Introduction à la programmation objet en Python

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Objectifs et structure de ce support

Objectifs de ce support :

- une introduction à la programmation objet
- en l'illustrant avec le langage Python
- et le diagramme de classe de la notation UML (Unified Modelling Language)

Principaux éléments :

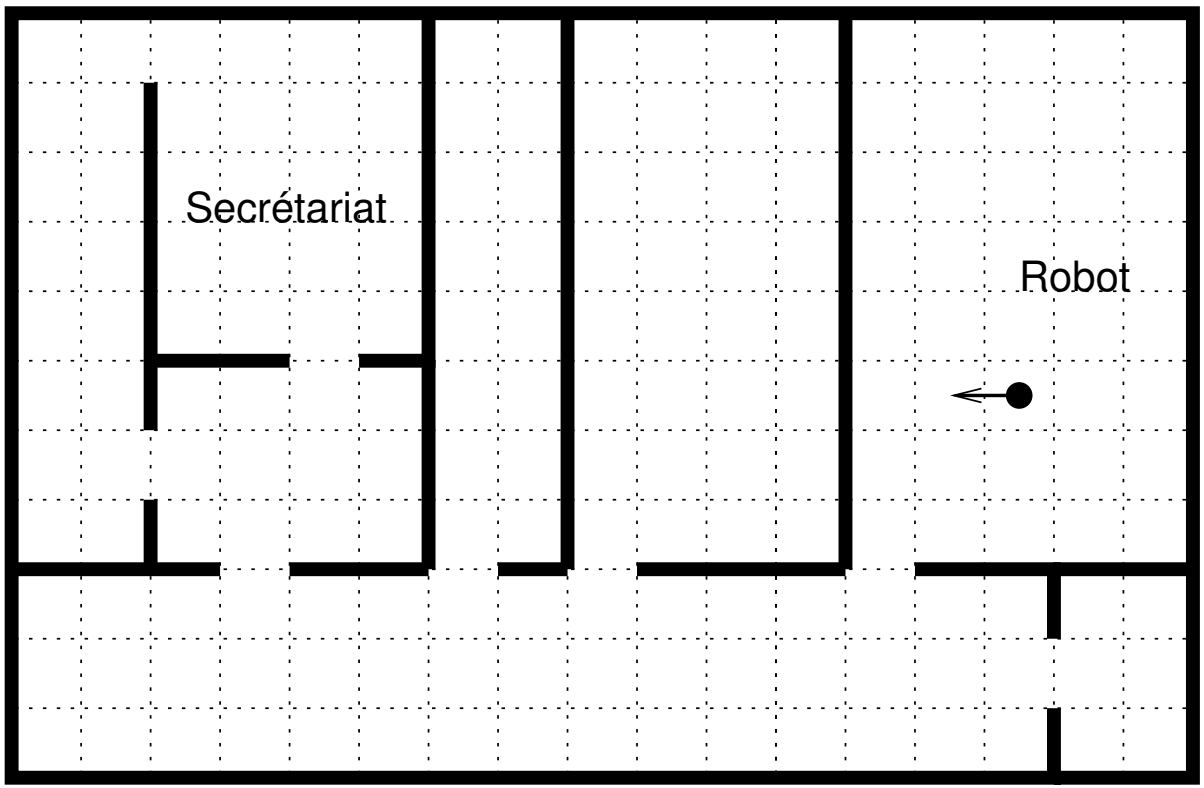
- Exemple introductif : passage de l'impératif à une approche objet
- Encapsulation : classe, objets, attributs, méthodes, etc.
- Relations entre classes
  - Relations d'utilisation
  - Relation d'héritage
- Des éléments méthodologiques

Sommaire

- Les robots

Modéliser un robot

Exercice 1 Modéliser un robot capable d'avancer d'une case et de pivoter de 90° vers la droite. On pourra alors le guider de la salle de cours (position initiale du robot) jusqu'au secrétariat.



Types et sous-programmes associés (pseudo-code)

1. On sait définir un type Robot :

```
RobotType1 =
    Enregistrement
        x: Entier;     -- abscisse
        y: Entier;     -- ordonnée
        direction: Direction
    FinEnregistrement


Direction = (NORD, EST, SUD, OUEST)
```

3. On sait utiliser des robots :

```
  Variable
      r1, r2: RobotType1;
  Début
      initialiser(r1, 4, 10, EST)
      initialiser(r2, 15, 7, SUD)
      avancer(r1)
      pivoter(r2)
  Fin
```

2. On sait modéliser ses opérations :

```
Procédure avancer(r: in out RobotType1)
    -- faire avancer le robot r
  Début
    ...
  Fin

Procédure pivoter(r: in out RobotType1)
    -- faire pivoter le robot r
    -- de 90° à droite
  Début
    ...
  Fin

Procédure initialiser(r: out RobotType1
        x, y: in Entier, d: in Direction)
    -- initialiser le robot r...
   Début
     r.x <- x
     r.y <- y
     r.direction <- d
   Fin
```

Module Robot en Python (fichier robot_dict.py)

```
1  """ Modèle très limité de robot qui ne sait qu'avancer d'une
2      case et pivoter à droite de 90°.  Il est repéré par son abscisse x
3      son ordonnée y et sa direction (un entier dans 0..3).
4  """
5  _directions = ('Nord', 'Est', 'Sud', 'Ouest')   # direction en clair
6  _dx = (0, 1, 0, -1)     # incrément sur X en fonction de la direction
7  _dy = (1, 0, -1, 0)     # incrément sur Y en fonction de la direction
8
9  def robot(x, y, direction):
10     """ Créer un nouveau robot. """
11     return dict(x=x, y=y, direction=_directions.index(direction))
12
13 def avancer(r):
14     """ Avancer le robot r d'une case dans la direction courante. """
15     direction = r['direction']
16     r['x'] += _dx[direction]
17     r['y'] += _dy[direction]
18
19 def pivoter(r):
20     """ Pivoter de 90° à droite le robot r. """
21     r['direction'] = (r['direction'] + 1) % 4
22
23 def afficher(r):
24     """ Afficher le robot r. """
25     print('Robot(x=', r['x'], ',␣y=', r['y'],   \
26           ',␣direction=', _directions[r['direction']], ')', sep='')
```

Commentaires sur le module Robot en Python

- Pas de définition explicite du type Robot
  - les données du robot sont stockées dans un type dict (dictionnaire), tableau associatif
  - syntaxe un peu lourde : `r['x']`
  - les notions d'objet et de classe seront de meilleures solutions
- Type énuméré Direction représenté par un entier.
- `_dx` et `_dy` sont des tuples indicés par la direction du robot
  - simplification du code de translater, évite des **`if`**.
- `_directions` permet d'obtenir le nom en clair de la direction
  - utilisé à la création
  - utilisé à l'affichage
- les éléments qui commencent pas un souligné (_) sont considérés comme locaux
  - un utilisateur du module ne devrait pas les utiliser

Utilisation du module Robot en Python

```
1  from robot_dict import *
2
3  r1 = robot(4, 10, 'Est')
4  print('r1 =', end=' '); afficher(r1)
5  r2 = robot(15, 7, 'Sud')
6  print('r2 =', end=' '); afficher(r2)
7  pivoter(r1)
8  print('r1 =', end=' '); afficher(r1)
9  avancer(r2)
10 print('r2 =', end=' '); afficher(r2)
```

Et le résultat :

```
r1 = Robot(x=4, y=10, direction=Est)
r2 = Robot(x=15, y=7, direction=Sud)
r1 = Robot(x=4, y=10, direction=Sud)
r2 = Robot(x=15, y=6, direction=Sud)
```

Module : Encapsulation des données (types) et traitements (SP)

- **Principe :** Un type (prédéfini ou utilisateur) n'a que peu d'intérêt s'il n'est pas équipé d'opérations !
- **Justifications :**
  - éviter que chaque utilisateur du type réinvente les opérations
  - favoriser la réutilisation
- **Module :** Construction syntaxique qui regroupe des types et les opérations associées
- **Intérêt des modules :**
  - structurer l'application à un niveau supplémentaire par rapport aux sous-programmes (conception globale du modèle en V) ;
  - factoriser/réutiliser le code (type et SP) entre applications ;
  - améliorer la maintenance : une évolution dans l'implantation d'un module n'a pas d'impact sur les autres modules d'une application ;
  - améliorer les temps de compilation : compilation séparée de chaque module.
- **Question :** Pourquoi séparer données et traitements ?
- **Approche objet :** regrouper données et SP dans une entité appelée classe

Version objet

1. Modéliser les robots :

Classe = Données + Traitements

RobotType1
x : Entier y : Entier direction : Direction
avancer pivoter
initialiser(**in** x, y : int, **in** d : Direction)

Notation UML :

Nom de la classe	
attributs	*(état)*
opérations	*(comportement)*
constructeurs	*(initialisation)*

2. Utiliser les robots

```
Variable
    r1, r2: RobotType1
Début
    r1.initialiser(4, 10, EST)
    r2.initialiser(15, 7, SUD)
    r1.pivoter
    r2.avancer
Fin
```

r1 : RobotType1
x = 4 y = 10 direction = ~~EST~~ SUD
avancer pivoter initialiser(**in** x, y : int, **in** d : Direction)

(objet r1)

r2 : RobotType1
x = 15 y = ~~7~~ 6 direction = SUD
avancer pivoter initialiser(**in** x, y : int, **in** d : Direction)

(objet r2)

Classes et objets

Classe : moule pour créer des données appelées objets

- spécifie l'état et le comportement des objets
- équivalent à un type équipé des opérations pour en manipuler les données

Objet : donnée créée à partir d'une classe (instance d'une classe)

- une identité : distinguer deux objets différents
  - r1 et r2 sont deux objets différents, instances de la même classe Robot
  - en général, l'identité est l'adresse de l'objet en mémoire
- un état : informations propres à un objet, décrites par les attributs/champs
  - exemple : l'abscisse, l'ordonnée et la direction d'un robot
  - propre à chaque objet : r1 et r2 ont un état différent
- un comportement : décrit les évolutions possibles de l'état d'un objet sous la forme d'opérations (sous-programmes) qui lui sont appliquables
  - exemples : avancer, pivoter, etc.
  - une opération manipule l'état de l'objet

Remarque : On ne sait pas dans quel ordre les opérations seront appelées... mais il faut commencer par initialiser l'objet : c'est le rôle du constructeur (initialiser).

Classe Robot en Python (fichier robot.py)

```
1   class Robot:
2       """ Modèle très limité de robot qui ne sait qu'avancer d'une case et
3           case et pivoter à droite de 90°.  Il est repéré par son abscisse x
4           son ordonnée y et sa direction (un entier dans 0..3).
5       """
6       # des attributs de classe
7       _directions = ('Nord', 'Est', 'Sud', 'Ouest')   # direction en clair
8       _dx = (0, 1, 0, -1)     # incrément sur X en fonction de la direction
9       _dy = (1, 0, -1, 0)     # incrément sur Y en fonction de la direction
10
11      def __init__(self, x, y, direction):
12          """ Initialiser le robot self à partir de sa position (x, y) et sa direction. """
13          self.x = x
14          self.y = y
15          self.direction = Robot._directions.index(direction)
16
17      def avancer(self):
18          """ Avancer d'une case dans la direction courante.  """
19          self.x += Robot._dx[self.direction]
20          self.y += Robot._dy[self.direction]
21
22      def pivoter(self):
23          """ Pivoter ce robot de 90° vers la droite.  """
24          self.direction = (self.direction + 1) % 4
25
26      def afficher(self):
27          print('Robot(x=', self.x, ', y=', self.y, ', direction=', \
28                  Robot._directions[self.direction], ')', sep='');
```

Utilisation de la classe Robot en Python

```
1   from robot import Robot
2
3   r1 = Robot(4, 10, 'Est')
4   print('r1 =', end=' '); r1.afficher()
5   r2 = Robot(15, 7, 'Sud')
6   print('r2 =', end=' '); r2.afficher()
7   r1.pivoter()
8   print('r1 =', end=' '); r1.afficher()
9   r2.avancer()
10  print('r2 =', end=' '); r2.afficher()
11  Robot.pivoter(r2)                    # syntaxe "classique"
12  print('r2 =', end=' '); r2.afficher()
```

Et le résultat :

```
r1 = Robot(x=4, y=10, direction=Est)
r2 = Robot(x=15, y=7, direction=Sud)
r1 = Robot(x=4, y=10, direction=Sud)
r2 = Robot(x=15, y=6, direction=Sud)
r2 = Robot(x=15, y=6, direction=Ouest)
```

Exercices

Exercice 2 : Fraction
Proposer une modélisation UML d'une classe Fraction.

Exercice 3 : Date
Proposer une modélisation UML d'une classe Date.

Sommaire

- Classe
- Objet
- Attributs
- Méthodes
- Information privées
- Python : un langage hautement dynamique
- Comment définir une classe

Classe

Une classe définit :

- un **Unité d'encapsulation** : elle regroupe la déclaration des attributs et la définition des méthodes associées dans une même construction syntaxique

```
class NomDeLaClasse:
    # Définition de ses caractéristiques
```

  - *attributs* = stockage d'information (état de l'objet).
  - *méthodes* = unités de calcul (sous-programmes : fonctions ou procédures).

  C'est aussi un espace de noms :
  - deux classes différentes peuvent avoir des membres de même nom
- un **TYPE** qui permet de :
  - créer des objets (la classe est un moule, une matrice) ;

Les méthodes et attributs définis sur la classe comme « unité d'encapsulation » pourront être appliqués à ses objets.

Objet

- Objet : donnée en mémoire qui est le représentant d'une classe.
  - l'objet est dit instance de la classe
- Un objet est caractérisé par :
  - un état : la valeur des attributs (x, y, etc.) ;
  - un comportement : les méthodes qui peuvent lui être appliquées (avaner, etc.) ;
  - une identité : identifie de manière unique un objet (p.ex. son adresse en mémoire).
- Un objet est créé à partir d'une classe (en précisant les paramètres effectifs de son constructeur, sauf `self`) :

```
Robot(4, 10, 'Est')                  # création d'un objet Robot
Robot(x=15, y=7, direction='Sud')    # c'est comme pour les sous-programmes
```

  - retourne l'identité de l'objet créé
- En Python, les objets sont soumis à la sémantique de la référence :
  - la mémoire qu'ils occupent est allouée dans le tas (pas dans la pile d'exécution)
  - seule leur identité permet de les désigner
- Les identités des objets sont conservées dans des variables :

```
r1 = Robot(4, 10, 'Est')
r2 = Robot(x=15, y=7, direction='Sud')
```

Attributs

On distingue :

- les attributs d'instance : spécifique d'un objet (x, y, direction)
- les attributs de classe : attaché à la classe et non aux objets (_dx, _dy, etc.)

```
1  class A:
2      ac = 10 # Attribut de classe
3      def __init__(self, v):
4          self.ai = v # ai attribut d'instance
5  a = A(5)    #------------------------------
6  assert a.ai == 5
7  assert A.ac == 10
8  assert a.ac == 10   # on a accès à un attribut de classe depuis un objet
9  # A.ai  # AttributeError: type object 'A' has no attribute 'ai'
10 b = A(7)    #------------------------------
11 assert b.ai ==  7
12 assert b.ac == 10
13 b.ai = 11   #------------------------------
14 assert b.ai ==  11  # normal !
15 assert a.ai == 5    # ai est bien un attribut d'instance
16 A.ac = 20   #------------------------------
17 assert A.ac == 20   # logique !
18 assert b.ac == 20   # l'attribut de classe est bien partagé par les instances
19 b.ac = 30   #-------------- Est-ce qu'on modifie l'attribut de classe ac ?
20 assert b.ac == 30   # peut-être
21 assert A.ac == 20   # mais non ! b.ac = 30 définit un nouvel attribut d'instance
22
23 assert A == type(a) and A == a.__class__    # obtenir la classe d'un objet
```

Méthodes

- Une méthode d'instance est un sous-programme qui exploite l'état d'un objet (en accès et/ou en modification).
  - Le premier paramètre désigne nécessairement l'objet. Par convention, on l'appelle self.
- Une méthode de classe est une méthode qui travaille sur la classe (et non l'objet).
  - Elle est décorée `@classmethod`.
  - Son premier paramètre est nommé `cls` par convention.

```
@classmethod
def changer_langue(cls, langue):
    if langue.lower == 'fr':
        cls._directions = ('Nord', 'Est', 'Sud', 'Ouest')
    else:
        cls._directions = ('North', 'East', 'South', 'West')

Robot.changer_langue('en')  # Utilisation
```

- Une méthode statique est une méthode définie dans l'espace de nom de la classe mais est indépendante de cette classe.
  - Elle est décorée `@staticmethod`

```
class Date:
    ...
    @staticmethod
    def est_bissextile(annee):
        return annee % 4 == 0 and (annee % 100 != 0 or annee % 400 == 0)
```

Quelques méthodes particulières

- `__init__` : le constructeur : méthode d'initialisation nécessairement appelée quand on crée un objet.
  - C'est le constructeur qui devrait définir les attributs d'instance de la classe.
- `__str__` : utilisée par `str()` pour obtenir la représentation sous forme d'une chaîne de caractères de l'objet.
  - Remplace avantageusement la méthode afficher !
- `__eq__` : définir l'égalité logique (==) entre objet
  (par opposition à l'égalité physique, comparaison des identités, id(o1) == id(o2))
- ...

Information privées (locales)

- Élément essentiel pour l'évolution d'une application :
  - Tout ce qui est caché peut encore être changé.
  - Tout ce qui est visible peut avoir été utilisé. Le changer peut casser du code existant.
- Pas d'information privées en Python mais une convention :
  - ce qui commence par un souligné « _ » ne devrait pas être utilisé.
  - Traitement spécifique pour un élément qui commence par deux soulignés (nom préfixé par celui de la classe). Simule une information privée.
- Propriété sur les objets :
  - avoir une syntaxe identique à la manipulation d'un attribut (en accès ou modification) mais par l'intermédiaire de méthodes et donc avec contrôle !
  - Exemple : contrôler l'« attribut » mois d'une Date (compris entre 1 et 12).

Exemple de propriété : le mois d'une Date

```
1  # -*- coding: iso-8859-1 -*-
2  class Date:
3      def __init__(self, jour, mois, annee):
4          self.__jour = jour
5          self.__mois = mois
6          self.__annee = annee
7      @property       # accès en lecture à mois, comme si c'était un attribut
8      def mois(self):
9          return self.__mois
10     @mois.setter    # accès en écriture à mois, comme si c'était un attribut
11     def mois(self, mois):
12         if mois < 1 or mois > 12:
13             raise ValueError
14         self.__mois = mois
15
16 if __name__ == "__main__":
17     d1 = Date(25, 4, 2013)
18     assert d1.mois == 4
19     # d1.__mois # AttributeError: 'Date' object has no attribute '__mois'
20     assert d1._Date__mois == 4  # à ne pas utiliser !
21     d1.mois = 12
22     assert d1.mois == 12
23     d1.mois = 13    # ValueError
```

Question : Est-ce que les attributs choisis pour la date sont pertinents ?

Python : un langage hautement dynamique

En Python, on peut ajouter ou supprimer des attributs (et des méthodes) sur un objet, une classe, etc.

```
1  from robot import Robot
2
3  r1 = Robot(4, 10, 'Est')
4  print('r1 =', end=' '); r1.afficher()
5  r2 = Robot(15, 7, 'Sud')
6  print('r2 =', end=' '); r2.afficher()
7
8  r1.nom = "D2R2"          # on a ajouté un nom à r1 mais r2 n'a pas de nom
9
10 def tourner_gauche(robot):
11         robot.direction = (robot.direction + 3) % 4
12
13 Robot.pivoter_gauche = tourner_gauche
14 r2.pivoter_gauche()              # direction devient 'Est'
15 print('r2 =', end=' '); r2.afficher()
16
17 del r1.x                        # suppression d'un attribut
18 r1.avancer()             # AttributeError: 'Robot' object has no attribute 'x'
```

À éviter !

Comment définir une classe

Principales étapes lors du développement d'une classe :

1. Spécifier la classe du point de vue de ses utilisateurs (QUOI)
   - Spécifier les méthodes
   - Spécifier les constructeurs
   - Définir des programmes de test

   $\Longrightarrow$ On peut écrire la classe (en mettant un **pass**). Elle contient la documentation et les tests.

   $\Longrightarrow$ On peut l'exécuter... et bien sûr, les tests échouent.
2. Choisir les attributs (COMMENT)
   - Les attributs doivent permettre d'écrire le code des méthodes identifiées.
3. Implanter le constructeur et les méthodes (COMMENT)
   - De nouvelles méthodes peuvent être identifiées. Elles seront locales et donc avec un identifiant commençant par un souligné.
4. Tester
   - Les tests peuvent être joués au fur et à mesure de l'implantation des méthodes.

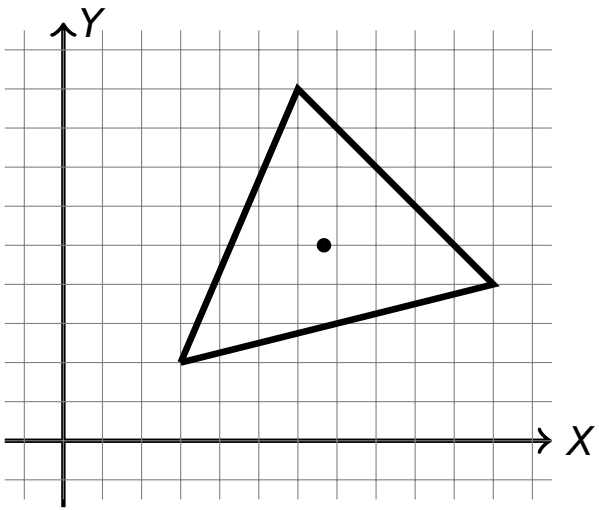
Sommaire

- Relations d'utilisation
- Héritage
- Classes abstraites et interfaces

Préambule

On souhaite faire un éditeur qui permet de dessiner des points, des segments, des polygones, des cercles, etc.

Exemple de schéma composé de 3 segments et un point (le barycentre)



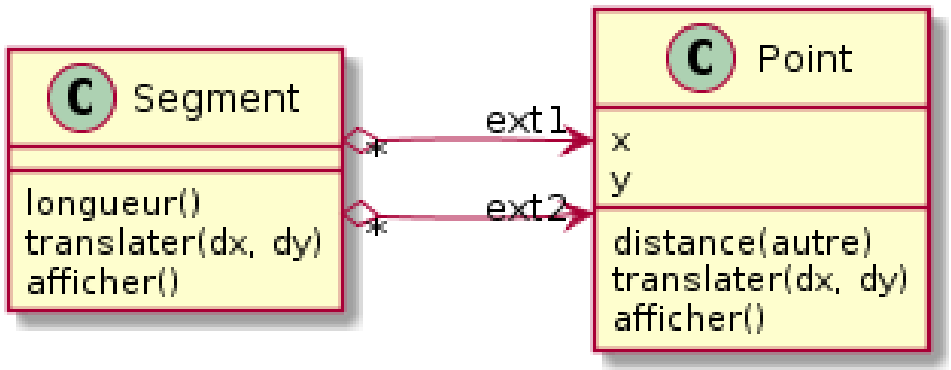
Documentation omise pour gain de place !

On considère la classe Point suivante.

```
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 translater(self, dx, dy):
11          self.x += dx
12          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

Principe : Une classe utilise une autre classe (en général, ses méthodes).



Exemple : La classe Segment utilise la classe Point. Un segment est caractérisé par ses deux points extémités :

- le *translater*, c'est translater les deux points extrémités
- sa *longueur* est la distance entre ses extrémités
- l'*afficher*, c'est afficher les deux points extrémités

Remarque : UML définit plusieurs niveaux de relation d'utilisation (dépendance, association, agrégation et composition) qui caractérisent le couplage entre les classes (de faible à fort).

La classe Segment

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

Le schéma donné en exemple

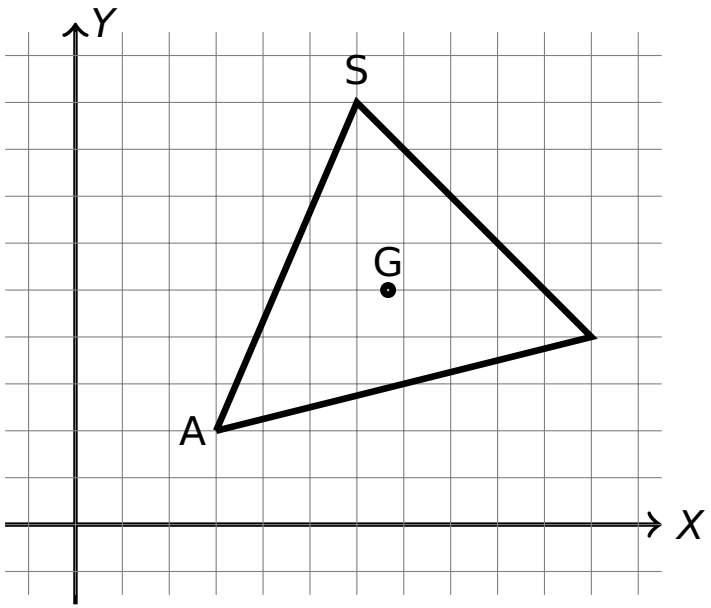
```
1  def exemple():
2      # créer les points sommets du triangle
3      p1 = Point(3, 2)
4      p2 = Point(6, 9)
5      p3 = Point(11, 4)
6
7      # créer les trois segments
8      s12 = Segment(p1, p2)
9      s23 = Segment(p2, p3)
10     s31 = Segment(p3, p1)
11
12     # créer le barycentre
13     sx = (p1.x + p2.x + p3.x) / 3.0
14     sy = (p1.y + p2.y + p3.y) / 3.0
15     barycentre = Point(sx, sy)
16
17     # construire le schéma
18     schema = [s12, s23, s31, barycentre];
19
20     # afficher le schéma
21     for elt in schema:
22         print(elt)
```

Le résultat obtenu :

```
[(3.0 ; 2.0) - (6.0 ; 9.0)]
[(6.0 ; 9.0) - (11.0 ; 4.0)]
[(11.0 ; 4.0) - (3.0 ; 2.0)]
(6.666666666666667 ; 5.0)
```

Évolution de l'application

Objectif : On souhaite pouvoir nommer certains points.



Comment faire ?

Idée : faire une nouvelle classe PointNommé, un point avec un nom.

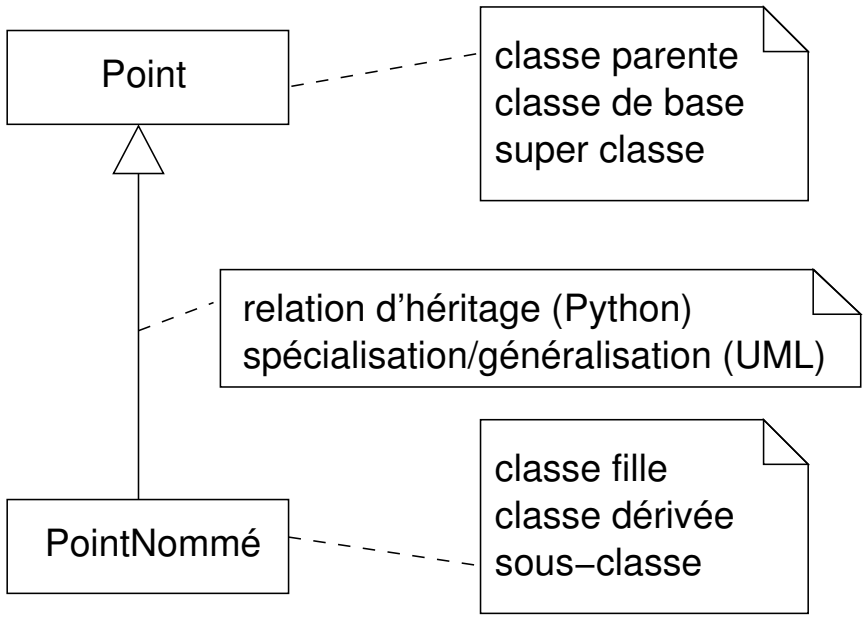
Héritage

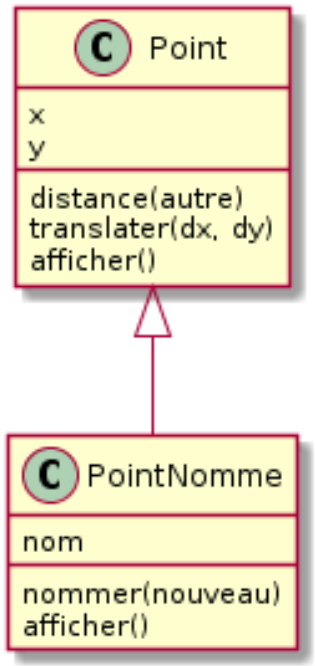
- **Principe :** définir une nouvelle classe par spécialisation d'une (ou plusieurs) classes existantes.
- **Exemple :** Définir une classe PointNommé sachant que la classe Point existe.
  La classe PointNommé :
  - hérite (récupère) tous les éléments de la classe Point
  - ajoute un nom et les opérations pour manipuler le nom
  - redéfinit la méthode afficher (pour afficher le nom et les coordonnées du point)
  - Point est la super-classe, PointNommé la sous-classe
- Souvent associé au sous-typage :

  Partout où on attend un Point, on peut mettre un PointNommé.
  - Attention, ceci dépend des langages !
  - A priori, pas de sens en Python car on peut toujours mettre n'importe quel objet dans n'importe quelle variable... Mais on aura certainement des erreurs à l'exécution !
- **Redéfinition :** Donner une nouvelle implantation à une méthode déjà présente dans une super-classe (*override*, en anglais).
  - Certains auteurs parlent de surcharge (*overload*).
- **Polymorphisme :** plusieurs forme pour la même méthode.
  - plusieurs versions de la méthode afficher dans PointNommé, la sienne et celle de Point

Notation et vocabulaire

Notation UML





Notation en Python

```
class PointNommé(Point):          # PointNommé hérite de Point
    ...
```

Vocabulaire : On parle d'ancêtres et de descendants (transitivité de la relation d'héritage)

La classe PointNommé

```
1   class PointNomme(Point):    # La classe PointNommé hérite de Point
2       def __init__(self, nom, x=0, y=0):
3           Point.__init__(self, x, y)  # initialiser la partie Point du PointNommé
4           self.nom = nom              # un nouvel attribut
5
6       def __str__(self):              # redéfinition
7           return "%s:%s" % (self.nom, Point.__str__(self))
8                                       # utilisation de la version de __str__ dans Point
9
10      def nommer(nouveau_nom):        # une nouvelle méthode
11          self.nom = nouveau_nom
```

Remarques :

- Il y a bien redéfinition de la méthode `__str__` de Point dans PointNommé.
- Les méthodes de Point sont héritées par PointNommé (ex : translater, distance)

Le nouveau schéma avec les point nommés

```
1  def exemple():
2      # créer les points sommets du triangle
3      p1 = PointNomme("A", 3, 2)
4      p2 = PointNomme("S", 6, 9)
5      p3 = Point(11, 4)
6
7      # créer les trois segments
8      s12 = Segment(p1, p2)
9      s23 = Segment(p2, p3)
10     s31 = Segment(p3, p1)
11
12     # créer le barycentre
13     sx = (p1.x + p2.x + p3.x) / 3.0
14     sy = (p1.y + p2.y + p3.y) / 3.0
15     barycentre = PointNomme("G", sx, sy)
16
17     # construire le schéma
18     schema = [s12, s23, s31, barycentre];
19
20     # afficher le schéma
21     for elt in schema:
22         print(elt)
```

Remarques :

- création de PointNommé au lieu de Point
- aucune modification sur la classe Point
- aucune modification sur la classe Segment
- ... mais on a ajouté PointNommé

Le résultat obtenu :

```
[A:(3.0 ; 2.0) - S:(6.0 ; 9.0)]
[S:(6.0 ; 9.0) - (11.0 ; 4.0)]
[(11.0 ; 4.0) - A:(3.0 ; 2.0)]
G:(6.666666666666667 ; 5.0)
```

Aspect méthododologique

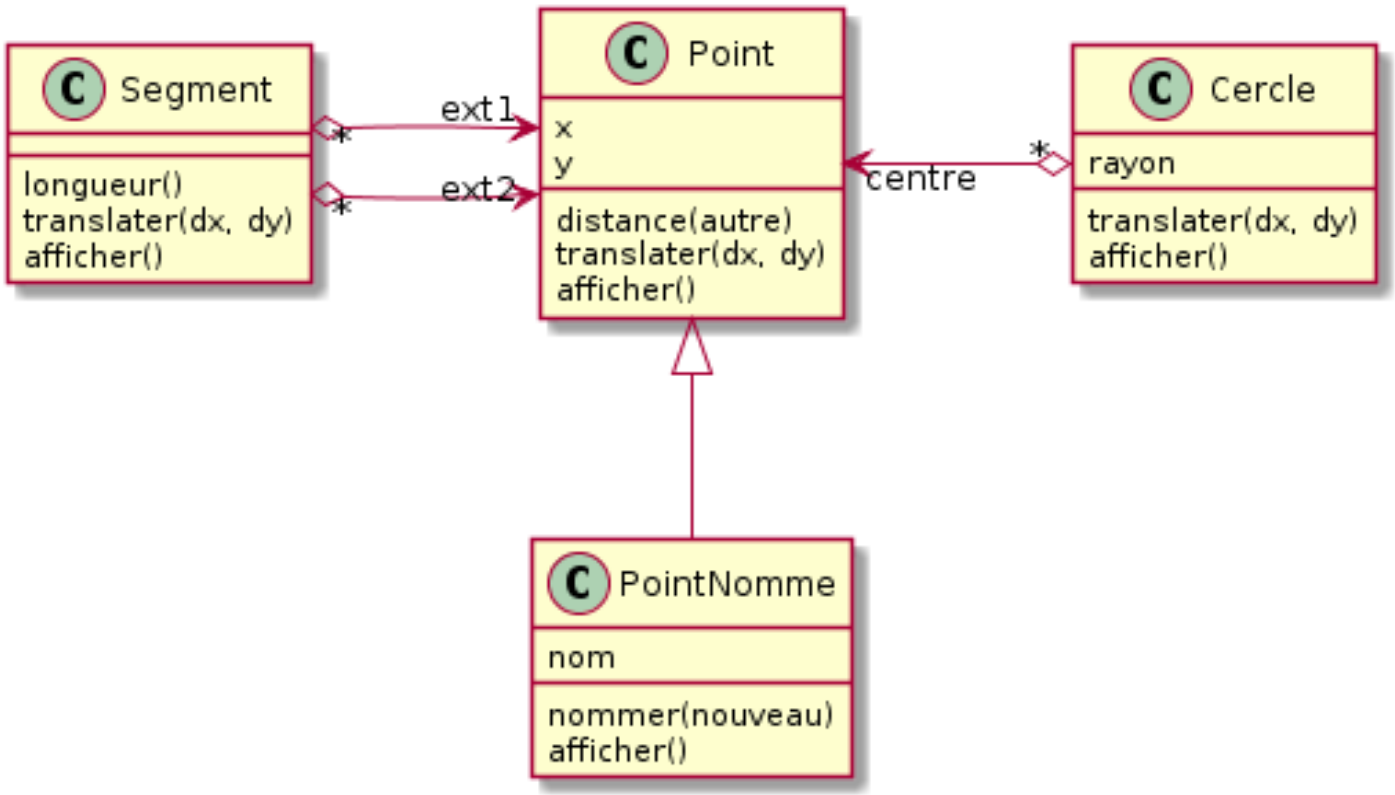
Comment définir une classe par spécialisation ?

1. Est-ce qu'il y a des méthodes de la super-classe à adapter ?
   - redéfinir les méthodes à adapter !
   - attention, il y a des règles sur la redéfinition !
2. Enrichir la classe des attributs et méthodes supplémentaires.
3. Tester la classe
   - Comme toute classe !

Comment ajouter un nouveau type de point, point pondéré par exemple ?

1. Choisir de faire une spécialisation de la classe Point.
   - Une nouvelle classe à côté des autres
   - On ne risque pas de casser le système
2. Écrire la classe
   - Redéfinir la méthode afficher (afficher)
   - Ajouter masse et set_masse
   - Tester (y compris en tant que Point)
3. Intégrer dans le système en proposant de créer des PointPondérés

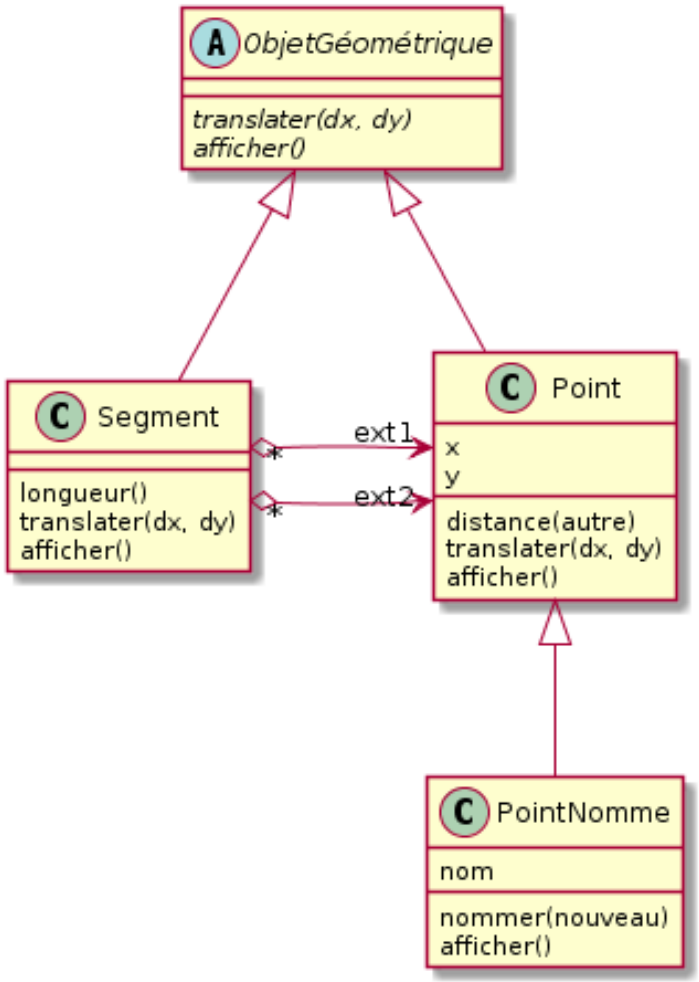
Diagramme de classe... avec une nouvelle classe



Comment imposer que sur chaque nouvel objet géométrique, soient définies les méthodes translater et afficher ?

La classe ObjetGéométrique

Solution : Définir une nouvelle classe qui généralise Point, Segment, Cercle, etc.



Première idée :

```
1  class ObjetGeometrique:
2      def translater(self, dx, dy):
3          pass
4      def __str__(self):
5          pass
```

Meilleure idée :

```
1  class ObjetGeometrique:
2      def translater(self, dx, dy):
3          raise NotImplementedError
4      def __str__(self):
5          raise NotImplementedError
```

Question : On oublie de redéfinir la méthode translater dans Cercle. Comment le détecter ?

Classes abstraites et interfaces

Problème : La méthode translater étant définie dans ObjetGéométrique, pour détecter qu'on a oublié de la définir dans Cercle, il faut écrire un programme qui :

- crée un cercle
- translate le cercle

et constater que ça n'a pas eu l'effet escompté !

La bonne solution : dire que les méthodes sont abstraites et donc la classe abstraite ! Utilisation du module abc (Abstract Base Classes), PEP 3119.

```
1  import abc
2  class ObjetGeometrique(metaclass=abc.ABCMeta):
3      @abc.abstractmethod
4      def translater(self, dx, dy):
5          pass
6      @abc.abstractmethod
7      def __str__(self):
8          pass
```

L'erreur sera détectée dès la création de l'objet ! Beaucoup plus sûr !

Interface : classe complètement abstraite $\Rightarrow$ **spécifier un comportement**, définir un contrat.

Sommaire

- Unified Modeling Language (UML)

Unified Modeling Language (UML)

Principales caractéristiques d'UML [1, 2]

- notation graphique pour décrire les éléments d'un développement (objet)
- normalisée par l'OMG [3] (Object Management Group)
- version 1.0 en janvier 1997. Version actuelle : 2.4.1, mai 2012.
- *s'abstraire du langage de programmation et des détails d'implantation*

Utilisations possibles d'UML [2]

- **esquisse (*sketch*) :** communiquer avec les autres sur certains aspects
  - simplification du modèle : seuls les aspects importants sont présentés
  - échanger des idées, évaluer des alternatives (avant de coder), expliquer (après)
  - n'a pas pour but d'être complet
- **plan (*blueprint*) :** le modèle sert de base pour le programmeur
  - le modèle a pour but d'être complet
  - les choix de conception sont explicités
  - seuls les détails manquent (codage)
- **langage de programmation :** *le modèle est le code !*
  - pousser à l'extrême l'approche « plan »
    $\Longrightarrow$ mettre tous les détails dans le modèle
  - engendrer automatiquement le programme à partir du modèle.

Références

Pierre-Alain Muller and Nathalie Gaertner.

Modélisation objet avec UML.

Eyrolles, 2è edition, 2003.

Martin Fowler.

UML 2.0.

CampusPress Référence, 2004.

OMG.

UML Resource Page.

http ://www.omg.org/uml/.