



Programmation Orientée Objet (Python)

TP4 : Introduction à la programmation orientée-objet (POO)

[A.Martin](#)

2026-2027



Visual Studio Code



Objectif

Cette première séance de programmation orientée objet (POO) a pour but de vous initier à l'organisation d'un projet Python sous forme de classes, et à l'utilisation de l'héritage et du polymorphisme.

À la fin de ce TP, vous saurez :

- lire et interpréter un diagramme de classes UML (*Unified Modeling Language*) ;
- implémenter des classes avec des attributs et des méthodes ;
- manipuler des objets géométriques simples (points, cercles, rectangles) ;
- appliquer des transformations (translations, rotations, mises à l'échelle) ;
- visualiser graphiquement vos résultats pour valider vos implémentations.

Durée estimée : 4 heures

1. Introduction

L'objectif de ce TP est de concevoir un petit module Python qui manipule des objets géométriques dans le plan. Nous allons suivre le diagramme UML présenté sur la figure 1.

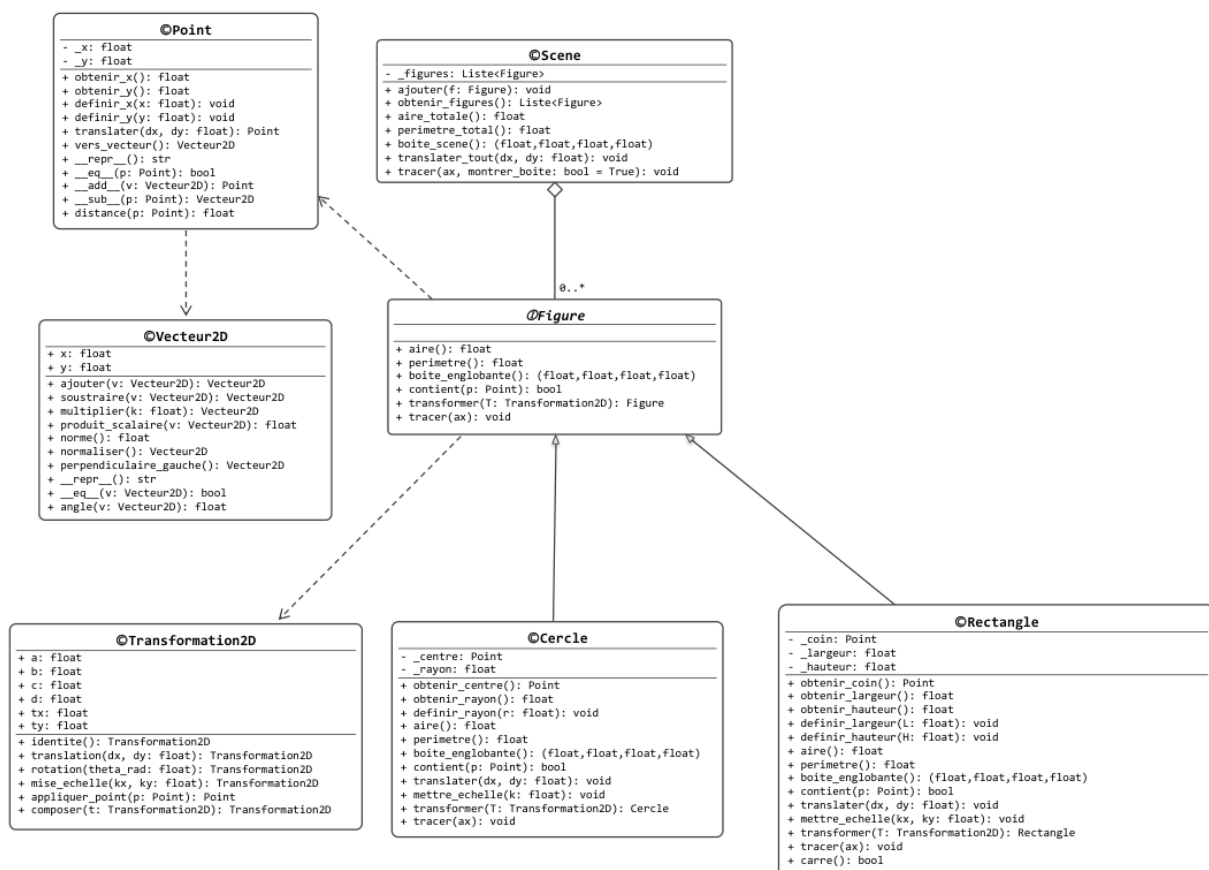


Figure 1 – Diagramme de classes du module « géométrie ».

Ce diagramme représente les classes et leurs relations. Avant de coder, il est essentiel de comprendre quel est le rôle de ces classes et pourquoi elles ont été choisies.

- **Vecteur2D** représente une direction ou une différence entre deux points. La classe regroupe les opérations vectorielles de base (addition, soustraction, produit scalaire, normalisation). En séparant positions (points) et directions (vecteurs), cette classe participe à clarifier le code dans son ensemble.

- **Point** représente une position dans le plan. On peut traduire un point, l'afficher, ou le convertir en vecteur. La classe a été choisie pour distinguer une position fixe d'un déplacement.
- **Transformation2D** encode une transformation affine (translation, rotation, mise à l'échelle). On peut l'appliquer à un point et composer plusieurs transformations. Le fait de centraliser les transformations évite de réécrire les formules à chaque fois.
- **Figure** (interface) définit un « contrat commun » pour toutes les figures. Chaque figure doit savoir calculer son aire, son périmètre, sa boîte englobante (bounding box), tester l'appartenance d'un point, être transformée et être tracée.

De plus, grâce au polymorphisme, la classe **Scene** pourra manipuler indifféremment des cercles et rectangles.

- **Cercle** et **Rectangle** sont deux exemples concrets de figures. Ils couvrent deux cas de base : une forme courbe et une forme polygonale.
- **Scene** regroupe plusieurs figures. Elle permet de calculer l'aire totale, le périmètre total, la boîte englobante et de traduire toutes les figures en bloc. C'est un exemple typique de « composition d'objets ».

2. Travail à réaliser

2.1 Étape préparatoire

1. Ouvrez un terminal.
2. Dans votre dossier de travail « tp_python », désarchivez le fichier ressources « tp4.zip » disponible sur Moodle :

```
unzip tp4.zip
```

3. Déplacez-vous dans le dossier « tp4 ». Vous devriez voir l'arborescence suivante :

```
tp4/
|-- src/
|   |-- geometrie.py      # fichier Python à compléter
|   |-- __init__.py
|-- tests/
|   |-- test_geometrie.py # tests fournis
|   |-- __init__.py
|-- requirements.txt
```

4. Créez l'environnement virtuel « env_tp4 » :

```
python -m venv env_tp4
```

L'arborescence devient :

```
tp4/
|-- env_tp4/              # fichiers de l'environnement virtuel
|-- src/
|   |-- geometrie.py
|-- tests/
|   |-- test_geometrie.py
|-- requirements.txt
```

5. Activez l'environnement virtuel et installez les bibliothèques nécessaires :

```
pip install -r requirements.txt
```

6. Désactivez ensuite l'environnement virtuel (vous l'activerez depuis VSCode).

2.2 Développement des classes

Vous allez implémenter les classes dans l'ordre ci-dessous. Après chaque étape, vous lancerez le test fourni pour visualiser vos résultats :

```
pytest --cov=src --cov-report=term-missing -v
```

Les méthodes `tracer(ax)` sont déjà implémentées dans le squelette. Elles servent uniquement à visualiser vos objets.

Étape 1 – Classe Vecteur2D

Implémentez toutes les méthodes de la classe Vecteur2D, puis exécutez le test fourni.

Le test trace le vecteur d'origine et sa version normalisée. Sur la figure 2, on observe en **bleu** le vecteur $\mathbf{v}$ brut et en **orange** sa version unitaire $\mathbf{v}_u$ (de norme 1).

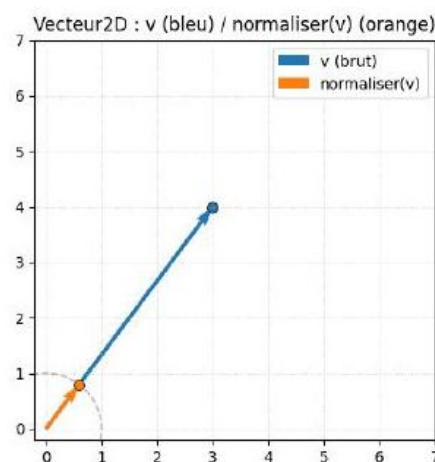


Figure 2 – Résultat du test pour la classe Vecteur2D.

Rappel :

Si $\mathbf{v} = (v_x, v_y)$, alors sa norme vaut $\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2}$, et son vecteur unitaire $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$.

Évitez la division par zéro lors de la normalisation : vous pouvez lever une exception si $\|\mathbf{v}\| = 0$.

Étape 2 – Classe Point

Implémentez toutes les méthodes de la classe Point, puis exécutez le test fourni.

Le test affiche un point avant et après translation.

Sur la figure 3, on distingue en **bleu**, le point initial $\mathbf{P}$ et en **orange**, le point translaté $\mathbf{P}' = \mathbf{P} + \mathbf{v}$.

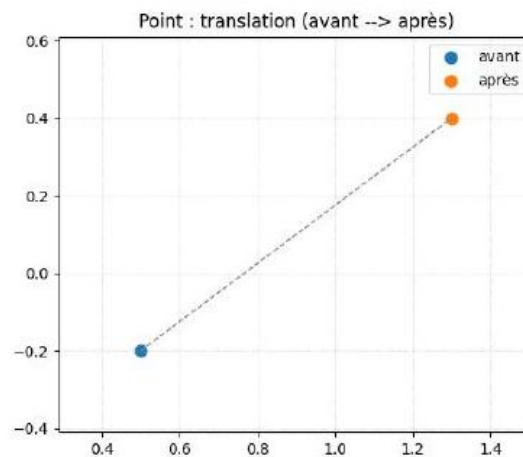


Figure 3 – Résultat du test pour la classe Point.

Astuce :

Surchargez les opérateurs pour simplifier l'écriture :

```
P2 = P1 + v    # translation
v = P2 - P1    # vecteur reliant deux points
```

Étape 3 – Classe Transformation2D

Implémentez l'ensemble des méthodes de la classe Transformation2D, puis testez votre code. Le test ne trace rien ici.

Rappel : composition de deux transformations affines $C = S \circ T$

Composer deux transformations affines revient à combiner leurs matrices et leurs vecteurs de translation.

$$M_C = M_S \cdot M_T ; t_C = M_S \cdot t_T + t_S$$

- La matrice M_C s'obtient en multipliant la matrice de la transformation appliquée **en second** (S) par celle appliquée **en premier** (T). Autrement dit, on applique d'abord T , puis S .
- Le vecteur de translation t_C s'obtient en appliquant M_S à t_T , puis en ajoutant t_S .

Ainsi, la transformation composée agit comme une seule transformation affine :

$$C(p) = M_C p + t_C.$$

Exemple : une rotation de 90° autour de l'origine, suivie d'une translation de +2 en x, donne une matrice de rotation et un vecteur de translation (2, 0).

Étape 4 – Classe Cercle

Implémentez toutes les méthodes de la classe Cercle, puis exécutez le test fourni.

Le test affiche un cercle et deux points de test.

Sur la figure 4, on distingue en **rouge** un point intérieur au cercle et en **orange** un point extérieur.

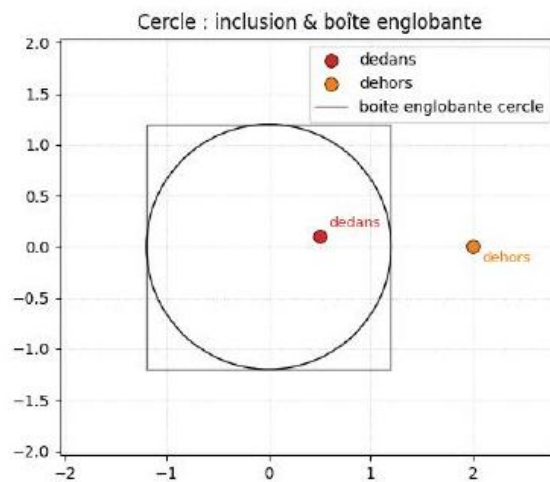


Figure 4 – Résultat du test pour la classe Cercle.

Rappel mathématique :

Un point $P(x, y)$ appartient au cercle de centre $C(x_c, y_c)$ et de rayon R si :

$$\|P - C\|^2 = (x - x_c)^2 + (y - y_c)^2 \leq R^2$$

Étape 5 – Classe Rectangle

Implémentez toutes les méthodes de la classe `Rectangle`, puis exécutez le test fourni.

Le test affiche un rectangle et deux points. Et ensuite une figure sur laquelle plusieurs transformations sont effectuées sur un rectangle d'origine.

Sur la figure 5, on distingue en **rouge** un point intérieur et en **orange** un point extérieur.

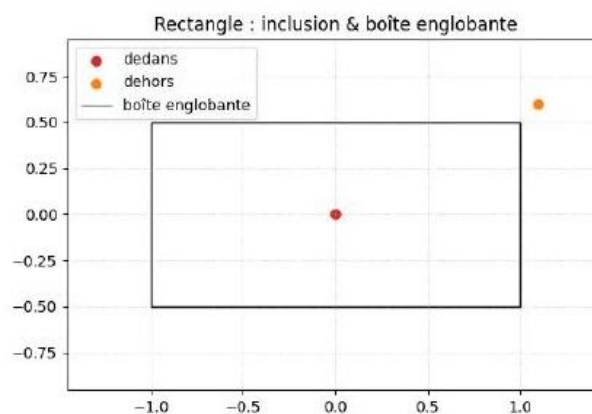


Figure 5 – Résultat du test pour la classe Rectangle.

Condition d'appartenance :

$$x_{min} \leq x_P \leq x_{max} \text{ et } y_{min} \leq y_P \leq y_{max}$$

où $P(x_P, y_P)$ est le point à tester.

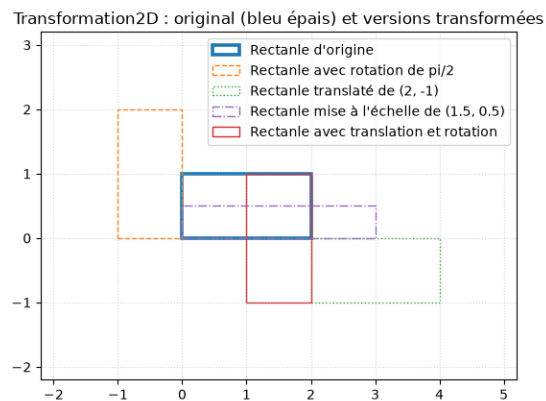


Figure 6 – Résultat du test pour la classe Rectangle avec Transformation.

Étape 6 – Classe Scene

Implémentez toutes les méthodes de la classe Scene, puis exécutez le test fourni.

Le test affiche une scène composée d'un cercle et d'un rectangle, ainsi que leur boîte englobante (en bleu) et leur version translatée (en orange).

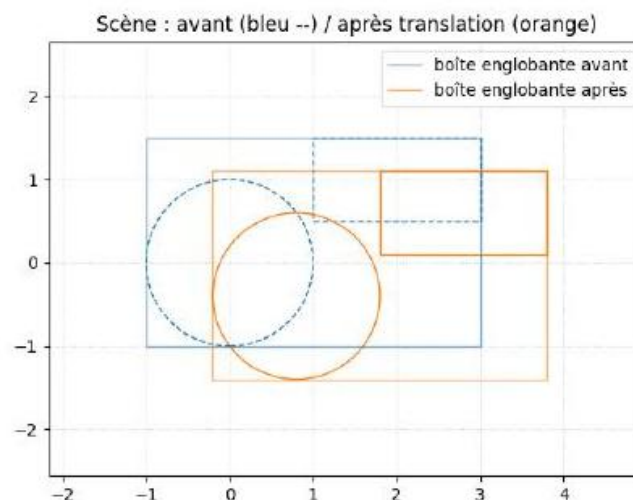


Figure 7 – Résultat du test pour la classe Scene.

Principe de composition :

Une scène regroupe plusieurs objets géométriques. Chaque figure garde son comportement propre (calculs, tracé), tandis que la scène agit comme un conteneur coordonnant les opérations globales.

2.3 Travail facultatif

1. Dans la classe Point, ajoutez une méthode `distance()` qui calcule la distance du point à un autre point passé en paramètre.
2. Dans la classe Vecteur2D, ajoutez une méthode `angle()` qui calcule l'angle entre le vecteur et un autre vecteur passé en paramètre.
3. Dans la classe Rectangle, ajoutez une méthode `carre()` qui teste si le rectangle est un carré.

Testez vos méthodes.

3. Résumé

Dans ce TP, vous avez appris à :

- lire un diagramme de classes UML et en extraire les classes à coder ;
- implémenter des classes et leurs méthodes ;
- voir le principe d'héritage, notamment au travers de l'interface Figure ;
- manipuler et transformer des objets géométriques simples ;
- vérifier vos résultats par des tests et des tracés graphiques.

Ces notions serviront de base au TP suivant dans lequel vous enrichirez ce code pour modéliser des phénomènes optiques simples (technique du « tracé de rayons »).