes deux un seul et unique objet.
- Ce qui implique que si on modifie l'attribut de p1, l'attribut de p2 sera aussi tôt modifié aussi.
- Dans ce cas-ci ils pointent tous les deux vers le même emplacement mémoire.
- Une modification de l'un d'entre eux génèrerait une modification de l'autre .

Objets : valeurs de retour fonction

- Cas dans lequel la fonction transmet une instance comme valeur de retour.

Par exemple, la fonction translation en dehors de la classe va retourner un objet dont les coordonnées de départ seront translatées.

```
#objet

def translations(Point,dx,dy):
    P = Point(Point.getX()+dx,Point.getY()+dy)
    return P
```

Renvoie un objet, de type P()

`Translations()` prend comme argument : un objet et 2 nombres et il renvoie un **nouvel** objet
→ /!\ on ne modifie pas l'objet passé en argument ! On récupère ses données que l'on modifie pour les données à un nouvel objet que la fonction instancie

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).
- Les chaînes de caractères sont des objets
- Il est possible d'effectuer de nombreux traitements sur les chaînes en utilisant des 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()                    #trier la liste (croissante)
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

- Il existe aussi pour les listes, des instructions intégrées telles que `del` qui permet d'effacer un ou plusieurs éléments à partir de leur index:

```
#objet

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` travail avec un index ou une tranche d'index.

30

remove()/ del

- Exemple :

```
#remove()/del

nombres = [17, 38, 10, 25, 72]
nombres.remove(38)
nombres
[17, 10, 25, 72]

nombres = [17, 38, 10, 25, 72]
del nombres[1]
nombres
[17, 10, 25, 72]
```

Je mentionne le chiffre que je veux supprimer

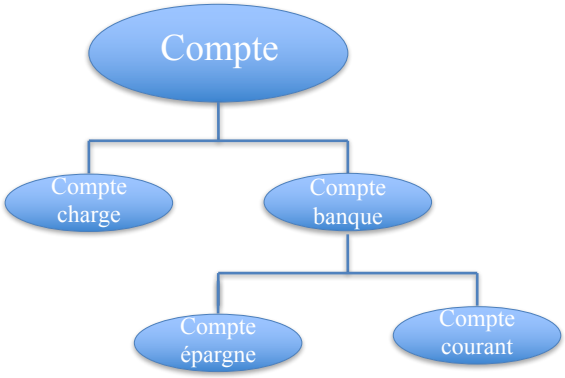
Je mentionne l'indice où se trouve le chiffre 38 dans la liste

31

Héritage

Héritage

- Une classe peut être utilisée pour décrire une autre via l'héritage
- Les classes peuvent être organisées dans les hiérarchies d'héritages



33

Héritage

- Une classe préexistante peut être utilisée pour créer une nouvelle.
- Cette nouvelle classe héritera de toutes ses propriétés mais pourra modifier certaines d'entre elles et/ou y ajouter des nouvelles propriétés propres à cette nouvelle classe
- Ce procédé s'appelle *dérivation*
- Cela permet de créer une hiérarchie de classes allant du général au particulier

Terminologie :

- la classe existante = super-classe ou classe parent
- la nouvelle classe = sous-classe ou classe enfant

34

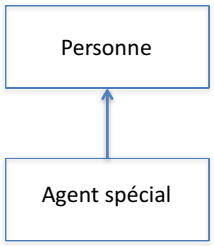
Héritage

- Le programmeur peut ajouter de nouvelles variables ou méthodes, ou peut modifier celui qui a été hérité
- En utilisant des composants de programmes existants pour en créer de nouvelles, nous épargnons tous les efforts concernant le design, l'implémentation et le test du programme existant

35

Héritage

- La relation d'héritage est souvent représentée par un diagramme avec un flèche de la classe enfant vers la classe parent



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

36

Héritage

Pour bien comprendre :
Disons que nous avons la classe : **Félin**. Tous les objets « Félin » ont des attribues et méthode commune.

Nous voulons créer des classes plus spécifiques : **Chat** et **Tigre**. Ces deux classes ont des choses communes (attribues et méthodes) qui se retrouvent dans la class **Félin**. C'est pourquoi elles étendent cette classe et ont le droit d'utiliser toutes les méthodes déjà coder de cette classe, exemple : courir(). Bien entendue, si une méthode existe déjà dans la classe **Félin**, on peut la recoder dans une class fille si elle doit lui être spécifique.

37

Héritage

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

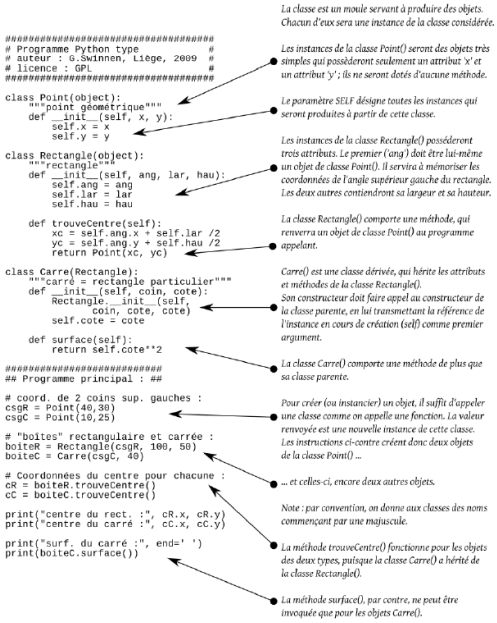
```
#héritage

class Personne:
    def __init__(self,nom):
        self.nom = nom
        self.prenom = "Martin'
    def __str__(self):
        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 une agent spécial. Elle hérite de la classe Personne

Appel explicite du constructeur de Personne

38

Exemple récapitulatif – cf. Swinnen, p. 186



```
#####################################
# Programme Python type             #
# auteur : G.Swinnen, Liège, 2009   #
# licence : GPL                     #
#####################################

class Point(object):
    """point géométrique"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Rectangle(object):
    """rectangle"""
    def __init__(self, ang, lar, hau):
        self.ang = ang
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        xc = self.ang.x + self.lar /2
        yc = self.ang.y + self.hau /2
        return Point(xc, yc)

class Carre(Rectangle):
    """carré = rectangle particulier"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self,
                           coin, cote, cote)
        self.cote = cote

    def surface(self):
        return self.cote**2

############################
## Programme principal : ##

# coord. de 2 coins sup. gauches :
csgR = Point(40,30)
csgC = Point(10,25)

# "boites" rectangulaire et carrée :
boiteR = Rectangle(csgR, 100, 50)
boiteC = Carre(csgC, 40)

# Coordonnées du centre pour chacune :
cR = boiteR.trouveCentre()
cC = boiteC.trouveCentre()

print("centre du rect. :", cR.x, cR.y)
print("centre du carré :", cC.x, cC.y)

print("surf. du carré :", end=' ')
print(boiteC.surface())
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une instance de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui posséderont seulement un attribut 'x' et un attribut 'y' ; ils ne seront dotés d'aucune méthode.

Le paramètre SELF désigne toutes les instances qui seront produites à partir de cette classe.

Les instances de la classe Rectangle() posséderont trois attributs. Le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle. Les deux autres contiendront sa largeur et sa hauteur.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance en cours de création (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

Pour créer (ou instancier) un objet, il suffit d'appeler une classe comme on appelle une fonction. La valeur renvoyée est une nouvelle instance de cette classe. Les instructions ci-contre créent donc deux objets de la classe Point() ...

... et celles-ci, encore deux autres objets.

Note : par convention, on donne aux classes des noms commençant par une majuscule.

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de la classe Rectangle().

La méthode surface(), par contre, ne peut être invoquée que pour les objets Carre().

39

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

Personne n'hérite pas d'AgentSpecial

40

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

41

Héritage Multiple

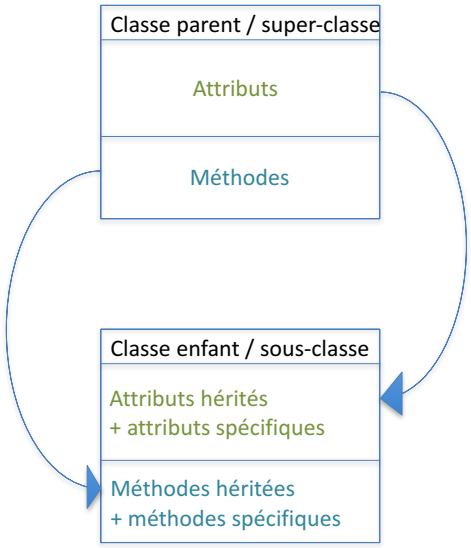
- Python inclut un mécanisme permettant l'héritage multiple.
- L'idée est simple, au lieu d'hériter d'une seule classe, on peut hériter de plusieurs.

```
#Héritage multiple

class MaClasseHeritee(MaClasseMere1, MaClasseMere2)
```

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

42

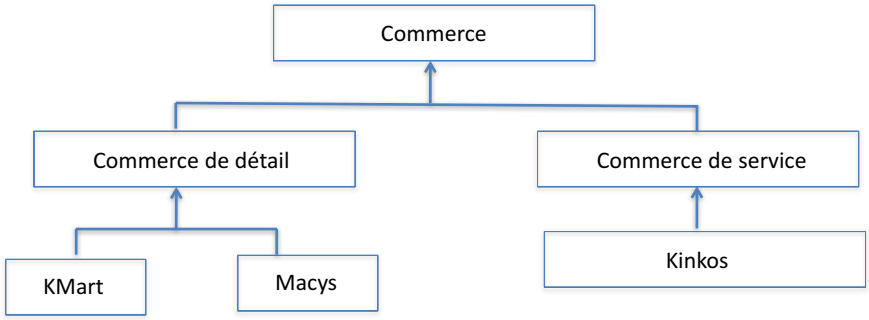
Héritage



43

Héritage

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



44

Héritage

- Une sous-classe:

  -Hérite des attributs et méthodes de la super-classe

  -Ajoute en plus des attributs et méthodes qui lui est propre

- 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.

Application : classe

Application

Définissez une classe Satellite() qui permette d'instancier des objets simulant des satellites artificiels lancés dans l'espace, autour de la terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées : masse = 100, vitesse = 0.
Lorsque l'on instanciera un nouvel objet Satellite(), on pourra choisir son nom, sa masse et sa vitesse.
Les méthodes suivantes seront définies :

- impulsion(force, duree) permettra de faire varier la vitesse du satellite. Pour savoir comment, rappelez-vous votre cours de physique : la variation de vitesse Δv subie par un objet de masse m soumis à l'action d'une force F pendant un temps t vaut $\Delta v = (F * t)/m$. Par exemple : un satellite de 300 kg qui subit une force de 600 Newtons pendant 10 secondes voit sa vitesse augmenter (ou diminuer) de 20 m/s.
- affiche_vitesse() affichera le nom du satellite et sa vitesse courante.
- energie() renverra au programme appelant la valeur de l'énergie cinétique du satellite. Rappel : l'énergie cinétique se calcule à l'aide de la formule $Ec = (m * v^2)/2$

47

Application

➢ Exemple d'utilisation de cette classe :

```
s1 = Satellite('Zoé', masse = 250, vitesse = 10)

s1.impulsion(500, 15)

s1.affiche_vitesse()

print("énergie =", s1.energie())

s1.impulsion(500, 15)

s1.affiche_vitesse()

print("nouvelle énergie =", s1.energie())
```

48

Application 7

➢ Solution :

```
class Satellite(object):
    def __init__(self, nom, masse =100, vitesse =0):
        self.nom, self.masse, self.vitesse = nom, masse, vitesse

    def impulsion(self, force, duree):
        self.vitesse = self.vitesse + force * duree / self.masse

    def energie(self):
        return self.masse * self.vitesse**2 / 2

    def affiche_vitesse(self):
        print("Vitesse du satellite {} = {} m/s".format(self.nom,
self.vitesse))
```

49

Application : héritage

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
```

51

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())
```

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

Process finished with exit code 0
```

52