Géolib

Dernière révision le 13 septembre 2006.

Geolib est la librairie utilisée par WxGéométrie pour définir et manipuler les objets géométriques usuels. Bien qu'elle soit étroitement liée à WxGéométrie, elle peut éventuellement être utilisée seule.

Plus précisemment, il s'agit même d'une spécification de geolib : **tout objet doit pouvoir être manipulé en ligne de commande, dans un interpréteur python quelconque**[1].

Ceci, pour plusieurs raisons.

- Tout d'abord, je pense que cette contrainte a assaini WxGéométrie, en clarifiant la répartition des tâches entre chaque partie du projet.
- Plus évident, cela facilite le débogage. Inutile de lancer tout un contexte graphique pour déboguer un objet... d'autant qu'un objet mal défini risque de perturber l'affichage, et de générer tout un tas d'erreurs annexes qui viendront polluer l'origine du problème.
- Enfin, il peut être utile de pouvoir travailler avec ces objets en ligne de commande, par exemple pour effectuer rapidement un grand nombre de tâches.

A long terme, il est de tout manière prévu que WxGéométrie puisse autant que possible travailler en ligne de commande, ce qui reste la méthode de travail la plus austère mais souvent la plus efficace, y compris dans les environnements de travail moderne[2].

Au niveau de sa structure, geolib contient actuellement (version 0.106) deux sous-librairies, *objets.py* et *feuille.py*.

Le premier fichier définit les objets géométriques, et le deuxième fournit un contexte de travail (une « feuille ») qui est utilisé par Wxgéométrie, mais doit pouvoir une fois encore être utilisé dans un interpréteur Python.

I. Les objets géométriques

Un objet géométrique doit être défini dans le fichier *objets.py* de geolib.

Concrêtement, c'est un sous-classe de la classe *Objet*.

Si l'on doit créer plusieurs objets apparentés, il est conseillé de préalablement créer une classe mère.
Par exemple, une classe regroupe les différents cercles.
Les classes qui servent à regrouper des objets, et qui ne doivent pas être utilisées directement, doivent[3] porter le suffixe *_generique*. Exemple : *Cercle_generique*.

On notera qu'en Python, un objet peut hériter de plusieurs classes, mais cela complique un peu le processus de création d'objet.

1 A noter cependant que geolib depend de pylib, et donc d'un certain nombres de librairies, soit internes au projet Wxgéométrie, soit externes, comme matplotlib et Numeric.
2 Apple®, le champion du graphisme, n'hésite pas à mettre en avant que MacOS X® est compatible POSIX.
3 Le terme est un peu fort... Il s'agit d'une convention, par souci de clarté, et non d'une obligation liée à l'architecture du programme.

1. Format

Pour commencer, nous allons étudier ensemble deux objets : l'objet Cercle, qui est un exemple d'objet relativement simple, et l'objet Droite, plus typique de ce que l'on rencontre habituellement.

a) Etude d'un exemple simple : l'objet Cercle

Voici le code associé à l'objet Cercle.

Comme le code n'est pas stabilisé, il est possible que celui-ci ait un peu évolué dans votre version de WxGéométrie (ajout de tests sur le rayon par exemple).

Néanmoins, l'analyse ci-dessous reste d'actualité.

```
class Cercle(Cercle_generique):
    """Un cercle.

    Un cercle defini par son centre et son rayon"""

    def __init__(self, centre, rayon = None, **kw):
        if rayon == None:
            self.synchroniser_feuille()
            if self.__feuille__:
                xmin, xmax, ymin, ymax = self.__canvas__().fen()
                rayon=random.uniform(0,min(abs(xmin-xmax),abs(ymin-ymax)))
            else:
                rayon = 1
        rayon = self.nb2var(rayon)
        Cercle_generique.__init__(self)
        self.enregistre(Point_generique, Variable)
```

Décortiquons le pas à pas :

```
class Cercle(Cercle_generique):
```

Le nom de classe doit etre **court et explicite**. Eventuellement, on utilisera la **tiret bas** « _ » pour plus de clarté, si plusieurs mots interviennent dans le nom.

Exemple : *Cercle_diametre* pour un cercle défini par la donnée d'un diamètre.

Il sert pour passer des commandes dans WxGéométrie par exemple (ou pour les sauvegardes).
Il est aussi utilisé tel quel, par endroits, dans l'interface graphique de WxGéométrie (de moins en moins).

On remarque que *Cercle* n'hérite pas directement d'*Objet*, mais de *Cercle_generique* (voir l'introduction du **I.** à ce propos).

```
    """Un cercle.

    Un cercle defini par son centre et son rayon"""
```

Juste après l'introduction de la classe, vient une ou plusieurs lignes d'information sur celle-ci.

Pour tout objet Python, ces lignes de documentation sont conseillées, mais néanmoins en général facultatives.

Ici, **dans geolib, la première ligne est obligatoire**, et fait l'objet d'une **spécification précise**, à savoir :

- elle doit commencer par l'article indéfini « Un » ou « Une »[4] immédiatement après les 3 guillemets, et suivi d'un espace.
- elle doit contenir en français correct le nom de l'objet (et rien d'autre).
- Elle se termine par un point.

Les lignes suivantes sont facultatives, quoique bienvenues. Elles expliquent la signification des arguments, et éventuellement donne des informations (d'ordre mathématique par exemple) sur l'objet.

Une ligne vierge doit séparer ces lignes de la première.

```
    def __init__(self, centre, rayon = None, **kw):
```

La méthode __init__ est appelée lors de la création de l'objet.

Ses paramètres ont être ceux utilisés pour créer l'objet.

Ici, je créerai mon objet par une instruction du style :

```
C = Cercle(A, 3)
```

...pour créer un cercle de centre A et de rayon 3, où A est un objet de type *Point_generique* déjà créé.

A ira dans la variable *centre*, et 3 dans la variable *rayon*.

Les valeurs facultatives doivent être suivies de « = None »[5].

Ici, l'utilisateur n'est pas obligé de rentrer un rayon, le code qui suit générera une valeur par défaut.

```
        if rayon == None:
            self.synchroniser_feuille()
            if self.__feuille__:
                xmin, xmax, ymin, ymax = self.__canvas__().fen()
                rayon=random.uniform(0,min(abs(xmin-xmax),abs(ymin-ymax)))
            else:
                rayon = 1
```

Enfin, ****kw** est obligatoire ; il permet de rentrer comme arguments de l'objet différentes informations de style (couleur, épaisseur), utilisées en mode graphique.

Exemple :

```
C = Cercle(A, 3, visible = False, couleur = "blue")
```

```
        rayon = self.nb2var(rayon)
```

Les valeurs numériques sont stockées dans des objets spéciaux de type *Variable*.

Le fonctionnement des objets de type *Variable* est assez complexe, et nécessiterait à lui tout seul une section de cette documentation (**à faire !**).

Tout ce qu'il faut savoir en général, c'est qu'une valeur numérique entrée comme argument doit être convertie en objet *Variable*, après lui avoir éventuellement fait subir un traitement ou des vérifications (par exemple, ici,

4 Avec majuscule de préférence, mais ce n'est que pour des raisons esthétiques !...

5 Il est bon, par souci d'homogénéité du code, que cette règle soit toujours respectée, bien qu'on puisse envisager dans l'absolu d'autres manières de traiter les arguments facultatifs.
Par ailleurs, cette règle est conforme aux usages de programmation en Python généralement en vigueur.

il aurait été utile de vérifier que le rayon est positif).

Pour cela, on utilise la méthode *nb2var*, reçu par héritage de la classe *Objet*.

```
        Cercle_generique.__init__(self)
```

On initialise l'objet comme étant un Cercle générique.

(Pour cela, on appelle la méthode *__init__* de la classe mère).

```
       self.enregistre(Point_generique, Variable)
```

La méthode *enregistre* (héritée de la classe Objet) effectue un certain nombre d'actions :

- Elle enregistre l'objet dans la feuille courante (si il y en a une de définie).
- Elle vérifie que les arguments entrés en paramètres par l'utilisateur sont bien des types mentionnés ci-dessus: *Point_generique* pour le centre, et *Variable* pour le rayon (qui a été converti avec la méthode *nb2var*).
- Elle enregistre toutes les variables de l'espace de noms courant dans l'espace de noms de l'objet.
  Par exemple, ici, cela revient à effectuer :
  self.rayon = rayon
  self.centre = centre
  (exceptions : *self* lui-même et *kw* ne sont jamais enregistrés).
- Elle met à jour la liste des dépendances (l'objet cercle dépend de l'objet centre et de l'objet rayon ; donc par exemple, si dans le futur l'objet centre est effacé, il faudra que l'objet cercle le soit aussi).

La méthode *enregistre* doit obligatoirement être appelée tout à fait à la fin de *__init__*, sauf dans de rarissimes exceptions qui correspondent à des cas de figure très précis et très avancés[6].

Elle est facultative pour les objets génériques[7], mais indispensable pour tous les autres.

b) Etude d'un exemple plus consistant : l'objet Droite

Voici le code associé à l'objet Droite (il peut varier un peu d'une version à l'autre de geolib, mais l'architecture reste globalement la même).

```
class Droite(Ligne):
    """Une droite.

    Une droite definie par deux points"""

    def __init__(self, point1, point2, **kw):
        Ligne.__init__(self, point1 = point1, point2 = point2)
        self.style(**param.droites)
        self.enregistre(Point_generique, Point_generique)
```

6 Redéfinition d'un objet à la volée, ou création, lors de l'initialisation, d'un autre objet, qui dépend de l'objet en cours d'initialisation. Voir l'exemple des classes *Triangle* et *Polygone*.

7 Ou pour les objets qui n'auraient pas de méthode *__init__*, bien sûr.

```
    def _conditions_existence(self):
        return [sum(abs(self.point1.coo() - self.point2.coo())) > 1e-14]


    def x_y(self, nbr = 0, flag = 1):
        #si flag = 1, donne le point (x,y) sur la droite en fonction de x.
        #si flag = 0, donne le point (x,y) sur la droite en fonction de y.
        [x1, y1], [x2, y2] = self.point1.coordonnees(),
self.point2.coordonnees()

        # si la pente est presque verticale et que flag = 1, ou presque
horizontale et que flag = 0, le calcul peut exploser... l'option flag = 2
adapte flag a la pente.
        if flag == 2:  flag = self.pente(1)

        # dans le cas d'une droite horizontale ou verticale, flag est
ignore et prend la seule valeur correcte :
        flag = not (y1 - y2) and 1 or ((x1 - x2) and flag or 0)

        if flag:  return (nbr,((y1-y2)*(nbr-x1)+(x1-x2)*y1)/(x1-x2))
        else:  return (((x1-x2)*(nbr-y1)+(y1-y2)*x1)/(y1-y2),nbr)


    def pente(self, flag = 0):
         # si flag = 1, retourne la "tendance" de la pente (verticale : 0,
horizontale : 1)
        [x1, y1], [x2, y2] = self.point1.coordonnees(),
self.point2.coordonnees()
        if flag: return (abs(x2 - x1) > abs(y2 - y1)) and 1 or 0
        try: return (y2 - y1)/(x2 - x1)
        except ZeroDivisionError: return None


    def _affiche(self):
        if not self.__representation__:
            self.__representation__ = [self.__canvas__().plot()]
        x1, y1 = self.point1.coordonnees()
        x2, y2 = self.point2.coordonnees()
        p = self.pente(1)
        fen = self.__feuille__.fenetre() # il faut tracer depuis le bord
de fenetre
        x1, y1 = self.x_y((fen[2], fen[0])[p], p)
        x2, y2 = self.x_y((fen[3], fen[1])[p], p)
        plot = self.__representation__[0]
        plot._x = array([x1, x2]) ; plot._y = array([y1, y2])
        plot._color = self.style("couleur")
        plot._linestyle = self.style("style")
        plot._linewidth = self.style("epaisseur")


    def _distance_inf(self, x, y, d):
        x0, y0 = self.point1.position()
        x1, y1 = self.point2.position()
        A = Point(x0, y0) ; B = Point(x1, y1) ; M = Point(x, y)
        AB = Droite(A, B)
        return Projete_droite(M, AB).distance() < d
```

```
    def __contains__(self, M):
        if not isinstance(M, Point_generique):
            return False
        A = self.point1
        B = self.point2
        u = Vecteur(A, B)
        v = Vecteur(A, M)
        xu, yu = u.coordonnees()
        xv, yv = v.coordonnees()
        return abs(xu*yv-xv*yu) < 1e-14
```

Décortiquons le un peu (nous rentrerons moins dans les détails) :

```
class Droite(Ligne):
```

Exceptionnellement, Ligne ne porte pas le nom de Ligne_generique comme il se devrait (héritage d'un vieux code, mais il est possible que cela change à l'avenir).

```
    """Une droite.

    Une droite definie par deux points"""
```

Là encore, la première ligne est obligatoire, et doit commencer par « Un » ou « Une », sans espace ni saut de ligne entre les guillemets et le déterminant.

La 3ème ligne, par contre, n'est qu'un simple commentaire facultatif[8].

```
    def __init__(self, point1, point2, **kw):
        Ligne.__init__(self, point1 = point1, point2 = point2)
        self.style(**param.droites)
        self.enregistre(Point_generique, Point_generique)
```

8 Pour l'instant en tout cas.
Dans le doute, il vaut mieux respecter la « convention » suivante :
la 3ème ligne explicite succintement l'objet et éventuellement ses arguments.
Il se pourrait qu'elle apparaisse à terme dans une info-bulle par exemple, ou dans la barre de statu.
Les autres lignes sont des commentaires plus avancés, à destination des développeurs par exemple, pour expliquer l'implémentation de l'objet.