Cours Python

—

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 :

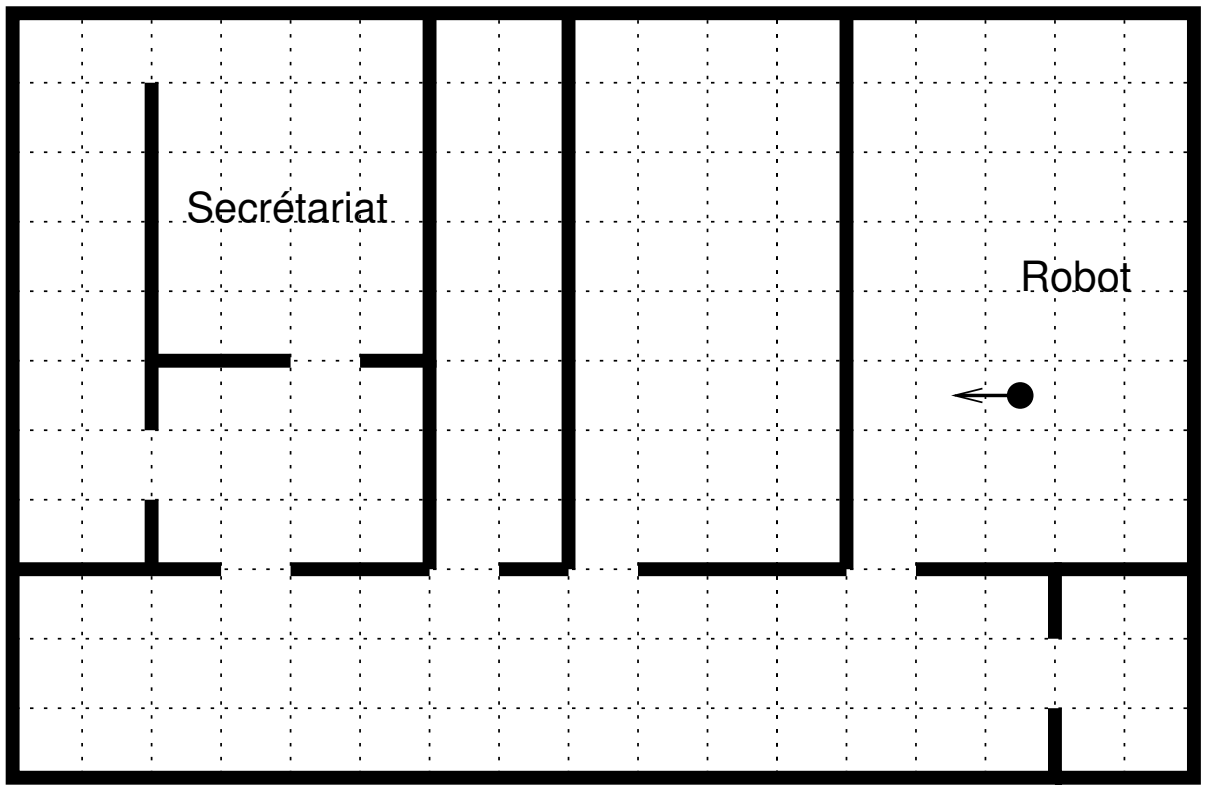
- Exemple introductif : passage de l'impératif à une approche objet
- Encapsulation : classe, objets, attributs, méthodes, etc.
- Relations entre classes
  - Relations d'utilisation
  - Relation d'héritage
- Quelques éléments spécifiques à Python
  - Introspection
  - Méthodes spéciales et redéfinitions des opérateurs
  - Métaclasses
- 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 { '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, *, prefix=''):
24     """ Afficher le robot r. """
25     print('{}Robot(x={}, y={}, direction={})'   \
26             .format(prefix, r['x'], r['y'], _directions[r['direction']]))
```

Commentaires sur le module Robot en Python

- Pas de définition explicite du type Robot
  - les données du robot sont stockées dans un dictionnaire (type dict), tableau associatif
  - syntaxe un peu lourde : `r['x']`
  - les notions d'objet et de classe seront de meilleures solutions
  - *Remarque :* on pourrait utiliser le type `namedtuple` de `collections` (mais immuable).
- Type énuméré Direction représenté par un entier.
  - *Remarque :* Python offre le module `enum` pour définir des types énumérés.
- `_dx` et `_dy` sont des tuples indicés par la direction du robot
  - donnent l'entier à ajouter à x ou y pour déplacer le robot en fonction de sa direction
  - 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  afficher(r1, prefix='r1 = ')
5  r2 = robot(15, 7, 'sud')
6  afficher(r2, prefix='r2 = ')
7  pivoter(r1)
8  afficher(r1, prefix='r1 = ')
9  avancer(r2)
10 afficher(r2, prefix='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 ne devrait pas avoir d'impact sur les autres modules d'une application ;
  - faciliter le test : le modules sont testés individuellement.
- **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).

Première version de la classe Robot en Python (robot.py)

```
class Robot:
    """ Robot qui sait avancer d'une case et pivoter à droite de 90°.
        Il est repéré par son abscisse x, son ordonnée y et sa direction.
    """
    def __init__(self, x, y, direction):
        """ Initialiser le robot self à partir de sa position (x, y) et sa direction. """
        self.x = x
        self.y = y
        self.direction = 'nord est sud ouest'.split().index(direction)

    def avancer(self):
        """ Avancer d'une case dans la direction.  """
        # mettre à jour self.x
        if self.direction == 1: # est
            self.x += 1
        elif self.direction == 3: # ouest
            self.x -= 1
        # mettre à jour self.y
        if self.direction == 0: # nord
            self.y += 1
        elif self.direction == 2: # sud
            self.y -= 1
```

Première version de la classe Robot en Python (robot.py) (2)

```
def pivoter(self):
    """ Pivoter ce robot de 90° vers la droite.  """
    self.direction = (self.direction + 1) % 4

def afficher(self, *, prefix=''):
    print(prefix + '({},␣{})>{}'.format(self.x, self.y, \
            'nord␣est␣sud␣ouest'.split()[self.direction]))
```

1. Expliquer les différents éléments qui apparaissent sur cette classe.
2. Expliquer comment a été modélisé le robot, en particulier que représente direction.
3. Indiquer les critiques que l'on peut faire sur ce code et comment l'améliorer.

Utilisation de la classe Robot en Python (fichier exemple_robot.py)

```
1  from robot import Robot
2
3  r1 = Robot(4, 10, 'est')
4  r1.afficher(prefix='r1 = ')
5  r2 = Robot(15, 7, 'sud')
6  r2.afficher(prefix='r2 = ')
7  r1.pivoter()
8  r1.afficher(prefix='r1 = ')
9  r2.avancer()
10 r2.afficher(prefix='r2 = ')
11 Robot.pivoter(r2)           # syntaxe "classique"
12 Robot.afficher(r2, prefix='r2 = ')
13 print('Robot.pivoter :', Robot.pivoter)
14 print('r2.pivoter :', r2.pivoter)
```

Et le résultat :

```
r1 = (4, 10)>est
r2 = (15, 7)>sud
r1 = (4, 10)>sud
r2 = (15, 6)>sud
r2 = (15, 6)>ouest
Robot.pivoter : <function Robot.pivoter at 0x7f437a4daa60>
r2.pivoter : <bound method Robot.pivoter of <robot.Robot object at 0x7f437a4fdd68>>
```

Classe Robot en Python (fichier robot.py)

```
1  class Robot:
2      """ Robot qui sait avancer d'une case et pivoter à droite de 90°.
3          Il est repéré par son abscisse x, son ordonnée y et sa direction.
4      """
5      # des attributs de classe
6      _directions = ('nord', 'est', 'sud', 'ouest')   # direction en clair
7      _dx = (0, 1, 0, -1)     # incrément sur X en fonction de la direction
8      _dy = (1, 0, -1, 0)     # incrément sur Y en fonction de la direction
9
10     def __init__(self, x, y, direction):
11         """ Initialiser le robot self à partir de sa position (x, y) et sa direction. """
12         self.x = x
13         self.y = y
14         self.direction = Robot._directions.index(direction)
15
16     def avancer(self):
17         """ Avancer d'une case dans la direction.  """
18         self.x += Robot._dx[self.direction]
19         self.y += Robot._dy[self.direction]
20
21     def pivoter(self):
22         """ Pivoter ce robot de 90° vers la droite.  """
23         self.direction = (self.direction + 1) % 4
24
25     def afficher(self, *, prefix=''):
26         print(prefix, self, sep='')
27
28     def __str__(self):
29         return '({},␣{})>{}'.format(self.x, self.y, Robot._directions[self.direction])
```

Passage d'UML à Python

Le diagramme UML donne le squelette de la classe Python

RobotType1
+ x : Entier + y : Entier + direction : Direction - dx : String [4] - dy : String [4] - directions : String [4]
+ avancer + pivoter
+ initialiser(**in** x, y : int, **in** d : Direction)

— : privé (local à la classe)
+ : public (accessible de tous)
souligné : niveau classe

```
class Robot:
    """ ... """

    _directions = ...   # les attributs de classe
    _dx = ...
    _dy = ...

    def __init__(self, x, y, direction):    # constructeur
        """ ... """
        self.x = ...    # définir les attributs d'instance
        self.y = ...
        self.direction = ...

    def avancer(self):
        """ ... """
        pass

    def pivoter(self):
        """ ... """
        pass

    def __str__(self):  # pour convertir en chaîne
        return ...
```

Exercices

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

Exercice 3 : Monnaie

1. Modéliser en UML une classe Monnaie. Une monnaie est caractérisée par une valeur (int) et une devise (str) et possède les opérations ajouter et retrancher pour respectivement ajouter et retrancher à cette monnaie une autre monnaie. Les deux monnaies doivent avoir même devise sinon l'exception TypeError est levée.
2. Écrire deux tests. Le premier ajoute à la monnaie m1 de 5 euros la monnaie m2 de 7 euros et vérifie que m1 vaut bien 12 euros. Le deuxième retranche à m1 m2 et vérifie que la valeur de m1 est -2.
3. Écrire en Python la classe Monnaie.

Exercice 4 : Fraction
Proposer une modélisation UML d'une classe Fraction.

Sommaire

- Classe
- Objet
- Attributs
- Méthodes
- Python : un langage hautement dynamique
- Information privées
- 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 le premier, `self`) :

```
Robot(4, 10, 'Est')                  # création d'un objet Robot
Robot(direction='Sud', x=15, y=7)    # 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 (des noms) :

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

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, _directions, 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 is type(a) and A is 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 (méthodes spéciales)

- `__init__(self, ...)` : le constructeur : méthode d'**initialisation** nécessairement appelée quand on crée un objet. Voir aussi `__new__`, T. 23.
  - C'est le constructeur qui devrait définir les attributs d'instance de la classe.
- `__del__(self)` : le destructeur, appelé quand une instance est sur le point d'être détruite.
- `__repr__(self)` : chaîne représentant cet objet et qui devrait correspondre (quand c'est possible) à une expression Python valide pour créer l'objet.
- `__str__(self)` : utilisée par **`str()`** pour obtenir la représentation sous forme d'une chaîne de caractères *lisible* de l'objet. Si non définie, retourne le résultat de `__repr__`.
  - Remplace avantageusement la méthode afficher !
- `__new__(cls, ...)` : méthode implicitement statique qui crée l'objet (et appelle `__init__`).
- `__bool__(self)` : utilisée quand l'objet est considéré comme booléen et avec la fonction prédéfinie bool().
- ...

Voir `https://docs.python.org/3/reference/datamodel.html#special-method-names`

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 =', r1)
5  r2 = Robot(15, 7, 'sud')
6  print('r2 =', r2)
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
15 r2.pivoter_gauche()     # direction devient 'est'
16 print('r2 =', r2)
17
18 del r1.x                # suppression d'un attribut
19 r1.avancer()            # AttributeError: 'Robot' object has no attribute 'x'
```

À éviter !

__slots__

Motivation :

- diminuer la taille mémoire utilisée par des objets (le dictionnaire `__dict__` n'est pas conservé)

Conséquence :

- On ne peut pas ajouter de nouveaux attributs.
- Remarque : on peut en ajouter si on ajoute `__dict__` dans `__slots__`.

Exemple :

```
1  class A:
2      __slots__ = [ 'a', 'b' ]
3
4  a = A()
5  print(a.b)  # AttributeError: b
6  a.a = 1 # ok
7  a.b = 2 # ok
8  print('__slots__' in dir(a)) # True
9  print('__dict__' in dir(a))  # False
10 a.c = 3 # AttributeError: 'A' object has no attribute 'c'
```

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).
- Principe de Python :
  1. On commence par définir un attribut normal.
  2. Si on a besoin de le contrôler, on passer à une propriété : cette modification ne remet pas en cause le code client qui reste inchangé.

Exemple de propriété : le mois d'une Date

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

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

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.

Comptes bancaires

Exercice 5 : Compte simple

Nous nous intéressons à un compte simple caractérisé par un solde exprimé en euros, positif ou négatif, et son titulaire.[1] Il est possible de créditer ce compte ou de le débiter d'un certain montant.

5.1 Donner le diagramme de classe de la classe CompteSimple.

5.2 Écrire un programme de test de CompteSimple.

5.3 Écrire (puis tester) la classe CompteSimple.

Exercice 6 : Banque

En suivant la même démarche, définir une classe Banque qui gère des comptes. Elle offre les opérations suivantes :

- ouvrir un compte pour un client
- calculer le total de l'argent géré par la banque (la somme des soldes de tous les banques)
- prélever des frais sur l'ensemble des comptes

Exercice 7 : Compte Simple (suite)

Modifier la classe CompteSimple pour que le solde ne soit plus accessible en écriture. Il ne faut pas que cette modification ait un impact sur les programes qui utilisent CompteSimple, en particulier Banque (et les programmes de test).

1. Nous simplifions le problème en considérant que tout compte a un et un seul titulaire correspondant à une personne physique modélisée par une classe Personne.

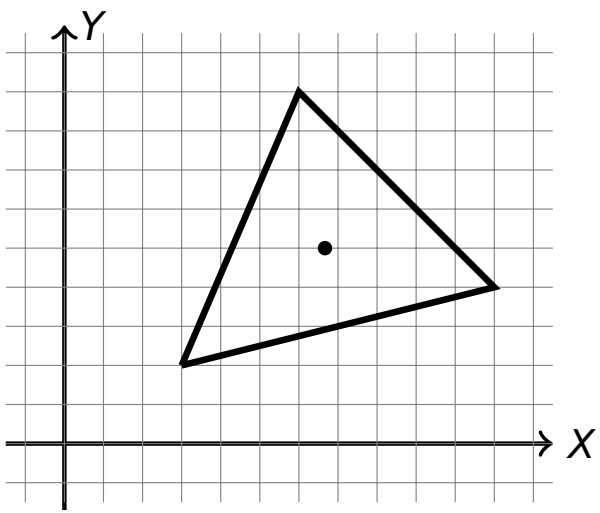
Sommaire

- 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)



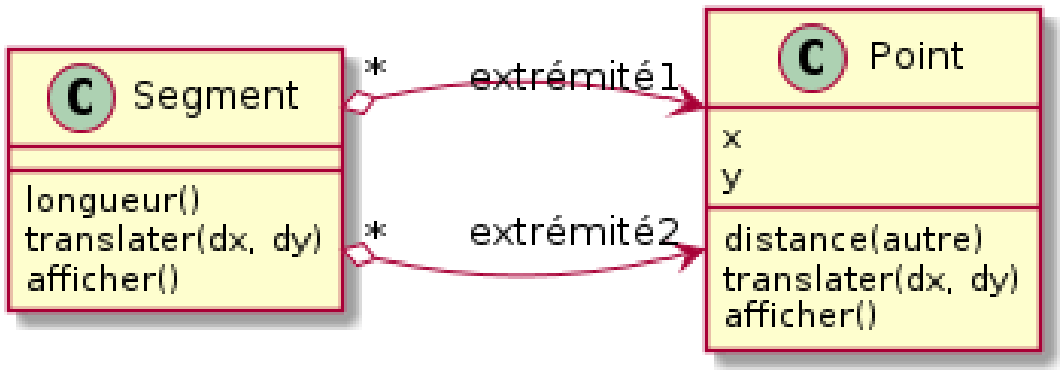
Documentation omise pour gain de place !

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 __repr__(self):
8           return f"Point({self.x}, {self.y})"
9
10      def __str__(self):
11          return f"({self.x} ; {self.y})"
12
13      def translater(self, dx, dy):
14          self.x += dx
15          self.y += dy
16
17      def distance(self, autre):
18          dx2 = (self.x - autre.x) ** 2
19          dy2 = (self.y - autre.y) ** 2
20          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 __repr__(self):
7           return f"Segment({repr(self.extremite1)}, {repr(self.extremite2)})"
8
9       def __str__(self):
10          return f"[{self.extremite1} - {self.extremite2}]"
11
12      def translater(self, dx, dy):
13          self.extremite1.translater(dx, dy)
14          self.extremite2.translater(dx, dy)
15
16      def longueur(self):
17          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)
```

Exercice

Exercice 8 : Définition d'un point nommé

Un point nommé est un point, caractérisé par une **abscisse** et une **ordonnée**, qui possède également un **nom**. Un point nommé peut être **translaté** en précisant un déplacement suivant à l'axe des X (abscisses) et un déplacement suivant l'axe des Y (ordonnées). On peut obtenir sa **distance** par rapport à un autre point. Il est possible de **modifier** son abscisse, son ordonnée ou son nom. Enfin, ses caractéristiques peuvent être **affichées**.

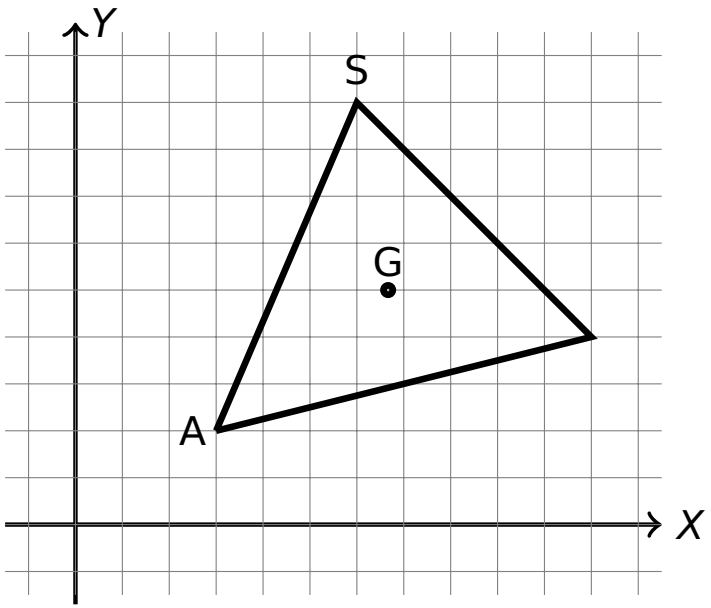
8.1 Modéliser en UML cette notion de point nommé.

8.2 Une classe Point a déjà été écrite. Que constatez-vous quand vous comparer le point nommé et la classe Point des transparents suivants ?

8.3 Comment écrire la classe PointNommé ?

Évolution de l'application

Objectif : On souhaite pouvoir nommer certains points.



Comment faire ?

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

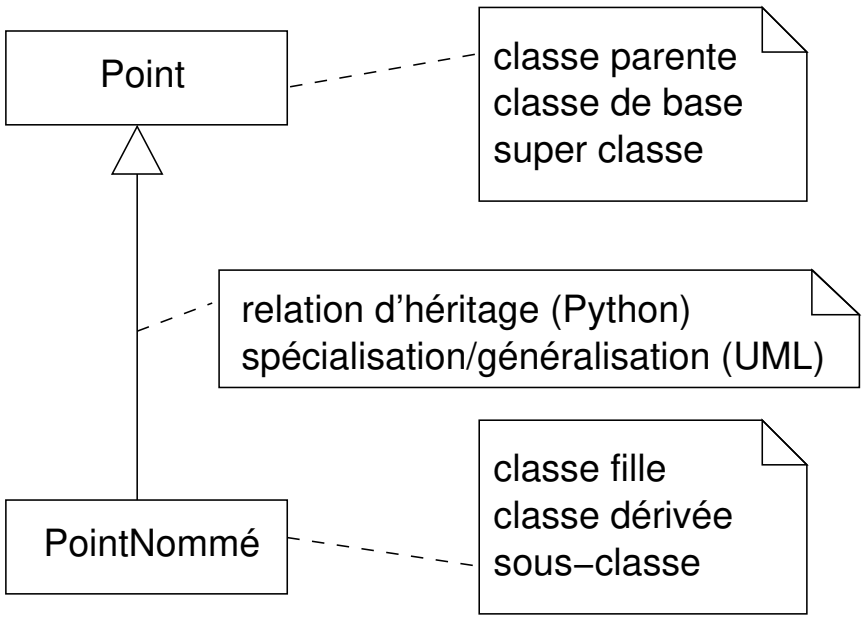
Héritage

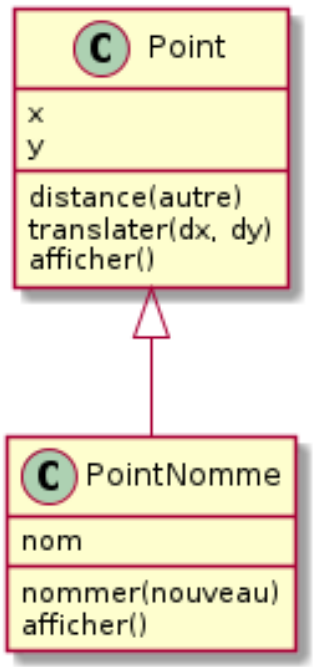
- **Principe :** définir une nouvelle classe par spécialisation d'une (ou plusieurs) classes existantes.
- **Exemple :** Définir une classe PointNommé sachant que la classe Point existe.
  La classe PointNommé :
  - hérite (récupère) tous les éléments de la classe Point
  - ajoute un nom et les opérations 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 !
  - En Python, c'est juste de la réutilisation même si c'est souvent associé au sous-typage. Python pratique le *duck typing* : le type est défini par l'ensemble des méthodes et attributs d'un objet.
- **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 formes 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          super().__init__(x, y)  # initialiser la partie Point du PointNommé
4          self.nom = nom              # un nouvel attribut
5
6      def __repr__(self):
7          return f"PointNomme({repr(self.nom)}, {self.x}, {self.y})"
8
9      def __str__(self):              # redéfinition
10         return f"{self.nom}:{super().__str__()}"
11                                     # utilisation de la version de __str__ dans Point
12
13     def nommer(self, nouveau_nom):      # une nouvelle méthode
14         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)
- **`super()`** est recommandé pour appeler la méthode des super classes. Voir `https://rhettinger.wordpress.com/2011/05/26/super-considered-super/`
  `super()` raccourci pour **`super`**`(PointNomme, self)` ici
- Ancienne solution : `Point.__init__(self, x, y)` et `Point.__str__(self)`

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

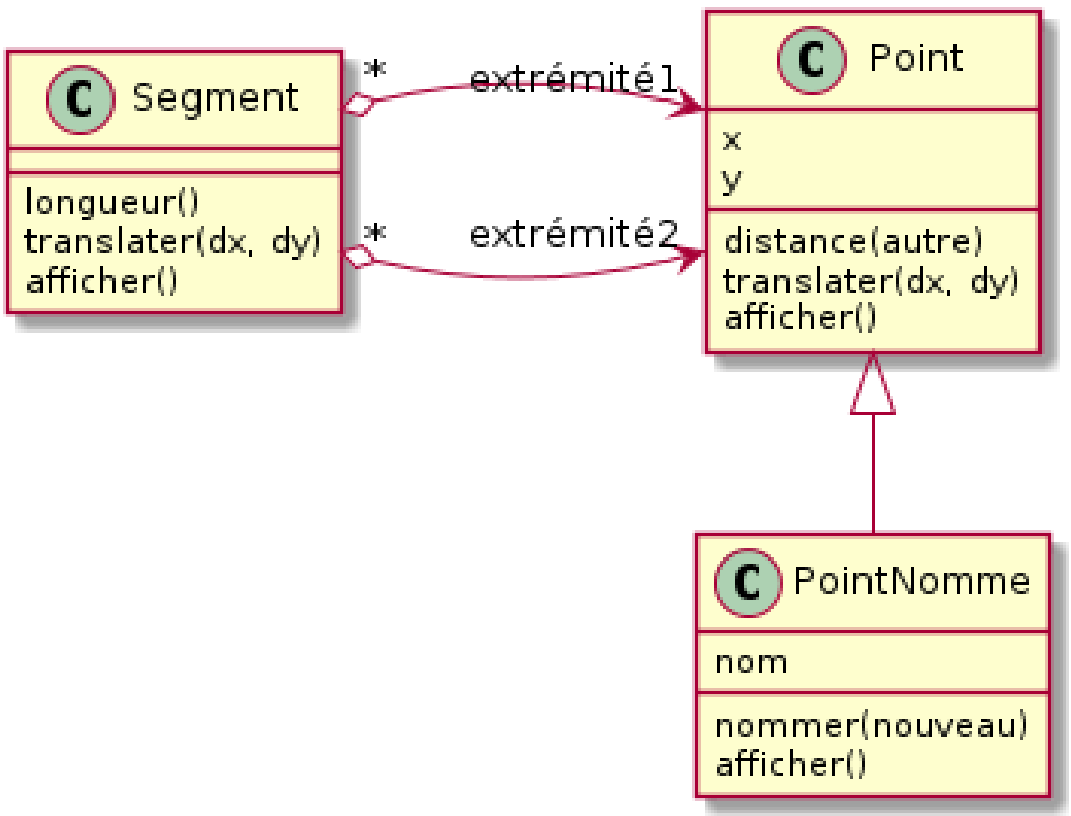
Le résultat obtenu :

```
[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)
```

Remarque :

```
assert isinstance(p1, Point)
assert isinstance(p1, PointNomme)
assert isinstance(p3, Point)
assert not isinstance(p3, PointNomme)
assert issubclass(PointNomme, Point)
```

Nouveau diagramme de classe



Autres aspects

La classe `object`

- C'est l'ancêtre commun à toutes les classes.
- Quand on ne précise aucune superclasse, la classe hérite implicitement de **`object`**

Héritage multiple

- Python permet l'héritage multiple.
- Il suffit de lister toutes les super classes séparées par des virgules

```
class S(A, B, C):        # la classe S hérite de A, B et C
    pass
```

- L'ordre des classes à une importance et influt sur le MRO (Method Resolution Order) : linéarisation des classes parentes pour permettre le call-next-method :
  - il préserve l'ordre de gauche à droite précisé dans chaque classe
  - une sous-classe apparaît avant ses superclasses
  - une même classe n'apparaît qu'une fois !
  - il est monotone : l'ajout de nouvelle sous-classes ne modifie par le MRO des superclasses.

Héritage multiple et MRO

Exercice 9 : Calcul MRO

Dans l'exemple suivant, donner le MRO (de la classe considéré jusqu'à object) des classes D et F.

```
class A: pass
class B: pass
class C(A, B): pass
class D(C, A): pass

print(D.mro())

class E(B, A): pass
class F(C, E): pass

print(F.mro())
```

Héritage multiple : intérêt du MRO

```
class Element:
    def f(self):
        print('Element.f')

class Good(Element):
    def f(self):
        print('Good.f')
        return super().f()

class Bad(Element):
    def f(self):
        print('Bad.f')
        return Element.f(self)

class Log(Element):
    def f(self):
        print('Do something: logging...')
        return super().f()

class WithGood(Good, Log):
    def f(self):
        print('WithGood.f')
        return super().f()

class WithBad(Bad, Log):
    def f(self):
        print('WithBad.f')
        return super().f()

print('\nWithGood().f() :') ; WithGood().f()
print('\nWithBad().f() :')  ; WithBad().f()
```

```
WithGood().f() :
WithGood.f
Good.f
Do something: logging...
Element.f

WithBad().f() :
WithBad.f
Bad.f
Element.f
```

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 : fonctionner au moins dans les mêmes cas et faire au moins ce qui est attendu
2. Enrichir la classe : définir de nouveaux attributs et méthodes.
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

Exercice 10 Écrire la classe PointPondéré.

Comptes bancaires... la suite

Exercice 11 : Comptes courants

La banque gère aussi des comptes courants. Chaque compte courant conserve l'historique des opérations qui le concernent (on se limite ici aux opérations de crédit et de débit). En plus des opérations déjà disponibles sur un compte simple, on peut éditer un relevé de compte qui fait apparaître toutes les opérations réalisées sur le compte et le solde final.

11.1 Compléter le diagramme UML pour faire apparaître la classe CompteCourant.

11.2 Écrire et tester la classe CompteCourant.

Exercice 12 : Compléter la banque

12.1 Ajouter une opération pour permettre d'ouvrir un compte courant.

12.2 Tester que les opérations existantes de la banque fonctionnent bien, que l'on ouvre des comptes simples ou des comptes courants.

12.3 Ajouter une opération pour éditer le relevé des comptes gérés par la banque.

Exercice 13 : Numéros de comptes

Pour les différencier, chaque compte possède un numéro unique. On suppose que les numéros sont des entiers et qu'ils sont attribués par ordre croissant en commençant à 10001. Les numéros de compte sont donc 10001, 10002, 10003, etc. Dans la suite, nous envisageons deux solutions pour attribuer les numéros de compte.

13.1 On souhaite que l'attribution du numéro de compte soit de la responsabilité des classes `CompteSimple` et `CompteCourant`. Indiquer les modifications à apporter à l'application.

13.2 On suppose maintenant que c'est la banque qui gère et attribue les numéros de compte. Indiquer les modifications à apporter à l'application.

Duck typing (typage canard)

Principe : Si un objet dispose des attributs et des méthodes que l'on souhaite utiliser, c'est qu'il a le bon type.

« Si je vois un oiseau qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard » (James Whitcomb Riley)
Exemple : (Inspiré de La programmation orientée objet en Python)

```
class StrangeFile:
    def read(self, size=0):
        return ''
    def write(self, s):
        print(s[::-1], end='')
    def close(self):
        pass

f = StrangeFile()
print('foo', file=f)
```

affiche `'oof'`.
Ça marche... car les méthodes attendues par **`print`** sont présentes.

Héritage ou délégation

En Python, on peut facilement écrire la classe PointNommée en utilisant la délégation (utilisation de la classe Point) plutôt que l'héritage.

```
1  class PointNommeDelegation:
2      def __init__(self, nom, x=0, y=0):
3          self.pt = Point(x, y)   # le Point du PointNommé
4          self.nom = nom          # le nom
5
6      def __repr__(self):
7          return f"PointNomme({repr(self.nom)}, {self.x}, {self.y})"
8
9      def __str__(self):              # adaptation
10         return f"{self.nom}:{self.pt}"
11
12     def nommer(self, nouveau_nom):      # une nouvelle méthode
13         self.nom = nouveau_nom
14
15     def __getattr__(self, name):
16         return getattr(self.pt, name)
```

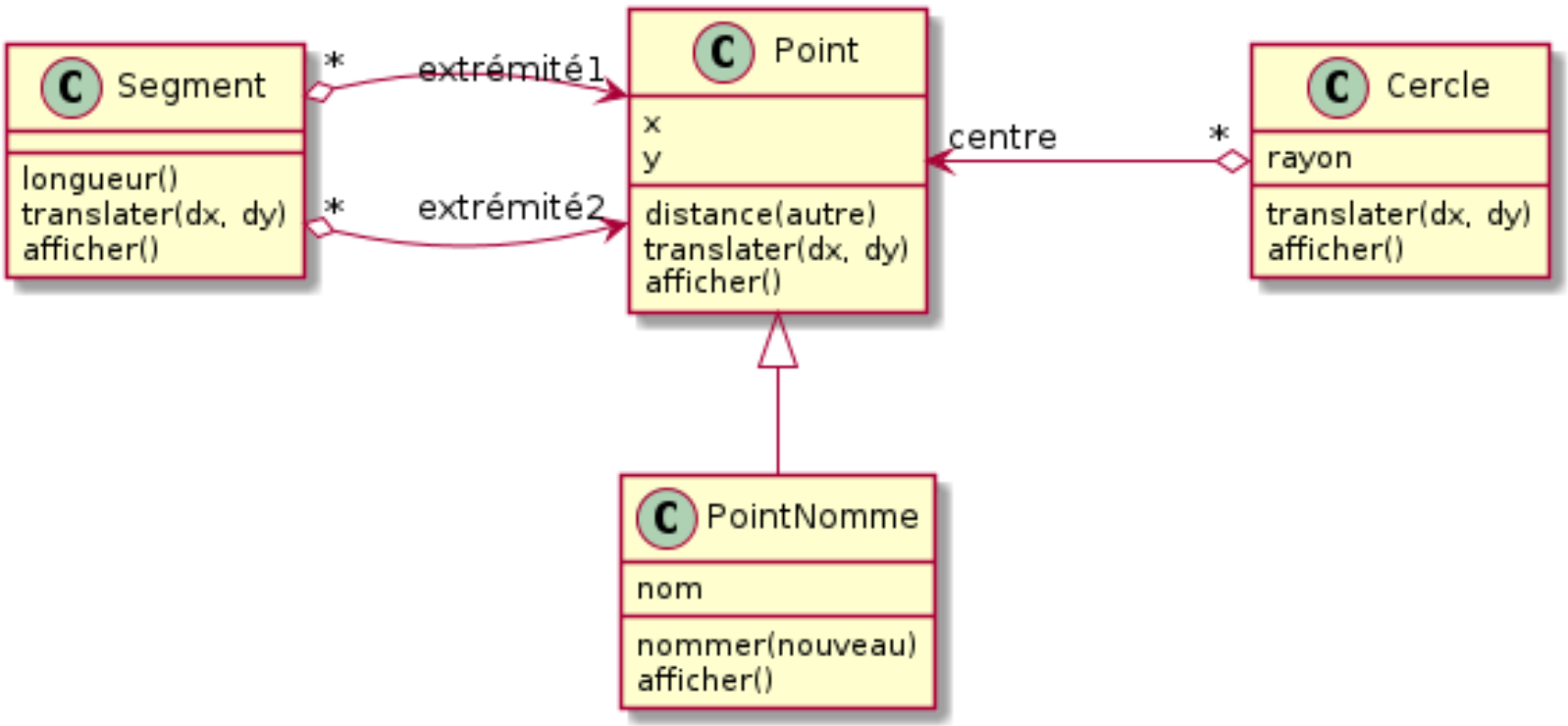
Explications : Lorsqu'un attribut (au sens Python : attribut ou méthode au sens objet) n'est pas trouvé sur l'objet, la méthode spéciale `__getattr__` est appelée. Ici elle cherche dans l'objet `pt`.

Héritage ou délégation (suite)

```
1  def test_PointNommeDelegation():
2      pn = PointNommeDelegation('A', 10, 20)
3      assert pn.nom == 'A'
4      assert pn.x == 10
5      assert pn.y == 20
6      assert str(pn) == 'A:(10.0 ; 20.0)'
7      assert Point.__str__(pn) == '(10.0 ; 20.0)'
8      assert Point.__repr__(pn) == 'Point(10.0, 20.0)'
9      assert isinstance(pn, PointNommeDelegation)
10     assert not isinstance(pn, Point)
```

Inconvénient : `isinstance(pn, Point)` est fausse
Si on se sert de cette propriété pour vérifier qu'un Segment est bien construit à partir de Point, on ne pourra pas fournir un objet construit à partir de PointNommeDelegation.

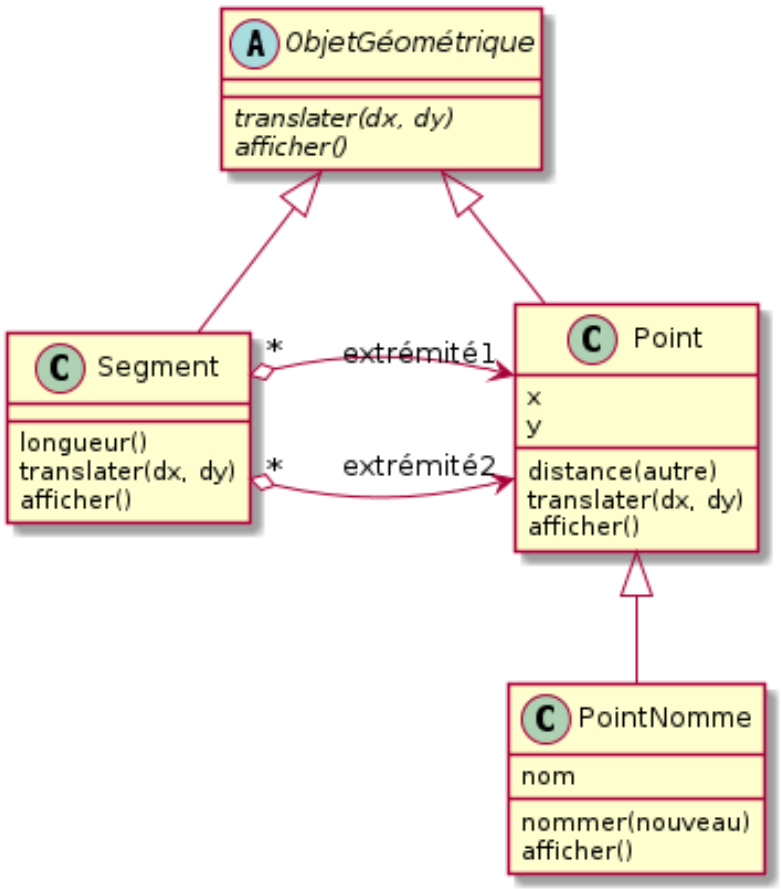
Diagramme de classe... avec une nouvelle classe



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 ⇒ **spécifier un comportement**, définir un contrat.

Sommaire

Introspection

Principe :

- Utiliser les noms des attributs sous forme de chaînes de caractères pour les manipuler.

Intérêt :

- Permet de faire des opérations sur ces noms.

Moyen :

- **`hasattr(object, name)`** : vrai si object possède un attribut du nom précisé, faux sinon.
- **`getattr(object, name[, default])`** : l'attribut nommé name dans object s'il existe sinon la valeur default si elle est fournie ou lève l'exception AttributeError.
- **`setattr(object, name, value)`** : associe value à l'attribut appelé name de object (si object l'autorise).

Exemple naïf

```
liste = [1, 2, 3]
op = getattr(liste, 'pop')  # récupérer l'attribut qui s'appelle 'pop'
print(op)       # <built-in method pop of list object at 0x7f1d6fa1a908>
x = op()        # exécuter op comme une fonction
print(x)        # 3
print(liste)    # [1, 2]
assert hasattr(liste, 'remove')
assert not hasattr(liste, 'size')
setattr(liste, 'foo', 10)   # AttributeError: 'list' object has no attribute 'foo'

class A:
    pass

o = A()
setattr(o, 'foo', 10)
print(o.foo)    # 10
```

Exemples d'utilisation

- Vérifier qu'un objet possède les propriétés attendues :

```
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
```

- calculer la méthode à appeler

```
import outputter

def output(data, format='text'):
    fn = getattr(outputter, 'output_' + format, outputter.output_text)
    return fn(data)
```

- récupérer le nom de toutes les méthodes définies sur un objet :

```
def all_methods_name(object):
    return (name for name in dir(object) if callable(getattr(object, name)))

print(list(all_methods_name(tuple())))
```

Sommaire

Méthodes de comparaison

```
__lt__(self, other)     # < (lesser than)
__le__(self, other)     # <= (lesser equal)
__eq__(self, other)     # == (equal)
__ne__(self, other)     # != (non equal)
__gt__(self, other)     # > (greater than)
__ge__(self, other)     # >= (greater equal)
```

Remarques :

- `__eq__` : définir l'égalité logique (==) entre objet
  (par opposition à l'égalité physique, comparaison des identités, (avec **`is`** : id(o1) == id(o2))
- doit retourner `NotImplemented` si l'opération n'est pas implanté pour le couple d'objets.
- par défaut, `__ne__()` utilise `__eq()__`
- `functools.total_ordering()` : engendrer les autres fonctions à partir d'une seule.
- `__hash__()` : donne un hash de l'objet. Si deux objets sont égaux, ils doivent avoir le même hash. Ne pas la définir si les objets sont modifiables !

Surcharge des opérateurs arithmétiques

```
__add__(self, other)          # +
__sub__(self, other)          # -
__mul__(self, other)          # *
__matmul__(self, other)       # @
__truediv__(self, other)      # /
__floordiv__(self, other)     # //
__mod__(self, other)          # %
__pow__(self, other)          # **
__lshift__(self, other)       # <<
__rshift__(self, other)       # >>
__and__(self, other)          # &
__xor__(self, other)          # ^
__or__(self, other)           # |
```

```
__neg__(self)     # - (unaire)
__pos__(self)     # + (unaire)
__abs__(self)     # abs()
__invert__(self)  # ~
```

Pour chacune des précédentes méthodes, il existe deux autres méthodes :

- `__iXXX__` : pour la forme contractée de l'affectation (`a += b` correspond à `__iadd__`)
- `__rXXX__` : utilisé quand l'objet est à droite de l'opérateur. `1 + a` correspond à `__radd__`

Exercice 14 Surcharger l'opérateur '+' pour additionner deux monnaies et '*' pour multiplier une monnaie par nombre.

Émuler un container

Les méthodes spéciales suivantes permettent d'émuler un container (opérateur `[]`) :

```
__len__(self)           # la longueur de l'objet (nombre d'éléments contenus)
__getitem__(self, key)  # récupérer
    # la clé est un objet (un entier pour une séquence) ou un slice.
__setitem__(self, key, value)   # changer la valeur associée à la clé
__delitem__(self, key)  # suppression de self[key]
__iter__(self)          # retourne un itérateur
__reversed__(self)      # itérateur inversé, de la fin au début
__contains__(self, item)        # appartenance (opérateur in)
    # si non définie, __iter__ est utilisé
```

Émuler une fonction : `__call__`

Motivation : Si une classe définit la méthode spéciale `__call__`, ses instances peuvent être utilisées comme des fonctions (opérateur `()`).

```
def fib(n):
    if n <= 1: return n
    else: return fib(n-1) + fib(n-2)

class Fib:
    __cache = {0: 0, 1: 1}

    def __call__(self, n):
        if n not in self.__cache:
            self.__cache[n] = self(n - 1) + self(n - 2)
        return self.__cache[n]

import timeit
print('fib(30) :', timeit.timeit('fib(30)', globals=globals(), number=10))

f = Fib()
print('f(30)  :', timeit.timeit('f(30)', globals=globals(), number=10))
print('f(200) :', timeit.timeit('f(200)', globals=globals(), number=10))
```

```
fib(30) : 2.960786707999432
f(30)  : 3.261299934820272e-05
f(200) : 0.00013471199963532854
```

__new__ vs __init__

Motivation : On redéfinit __new__ car on veut contrôler la création des instances

```
class Singleton(object):
    _instance = None    # accessed if this attribute is not defined in the subclass yet
    def __new__(cls, *args, **kwargs):
        if not isinstance(cls._instance, cls):  # case of a subclass created after a supercl
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class BaseClass: pass
class MyClass(BaseClass, Singleton): pass
class SubMyClass(MyClass): pass
class MySecondClass(Singleton): pass

o1 = MyClass() ; o2 = MyClass()
assert o1 is o2
s1 = SubMyClass() ; s2 = SubMyClass()
assert s1 is s1
assert o1 is not s1
n1 = MySecondClass() ; n2 = MySecondClass()
assert n1 is n2
assert o1 is not n1
assert s1 is not n1
# inspiration https://stackoverflow.com/questions/6760685/creating-a-singleton-in-python
```

Question : Que se passe-t-il si une sous-classe redéfinit __new__?

Autre utilisation de __new__ : hériter d'un type immuable

```
class Point2D(tuple):
    __slots__ = []

    def __new__(cls, x=0, y=0):
        return super().__new__(cls, (x, y))

    @property
    def x(self):
        return self[0]

    @property
    def y(self):
        return self[1]

    def __repr__(self):
        return 'Point2D' + super().__repr__()

p1 = Point2D(1, 2)
print(p1.x) # 1
print(p1.y) # 2
print(p1)   # Point2D(1, 2)
# p1.x = 2  # AttributeError: can't set attribute
# p1.z = 5  # AttributeError: 'Point2D' object has no attribute 'z'
```

Remarque : `__slots__` permet d'interdire d'ajouter des attributs (au sens python).

Sommaire

Intérêt des métaclasses

Intérêt des métaclasses

- Pouvoir intervenir sur la création d'une classe.
- Engendrer automatiquement de nouvelles fonctionnalités, etc.
- Aller vers la définition de DSL
- Voir `https://stackoverflow.com/questions/392160/what-are-your-concrete-use-cases-for-metaclasses-in-python`

Avertissement :

> *Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why). Tim Peters (c.l.p post 2002-12-22)*

Voir `https://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/6581949#6581949` et `http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example`

Les classes sont aussi des objets !

Rappel : Une classe est une « fabrique » à objets.

```
class A:
        pass

a1 = A()          # un premier objet
a2 = A()          # un deuxième objet
print(a1)         # <__main__.A object at 0x7fad95e48470>
type(a1)          # <class '__main__.A'>
```

Mais les classes sont aussi des objets ! Définir la classe A, c'est définir un objet qui correspond à la classe A.

```
A                 # <class '__main__.A'>
B = A             # On peut l'affecter à des noms, y ajouter des attributs, etc.
print(A)          # le passer en paramètre. Ici affiche : <class '__main__.A'>
```

Une classe doit donc être engendrée par quelque chose (comme les objets).

```
type(A)           # <class 'type'>
type(type(A))     # <class 'type'>
```

type a plusieurs formes. On a utilisé celle qui permet de retrouver le type d'un objet. Voyons celle qui permet de créer une classe.

Autre façon de créer une classe

Pour créer une classe, on fournit 3 paramètres à type :

1. le nom de la classe à créer (une chaîne)
2. les classes de base dont elle hérite (un tuple)
3. un dictionnaire des attributs de la classe (dict)

```
A = type('A', (), {})   # équivalent de : class A: pass
print(A)                # <class '__main__.A'>
B = type('B', (A,), {'a': True})
b = B()          # on peut créer des objets !
print(b)         # <__main__.B object at 0x7fad95e50b70>
print(b.a)       # True
```

équivalent de :

```
class B(A):
        a = True
```

Définition : Une metaclasse est une classe (ou callable) qui crée des classes (ex : 'type').

Et en regardant l'attribut `__class__` :

```
n = 5
n.__class__              # <class 'int'>
n.__class__.__class__    # <class 'type'>
b.__class__              # <class '__main__.B'>
b.__class__.__class__    # <class 'type'>
```

Choisir une metaclasse

Lors de la création d'une classe, on peut choisir la métaclasse à utiliser en utilisant le paramètre nommé 'metaclass' dans la liste des parentes.

```
class D(metaclass=type):
        pass

D.__class__     # <class 'type'>
```

On peut aussi définir l'attribut `__metaclass__` dans la classe :

```
class D:
        __metaclass__ = type
```

Sinon, la valeur de `__metaclass__` du module qui est utilisée.
Sinon, la metaclasse de sa première classe parente est utilisée.

Utiliser **type** comme valeur de la métaclasse n'a pas d'intérêt ! C'est ce qui est fait par défaut !

Un premier exemple de « metaclasse »

Commençons par comprendre ce que fait (et donc attend de la metaclasse) l'interpréteur lors de la définition d'une classe :

```
from trace import trace

@trace
def fn(*argv, **kwargs):
    pass

class A(metaclass=fn):
    ac = 1
    def __init__(self, v):
        self.ai = v

print('A =', A)
```

qui affiche :

```
--> fn('A', (), {'__module__': '__main__', '__qualname__': 'A', 'ac': 1,
        '__init__': <function A.__init__ at 0x7f252cf351e0>})
<-- None
A = None
```

On constate, sans surprise, que ce sont bien les trois paramètres attendus par type qui sont fournis à la métaclasse !

Un exemple de metaclasse... avec une fonction

```
# https://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/6581949#6581949
def upper_attr(future_class_name, future_class_parents, future_class_attr):
  """
    Return a class object, with the list of its attribute turned
    into uppercase.
  """
  # pick up any attribute that doesn't start with '__' and uppercase it
  uppercase_attr = {}
  for name, val in future_class_attr.items():
      if not name.startswith('__'):
          uppercase_attr[name.upper()] = val
      else:
          uppercase_attr[name] = val
  # let 'type' do the class creation
  return type(future_class_name, future_class_parents, uppercase_attr)

class A(metaclass=upper_attr):
    ac = 1
    def __init__(self, v):
        self.ai = v

print('A ', A)
assert not hasattr(A, 'ac')
assert hasattr(A, 'AC')
assert A.AC == 1
```

Une classe peut hériter de type : metaclasse

```
# https://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/6581949#6581949
class UpperAttrMetaclass(type):
    # __new__ is the method called before __init__
    # it's the method that creates the object and returns it
    # while __init__ just initializes the object passed as parameter
    # you rarely use __new__, except when you want to control how the object is created.
    # here the created object is the class, and we want to customize it so we override __new_
    # you can do some stuff in __init__ too if you wish
    # some advanced use involves overriding __call__ as well, but we won't see this
    def __new__(metacls, clsname, bases, dct):
        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val
        return super().__new__(metacls, clsname, bases, uppercase_attr)

class A(metaclass=UpperAttrMetaclass):
    ac = 1
    def __init__(self, v):
        self.ai = v
assert not hasattr(A, 'ac')
assert hasattr(A, 'AC')
assert A.AC == 1
```

Exemple de metaclasse : Singleton

```
class Singleton(type):  # a metaclass is being defined
        __instances = {}

        def __call__(cls, *args, **kwargs):
                if cls not in Singleton.__instances:
                        Singleton.__instances[cls] = super().__call__(*args, **kwargs)
                return Singleton.__instances[cls]

class BaseClass: pass
class MyClass(BaseClass, metaclass=Singleton): pass
class SubMyClass(MyClass): pass
class MySecondClass(metaclass=Singleton): pass

o1 = MyClass() ; o2 = MyClass()
assert o1 is o2
s1 = SubMyClass() ; s2 = SubMyClass()
assert s1 is s1
assert o1 is not s1
n1 = MySecondClass() ; n2 = MySecondClass()
assert n1 is n2
assert o1 is not n1
assert s1 is not n1
print(MyClass._Singleton__instances)
# inspiration https://stackoverflow.com/questions/6760685/creating-a-singleton-in-python
```

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.

Quelques outils UML

- Une liste des outils UML : `https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools`
- Outil en SaaS : `https://www.genmymodel.com/`
- Production de diagramme UML à partir d'une description textuelle du diagramme : PlantUML `http://plantuml.com/`
- Dans le monde Eclipse : Papyrus `https://eclipse.org/papyrus/` et UML Designer `https://marketplace.eclipse.org/content/uml-designer`
- Umbrello `https://umbrello.kde.org/`
- BOUML `http://www.bouml.fr/`

Références

[1] P.-A. Muller and N. Gaertner, *Modélisation objet avec UML*.
Eyrolles, 2è ed., 2003.

[2] M. Fowler, *UML 2.0*.
CampusPress Référence, 2004.

[3] OMG, “UML Resource Page.” http ://www.omg.org/uml/.