chap 7

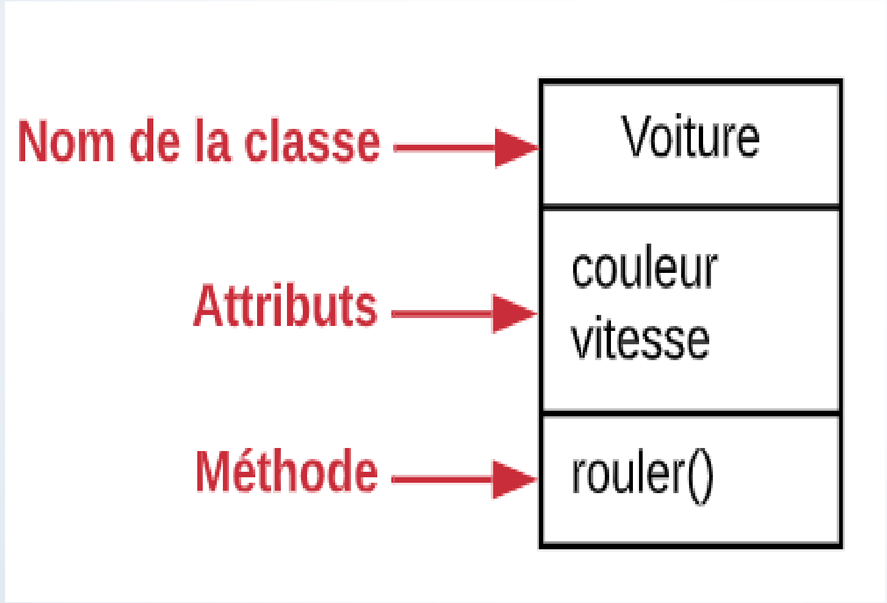
Les classes

Enseignante: Mme Lamia MANSOURI

1. Présentation des classes

Définition 1

Une classe est un ensemble incluant des attributs et des méthodes. Les attributs sont des variables accessibles depuis toute méthode de la classe où elles sont définies. En Python, les classes sont des types **modifiables**.



- Les champs d'une classe peuvent être de type quelconque
- Ils peuvent faire référence à d'instances d'autres classes

Syntaxe:

```
class nom_classe :
    # corps de la classe
    # ...
```

Le corps d'une classe peut être vide, inclure des variables ou attributs, des fonctions ou méthodes.
Le mot-clé **pass** permet de préciser que le corps de la classe ne contient rien.

Exemple :

```
class classe_vide:
    pass
```

Définition 2 : instantiation

Une instance d'une classe C désigne une variable de type C. Le terme instance ne s'applique qu'aux variables dont le type est une classe.

Une instance de la classe est tout simplement un ***objet*** de cette classe.

Syntaxe:

C1=nom_classe()

2. Attributs et méthodes

- Les attributs sont des variables qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

- Une méthode s'écrit comme une fonction dans le corps de la classe avec un premier paramètre ***self*** obligatoire, où **self** représente l'objet sur lequel la méthode sera appliquée. Autrement dit ***self*** est la référence d'instance.

- La déclaration d'une méthode inclut toujours un paramètre **self** de sorte que **self.nom_attribut** désigne un attribut de la classe.

  Nom_attribut seul désignerait une variable locale sans aucun rapport avec un attribut portant le même nom. Les attributs peuvent être déclarés à l'intérieur de n'importe quelle méthode, voire à l'extérieur de la classe elle-même.

```
class nom_classe :
    def nom_methode (self, param_1, ..., param_n) :
        self.nom_attribut = valeur
```

Appel d'une méthode

```
cl = nom_classe() # objet c1 qui est une instance de  nom_classe
t = cl.nom_methode(valeur_1, ..., valeur_n)
```

Exemple

```
class compte:
    #gestion d'un compte bancaire
    def crediter(self,x):
        self.solde += x
    def afficherSolde(self):
        print("Le solde vaut :",self.solde)

monCompte=compte()
monCompte.nom = "comptePerso"
monCompte.numero = 666
monCompte.solde = 1000
```

3. Constructeur et destructeur

Constructeur

Le constructeur se définit dans une classe comme une fonction avec deux particularités :

- le nom de la fonction doit être __init__ ;
- la fonction doit accepter au moins un paramètre, dont le nom doit être self, et qui doit être le premier paramètre et ne doit pas retourner de résultat

Le paramètre ***self*** représente en fait l'objet cible, c'est-à-dire que c'est une variable qui contient une référence vers l'objet qui est en cours de création. Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

```
class compte:
    """gestion d'un compte bancaire"""
    def __init__(self,nom,numero,valeur):
        self.nom = nom
        self.numero = numero
        self.solde = valeur
    def afficherSolde(self):
        print("Le solde vaut :",self.solde)

monCompte = compte(" comptepro",777,10000)
monCompte.afficherSolde()
```

Destructeur

Pour libérer la mémoire d'un objet que l'on n'utilise plus, on va rajouter à nos classes une méthode particulière : un destructeur.
En Python, son nom est imposé : ***__del__.***

Il pourra également servir à mettre à jour certains attributs de classes, comme par exemple un compteur d'objets instanciés.
On ne fait quasiment jamais d'appel explicite à la méthode **__del__**. Elle est appelée **automatiquement** quand un objet cesse d'être référencé (comme par exemple si l'on crée une instance dans une fonction ou procédure).

La syntaxe de la déclaration de la méthode ***__del__*** est la même que celle des autres méthodes :

```
class nomDeLaClasse:
    #documentation de la classe
    def __init__(self,para1,para2,...):
        ...
    def __del__(self):
        ...
    def methode1(self,para1,para2,...):
        ...
```

Attributs privés

9

On réalise la protection des attributs d'une classe grâce à l'utilisation d'attributs privées. Pour avoir des attributs privés, leur nom doit débuter par __ (deux fois le symbole underscore _).

Une fois déclarés privés, il n'est pas plus possible de faire appel aux attributs __x et __y depuis l'extérieur de la classe Point. Il faut donc disposer de méthodes qui vont permettre par exemple de modifier ou d'afficher les informations associées à ces variables.

Exemple

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

p = Point(1, 2)
p.__x

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in   p.__x
AttributeError: Point instance has no attribute '__x'
```

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

    def deplace(self, dx, dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy

    def affiche(self):
        print("abscisse =", self.__x, "ordonnee =", self.__y)

a = Point(2, 4)
a.affiche()
a.deplace(1, 3)
a.affiche()
```

Définition des classes dans des modules

On a toujours intérêt à définir les classes dans des modules (pas obligatoire). On peut avoir plusieurs classes dans un module. Ex. ModuleCompte.py

Contenu du module ModuleCompte.py

```
class compte:
    #gestion d'un compte bancaire
    def __init__(self,nom,numero,solde):
        self.nom = nom
        self.numero = numero
        self.solde = solde

    def crediter(self,x):
        self.solde += x
    def afficherSolde(self):
        print("Le solde vaut :",self.solde)
```

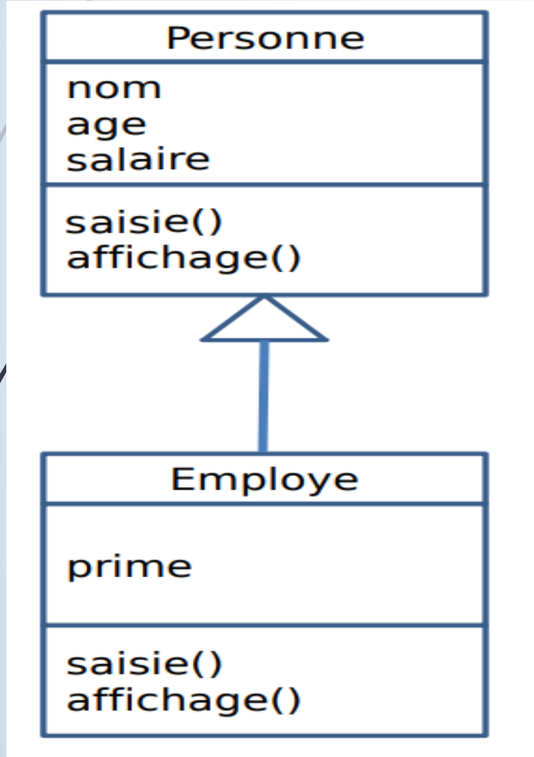
Le programme principal

```
import ModuleCompte as MC
#instanciation
monCompte= MC.compte('moncompte personel',666,1000)
print(monCompte.nom)
monCompte.crediter(2000)
monCompte.afficherSolde()
```

- Il faut d'abord importer le module contenant la classe, nous lui attribuons l'alias MC ici
- Pour la création de l'instance c, nous spécifions le module puis le nom de la classe
- print(dir(c)) :affiche tous les membres de c

4. l'héritage

- L'héritage permet de construire une hiérarchie de classes. Les classes héritières héritent des champs et méthodes de la classe ancêtre.
- Ce mécanisme nécessite des efforts de modélisation et de conception. Mais au final, on améliore la lisibilité et la réutilisabilité du code.



La classe Employe est une Personne, avec le champ supplémentaire prime.
Cela va nécessiter la reprogrammation des méthodes saisie() et affichage(). On peut éventuellement ajouter d'autres méthodes spécifiques à Employe.

Déclaration en Python:

```
class classeMere:
    #une classe mère
    ...

class classeFille(classeMere):
    #une classe fille héritant de la classe mère
    ...
```

Constructeur de la classe fille

On devra respecter une règle fondamentale : le constructeur de la classe fille doit faire un appel explicite au constructeur de la classe mère afin d'initialiser les attributs hérités de celle-ci.

Pour cela on aura deux syntaxes à notre disposition:

- Dans la première syntaxe possible, on fait précéder __init__ du nom de la classe mère :

```
class classeFille(classeMere):
    #documentation de la classe fille
    def __init__(self,para1,para2,...):
        classeMere.__init__(self,para1,...)
        ...
```

- Dans la seconde syntaxe autorisée, on fait précéder __init__ du mot clé super() :

```
class classeFille(classeMere):
    #documentation de la classe fille
    def __init__(self,para1,para2,...):
        super().__init__(para1,...)
```

Héritage multiple

L'héritage multiple est la possibilité pour une classe de posséder plusieurs classes mères.

Comme pour un héritage simple, la classe fille possède les attributs et les méthodes de ses classes mères plus d'autres qui lui sont propres.

```
class mere1:
    ...
class mere2:
    ...
class classeFille(mere1,mere2,...):
    ...
```

On devra respecter une règle fondamentale : le constructeur de la classe fille doit faire un appel explicite au constructeur des classes mères afin d'initialiser les attributs hérités de celles-ci :

```
class classeFille(mere1,mere2,...):
    def __init__(self,para1,para2,...):
        mere1.__init__(self,para1,...)
        mere2.__init__(self,para1,...)
```

Exemple


```
class carnivore:
    def __init__(self,p):
        self._poidsViande = p
    def devorer(self):
        print("Je mange",self._poidsViande,"kilos de viande par jour")

class herbivore:
    def __init__(self,p):
        self._poidsHerbe = p
    def brouter(self):
        print("Je mange",self._poidsHerbe,"kilos d’herbe par jour")

class omnivore(carnivore,herbivore):
    def __init__(self,pv,ph,h):
        carnivore.__init__(self,pv)
        herbivore.__init__(self,ph)
        self.__humain = h

o = omnivore(10,5,False)
o.devorer()
o.brouter()
```

5. Surcharge des méthodes

ModulePersonne.py

```
class Personne:
    def __init__(self):
        self.nom = ""
        self.age = 0
        self.salaire = 0.0

    def saisie(self):
        self.nom = input("Nom : ")
        self.age = int(input("Age : "))
        self.salaire = float(input("Salaire : "))
    def affichage(self):
        print("Son nom est ", self.nom)
        print("Son âge : ", self.age)
        print("Son salaire : ", self.salaire)

class Employe(Personne):
    def __init__(self):
        Personne.__init__(self)
        self.prime = 0.0
    def saisie(self):
        Personne.saisie(self)
        self.prime = float(input("Prime : "))
    def affichage(self):
        Personne.affichage(self)
        print("Sa prime : ", self.prime)
```

Le programme principal

```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#instanciation
e = MP.Employe()
#saisie
e.saisie()
print(" ** Affichage **")
e.affichage()
```

6. La gestion des collections d'objets


```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#liste vide
liste = []
n = int(input("Nb de pers : "))
#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)
#affichage
print(" début affichage ")
for p in liste:
    print("-------")
    p.affichage()
```

Créer l'objet référencé par a, effectuer la saisie. Ajouter la référence dans la liste.

Le typage est automatique, p est bien de type Personne.