Université Claude Bernard Lyon 1

Programmation orientée objet en Python

Nicolas Pronost

LIRIS

Philosophie de la POO

- Dans la programmation orientée objet (POO), toutes les variables sont des objets associés à une classe
- Une classe est un type
  - qui se veut plus complexe que juste un nombre ou un caractère
- Un objet est une collection
  - de données membres (ou attributs)
  - de fonctions membres manipulant les données membres (ou méthodes)
- On peut créer autant d'objets de classe (i.e. de variables du type décrit par la classe) que l'on veut
- Un objet est aussi appelé une instance de classe
- En Python, tout est objet

Définition d'une classe

- Pour définir une classe on utilise le mot clé **class**

```
class MaClasse :
   '''Documentation brève de la classe'''
```

- On peut ensuite ajouter des données membres et des fonctions membres

```
class MaClasse :
   '''Documentation brève de la classe'''
   def fonctionMembre(self) :
      print('Hello')
```

  - **fontionMembre** est une fonction membre que l'on peut appeler sur une instance de la classe **MaClasse**

Création d'un objet

- On peut créer une instance (ou objet) de classe en utilisant le nom de la classe suivie de parenthèses

```
uneInstance = MaClasse()
```

- On peut ensuite accéder aux données membres et fonctions membres en utilisant l'opérateur . (point)

```
uneInstance.fonctionMembre()     #affiche 'Hello'
```

Référence à l'instance

- Vous avez noter la présence du paramètre **self** dans la fonction membre

```
    def fontionMembre (self) :
```

- Pourtant il n'est pas passé en paramètre lors de l'appel

```
uneInstance.fonctionMembre()
```

- Lorsqu'une fonction membre est appelée sur une instance de classe, cette instance est automatiquement ajoutée en tant que premier paramètre
  - L'appel

```
uneInstance.fontionMembre()
```

    est traduit automatiquement en

```
MaClasse.fonctionMembre(uneInstance)
```

Référence à l'instance

- Cela veut aussi dire, que toute fonction membre d'une classe a au moins un paramètre : l'instance
  - Par convention on le nomme **self**
- A l'intérieur de la fonction membre, **self** désigne l'instance sur laquelle la fonction est appelée, on peut donc l'utiliser pour accéder aux données membres et aux fonctions membres spécifiques à cette instance

```
class MaClasse :
   donneeCommune = 0

   def fctSurInstance (self) :
      self.donneeInstance = 1
      MaClasse.donneeCommune += 1

instance1 = MaClasse()      # dC = 0 , i1.dI n'existe pas
instance1.fctSurInstance() # dC = 1 , i1.dI = 1
instance2 = MaClasse()      # dC = 1 , i2.dI n'existe pas
instance2.fctSurInstance() # dC = 2 , i2.dI = 1
```

Membres spéciaux

- Les données et fonctions qui commencent par un double tiret-bas (__) sont des membres spéciaux, ils ont un sens particulier
- Il existe plusieurs données membres spéciales dont
  - **__doc__** : documentation de la classe (donnée par la docstring, i.e. le texte entre ''' en dessous du nom de la classe)
  - **__name__** : nom de la classe
  - **__dict__** : dictionnaire où les clés sont les noms des données membres et les valeurs sont les valeurs des données membres

Constructeur

- Parmi les fonctions membres spéciales, il y a le constructeur de la classe
- Il porte le nom suivant : **__init__**
- Cette fonction est appelée quand une instance de la classe est créée
  - elle renvoie l'objet (**self**) après création

Constructeur

- Le constructeur est habituellement utilisé pour initialiser les valeurs des données membres

```
nc = NombreComplexe()
```

```
class NombreComplexe :
   def __init__ (self) :
      self.Re = 0
      self.Im = 0
```

- Il prend donc habituellement les valeurs des données membres à donner à cette instance en paramètre

```
nc = NombreComplexe(2,4)
```

```
class NombreComplexe :
   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i
```

Création à la volée

- Rappeler vous que Python est un langage interprété, donc il est capable d'ajouter des données membres à la volée

```
nc1 = NombreComplexe()
nc2 = NombreComplexe()

#création à la volée d'une donnée membre texte pour nc1
nc1.texte = 'ceci est un nombre complexe'

print(nc2.texte)
# produit une erreur (pas de membre nommé texte dans nc2)
```

- Il peut aussi en supprimer (instruction **del**)

```
nc1 = NombreComplexe()
del nc1.Im
```

  - cela fonctionne sur les données membres, sur les fonctions membres et sur les instances elles-mêmes

Fonctions membres d'une classe

- On a vu des données membres de classe (communes aux instances), des données membres d'instance (spécifiques à une instance) et des fonctions membres d'instances (avec le paramètre **self**)
- Il existe la possibilité, plus rarement utilisée, d'avoir des fonctions membres de classe (communes aux instances)
  - le premier paramètre s'appelle alors **cls** (pour classe)
  - on indique à Python que c'est une fonction de classe grâce à la fonction **classmethod**
  - on l'appelle avec l'opérateur . (point) sur la classe (comme pour les données membres de la classe) ou sur une des instances de la classe

Fonctions membres d'une classe

```
class NombreComplexe :

   nbComplexesCrees = 0

   def __init__ (self) :
      self.Re = 0
      self.Im = 0
      NombreComplexe.nbComplexeCrees += 1

   def afficheNbComplexe (cls) :
      print(NombreComplexe.nbComplexeCrees)
   afficheNbComplexe = classmethod(afficheNbComplexe)


NombreComplexe.afficheNbComplexe()   # affiche 0
nc1 = NombreComplexe()
NombreComplexe.afficheNbComplexe()   # affiche 1
nc2 = NombreComplexe()
NombreComplexe.afficheNbComplexe()   # affiche 2
# on peut aussi faire : nc2.afficheNbComplexe()
```

Destructeur

- Parmi les fonctions spéciales, il y a aussi le destructeur de la classe
- Il porte le nom suivant : **__del__**
- Cette procédure est appelée
  - lorsque vous demander vous-même la destruction `del nc1`
  - automatiquement en sortant de la porté de l'objet
- Le destructeur est habituellement utilisé pour effectuer des actions avant la destruction de l'objet

```
class NombreComplexe :

   def __init__ (self) :
      self.Re = 0
      self.Im = 0

   def __del__ (self) :
      print("Un nombre complexe a été détruit")
```

Représentation d'un objet

- Un simple **print** sur l'objet affiche ce type d'information

```
nc1 = NombreComplexe()
print(nc1)
# affiche quelque chose comme :
# <__main__.NombreComplexe object at 0x00AC45B70>
```

  - Pas très utile...
- Deux fonctions membres spéciales permettent de contrôler comment l'objet est représenté et affiché à l'écran

Représentation d'un objet

- La fonction membre spéciale **__repr__** retourne la chaîne de caractère qu'il faut afficher lorsque l'on tape directement le nom de l'objet (plutôt pour le debug)

```
class NombreComplexe :

   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i

   def __repr__ (self) :
      if self.Im >= 0 :
         return str(self.Re) + '+' + str(self.Im) + 'i'
      else :
         return str(self.Re) + str(self.Im) + 'i'


nc1 = NombreComplexe(5,2)
nc2 = NombreComplexe(5,-2)
nc1     # affiche 5+2i
nc2     # affiche 5-2i
```

Représentation d'un objet

- La fonction membre spéciale **__str__** retourne la chaîne de caractère qu'il faut afficher lorsque l'on appelle la fonction **print** sur l'objet

```
class NombreComplexe :

   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i

   def __str__ (self) :
      if self.Im >= 0 :
         return str(self.Re) + '+' + str(self.Im) + 'i'
      else :
         return str(self.Re) + str(self.Im) + 'i'


nc1 = NombreComplexe(5,2)
print(nc1)     # affiche 5+2i
```

Surcharge d'opérateur

- Une fois que vous avez défini une classe (i.e. un type), vous voulez pouvoir manipuler facilement les instances de cette classe
- Comme vous le faîtes déjà avec les types de base: ex. faire des opérations mathématiques entre deux entiers, accéder à un élément d'une liste etc.
- Il existe un moyen de spécifier ces opérations dans une classe sans avoir à inventer des noms de fonction
  - ça serait dommage d'avoir à faire `nc3 = nc1.additionnerAvec(nc2)`
  - si on pouvait faire `nc3 = nc1 + nc2`
  - par contre, il faut décider de ce que veut dire par faire l'addition de deux nombres complexes, l'ordinateur ne peut pas décider pour vous

Surcharge d'opérateur

```
class NombreComplexe :

   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i

   def __add__ (self, operandeDroite) :
      resultat = NombreComplexe(self.Re,self.Im)
      resultat.Re += operandeDroite.Re
      resultat.Im += operandeDroite.Im
      return resultat

nc1 = NombreComplexe(5,2)
nc2 = NombreComplexe(-1,1)
nc3 = nc1 + nc2      # nc3 représente le complexe 4+3i
```

Surcharge d'opérateur

- Vous pouvez ainsi surcharger la plupart des opérations mathématiques
  - **__sub__** pour la soustraction (**-**)
  - **__mul__** pour la multiplication (*****)
  - **__truediv__** pour la division (**/**)
  - **__floordiv__** pour la division entière (**//**)
  - **__mod__** pour le modulo (**%**)
  - **__pow__** pour la puissance (******)
  - etc.
- Vous pouvez surcharger les opérateurs combinés
  - **__iadd__** pour l'addition avec affectation (**+=**)
  - **__isub__** pour la soustraction avec affectation (**-=**)
  - etc.

Surcharge d'opérateur

- Les opérateurs de comparaison peuvent aussi être surchargés
  - **__eq__** pour la comparaison (**==**)
  - **__ne__** pour la différence (**!=**)
  - **__gt__** pour la supériorité stricte (**>**)
  - **__ge__** pour la supériorité (**>=**)
  - **__lt__** pour l'infériorité stricte (**<**)
  - **__le__** pour l'infériorité (**<=**)
- Exemple

```
class NombreComplexe :
   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i
   def __eq__ (self, nc) :
      return self.Re == nc.Re and self.Im == nc.Im

nc1 = NombreComplexe(5,2)
nc2 = NombreComplexe(-1,1)
estEgal = (nc1 == nc2)      # estEgal vaut False
```

Surcharge d'opérateur

- Les opérateurs logiques aussi
  - **__and__** pour le ET logique
  - **__or__** pour le OU logique
  - **__xor__** pour le OU EXCLUSIF logique
  - **__invert__** pour la NEGATION (~)
- Exemple

```
class NombreComplexe :
   def __init__ (self,r,i) :
      self.Re = r
      self.Im = i
   def __invert__ (self) :
      self.Im = -self.Im

nc1 = NombreComplexe(5,2)
~nc1   # nc1 représente le complexe (5,-2)
```

Héritage

- L'héritage sert à transférer les données et fonctions membres d'une classe (mère) à une autre classe (fille)
- Si une classe B hérite d'une classe A, les instances de la classe B auront accès aux membres de A plus ceux propres à B
- Une classe fille peut également redéfinir les fonctions membres héritées de la classe mère

Héritage simple

- Pour qu'une classe hérite des membres d'une autre classe, on ajoute le nom de la classe mère en parenthèse derrière la déclaration du nom de la fille

```
class Fille (Mere) :
```

- Par exemple, une classe Chien héritera d'une classe Animal
  - car un chien 'est un' animal, les 'caractéristiques' ou 'fonctionnalités' d'un animal sont toutes présentes dans un chien, on n'a pas envie de les re-décrire toutes une deuxième fois
  - par contre l'inverse n'est pas vrai, un chien a des 'caractéristiques' ou 'fonctionnalités' qui lui sont propres
  - mais il se peut qu'un chien ait des façons différentes d'implémenter certaines 'fonctionnalités' d'un animal

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)
```

```
class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

La classe Nombre est créée normalement. Elle contient une donnée membre nombreReel et une fonction membre pour l'affichage.

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

La classe NombreComplexe hérite de la classe Nombre. C'est-à-dire que les données et fonctions membres de Nombre sont transférées à NombreComplexe.

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Comme créer un nombre complexe est différent de créer un nombre réel, on redéfinit le constructeur hérité de Nombre.

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Mais comme on peut gérer la partie réelle d'un nombre complexe comme un nombre réel, alors on va réutiliser les caractéristiques et fonctions de Nombre. En particulier, on va utiliser la donnée membre nombreReel. Pour l'initialiser, on appelle le constructeur de la classe mère (qui n'est pas appelé sinon).

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Puis on gère la partie imaginaire, propre au nombre complexe, en ajoutant une donnée membre nombreImaginaire.

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

De la même manière, on veut redéfinir comment on affiche un nombre complexe, qui est différent de l'affichage d'un nombre réel.

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Noter que l'on peut accéder aux données membres de la classe mère directement via l'instance de la classe fille (self.donneeMere)

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

On aurait pu réutiliser la fonction membre de la classe mère :
return Nombre.__str__(self) + ', Partie imaginaire : ' + str(self.nombreImaginaire)
mais l'affichage aurait écrit : Nombre réel : XXX, Partie imaginaire : XXX

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Ceci créer une instance de la classe Nombre, avec le réel 4.0

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Ceci créer une instance de la classe NombreComplexe (2+5i)

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Affiche : Nombre réel : 4

Héritage simple

```
class Nombre :
   def __init__(self,n) :
      self.nombreReel = n
   def __str__ (self) :
      return 'Nombre reel : ' + str(self.nombreReel)

class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      Nombre.__init__(self,re)
      self.nombreImaginaire = im
   def __str__ (self) :
      return 'Partie reelle : ' + str(self.nombreReel) + 'Partie imaginaire : '
+ str(self.nombreImaginaire)


n1 = Nombre(4)
n2 = NombreComplexe(2,5)
print(n1)
print(n2)
```

Affiche : Partie réelle : 2, Partie imaginaire : 5

Résumé

- Dans la classe mère, on fait tout comme d'habitude
- Dans la classe fille
  - Déclaration de l'héritage `class Fille (Mere) :`
  - Appel à une fonction mère
    - si redéfinie dans la fille `Mere.fonction(self,...)`
    - sinon `self.fonction(...)`
  - Accès à une donnée fille ou mère `self.donnee`
- Dans un programme externe (ex. principal)
  - Création d'un objet fille ou mère `obj = nomClasse(...)`
  - Accès à un membre fille ou mère `fille.membre`
    - si le membre existe dans fille, c'est celui-là qui est accédé
    - s'il n'existe pas dans fille, ça accède à celui de la mère
    - s'il n'existe pas dans mère non plus, erreur (en fait remonte l'arbre d'héritage car la mère peut elle-même être une fille d'une autre)

Si vous êtes perdu dans votre code

- Vous pouvez toujours accéder à un membre par la syntaxe

```
nomClasse.membre[(self,[...])]
```

- Vous pouvez utiliser la fonction ci-dessous pour vérifier si une classe hérite d'une autre

```
issubclass(nomClasse1,nomClasse2)
```

  - retourne vrai si nomClasse1 hérite de nomClasse2, faux sinon

- Vous pouvez utiliser la fonction ci-dessous pour savoir si une instance est issue d'une classe ou d'une de ses filles

```
isinstance(nomInstance,nomClasse)
```

  - retourne vrai si nomInstance est une instance de nomClasse ou d'une de ses filles, faux sinon

Héritage multiple

- Une classe peut en fait hériter de plusieurs autres classes
- Les membres de cette classe fille seront donc l'union des membres de toutes les classes mères plus ceux propres à la classe fille
- Il suffit d'ajouter les noms des classes mères dans les parenthèses séparés par des virgules

```
class Fille (Mere1, Mere2, ...) :
```

- Lors des appels aux membres, on les cherche dans l'ordre défini dans les parenthèses
  - D'abord dans Fille, puis dans Mere1, puis dans Mere2 etc.
  - Attention si Mere1 hérite d'une ou plusieurs classes, on va chercher dedans avant de passer à Mere2

Héritage multiple

```
class A :
   def __init__(self,a) :
      self.donneeA = a

class B :
   def __init__ (self,b) :
      self.donneeB = b

class C (A,B) :
   def __init__ (self,a,b,c) :
      A.__init__(self,a)
      B.__init__(self,b)
      self.donneeC = c

objA = A(1)
objB = B(2)
objC = C(3,4,5)
print(objA.donneeA)  # affiche 1
print(objB.donneeB)  # affiche 2
print(objC.donneeA)  # affiche 3
print(objC.donneeB)  # affiche 4
print(objC.donneeC)  # affiche 5
```

Quelques remarques

- Les membres qui commencent par un ou deux tirets bas (_ ou __) hors membres spéciaux sont supposés être privés à la classe
  - interdit de les appeler en dehors de la classe
  - pour les données membres, il faut passer par des fonctions spéciales de manipulation (accesseur et mutateur : get et set)

```
class maClasse :
   def __init__ (self,d) :
      self.__donnee = d
   def getDonnee (self) :
      return self.__donnee
   def setDonnee (self,nd) :
      self.__donnee = nd
```

Quelques remarques

- La fonction **super()** est préférée pour accéder à un membre de la classe mère
  - car en cas de changement de classe mère, vous n'avez pas à changer le code

```
class NombreComplexe (Nombre) :
   def __init__ (self,re,im) :
      super().__init__(re)            # au lieu de Nombre.__init__(self,re)
      self.nombreImaginaire = im
```

- Toute classe hérite de la classe **object** de Python, même lorsque ça n'est pas explicitement indiqué
  - cette classe contient en autres les membres **__init__**, **__str__** etc.

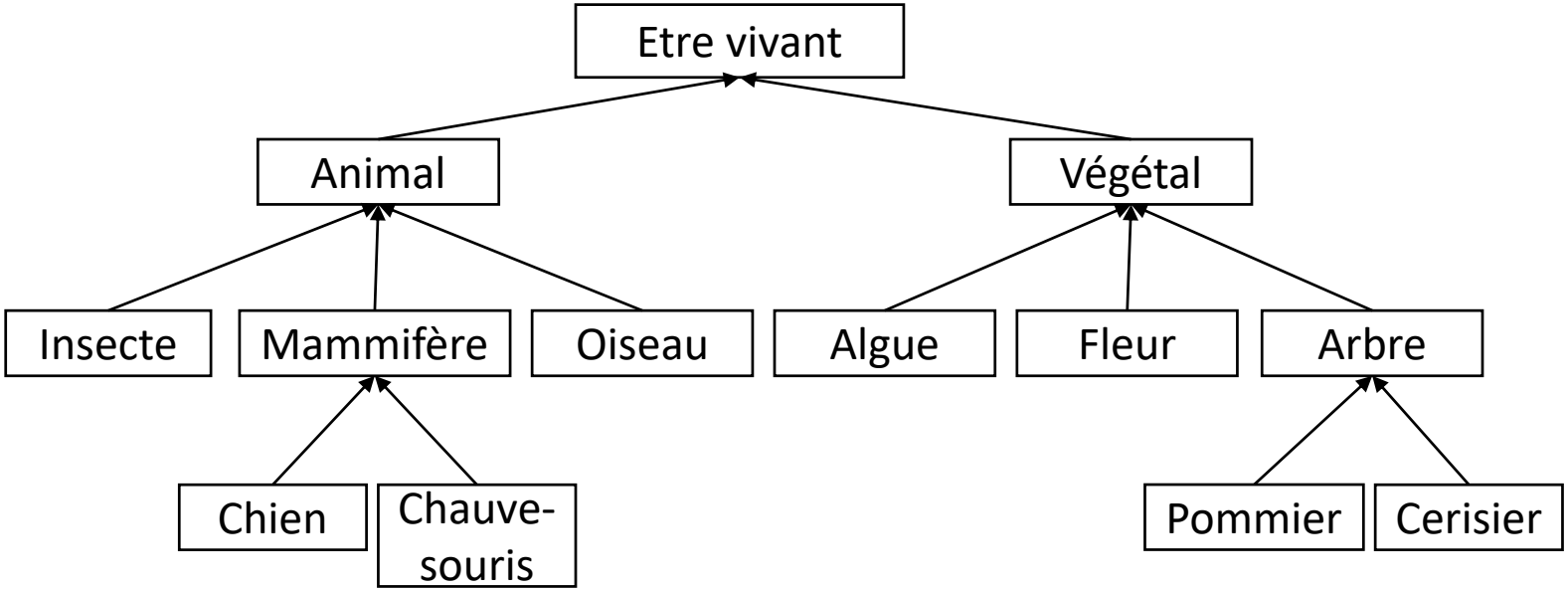
Quelques remarques

- Attention, en cas d'héritage multiple où plusieurs classes contiennent le même nom de fonction membre, il faut le préfixer par le nom de la classe pour lever l'ambiguïté

```
class A :
   def afficher(self) :
      print('A')

class B :
   def afficher(self) :
      print('B')

class C (A,B) :
   def afficher(self) :
      print('C')

objA = A()
objB = B()
objC = C()
objA.afficher()         # affiche A
objB.afficher()         # affiche B
objC.afficher()         # affiche C
A.afficher(objC)        # affiche A
B.afficher(objC)        # affiche B
C.afficher(objC)        # affiche C
```

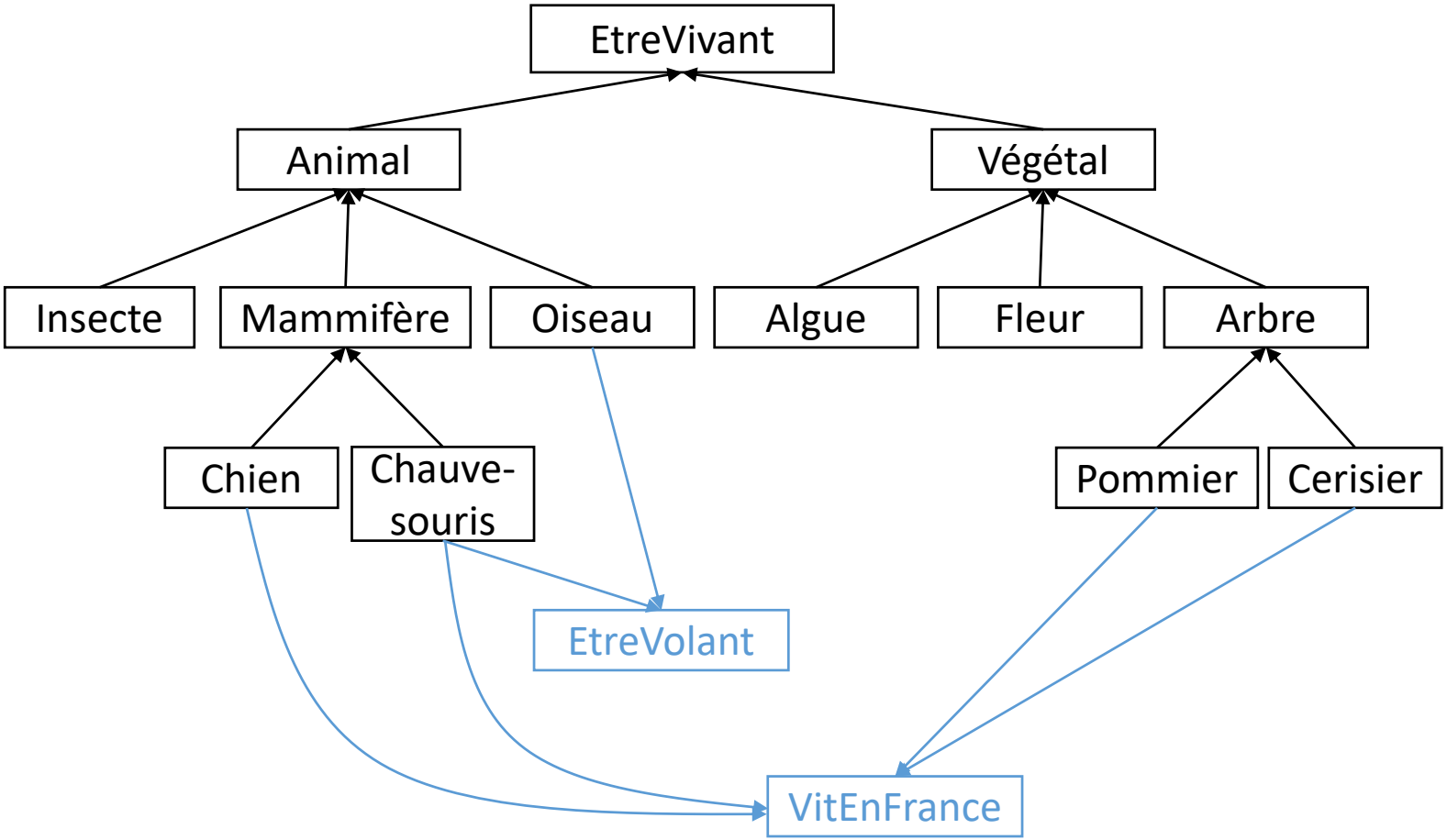
Quelques remarques

- L'héritage est aussi beaucoup observé avec de multiples classes qui héritent toutes d'une même classe mère
  - Les classes Chien, Chat, Poule, Cochon etc. hériteront toutes de Animal
- Les hiérarchies de classes sont souvent visuellement représentées dans un diagramme



Quelques remarques

- Exemple en introduisant de l'héritage multiple



Résumé de la terminologie

- Classe
  - un type défini par le programmeur regroupant des membres (données et fonctions, de classe et d'instance)
- Instance / objet de classe
  - une variable individuelle d'une certaine classe
- Donnée membre ou attribut
  - une variable qui stocke une donnée associée à une instance ou à une classe
- Fonction membre ou méthode
  - une fonction qui s'appelle sur une instance d'une classe ou sur une classe
- Surcharge de fonction membre (dont les opérateurs)
  - affectation de différents comportements à une fonction particulière
- Héritage de classe
  - transfert de caractéristiques d'une classe à une autre

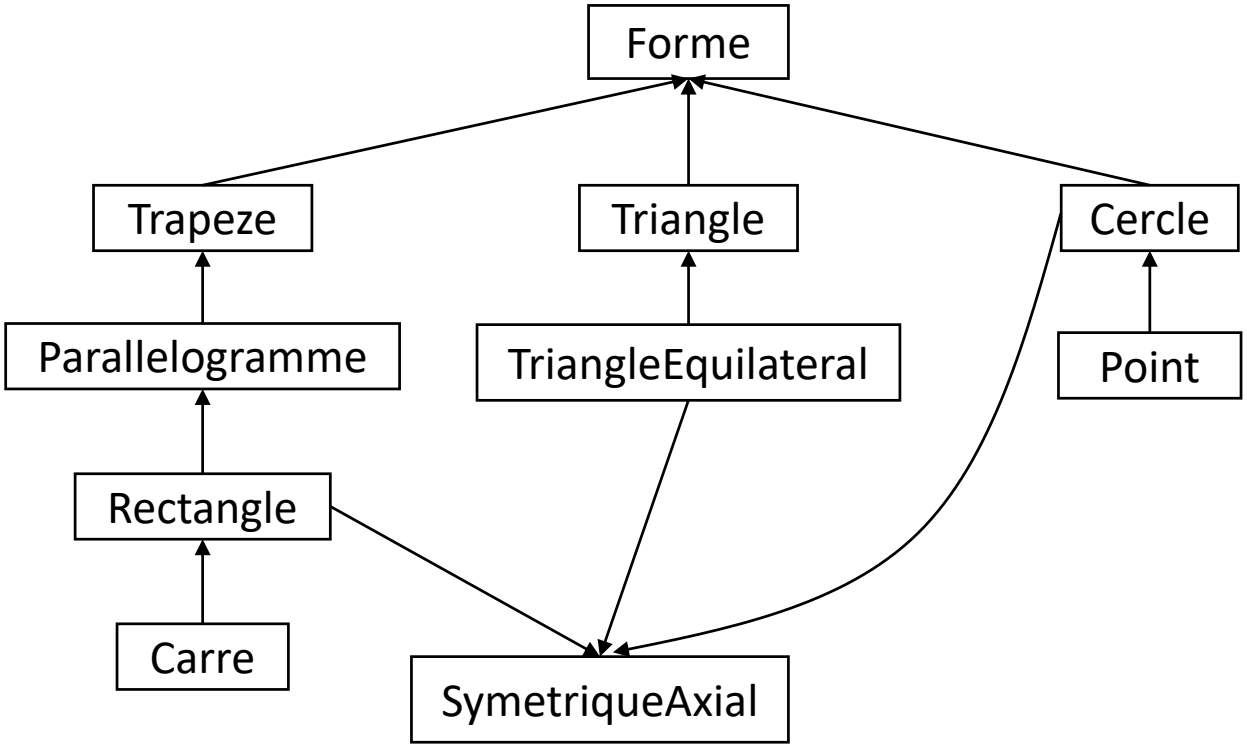
Exercice : géométrie 2D

- Une forme est caractérisée par une position 2D
  - Un trapèze est une forme à quatre côtés avec deux côtés opposés parallèles caractérisé par une longueur, largeur et hauteur
    - Un parallélogramme est un trapèze avec des côtés opposés parallèles deux à deux
      - Un rectangle est un parallélogramme avec des côtés perpendiculaires
        - Un carré est un rectangle avec des côtés de même longueur
  - Un triangle est une forme à trois côtés avec une base et une hauteur caractérisée par une hauteur et une base
    - Un triangle équilatéral est un triangle avec les côtés de même longueur
  - Un cercle est une forme avec un rayon
    - Un point est un cercle ponctuel
- Dessiner un diagramme des classes et l'implémenter en Python (données membres et constructeurs)

Exercice : géométrie 2D

- Ajouter à ces classes la fonctionnalité de pouvoir calculer l'aire de l'objet 2D
- Ajouter la classe SymetriqueAxial caractérisée par les paramètres a et b d'une équation de droite ($y = a \times x + b$) décrivant un des axes de symétrie axiale de l'objet

Exercice : géométrie 2D



Exercice : géométrie 2D

```
class SymetriqueAxial:
    def __init__(self, parama, paramb):
        self.a = parama
        self.b = paramb

class Forme:
    def __init__(self, x, y):
        self.posX = x
        self.posY = y

class Trapeze(Forme):
    def __init__(self, x, y, long, larg, haut):
        Forme.__init__(self, x, y)
        self.hauteur = haut
        self.longeur = long
        self.largeur = larg

    def aire(self):
        return (self.longeur + self.largeur) *
self.hauteur / 2.0

class Parallelogramme(Trapeze):
    def __init__(self, x, y, long, haut):
        Trapeze.__init__(self, x, y, long, long, haut)

class Rectangle(Parallelogramme, SymetriqueAxial):
    def __init__(self, x, y, long, larg):
        Parallelogramme.__init__(self, x, y, long, larg)
        SymetriqueAxial.__init__(self, 0, y)
```

```
class Carre(Rectangle):
    def __init__(self, x, y, cote):
        Rectangle.__init__(self, x, y, cote, cote)

class Triangle(Forme):
    def __init__(self, x, y, h, b):
        Forme.__init__(self, x, y)
        self.hauteur = h
        self.base = b

    def aire(self):
        return (self.hauteur * self.base) / 2.0

class TriangleEquilateral(Triangle, SymetriqueAxial):
    def __init__(self, x, y, cote):
        Triangle.__init__(self, x, y, math.sqrt(3) *
cote / 2.0, cote)
        SymetriqueAxial.__init__(self, 0, y)

class Cercle(Forme, SymetriqueAxial):
    def __init__(self, x, y, r):
        Forme.__init__(self, x, y)
        self.radius = r
        SymetriqueAxial.__init__(self, 0, y)

    def aire(self):
        return math.pi * self.radius * self.radius

class Point(Cercle):
    def __init__(self, x, y):
        Cercle.__init__(self, x, y, 0)
```