Programmation orientée objets avec Python

Cette section n'est qu'un ébauche, permettant de se familiariser avec quelques concepts simples de POO

Contrairement à ce que le titre de ce document laisse entendre, l'objet de ce document n'est pas de traiter les différents aspects de la POO, mais plutôt de détailler les différentes notations (et qques concepts) de POO qui sont utiles dans le cadre de l'utilisation du *binding* **PySide** (utilisé pour réaliser des interfaces graphiques). Il s'agit juste d'un bref survol, rapide et absolument pas exhaustif, de quelques éléments utiles à la compréhension du document Interfaces graphiques avec Python et Qt.

1 Des points et des droites

1.1 Une classe Point

Nous désirons manipuler des points du plan, en utilisant la POO. Pour cela, nous créons une nouvelle **classe**, qui servira de *moule* à la création des nouveaux points :

```
class Point :
    pass
```

Une fois ces lignes entrées, la classe `Point` existe est nous pouvons créer de nouveaux objets ainsi :

```
>>> p=Point()
>>> p
<__main__.Point object at 0x7f0354113a50>
```

Notre classe ne sert encore à rien. Une des premières choses à faire est de recenser :

- les caractéristiques de chaque objet (ce seront les attributs)
- les manipulations que nous leur appliquerons ou qu'ils s'appliqueront (ce seront les méthodes)

Nous devons aussi définir de quelle manière seront créés et initialisés nos objets.

Manifestement, un point du plan est bien représenté par ses coordonnées. Les coordonnées seront les attributs. Une des premières manipulations nécessaires est l'affichage. Enfin, nos objets seront créés en indiquant leurs coordonnées.

1.2 Méthode spéciale d'initialisation

La méthode spéciale `__init__` est appelée à l'initialisation des nouveaux objets (pour ceux qui connaissent la POO, c'est *presque* comme un constructeur, mais pas exactement). Elle prend en paramètre l'objet à initialiser, ainsi que les paramètres qu'on utilise lors de l'initialisation (x et y dans notre cas). En Python, la référence à l'objet est toujours le premier paramètre passé aux méthodes. Par convention, on l'appelle `self` :

```
class Point :
    def __init__(self,x,y) :
        self.x=x
        self.y=y
```

> Dans le code qui précède : `self.x=x` signifie : l'attribut x de l'objet `self` (`self` est l'objet que nous initialisons) doit valoir ce que valait x (le paramètre).

Voyons comment utiliser cet objet :

```
>>> p=Point()
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    p=Point()
TypeError: __init__() takes exactly 3 arguments (1 given)
```

En effet, lors de la création d'un nouvel objet, il faut donner une référence à l'objet (ce qui est fait automatiquement par Python **ainsi que 2 entiers** (ce qui fait bien au total 3 paramètres) :

```
>>> p=Point(2,3)
>>> p
<__main__.Point object at 0x7f8e80113b90>
>>> p.x
2
>>> p.y
3
>>> p.x=42
>>> p.x
42
```

> Nous voyons que les 2 attributs ont bien été initialisés aux valeurs passées lors de la création. Nous voyons aussi que ces valeurs peuvent être changées sans précaution (c'est un point parfois critiqué de Python).

1.3 Une première méthode

Nous allons maintenant rajouter une méthode d'affichage :

```
class Point :
    def __init__(self,x,y) :
        self.x=x
        self.y=y

    def affiche(self) :
        print("Point(",self.x,',',self.y,")")
```

Voyons comment cela fonctionne :

```
>>> p=Point(2,3)
>>> p.affiche()
Point( 2 , 3 )
```

1.4 Une classe Droite

Créons maintenant une autre classe, pour représenter des droites du plan, non verticales. Ces droites sont assimilables aux équations de type $y=ax+b$. Les attributs représentant nos objets droite seront donc a et b :

```
class Droite :
    def __init__(self,a,b) :
        self.a=a
        self.b=b
```

Nous pouvons maintenant rajouter une méthode à nos droites, permettant par exemple de savoir si un point est dessus :

```
class Droite :
    def __init__(self,a,b) :
        self.a=a
        self.b=b
    def appartient(self,p) :
        if p.y==self.a*p.x+self.b : return True
        return False
```

Voyons si cela fonctionne :

```
>>> d=Droite(2,-3)
>>> p1=Point(5,7)
>>> p2=Point(5,8)
>>> d.appartient(p1)
True
>>> d.appartient(p2)
```

```
False
>>> d.appartient(Point(4,5))
True
```

Ajoutons encore une méthode à la classe `Droite` permettant, connaissant l'abcisse d'un point, de renvoyer le point de cette droite ayant cette abcisse (il y en a toujours un puisque les droites ne sont pas verticales) :

```
class Droite
    ...
    def prendPoint(self,x) :
        y=self.a*x+self.b
        return Point(x,y)
```

Testons notre nouvelle methode :

```
>>> d=Droite(2,3)
>>> d=Droite(2,-3)
>>> p=d.prendPoint(5)
>>> p.affiche()
Point( 5 , 7 )
>>> d.prendPoint(5).affiche()
Point( 5 , 7 )
```

Inversement, ajoutons à la classe `Point` une méthode qui renvoie la droite médiatrice (on fournit le second point en paramètre). Comme nous avons besoin des coordonnées du milieu, nous ajoutons la méthode qui fait ce travail :

```
    def milieu(self,p) :
        return Point((self.x+p.x)/2,(self.y+p.y)/2)

    def mediatrice(self,p) :
        if self.y==p.y : return None
         m=self.milieu(p)
         co=(p.x-self.x)/(p.y-self.y)
         a=-co
         b=m.x*co+m.y
         return Droite(a,b)
```

> Il aurait été envisageable de créer une classe `Segment` contenant comme attributs les 2 points extrêmes et d'implanter les méthodes `mediatrice` et `milieu` dans la classe `Segment`.
>
> Faites ce travail en exercice.

1.5 Méthodes magiques pour afficher les objets

Pour avoir un affichage plus simple des objets, nous utilisons une des méthodes **magiques** de Python (ces méthodes sont encadrées par __, et `__init__` en était un example).

Cette méthode, nommée `__repr__` est utilisée automatiquement par Python pour fournir une représentation exécutable de l'objet. Une méthode un peu similaire `__str__` fournit une représentation pour l'utilisateur, pas forcément exécutable. Dans notre cas, si le point de coordonnées 5,3 s'affiche ainsi : `Point(5,3)` nous avons à la fois une représentation lisible et exécutable.

Cet ajout nous permet de supprimer la méthode `affiche` devenue inutile.

```
class Point
   ...
   def __repr__(self) :
       return 'Point('+str(self.x)+','+str(self.y)+')'
```

La méthode `__repr__`, comme `__str__` ne doit pas **afficher** mais doit **retourner une chaîne de caractères**.

Voyons ce que nous pouvons faire à présent :

```
>>> p=Point(5,3)
>>> p
Point(5,3)
>>> print(p)
Point(5,3)
>>> p1=Point(5,3)
>>> p1
Point(5,3)
>>> p2=Point(2,6)
>>> p3=p1.milieu(p2)
>>> p3
Point(3.5,4.5)
>>> d=p1.mediatrice(p3)
>>> d
<__main__.Droite object at 0x7fc12c1139d0>
>>> d.a,d.b
(1.0, -0.5)
```

C'est l'occasion de rajouter une méthode `__repr__` à la classe `Droite` aussi :

```
class Droite :
```

```
    def __repr__(self) :
        return 'Droite('+str(self.a)+','+str(self.b)+')'
```

1.6 Quelques méthodes en plus

Le cinquième postulat d'Euclide nous indique que par un point donné, il ne passe qu'une droite parallèle à une autre droite donnée. C'est l'occasion pour nous d'écrire une nouvelle méthode dans la classe `Droite`.

```
class Droite
    ...
    def parallele(self,p) :
        d=Droite(self.a,self.b)
        d.b=p.y-d.a*p.x
        return d
```

Ajoutez des méthodes permettant de trouver :

- la perpendiculaire à une droite passant par un point
- l'intersection de deux droites.

(Solution)

vous pourrez ensuite vérifier que tout fonctionne correctement :

```
>>> d1=Droite(2,-3)
>>> p1=d1.prendPoint(2)
>>> p1
Point(2,1)
>>> p2=Point(8,-3)
>>> d3=d1.perpendiculaire(p2)
>>> d3
Droite(-0.5,1.0)
>>> d2=d3.perpendiculaire(p2)
>>> d2
Droite(2.0,-19.0)
>>> d1.parallele(p2)
Droite(2,-19)
>>> p3=d1.intersection(d3)
>>> p3
Point(1.6,0.20000000000000018)
>>> m=p2.mediatrice(p3)
>>> m
Droite(2.0,-11.0)
>>> p2.milieu(p3)
Point(4.8,-1.4)
>>> d1.parallele(p2.milieu(p3))
Droite(2,-11.0)
```

1.7 Héritage

En plus des points du plan, nous désirons générer les points colorés. La couleur sera représentée par un triplet (r,v,b). Tout ce que peut faire un point, un point coloré peut le faire : un point coloré est donc un cas particulier de point. C'est dans ce cas précis que nous utilisons l'héritage :

```
class PointCouleur(Point) :
    def __init__(self,x,y,col) :
        super().__init__(x,y)
        self.col=col
```

La classe `PointCouleur` hérite de la classe `Point` car c'est mentionné sur la ligne de déclaration de la classe. Nous écrivons une nouvelle méthode d'initialisation pour `PointCouleur` et appelons à l'intérieur la méthode d'initialisation de la classe mère (grâce à `super()`)

Voyons comment utiliser cette nouvelle classe :

```
>>> p1=PointCouleur(2,3,(255,0,0))
>>> p2=PointCouleur(-5,1,(0,255,0))
>>> p1.mediatrice(p2)
Droite(-3.5,-3.25)
```

La classe `PointCouleur` hérite de la classe `Point` et donc la méthode `mediatrice` est disponible aussi pour les `PointCouleur`.

En revanche l'affichage n'est pas satisfaisant :

```
>>> p1=PointCouleur(2,3,(255,0,0))
>>> p1
Point(2,3)
```

Nous redéfinissons donc la méthode `__repr__` de la classe `PointCouleur` :

```
class PointCouleur(Point) :
    def __init__(self,x,y,col) :
        super().__init__(x,y)
        self.col=col
    def __repr__(self) :
        return
'PointCouleur('+repr(self.x)+','+repr(self.y)+','+repr(self.col)+')'
```

L'affichage est maintenant différent selon le type de point :

```
>>> p1=PointCouleur(2,3,(255,0,0))
>>> p2=PointCouleur(-5,1,(0,255,0))
```

```
>>> p1
PointCouleur(2,3,(255, 0, 0))
>>> p2
PointCouleur(-5,1,(0, 255, 0))
>>> p1.milieu(p2)
Point(-1.5,2.0)
```

2 Copie

Attention, l'affectation ne crée pas une copie d'un objet, mais une nouvelle référence sur l'objet :

```
>>> p=Point(1,2)
>>> p1=p
>>> p1
Point(1,2)
>>> p1.x=42
>>> p
Point(42,2)
```

Modifier `p` ou `p1` c'est la même chose. Si on souhaite **copier** l'objet et en avoir deux versions, il faut utiliser le module `copy` :

```
>>> import copy
>>> p=Point(42,2)
>>> p2=copy.copy(p)
>>> p2
Point(42,2)
>>> p2.x=99999
>>> p2
Point(99999,2)
>>> p
Point(42,2)
```

3 Héritage ou non

Il est important de différencier si un objet fait partie d'un autre, ou si un objet est un cas particulier d'un autre. Lors de la conception de la classe B, et si on dispose déjà de la classe A, on se demande :

- si B est une sorte de A (comme un rectangle est un quadrilatère ou un point coloré est un point), alors B doit probablement être une classe qui hérite de A
- si B possède un A (comme un cercle possède un centre ou un segment deux points extrémités), alors B doit probablement avoir un objet de type A comme attribut

4 Code pour démarrer les exercices

geom.py

```
class Point :
    def milieu(self,p) :
        return Point((self.x+p.x)/2,(self.y+p.y)/2)
    def __repr__(self) :
       return 'Point('+str(self.x)+','+str(self.y)+')'

    def mediatrice(self,p) :
        if self.y==p.y : return None
        m=self.milieu(p)
        co=(p.x-self.x)/(p.y-self.y)
        a=-co
        b=m.x*co+m.y
        return Droite(a,b)
    def __init__(self,x,y) :
        self.x=x
        self.y=y


class Droite :
    def __init__(self,a,b) :
        self.a=a
        self.b=b
    def appartient(self,p) :
        if p.y==self.a*p.x+self.b : return True
        return False
    def prendPoint(self,x) :
        y=self.a*x+self.b
        return Point(x,y)
    def __repr__(self) :
        return 'Droite('+str(self.a)+','+str(self.b)+')'
    def parallele(self,p) :
        d=Droite(self.a,self.b)
        d.b=p.y-d.a*p.x
        return d

class PointCouleur(Point) :
    def __init__(self,x,y,col) :
        super().__init__(x,y)
        self.col=col
    def __repr__(self) :
        return
'PointCouleur('+repr(self.x)+','+repr(self.y)+','+repr(self.col)+')'
```

5 Tips

la méthode `mro()` (*method resolution order*) utilisable sur une classe indique l'ordre dans lequel les classes (celle utilisée et ses ancètres) sont parcourues pour trouver la méthode à exécuter :

```
>>> PointCouleur.mro()
[<class '__main__.PointCouleur'>, <class '__main__.Point'>, <class
'object'>]
```

6 Documentation

Il est important de bien documenter les classes et méthodes grâce aux docstrings. Voici le type de renseignements qu'on obtient ensuite avec `help` :

```
>>> help(poogeom)
Help on module poogeom:

NAME
    poogeom

DESCRIPTION
    Module poogeom : illustration de quelques concepts
    de programmation orientée objets avec des objets géométriques

CLASSES
    builtins.object
        Droite
        Point
            PointCouleur
        Segment
    class Droite(builtins.object)
     |  Classe représentant des droites non verticales du plan
     |
     |  Methods defined here:
     |
     |  __init__(self, a, b)
     |      Une droite non verticale est définie pas son coefficient
     |      directeur et l'ordonnée à l'origine
     |
     |  __repr__(self)
     |
     |  contient(self, p)
     |      Indique si le point p appartien à self
     |
     |  intersection(self, d)
     |      Renvoie le point d'intersection de self et d
     |
     |  parallele(self, p)
```

```
     |      Renvoie la droite parallèle à self passant par p
     |
     |  perpendiculaire(self, p)
     |      Renvoie la droite perpendiculaire à self passant par p
     |
     |  prendPoint(self, x)
     |      Renvoie le point d'abscisse x situé sur self
     |
     |  ----------------------------------------------------------------------
--
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
    class Point(builtins.object)
     |  Classe pour représenter un point du plan
     |
     |  Methods defined here:
     |
     |  __init__(self, x, y)
     |      Un point est défini par ses deux coordonnées
     |
     |  __repr__(self)
     |
     |  mediatrice(self, p)
     |      Renvoie la médiatrice de self et p
     |
     |  milieu(self, p)
     |      Renvoie le point milieur de self et p
     |
     |  ----------------------------------------------------------------------
--
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
    class PointCouleur(Point)
     |  Classe représentant un point du plan d'une certains couleur
     |
     |  Method resolution order:
     |      PointCouleur
     |      Point
     |      builtins.object
     |
     |  Methods defined here:
```

```
     |
     |  __init__(self, x, y, col)
     |
     |  __repr__(self)
     |
     |  ----------------------------------------------------------------------
--
     |  Methods inherited from Point:
     |
     |  mediatrice(self, p)
     |      Renvoie la médiatrice de self et p
     |
     |  milieu(self, p)
     |      Renvoie le point milieur de self et p
     |
     |  ----------------------------------------------------------------------
--
     |  Data descriptors inherited from Point:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
    class Segment(builtins.object)
     |  Classe pour représenter un segment de droite
     |
     |  Methods defined here:
     |
     |  __init__(self, p1, p2)
     |      Un segment est défini par ses deux extrémités
     |
     |  mediatrice(self)
     |      Droite médiatrice d'un segment
     |
     |  milieu(self)
     |      Milieu d'un segment
     |
     |  support(self)
     |      Renvoie la droite support du segment
     |
     |  ----------------------------------------------------------------------
--
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```