ARTS ET MÉTIERS ParisTech
Bordeaux - Talence

# UED SIM – SysIn

## Programmation Orientée Objet : Exemples en langage *Python*™ - Séance 1

Jean-luc Charles (jean-luc.charles@ensam.eu)
Éric Ducasse (eric.ducasse@ensam.eu)

## Préambule

- L'archive `TD-OO-1.zip` est disponible sur SAVOIR (plate forme E-Learning de Arts & Métiers ParisTech)
- L'extraction du contenu de l'archive place tous les fichiers extraits dans le répertoire `TD-OO-1`.
- **Tout le travail qui suit doit être fait dans votre répertoire de travail `TD-OO-1`.**

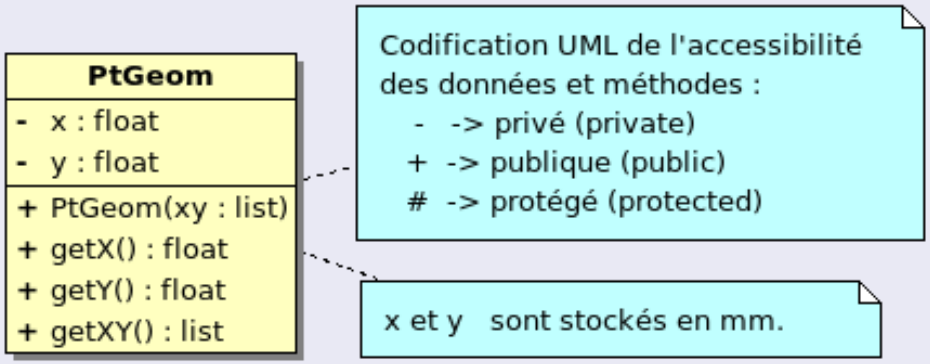
## Objectifs de la séance

Les objectifs de cette séance sont :

- ✓ Comprendre les concepts :
  - de classe,
  - d'objet,
  - d'héritage.
- ✓ Savoir lire et utiliser un diagramme de classes UML.
- ✓ Mettre en oeuvre la Programmation Orientée Objet :
  - savoir écrire une classe simple,
  - savoir implémenter des attributs/méthodes publiques et privés,
  - savoir créer des objets instances d'une classe,
  - savoir utiliser des objets,
  - savoir créer et utiliser une classe dérivée d'une classe de base.

# Partie I : Utiliser une classe pour construire un objet et s'en servir...

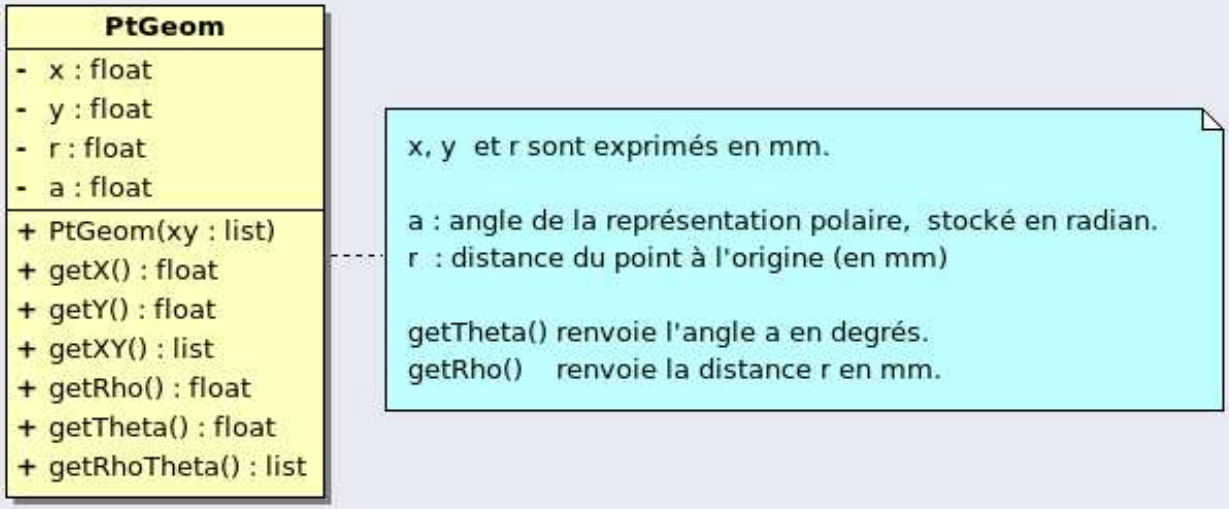
1. Lister les attributs (nom, type, visibilité...) et les méthodes (nom, arguments, valeur retournée...) de la classe **PtGeom**, telle que spécifiée sur le diagramme UML :




2. Charger le programme Python *PtGeom.py* dans l'éditeur IDLE :
   - Repérer les mots clefs... à quoi sert le mot clef `self` ?
   - À quoi reconnait-on le constructeur, à quoi sert-il ?
   - Comment se traduisent en Python les visibilités (private/protected/public) ?
   - Comment désigne-t-on dans le code Python les attributs d'une classe ?
3. Exécuter le programme Python *PtGeom.py*, analyser les sorties écran...
4. Implémenter la méthode d'accès `getXY()` ; ajouter des lignes de test de cette méthode.
5. Charger le programme Python *TracerNuage_11.py* dans l'éditeur IDLE :
   - Identifier les fonctions importées en début de fichier.
   - Localiser l'appel du constructeur de la classe PtGeom.
6. Exécuter le programme *TracerNuage_11.py*.

## Partie II : Évolution de la classe **PtGeom**...

1. Faire évoluer le code Python de la classe **PtGeom** conformément au diagramme UML :




   On pourra consulter l'aide sur les fonctions `radians` et `atan2` du module `math`.

2. Ajouter des lignes de tests *ad hoc*...
3. Charger le programme Python *TracerNuage_12.py* dans l'éditeur IDLE.
4. Exécuter le programme *TracerNuage_12.py*.

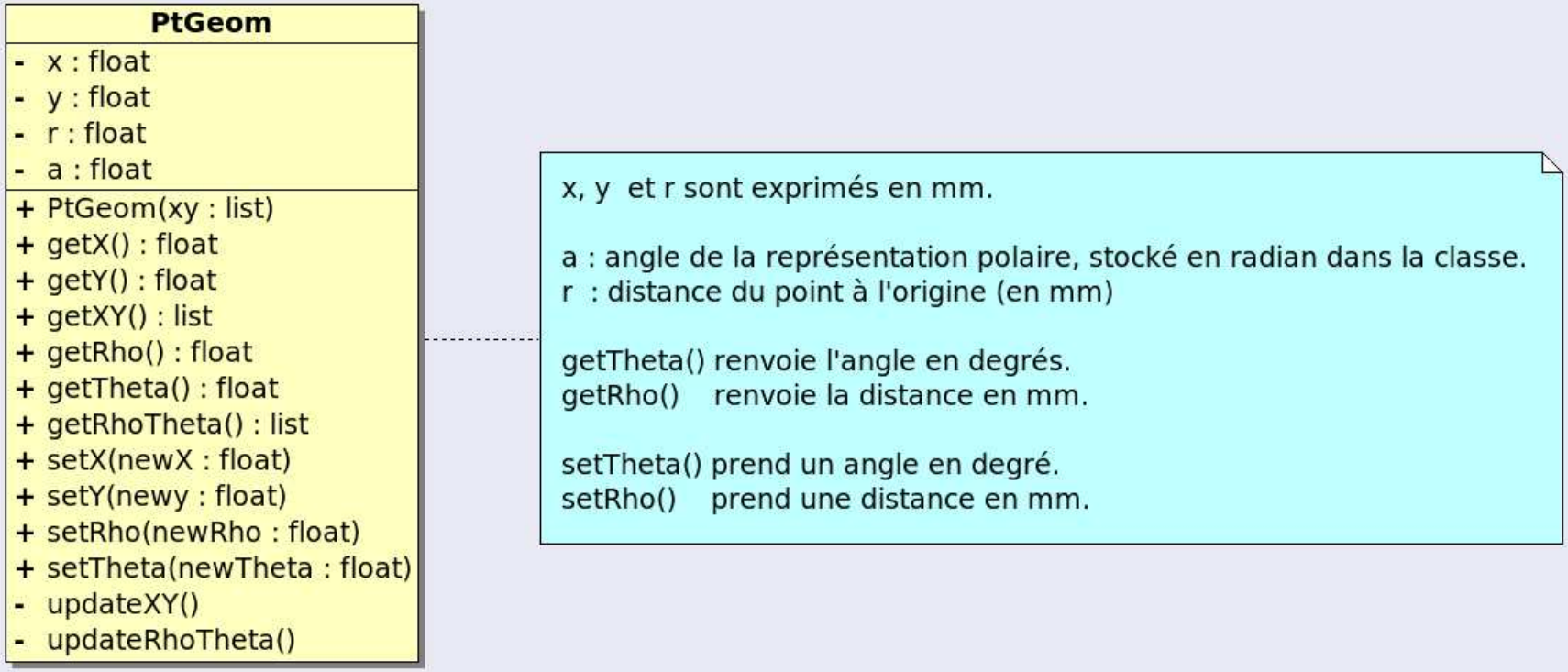
*Discussion : pour les attributs polaires 'a' et 'r', on a les problématiques 'redondance des représentations cartésienne/polaire' et 'stockage mémoire versus calcul'...*

## Partie III : Évolution de la classe **PtGeom** : méthodes d'accès en écriture...

1. Faire évoluer le code de la classe **PtGeom** pour ajouter la méthode `setTheta()` :
   - ▷ qui prend en argument **un angle en degré**,
   - ▷ qui modifie en conséquence la valeur de l'attribut `a`,
   - ▷ qui assure la cohérence des autres données de la classe...
2. Charger et exécuter le programme *TracerNuage_21.py*.
3. Faire évoluer le code de la classe **PtGeom** pour ajouter la méthode `setRho()` :
   - ▷ qui prend en argument **une distance en mm**,
   - ▷ qui modifie en conséquence la valeur de l'attribut `r`,
   - ▷ qui assure la cohérence des autres données de la classe...
4. Charger et exécuter le programme *TracerNuage_22.py*
   Compléter les méthodes de la classe **PtGeom** si besoin...

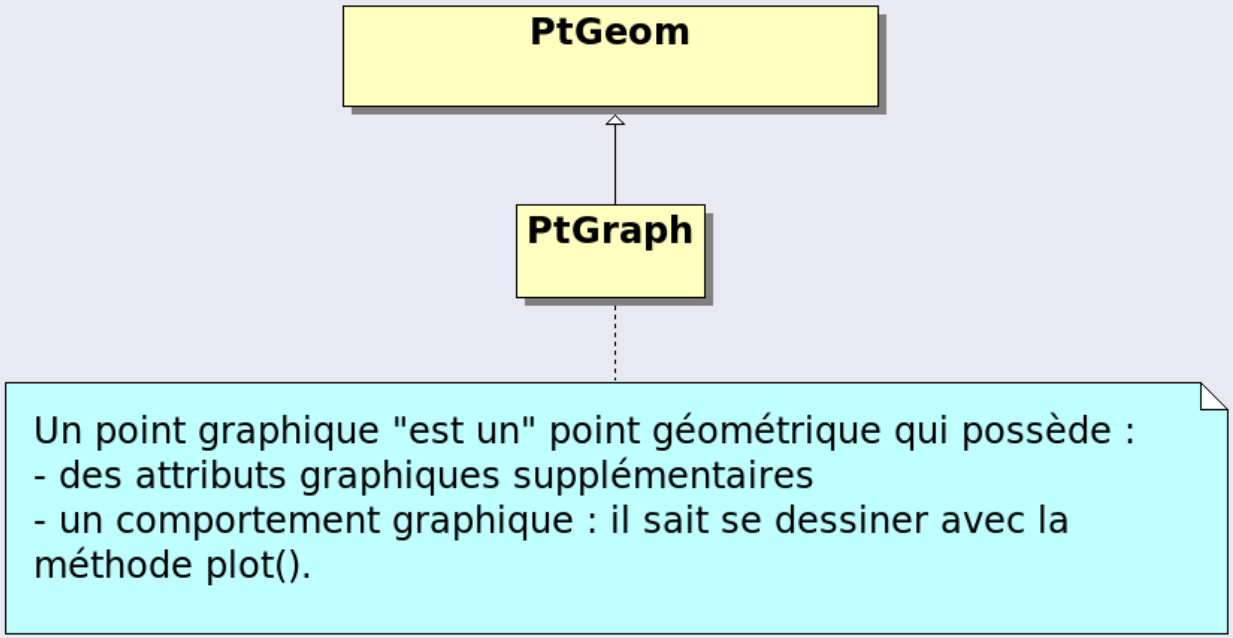
*Discussion : il pourra être utile de factoriser dans des méthodes privées les traitements qui se répètent à plusieurs endroits du code...*

# Récapitulatif : spécifications UML de la classe **PtGeom**



## Le concept d'**héritage**

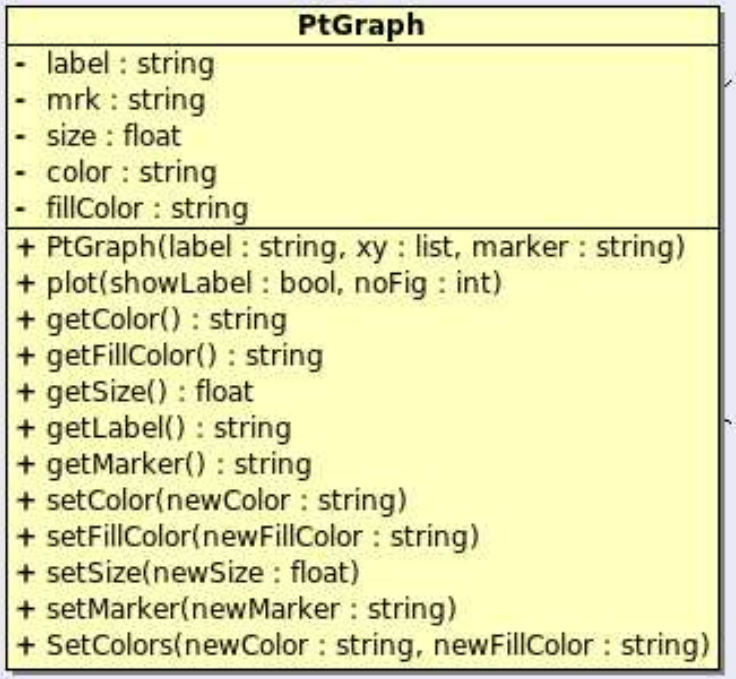
La classe **PtGraph** hérite (ou dérive) de la classe **PtGeom** :




Vocabulaire orienté objet :

- **PtGeom** est la classe mère, ou classe de base,
- **PtGraph** est la classe fille, ou la classe dérivée.

# Spécification UML de la classe **PtGraph**




Attributs
=====
label : nom du point
mrk : type de tracé (croix, rond... par défaut : rond)
size : taille du dessin point (par défaut : 7.0)
color : couleur du contour du marqueur (par défaut : noir)
fillColor : couleur de remplissage (par défaut : gris)

Méthodes
======
Le constructeur prend en argument
- le label (obligatoire)
- la liste (abscisse, ordonnée) (par défaut : (0,0))
- le type de marqueur (par défaut : un rond)

La méthode plot() prend 2 arguments :
- showLabel : si ce booléen est vrai, le label est affiché (par défaut : false)
- noFig : numéro de la figure où faire les tracés (par défaut : 1)

setColors() permet de modifier les couleurs du contour et du remplissage. Si un seul argument est donné, la même couleur est affectée au contour et au remplissage.

## Partie IV : Dérivation de la classe **PtGeom** en classe **PtGraph**...

1. Consulter les spécifications UML de la classe **PtGraph**, qui **hérite** de la classe **PtGeom**.
2. Charger le programme *TracerNuage_31.py* pour déterminer les méthodes de la classe **PtGraph** nécessaires à son exécution.
3. Charger le programme *PtGraph.py* qui définit la classe **PtGraph** :
   - ▷ identifier les mécanismes et les syntaxes supports de l'héritage,
   - ▷ compléter l'implémentation du constructeur,
   - ▷ coder les méthodes utiles à l'exécution du programme *TracerNuage_31.py*,
   - ▷ implémenter des tests élémentaires dans le fichier *PtGraph.py*.
4. Exécuter le programme *TracerNuage_31.py*.
5. Finir d'implémenter la classe **PtGraph** de façon à pouvoir exécuter le programme *TracerNuage_32.py*.

## Objectifs des compléments

Pour ceux qui veulent aller plus loin dans les concepts importants de l'orienté objet, ces compléments proposent les objectifs suivants :

✓ Comprendre les concepts :

- de variable de classe (par opposition à variable d'instance) ;
- de méthode de classe (par opposition à méthode d'instance) ;
- de constructeur polymorphe ;
- de constructeur par copie.

✓ Mise en oeuvre :

- savoir définir et utiliser une variable de classe ;
- savoir définir et utiliser une méthode de classe ;
- savoir utiliser la fonction intrinsèque Python `isinstance` pour définir un constructeur proposant plusieurs formes (polymorphe), dont la "construction par copie".

# Concept de variable de classe, de méthode de classe - Utilisation

- On propose d'ajouter dans **PtGraph** une donnée pour compter, lister, retouver... les points graphiques : cette donnée ne doit pas être présente dans tous les objets créés *(pourquoi ?)*
- La solution est d'utiliser une variable de classe (on parle aussi de variable statique), qui est une variable "factorisée" en un exemplaire unique au sein de la classe .
- La syntaxe de la définition d'une variable de classe est simple :
  - ▷ la définition se fait dans la classe, en-dehors de toute définition de méthode *(pourquoi ?)*,
  - ▷ le nom de la variable dans la classe n'est pas préfixé par `self`. *(pourquoi ?)*.

```
class PtGraph (PtGeom) :
    'Classe de points avec directives graphiques'
    __lptg = []  # Variable privée statique : liste des points créés
```

- Pour compter le nombre de points créés, on peut définir la méthode statique `nb` (on parle aussi de méthode de classe) en utilisant le **décorateur** `@staticmethod` :

```
class PtGraph(PtGeom) :
    ...
    @staticmethod
    def nb():
        return len(PtGraph.__lptg)
```

- L'intérêt d'une méthode statique est qu'elle peut être appelée sans avoir besoin d'un objet : en préfixant simplement son nom par le nom de la classe :

```
from PtGraph import PtGraph
...
print "Nombre de points graphiques créés : ",PtGraph.nb()
```

## Partie V : Évolution de la classe **PtGraph**

Faire évoluer le code Python de la classe **PtGraph** :

1. Ajouter la variable privée statique `__lptg`, de type liste (vide au départ).
2. Modifier le constructeur de la classe **PtGraph** pour ajouter à `__lptg` une référence sur le point graphique en cours de création.
3. Définir la méthode statique publique `nb`, qui retourne le nombre d'élements de la liste `__lptg`.
4. Tester les nouvelles lignes avec des tests unitaires inclus à la fin du fichier *PtGraph.py*.
5. Ajouter la méthode statique `plotAll`, qui effectue le tracé de tous les points référencés dans la liste `__lptg`. Cette méthode prendra un argument optionnel nommé `showLabel` (valeur par défaut : `False`) qui permettra d'afficher ou pas les labels des points.
6. Exécuter le programme *TracerNuage_41.py*.

## Partie VI : Évolution de la classe **PtGraph**

Définir la méthode statique `select` qui prend un argument `labels` et effectue le traitement :

1. En utilisant la fonction `isinstance`, tester si `labels` est du type `list` :
   si oui : former la liste des tous les points géométriques dont le label fait partie de `labels`
   si non : former la liste des tous les points géométriques dont le label est identique à `labels`
2. Dans tous les cas, afficher un message montrant le nombre de points trouvés, et renvoyer la liste des points ainsi selectionnés.
3. Écrire des test unitaires.
4. Essayer d'exécuter le programme *TracerNuage_42.py* : expliquer puis mettre en commentaire les 2 lignes qui provoquent une erreur !

## Constructeur polymorphe

Le but est d'utiliser la fonction intrinsèque Python `isinstance` pour tester le type des arguments passés, et pouvoir ainsi coder un traitement différent en fonction de ce type.

Par exemple dans la classe **PtGeom**, pour construire un nouveau point géométrique on peut accepter un argument :

- de type `list` : liste de 2 nombres flottants (cas du constructeur défini jusqu'ici),
- de type `PtGeom` : objet point géométrique existant qui possède 2 coordonnées accessibles grâce aux méthodes `getX()` et `getY()`.

Une écriture possible d'un tel constructeur polymorphe pourrait être :

```
class PtGeom :
    u'Classe de points géometriques (sans directive graphique)'
    # Constructeur polymorphe
    def __init__(self, xyOrPt = [0, 0]):
        if isinstance(xyOrPt, PtGeom) : ### copie d'un objet PtGeom existant
            pt = xyOrPt                 # le nom pt est plus lisible..
            self.__x = pt.getX()        # att. privé x : abscisse
            self.__y = pt.getY()        # att. privé y : ordonnée
        else :                          ### liste donnant abscisse et ordonnée
            xy = xyOrPt                 # le nom xy est plus lisible...
            self.__x = float(xy[0])     # att. privé x : abscisse
            self.__y = float(xy[1])     # att. privé y : ordonnée
        # MAJ des doord. polaires :
        self.__updateRhoTheta()
```

## Partie VII : Évolution de la classe **PtGeom**

Modifier le code de la classe **PtGeom** :

1. Modifier le constructeur pour permettre de créer un point géométrique en lui passant :
   - ▷ soit une liste fournissant abscisse et ordonnée (constructeur actuel),
   - ▷ soit un point géométrique existant (constructeur par **copie**).
2. Écrire des tests unitaires.
3. Ajouter une méthode `move` prenant les 2 arguments `dx` et `dy`, qui déplace (translation) le point des quantités passées en argument.
4. Écrire des tests unitaires.

## Partie VIII : Évolution de la classe **PtGraph**

Modifier en conséquence le code de la classe **PtGraph** :

1. Modifier le constructeur pour permettre de créer un point graphique en lui passant en premier argument :
   - ▷ soit soit un label (type `str`), auquel cas le deuxième argument (de valeur par défaut : `[0,0]`) pourra être une liste de 2 `float` ou un objet **PtGeom**,
   - ▷ soit un point graphique (type **PtGraph**).
2. Le troisième argument optionnel reste `marker` (valeur par défaut : `'o'`).
3. Écrire des tests unitaires.

Exécuter maintenant le programme *TracerNuage_42.py* dans sa version complète !