Programmation orientée objet en Python

Module M2207

Rushed Kanawati

A3 - LIPN UMR CNRS 7030
Université Paris 13
rushed.kanawati@lipn.univ-paris13.fr

February 10, 2017

Plan

Le module M2207

Organisation

- Intervenants : R. Kanawati, L. Saiu, Q. Chateiller.
- 4 cours, 3 TD, 5 TP
- Contrôle continue

Objectifs & Compétences visées

1. Proposer une solution logicielle orientée objet conforme à un cahier des charges
2. Concevoir une application sous forme d'objets et de relations
3. *Développer des applications client-serveur dans un langage orienté objet*

Programmation orientée objet : objectifs

Contexte

- Les domaines d'applications des logiciels sont de plus en plus **complexes** et **critiques**.
- *Système d'exploitation, Protocoles réseaux, Jeux, Pilotage de réacteurs nucléaires, Voiture autonome, ...*
- *Plus un système est complexe, plus il est susceptible d'effondrement*.
- Le coût de développement de logiciel devient très important.

Programmation orientée objet : objectifs

Qualités requises d'un logiciel

- **Simplicité** : *faciliter la maintenance*.
- **Extensibilité** : *faciliter l'adaptation aux changements du domaine d'application*.
- **Décentralisation** : *faciliter le développement collaboratif*.
- **Réutilisabilité** : *augmenter la fiabilité et réduire les coûts et les délais de développement*.

Programmation orientée objet : objectifs

Approches

- **La modularité** : Structurer un logiciel en ensemble de **modules**.
- Un module = ensemble de **fonctions** spécifiques pour traiter un problème donné

- **L'approche Objet** : Un objet = Un module autonome qui encapsule des données et des fonctions de manipulation de ces données.

Limites de la programmation procédurale : exemple

Gestion d'une base d'étudiants

Un étudiant est défini par : prénom, nom, num_étudiant, note.

Solution 1

```
1 etudiant1=["Max","LeGrand","100100",15]
2 etudiant2=["Min","Petit","100101",10]
3
4 etudiants=[]
5 etudiants.append(etudiant1)
6 etudiants.append(etudiant1)
7
8 def setNote(etudiant,note):
9     etudiant[3]=note
```

Limites de la programmation procédurale : exemple

Problème

```
1 employee=["Alexandre","LeGrand","10000",2000]
2 #employee (prenom, nom, numero, salaire)
3
4 setNote(employee,0)
```

Limites de la programmation procédurale : exemple

Approche objet

```
1 class Etudiant:
2
3     def __init__(self,prenom,nom,num,note=0):
4         self.prenom=prenom
5         self.nom=nom
6         self.num_etudiant=num
7
8     def setNote(self,note):
9         self.note=note
```

Limites de la programmation procédurale : exemple

Approche objet

```
1  class Employee:
2      def __init__(self,prenom,nom,num,salaire=1200):
3          self.prenom=prenom
4          self.nom=nom
5          self.num=num
6
7      def setSalaire(self,salaire):
8          self.salaire=salaire
9
10 e=Employer(prenom="Alexndre",nom="LeGrand",
11     num="100",salaire=1200)
12 e.setNote(e,0) # erreur ! opération non définie
```

Encore mieux !

```
1 class Personne:
2
3     def __init__(self,prenom,nom,num):
4         self.prenom=prenom
5         self.nom=nom
6         self.num=num
7
8     def getNum(self):
9         return (self.num)
```

Encore mieux !

```
1
2 class Etudiant(Personne):
3     def __init__(self,prenom,nom,num,note=0):
4         Personne.__init__(self,prenom,nom,num)
5         self.note=note
6
7     def setNote(self,note):
8         self.note=note
```

Encore mieux !

```
1 class Employee(Personne):
2     def __init__(self,prenom,nom,num,salaire):
3         Personne.__init__(self,prenom,nom,num)
4         self.salaire=salaire
5
6     def setSalaire(self,salaire):
7         self.salaire=salaire
```

Encore mieux !

```
1 class Apprenti(Etudiant,Employee):
2     def __init__(self,prenom,nom,num,note,salaire):
3         Etudiant.__init__(self,prenom,nom,num,note)
4         self.setSalaire(salaire)
```

Définitions

Programme orienté objet

Un ensemble d'objets qui communiquent entre eux par échange de messages.

Un objet

est défini par un état et un comportement

- **Etat** : ensemble d'attributs.
- **Comportement** un ensemble de services (fonctions) (méthodes) que l'objet peut exécuter.
- Dans un *modèle objet pur*, seules les méthodes d'un objet peuvent accéder aux attributs de l'état.

Définitions

Classe

- Un modèle à partir duquel on fabrique un objet.
- Création d'objet = instanciation
- Une classe doit fournir au moins une méthode dite **constructeur** qui permet de créer des objets.
- La méthode constructeur a la syntaxe suivante :

```
1 def __init__(self, ...):
2   /* initialisation de l'état de l'objet */
```

- Le premier argument d'une méthode est **self** une référence vers l'objet local.

Classe : exemple

```
1  class point:
2      def __init__(self): # constructeur sans parmètres
3          self.x = 0 # attribut d'instance
4          self.y =0  # attribut d'instance'
5      def move(self,dx,dy):
6          self.x=self.x+dx
7          self.y=self.y+dy
8
9  p1= point()  # création d'un point p1
10 p2= point() # création d'un point p2
```

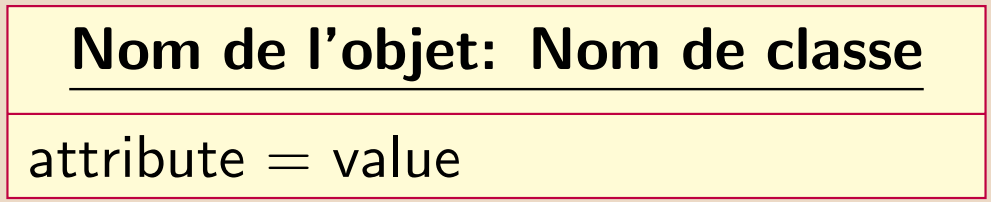
Notation graphique : UML (Unified Modeling Language)

Représentation de classes

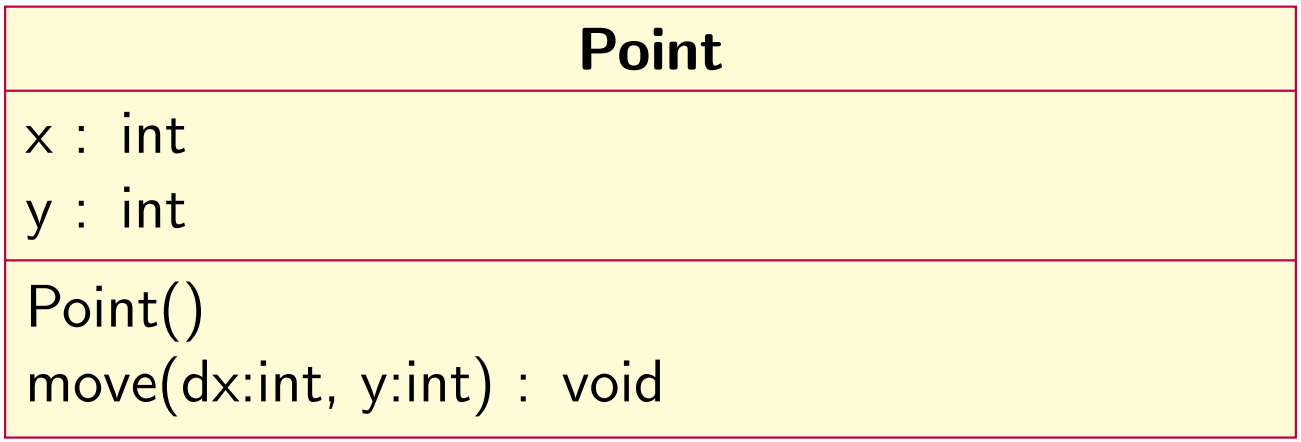
Nom de Classe
attribut1 : type attribut 2 : type, valeur par défaut
Constructeur(paramètres) methode1(parametèrs) : type de retour methode2(parametèrs) : type de retour

Notation graphique : UML

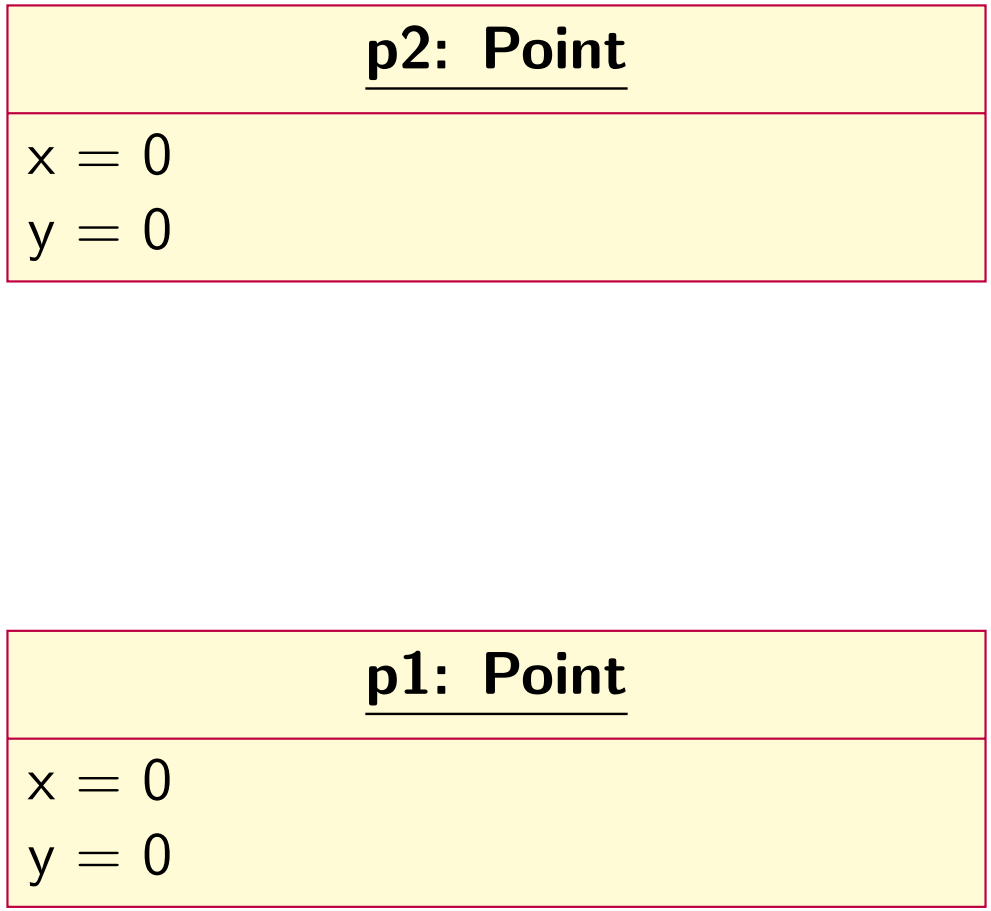
Représentation d'objets



UML : Exemple



UML : Exemple



UML

- Caractéristiques
  - Principe : s'abstraire du langage de programmation et des détails d'implantation
  - Normalisée par l'OMG (Object Management Group)
- Utilisations :
  - esquisse : communiquer avec les autres sur certains aspects
  - plan : le modèle sert de base pour le programmeur
  - Génération de code

Classes : fonctions prédéfinies

isinstance(objet,classe)

- Renvoie `True` si *objet* est instancié à partir de la classe *classe*

```
1 p1 = point()
2 isinstance(p1,point)     # returns True
3 isinstance(p1,Etudiant) # returns False
```

issubclass(classe1,calsse2

- Renvoie `True` si *classe1* est une sous-classe de la classe2.

```
1 issubclass(Etudiant,Personne)    # returns True
2 isinstance(Personne,Etudiant) # returns False
```

Types d'attributs

Attribut d'instance

- Attribut défini dans la méthode constructeur.
- La valeur d'un attribut d'instance est propre à chaque instance.
- L'accès à l'attribut d'instance est donnée par : `nomObjet.nomAttribut`

Attribut de classe

- Attribut défini au niveau de la classe
- **La valeur est partagée par tous les objets instanciés à partir de la classe**.
- L'accès à l'attribut de classe est donnée par : `nomDeClasse.nomAttribut`

Attributs : exemples

```
1  class CompteBancaire:
2      decouvert=0 #attribut de classe
3
4      def __init__(self,solde=0):
5          self.solde=solde  # attribut d'objet
6
7      def retirer(self,montant):
8          if (self.solde-montant>=CompteBancaire.decouvert):
9              self.solde=self.solde-montant
10         else:
11             print "solde insuffisent"
12
13     def setDecouvert(self,seuil):
14         CompteBancaire.decouvert= -seuil
```

Attributs : exemples

```
1 c1 = CompteBancaire(1000)
2 c2 = CompteBancaire(800)
3
4 c2.retirer(1000)  #solde insuffisant
5 c1.setDecouvert(300)
6 c2.retirer(1100) #succès
```

Protection d'attributs

- En `Python` tous les attributs sont des attributs publiques : accessibles à partir d'autres classes : L'encapsulation n'est pas respectée !
- Exemple : `c.solde = 10 000`
- il est possible de *brouiller* l'accès à un attribut en le nommant comme suit : `__nomAttribut`
- Un attribut *brouillé* est renommé automatiquement à `_NomClasse_NomAttribut`

Gestion des attributs d'instance

Accès en lecture

- `nomObjet.nomAttribut`
- **Fonction prédéfinie** : `getattr(objet,attribut)`
  - Exemple : `getattr(c,"solde")`
  - Attention le nom de l'attribut est passé comme un **str**
  - Si l'attribut n'existe pas la méthode `__getattr__(self,att)` sera appelée
- L'ensemble des attributs d'instance sont regroupés dans un attribut spéciale : `__`**dict**`__` de type dictionnaire
  - Exemple : `c.__dict__["solde"]`
  - `c.__dict__.keys() # retourne une liste des attributs définis`

Gestion des attributs d'instance

Accès en écriture

- `nomObjet.nomAttribut = valeur`

- **Fonction prédéfinie** : `setattr(objet,attr,val)`
  - **Attention : Si l'attribut n'existe pas il sera crée !!**
  - Les deux approches précédentes passent par un appel implicite à la fonction prédéfinie **__setattr__(self,att,val)**

- `nomObjet.__dict__["nomAttribut"] = val`

Gestion d'attributs de classes

- Même principe que pour les attributs d'instance.
- Les attributs de classes et les méthodes (d'instance ou de classes) sont dans le dictionnaire : `NomClasse.__dict__`.

```
1  class MathTools:
2      pi=3.141592
3      def isPair(n):
4          return (n%2==0)
5      isPair = staticmethod(isPair)
6
7  print Mathtools.__dict__
8
9
10 {'__module__': '__main__', 'pi': 3.141592, 'isPair': <
       staticmethod object at 0x1006ed670>, '__doc__': None}
```

Attributs : autres fonctions prédéfinies

hasattr(objet,att)

- Retourne True si objet possède un attribut att.
- Cette fonction fait appel à getattr et retourne False en cas de lever d'exception.

delattr(objet,name)

- Effacer l'attribut name de l'instance désigné par objet

```
1 class A:
2   def __init__(self,at):
3     self.at=at
4 a=A(5)
5 b=A(6)
6 delattr(a,at)
7 print b.at  # affiche 6
8 print a.at  # AttributeError: A instance has no attribute 'at
    '
```

Méthode d'instances

- Une méthode qui s'exécute dans l'espace d'un objet.
- A **self** comme premier argument.
- Appel : `nomObjet.nomMethod()`

```
1  class CompteBancaire:
2      def __init__(self,solde=0):
3          self.solde=solde  # attribut d'objet
4
5      def retirer(self,montant):
6          if (self.solde-montant>=0):
7              self.solde=self.solde-montant
8          else:
9              print "solde insuffisent"
10
11
12 c=CompteBancaire(2000)
13 c.retirer(1000)
```

Méthodes de classe

- Une méthode qui peut s'exécuter sans instancier la classe (pas de création d'objet)
- Souvent utilisée pour programmer des services génériques.
- Une méthode statique n'a pas **self** comme premier argument
- Elle doit être déclarée statique explicitement en utilisant l'instruction **staticmethod** :
  `nomMethode= staticmethod(nomMethode)`

Méthodes de classe : Exemple

```
1 class MathTools:
2     def isPair(n):
3         return (n%2==0)
4     isPair = staticmethod(isPair)
5
6 print MathTools.isPair(5)
```

Comparaison d'objets

```
1  class point:
2      def __init__(self): # constructeur sans paramètres
3          self.x = 0 # attribut d'instance
4          self.y =0  # attribut d'instance'
5      def move(self,dx,dy):
6          self.x=self.x+dx
7          self.y=self.y+dy
8
9  p1= point()  # création d'un point p1
10 p2 = point() # création d'un point p2
11
12
13 print p1==p2   # False
```

Comparaison d'objets

```
1  class point:
2      def __init__(self): # constructeur sans paramètres
3          self.x = 0 # attribut d'instance
4          self.y =0  # attribut d'instance'
5      def move(self,dx,dy):
6          self.x=self.x+dx
7          self.y=self.y+dy
8      def __cmp__(self,other):
9          if (self.x==other.x) and (self.y==other.y):
10             return 0
11         else:
12             return 1
13
14 p1= point()  # creation d'un point p1
15 p2 = point() # creation d'un point p2
16
17
18 print p1==p2   # True
```

Comparaison d'objets

`__cmp__`

- Permet de comparer deux objets (par nécessairement de la même classe)
- **Valeurs de retour :** { **-1, 0, 1**}
  - -1 si l'objet est plus petit que l'autre.
  - 0 si objets égaux
  - 1 si l'objet est plus grand que l'autre
- Fonction utilisée implicitement par la fonction prédéfinie `cmp()`

Autres fonctions de comparaison

- __lt__ : $<$
- __le__ : $\leq$
- __eq__ : $=$
- __gt__ : $>$
- __ge__ : $\geq$

Affichage d'objet

__str__

permet de retourner une chaîne de caractères pour afficher une représentation de l'objet (avec l'instruction `print`).

```
1  class Personne:
2      def __init__(self,prenom,nom,num):
3          self.prenom=prenom
4          self.nom=nom
5          self.num=num
6
7      def __str__(self):
8          """afficher plus joliment notre objet"""
9          return "{} {}, numero: {}".format(
10                     self.prenom, self.nom, self.num)
```

Exceptions

Principe

Séparer la détection d'une anomalie de son traitement.

Exception : Rappel

- Une *exception* est levée pour signaler une anomalie lors de l'exécution d'une instruction.
  - Division par zéro, conversion de type impossible, indice inexistant
  - ...
- On peut forcer la levée d'une exception en utilisant l'instruction : **raise**

```
1 if var < 0:
2   raise Exception
```

Exceptions & Objets

Exception : Rappel

- Si une exception est levée, l'exécution d'une méthode s'arrête et on retourne l'exception à l'appelant
- L'appelant *peut* traiter l'anomalie génératrice de l'exception si l'appel est fait dans un bloc `try/catch`

```
1 try:
2   instruction1
3   instruction2
4   ...
5 except exception:
6   actions
```

Exceptions

Exceptions prédéfinies

- `ZeroDivisionError` : division par zéro
- `ValueError` : valeur inacceptable
- `IndexError` : Accès à un indice inexistant
- `TypeError` : type d'argument incorrect
- `OSError` : erreur provoqué âr un appel système
- `MemoryError` : épuisement de la mémoire
- ...

Exceptions utilisateur

- Les exceptions sont des objets.
- On peut définir de nouvelles exceptions en **spécialisant** la classe `Exception`

Traitement des exceptions

- Chaque type d'exception peut avoir un traitement différent

```
try:
   ...
   instructuctions
   ...
except ExceptionType1 :
   actions
except ExceptionType2:
   actions
...
finally:
   actions à prendre dans le cas par défaut
```

Exceptions & Objets

- Utiliser les exceptions afin de vérifier la bonne utilisation des méthodes.
- Exemple : vérifier que les paramètres d'entrée d'une méthode ont les bon types et les bonnes valeurs
- Vérification simplifiée par l'instruction **assert**

assert

- Syntaxe : **assert Expression[, Arguments]**
- Exemple : assert (Temperature 0),"trop froid !!"
- Fonctionnement : Si *condition* est évaluée à `False`, alors lever `AssertionError`

assert : Exemple 1

Instanciation d'objets

```
1  class CompteBancaire:
2
3      def __init__(self,banque,titulaire,solde):
4          assert isinstance(banque,str) and len(banque)>0
5          assert isinstance(titulaire,str) and len(titulaire)>0
6          assert isinstance(float(solde),float) and solde>=0
7
8          self.banque=banque
9          self.titulaire=titulaire
10         self.solde=solde
```

assert : Exemple 2

Interdir la création dynamique d'attributs

```
1 classe A:
2   ....
3
4   def __setattr__(self,name,value):
5     if name not in self.__dict__.keys():
6       pass
7     ...
```

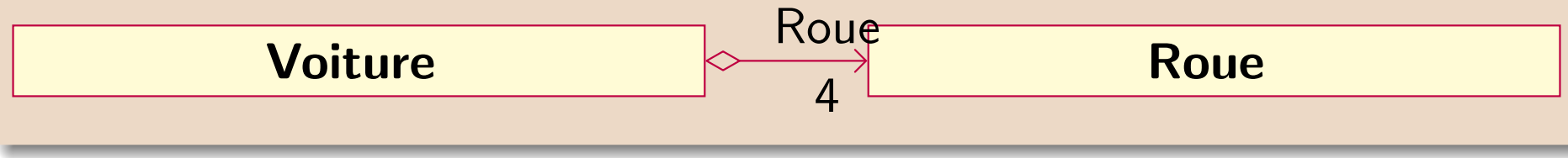
Relations entre classes

Deux types de relations

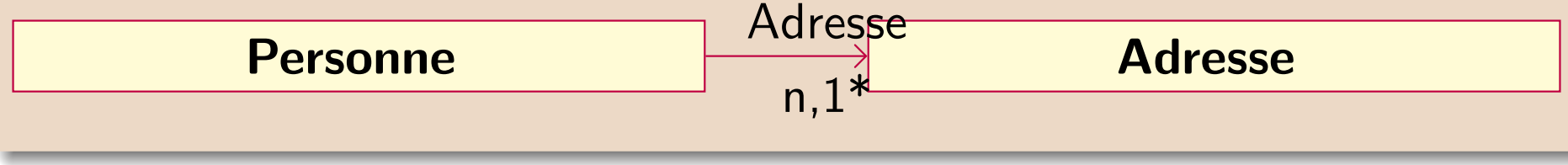
- **Relation d'utilisation**
  - C'est une relation de type : `has-a`
  - Une classe peut avoir des attributs qui sont d'instance d'autres classes.
  - Exemple : Une voiture a 4 roues.
  - Différents types de couplage entre les classes : composition, agrégation, association, etc.
- **Relation de spécialisation/généralisation**
  - C'est une relation de type : `is-a` ou une relation de **héritage**
  - A doit hériter de B si A est un **cas particulier** de B.
  - On parle de : Super classe/ sous classe ou classe mère / classe fille.
  - Exemple : `Chat, Lion, Félin, Animale, Chat Angora, Chien` : Quelles relations d'héritage ?

Relation d'utilisation : UML

Relation d'agrégation



Relation d'association



Relation d'utilisation : Exemple

Problem : un segment est défini par deux points

```
1  import math
2  class Point:
3    def __init__(self, x=0, y=0):
4      self.x = float(x)
5      self.y = float(y)
6
7    def __str__(self):
8      return "(%s ; %s)" % (self.x, self.y)
9
10   def move(self, dx, dy):
11     self.x= self.x+dx
12     self.y= self/y+dy
13
14   def distance(self, autre):
15     dx2 = (self.x - autre.x) ** 2
16     dy2 = (self.y - autre.y) ** 2
17     return math.sqrt(dx2 + dy2)
```

Relation d'utilisation : Exemple

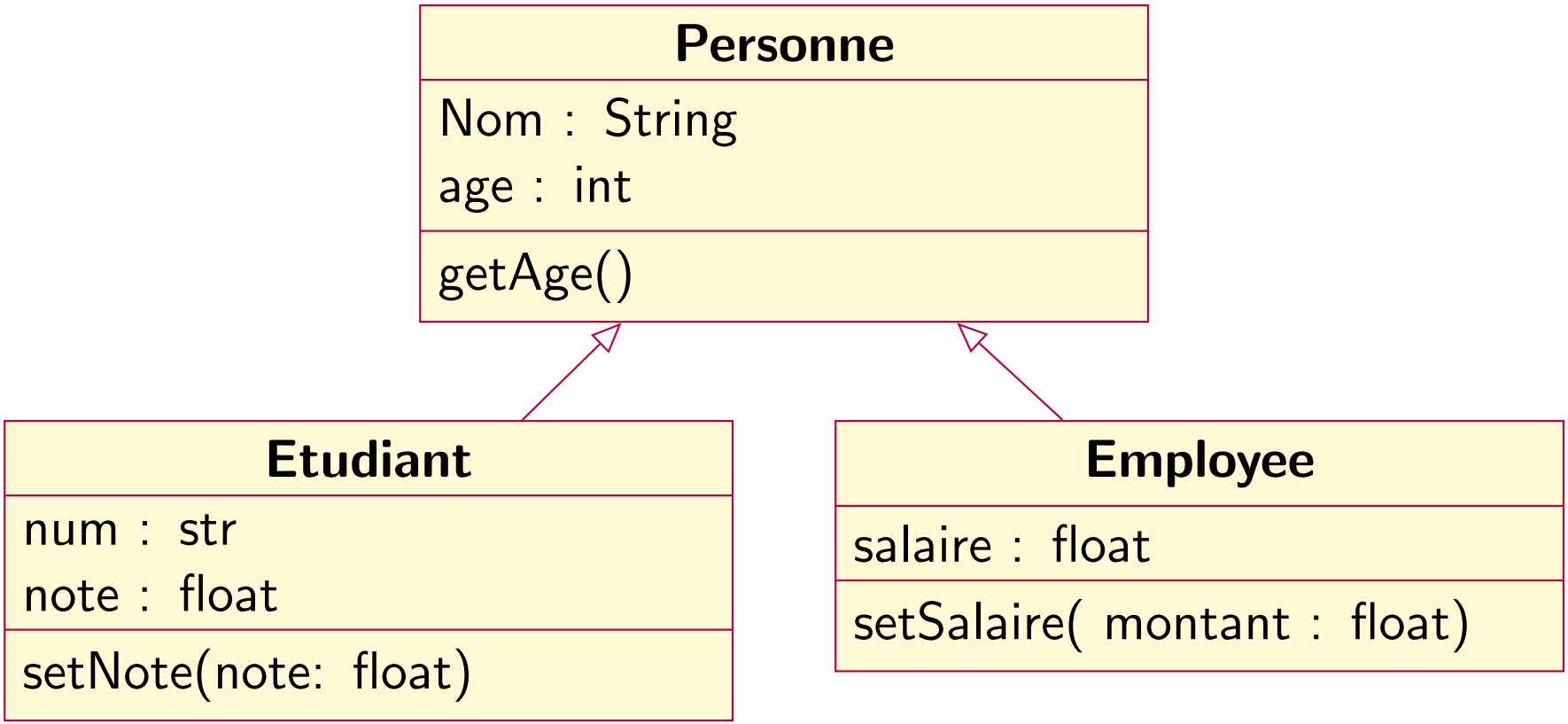
Problem : un segment est défini par deux points

```
1  class Segment:
2    def __init__(self, e1, e2):
3      assert isinstance(e1,Point) and isinstance(e2,Point)
4
5      self.extremite1 = e1
6      self.extremite2 = e2
7
8    def __str__(self):
9      return "[%s - %s]" % (self.extremite1, self.extremite2)
10
11   def move(self, dx, dy):
12     self.extremite1.move(dx, dy)
13     self.extremite2.move(dx, dy)
14
15   def longueur(self):
16     return self.extremite1.distance(self.extremite2)
```

Relation d'héritage

- Principe : définir une nouvelle classe par spécialisation d'une (ou plusieurs) classes existantes.
- La sous-classe :
  - récupère automatiquement tous les attributs et les méthodes des supers classes ;
  - peut enrichir la description de la classe par l'ajout des attributs et des méthodes ;
  - peut modifier les types des attributs hérités ;
  - peut modifier le comportement des méthodes héritées.

Relation d'héritage : UML



Relation d'héritage : syntaxe Python

Héritage Simple

```
1 class A(B):
2     ...
3     # A est une sous classe de B
```

Héritage multiple

```
1 class A(B,C,D):
2     ...
3     # A est une sous classe de B,C,D
```

Héritage multiple : gestion de conflits

```
1 class A(B,C,D):
2     ...
3     # A est une sous classe de B,C,D
```

Si les classes B; C,D ont les mêmes attributs, méthodes, la classe A hérite de l'attribut/méthode, cité en premier (ici de celui de B)

Héritage : exemple

Point coloré

```
1  class PointColore(Point):
2      def __init__(self,x=0, y=0,couleur):
3          Point.__init__(self, x, y)
4          self.couleur = nom
5
6      def __str__(self): # redéfinition
7          return "%s:%s" % (Point.__str__(self),self.couleur )
8          # utilisation de la version de __str__ dans Point
9
10     def setCouleur(couleur): # une nouvelle méthode
11         self.couleur = couleur
```

Classe abstraite

Classe abstraite

- **Quoi ?** : Une classe qui ne peut pas fabriquer des objets !
- **Pourquoi ?** : renforcer l'extensibilité des programmes en factorisant des attributs et des méthodes communs à différents classes.
- **Exemple :**
  - un point, un segment, un carré, un rectangle, un cercle, ..., sont des figures géométriques
  - On peut déplacer, colorier une figure, calculer sa superficie, son preimètre, ...
  - L'exécution des méthodes citées ci-dessus dépend de la nature de la figure
  - On peut définir ces méthodes d'une manière abstraite dans une classe abstraite !

Classe abstraite : Exemple

```
1  class FigureGeometrique:
2
3    def __init__(self,centre):
4      assert (isinstance(centre,Point)
5      self.centre=centre
6      raise NotImplementedError
7
8    def move(self,dx,dy):
9      self.centre.move(dx,dy)
10
11   def superficie(self):
12     raise NotImplementedError
13   def perimetre(self):
14     raise NotImplementedError
```

Classe abstraite : Exemple

```
1  class Carre(FigureGeometrique):
2    def __init__(self,centre,cote):
3      assert isinstance(centre,Point)
4      assert isinstance(float(cote),float) and float(cote)>=0
5
6      try:
7        super.__init__(centre)
8      except:
9        self.cote=cote
10
11   def superficie(self):
12     return (self.cote)*(self.cote)
13
14   def permimetre(self):
15     return(4*self.cote)
```

Classe abstraite : Exemple

```
1  class Circle(FigureGeometrique):
2    pi=3.14
3    def __init__(self,centre,rayone):
4      assert isinstance(centre,Point)
5      assert isinstance(float(rayon),float) and float(rayon)>=0
6
7      try:
8        super.__init__(centre)
9      except:
10       self.rayon=rayon
11
12   def superficie(self):
13     return Circle.pi*self.rayon*self.rayon
14
15   def permimetre(self):
16     return(2*Circle.pi*self.rayon)
```

Classe abstraite

Problem

- La détection de l'absence d'une méthode abstraite se fait lors de l'appel !! trop tard !
- Comment obliger une classe qui hérite d'une classe abstraite d'implementer les méthodes abstraites ?
- Solution : utiliser le module `abc` (Abstract Base Class) qui permet de déclarer **explicitement** qu'une classe/ou une méthode est abstraite.

Module abc : utilisation

```
1  import abc
2  class FigureGeometrique(metaclass=abc.ABCMeta):
3
4    @abc.abstractmethod
5    def __init__(self,centre):
6      assert (isinstance(centre,Point)
7      self.centre=centre
8      raise NotImplementedError
9
10   def move(self,dx,dy):
11     self.centre.move(dx,dy)
12
13   @abc.abstractmethod
14   def superficie(self):
15     raise NotImplementedError
16
17   @abc.abstractmethod
18   def perimetre(self):
19     raise NotImplementedError
```