TP Point dans un polygone

L'objectif final de ce TP est de réaliser un programme permettant de déterminer si un point est situé à l'intérieur d'un polygone. Pour y arriver, nous concevrons une bibliothèque permettant de créer diverses formes géométriques basiques (points, lignes, polygone, etc.) en implémentant pour chacune d'elles des comportements qui serviront à la résolution du problème global.

L'ensemble du travail sera réalisé en utilisant la programmation orienté objet et en veillant à rendre notre code le plus réutilisable possible pour d'autres problèmes que celui de l'inclusion d'un point dans un polygone.

Nous organiserons notre code en quatre fichiers :

— `base.py` contenant le code sur les formes géométriques et leurs enveloppes ;
— `point.py` : les points ;
— `polyligne.py` : les lignes et polylignes ;
— `polygone.py` : les polygones .

1 La classe Point

Nous commencerons par définir la classe représentant les points. Sa représentation UML est la suivante :

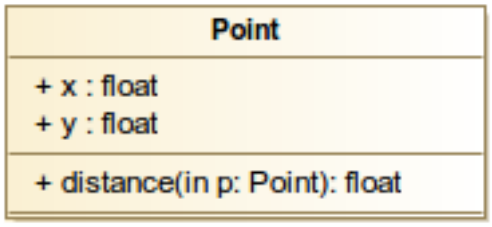

FIGURE 1 – La classe Point

1. Dans le fichier `point.py` définissez une classe `Point`. Le constructeur de cette classe attend deux paramètres : les coordonnées x et y du point.

Sauf mention contraire, dans ce TP, tous les attributs des classes seront de visibilité privés. Par convention, nous utiliseront un `_` devant le nom d'un attribut pour indiquer qu'il est privé.

2. Indiquez que les attributs `x` et `y` sont privés.

Nous souhaitons ajouter une méthode permettant de calculer la distance d'un point à un autre. Pour ce faire, nous aurons besoin d'avoir accès aux coordonnées `x` et `y` de cet autre point. Celle-ci ayant été rendues privées, il nous faut définir un moyen d'y accéder tout de même en dehors de l'objet.

3. Définissez les accesseurs des attributs `x` et `y` qui permettront de lire les valeurs des attributs en dehors de la classe.
4. Vous pouvez maintenant ajouter une méthode `distance(p)` retournant la distance à une autre instance de `Point` désignée par la variable `p`.

Lorsque nous créons une instance de point et essayons de l'afficher dans le console (`print(Point(2, 3))` par exemple), nous obtenons un quelque chose du type `<__main__.Point object at 0x7fe8f62dd550>`. Pour rendre cet affichage plus utile, il nous faut définir la méthode `__str__()`. Il s'agit de la méthode spéciale qui est appelée lorsque nous appelons la méthode `print()` sur un objet.

5. Définissez la méthode `__str__()` de manière à ce que l'affichage d'un point de coordonnée x=2 et y=3 soit le suivant :

```
POINT(3 2)
```

Créez deux instances de `Point` de la manière suivante :

```
p1 = Point(2, 3)
p2 = Point(2, 3)
```

6. Vérifiez que ces deux points sont bien identiques (`p1 == p2`). Quel est le résultat ?

Pour remédier au problème, nous allons définir la méthode spéciale `__eq__()` qui est appelée lorsque l'opérateur `==` est utilisé.

7. Définissez la méthode `__eq__(point)` retournant `True` si le point passé en paramètre possède les mêmes coordonnées que le point courant, et `False` sinon.

2 La classe FormeGeo

Nous allons maintenant définir une classe `FormeGeo` qui représente une forme géométrique quelconque. La classe `Point` hérite de `FormeGeo` : un point est un type particulier de forme géométrique. Nous n'instancierons jamais d'objet de la classe `FormeGeo` sans préciser de quel type d'objet il s'agit (un point, un polygone, etc.) : la classe `FormeGeo` est une classe abstraite.

8. Ajoutez une classe `FormeGeo` dans le fichier `base.py` et indiquez que la classe `Point` hérite de `FormeGeo` (il sera nécessaire d'importer le module `base` dans `point.py`).
9. Pour simuler le comportement d'une classe abstraite, levez une exception de type `NotImplementedError` dans le constructeur de `FormeGeo`.

De même que pour la classe `Point` nous allons ajouter une méthode `deplacer(dx, dy)` à la classe `FormeGeo`. Cette méthode sera abstraite. Son seul intérêt sera de contraindre toutes les classes héritant de `FormeGeo` à l'implémenter : toutes les formes géométriques pourront être déplacées de dx, dy.

10. Ajoutez la méthode `deplacer(dx, dy)` permettant de déplacer une forme géométrique de `dx` selon l'axe des `x` et `dy` selon l'axe des `y`.

Cette méthode sera également abstraite : l'implémentation dépendra du type de la forme géométrique. La classe `Point` héritant de `FormeGeo` et étant une classe contrète, il sera en revanche obligatoire d'implémenter la méthode `deplacer(dx, dy)` dans cette classe.

11. Rendez la méthode abstraite en levant là encore une exception.
12. Implémentez la méthode dans la classe `Point`.

3 La classe Enveloppe

Nous allons complétez notre modèle en ajoutant un classe `Enveloppe` représentant le rectangle englobant de n'importe quelle forme géométrique. Une enveloppe est par ailleurs elle-même une forme géométrique : la classe `Enveloppe` hérite de `FormeGeo`.

Le diagramme de classes que nous allons implémenter dans cette partie est le suivant :

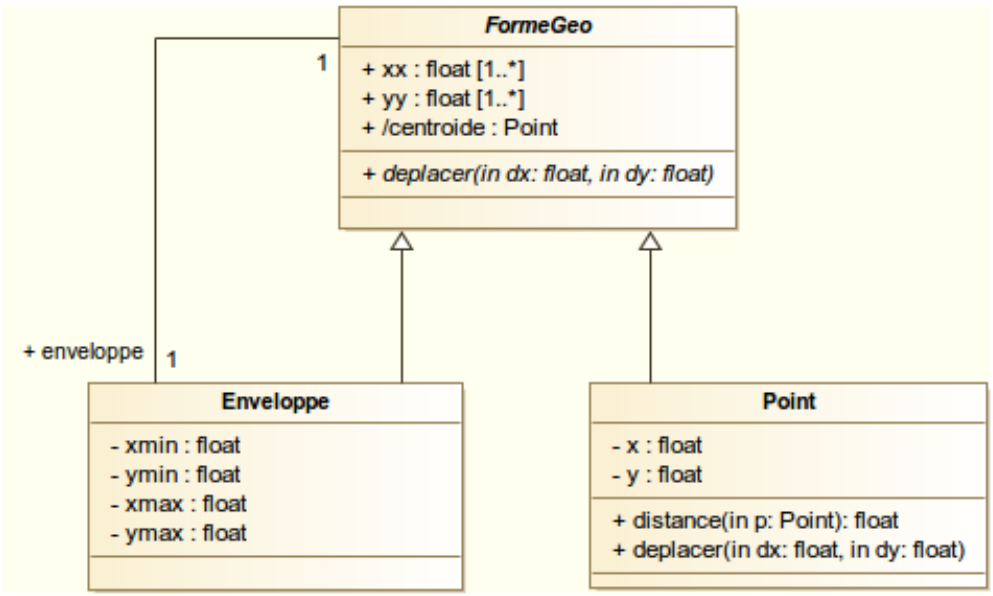

FIGURE 2 – Classes Enveloppe, FormeGeo et Point

Dans `FormeGeo` les attributs dérivés `xx` et `yy` contiennent les listes des coordonnées de l'ensemble des sommets d'une forme géométrique. Les attributs `xmin`, `ymin`, `xmax` et `ymax` de `Enveloppe` représentent les coordonnées minimales et maximales du rectangle.

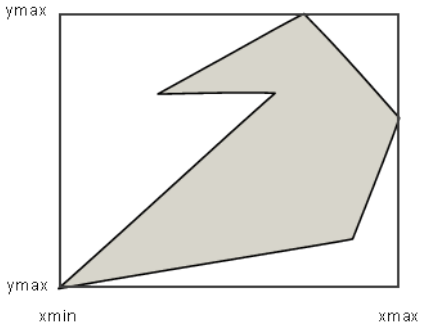

FIGURE 3 – Enveloppe d'une forme géométrique

13. Ajoutez la classe `Enveloppe` avec son constructeur (ses paramètres doivent être deux listes de coordonnées) et en indiquant que ses attributs sont privés.

14. Nous aurons besoin à la fin du TP de lire les coordonnées maximales de l'enveloppe d'une forme géométrique : ajoutez les accesseurs de `xmax` et `ymax` (vous avez le droit de le faire pour les autres attributs si vous le souhaitez).
15. Faites en sorte que l'affichage d'une enveloppe (appel de la méthode `print()`) retourne quelque chose du type :

```
ENVELOPPE(Xmin:2 Ymin:3 Xmax:7 Ymax:5)
```

16. Ajoutez les attributs `xx` et `yy` à la classe `FormeGeo`. Rien ne nous permet de définir ces attributs dans la classe `FormeGeo`. Rendez-les donc abstraits.
17. Ajoutez l'attribut `enveloppe` à la classe `FormeGeo`, en précisant ce qu'il contient.

Note : Remarquez qu'ici nous avons définit ce que contient l'attribut 'enveloppe' alors que nous n'avons rien définit d'autre dans la classe 'FormeGeo' (toutes les méthodes, y compris le constructeur, sont abstraites).

18. Implémentez le fait que l'enveloppe d'une enveloppe est l'enveloppe elle-même.

4 La classe Polyligne

Une polyligne est un type particulier de forme géométrique. Elle est composée d'un nombre indéterminé de points (au moins deux).

19. Créez la classe `Polyligne`, implémentez son constructeur et gérez l'affichage d'un objet polyligne (du type `POLYLIGNE((0 3)(2 4)(0 7))`)

Pour éviter que des erreurs surviennent dans la suite, nous allons interdire deux points successifs d'être identiques.

20. Modifiez le constructeur pour qu'un point ne soit pas pris en compte s'il est identique à son prédécesseur.
21. Implémentez la méthode `deplacer(dx, dy)` et le calcul des attributs `xx` et `yy`.
22. Ajoutez l'attribut dérivé `longueur` qui retourne la longueur de la polyligne.
23. Ajoutez également la méthode `pente(i)` retournant la pente du segment entre les i-ème et (i+1)-ème points.

A l'image de ce que nous pouvons faire pour une liste, en parcourant ses éléments dans une boucle *for*, il serait intéressant de pouvoir parcourir l'ensemble des points d'un polyligne en utilisant une syntaxe du type `for point in polyligne:`, c'est à dire en *itérant* sur une polyligne.

Le comportement d'un objet itérable est définissable à l'aide de la méthode spéciale `__iter__()`. Cette méthode est appelée à chaque passage dans la boucle *for* et renvoie l'élément utilisé à cette étape de la boucle. La méthode à employer vous est donnée ci-dessous.

24. Recopiez la méthode spéciale `__iter__()` dans votre classe `Polyligne`.

```
def __iter__(self):
    for point in self._points:
        yield point
```

Note : vous êtes encouragé à tester votre code régulièrement.

5 La classe Polygone

Notre modèle final est définit comme suit en UML :

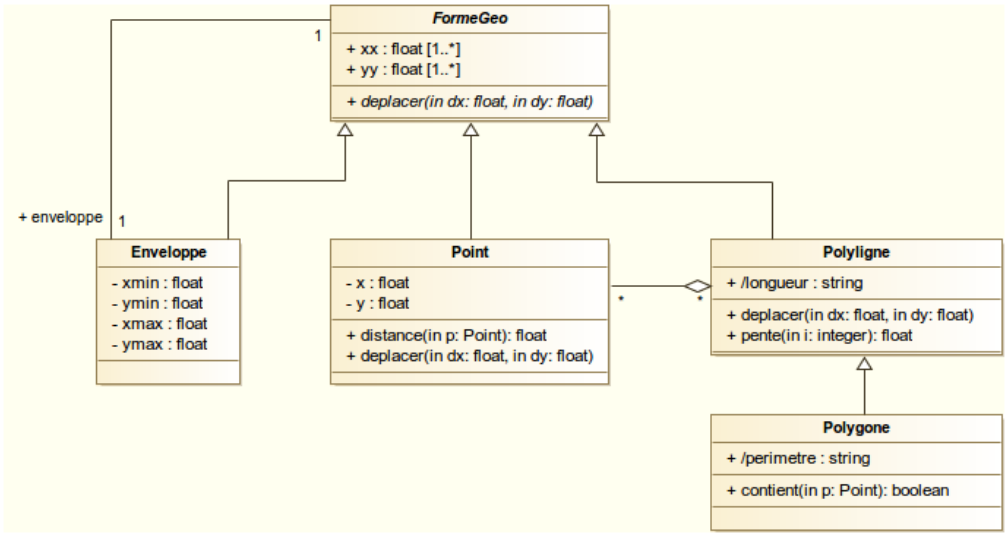

FIGURE 4 – Diagramme de classes complet

Un polygone est un type particulier de polyligne, dont le premier et le dernier point sont identiques.

25. Ajoutez une classe `Polygone` héritant de `Polyligne` et vous assurant que les méthodes héritées de `Polyligne` soient toujours correctes.
26. Pour plus de lisibilité, ajoutez un attribut dérivé `perimetre` égal à la longueur de la polyligne (du polygone).

6 La relation géométrique contient

L'algorithme que nous allons mettre en oeuvre pour déterminer si un point est contenu dans un polygone consiste à compter combien de fois une demi-droite partant de ce point et allant dans n'importe quelle direction intersecte le contour du polygone :

— si le nombre d'intersections est impair, le point est à l'intérieur du polygone ;
— si le nombre d'intersections est pair, le point est à l'extérieur du polygone.

Pour la mise en oeuvre de l'algorithme, nous effectuerons les opérations suivantes :

— définition d'un point très loin à l'extérieur de l'enveloppe (par exemple : `x=xmax+1000*(xmax-xmin)` et `y=ymax+1000*(ymax-ymin)`) ;
— création d'une ligne entre ce point et le point testé ;
— pour chaque segment du contour du polygone, calcul de l'intersection avec la ligne, si elle existe.

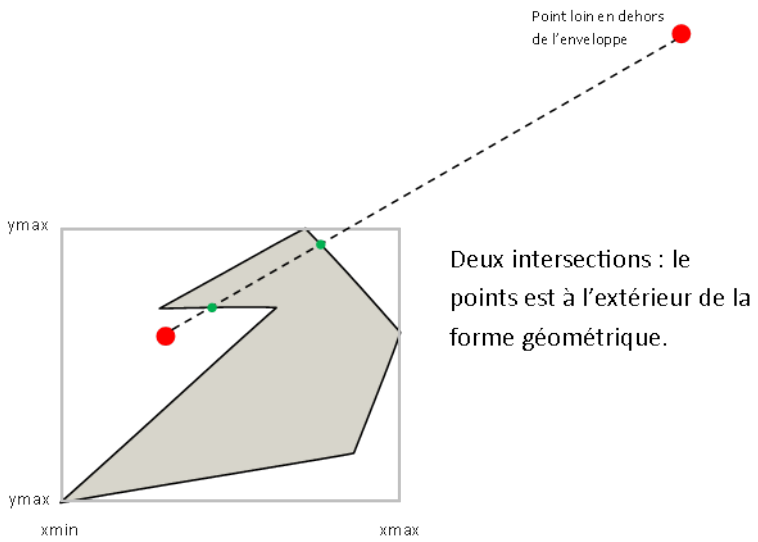

FIGURE 5 – Illustration de l'algorithme

27. A plusieurs reprises dans l'algorithme, nous allons cherche à calculer l'intersection de deux segments. Pour simplifier le code, nous vous proposons de créer une classe `Segment` héritant de `Polyligne` (un segment est une polyligne de deux points uniquement). Cette classe contiendra notamment des méthodes `pente()` (la pente du segment, calculable grace à la méthode `pente(i)` de `Polyligne`), `origine()` (ordonnée à l'origine) et `calculer_intersection(segment)` calculant l'intersection si elle existe avec un autre segment.

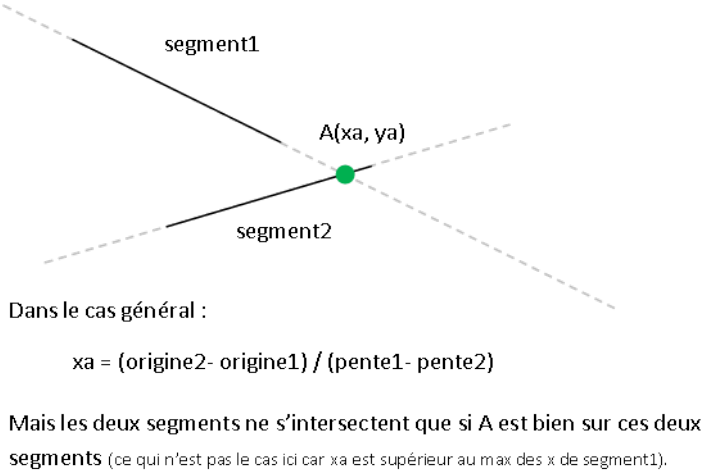

FIGURE 6 – Intersection de deux segments

25. Implémentez l'algorithme dans la méthode `contient(p)` de la classe `Polygone`
26. Testez votre méthode. Vous pouvez utiliser matplotlib pour vérifier que le résultat est correct.

7 Aller plus loin

Nous souhaiterions tester si le centroïde d'un polygone quelconque (points tirés aléatoirement) se situe à l'intérier du polygone où à l'extérieur de celui-ci. Cette fonctionnalité peut avoir un intérêt, dans un cadre cartographique, pour savoir si l'on peut attacher automatiquement des étiquettes au centroïdes de polygones où s'il faut utiliser d'autres points.

Ajoutez un attribut dérivé `centroide` à la classe forme géométrique (toutes les formes géométriques possèdent un centroïde), et implémentez son calcul dans les classes adaptées. Vous pouvez enfin complétez la classe `Polygone` avec une méthode indiquant si le centroïde est à l'intérieur du polygone ou pas.