TP Géométries

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.

1 La classe Point

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

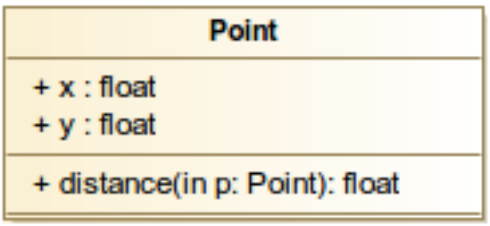

FIGURE 1 – La classe Point

1. Dans un fichier `geometrie.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`.
4. Vous pouvez maintenant ajouter une méthode `distance(p)`, où `p` est une instance de `Point`, et retournant la distance à cet autre point `p`.
5. Complétez la classe `Point` avec une méthode `deplacer(dx, dy)` permettant de déplacer un point de `dx` selon la coordonnée `x` et `dy` selon la coordonnée `y`.

Lorsque nous défissons une instance de point et essayons de l'afficher dans le console, 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.

6. Définissez la méthode `__str__()` pour l'ffichage d'un point de coordonnée x=2 et y=3 soit le suivant :

```
POINT(3 2)
```

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

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

Puis 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.

8. 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.

9. Ajoutez une classe `FormeGeo` à votre programme et indiquez que la classe `Point` hérite de `FormeGeo`.
10. La notion de classe abstraite n'existe pas en Python. Pour simuler ce comportement, 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 conrtaindre toutes les classes hériant de `FormeGeo` à l'implémenter : toutes les formes géométriques pourront être déplacées de dx, dy.

11. Ajoutez la méthode `deplacer(dx, dy)`. Pour la rendre abstraite, le corps de la méthode contiendra uniquement une levée de l'exception `NotImplementedError`.

Note : Grace au polymorphisme, nous pourront redéfinir cette méthode dans chacune des classes qui héritera de 'FormeGeo'. Cette méthode étant déjà définie dans 'Point' nous n'avons pour l'instant rien d'autre à faire.

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 géomé-

trique : la classe `Enveloppe` hérite de `FormeGeo`.

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

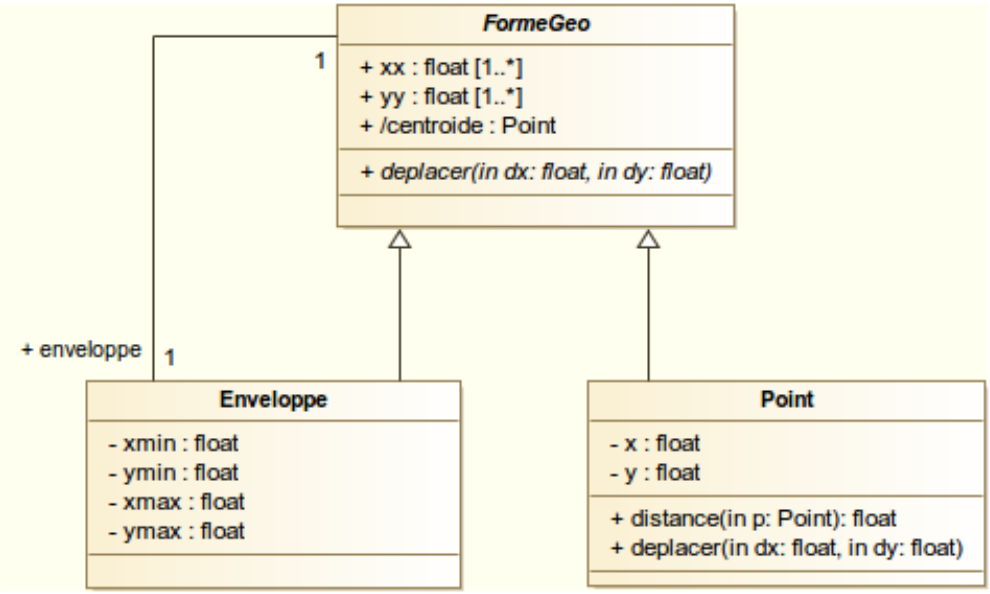

FIGURE 2 – Classes Enveloppe, FormeGeo et Point

Les attributs `xmin`, `ymin`, `xmax` et `ymax` de `Enveloppe` représentent les coordonnées minimales et maximales du rectangle.

Les attributs dérivés `xx` et `yy` contiennent les listes des coordonnées de l'ensemble des sommets d'une forme géométrique.

12. Ajoutez la classe `Enveloppe`, avec son constructeur et en utilisant bien des attributs privés.
13. 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)
```

14. Ajoutez les attributs `xx` et `yy` à la classe `FormeGeo`. Il ne contiennent pour l'instant rien (dans la classe `FormeGeo`).
15. 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 ormeGeo' (toutes les méthodes, y compris le constructeur, sont abstraites).

16. 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).

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

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

18. Modifiez le constructeur pour que si deux points successifs passés en paramètre sont identiques, l'un des deux points soit supprimés.
19. Implémentez la méthode `deplacer(dx, dy)`.
20. Ajoutez l'attribut dérivé `longueur` qui retourne la longueur de la polyligne.
21. 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.

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

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

— des tests de tout ça... ? (pente d'une droite verticale par ex...)

5 La classe Polygone

Notre modèle final est définit comme suit en UML :

Diagramme de classes complet

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

23. Ajoutez une classe `Polygone` héritant de `Polyligne` et vous assurant que les méthodes héritées de `Polyligne` soient toujours correctes.
24. 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 mide 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.

25. Implémentez l'algorithme dans la méthode `contient(p)` de la classe `Polygone`
26. Testez votre méthode.

7 Organisation du code

Jusqu'ici nous avons écrit tous notre code dans un unique fichier .py. Ce fichier commence à être volumineux et difficilement lisible. Pour faciliter le compréhension du code, nous allons répartir les classes sur plusieurs fichiers (aussi appelés modules). Ces modules seront regroupés dans un package Python.

=> Créez un nouveau dossier **API_geo**. Pour que ce dossier soit reconnu comme un package Python, créez à sa racine un fichier \lstinline{**init**.py}.

=> Créez ensuite dans ce package cinq modules : `base.py`, `point.py\`, `polyligne.py`, `polygone.py\` et \lstinline{enveloppe.py}.

=> Copiez alors dans chacun des modules les classes correspondantes :

— `base.py` : tout ce qui concerne les formes géométriques génériques
— `point.py` : la classe **Point**
— `polyligne.py` : la classe **Polyligne**
— `polygone.py` : la classe **Polygone** et ses dérivées
— `enveloppe.py` : la classe **Enveloppe**

Lorsque cela est fait, il reste à configurer proprement les imports pour que le code soit de nouveau fonctionnel. Par exemple, lorsque la classe Point est utilisée dans un module, on écrira au début du module : \lstinline{from API_geo.point import Point}.

=> Importez les classes utilisées dans chacun des modules.

=> Pour faciliter encore la compréhension du code, documentez le si ce n'est pas déjà fait et prévoyez des tests unitaires pertinents (les exemples des parties précédentes).