7 - Introduction à la POO

On place son rêve si loin, tellement loin, tellement hors des possibilités de la vie,
qu'on ne pourrait rien trouver dans la réalité qui le satisfasse ;
alors, on se fabrique, de toutes pièces, un objet imaginaire !

Roger Martin Du Gard (Jean Barois, p. 117)

Objectifs :

- Définir et utiliser une classe ;
- coder et utiliser un module de classes.

Fichiers à produire : `TP7_1_m.py` `TP7_2_m.py`.

7.1 -Manip

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui sont elles-mêmes des objets, ceux-ci étant constitués à leur tour d'objets plus simples, etc.

7.1.1 - Expérimenter les classes

Classe Point

En vous servant de votre cours, essayez et commentez cet exemple de la classe `Point` :

```
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-
"""Module de la classe Point."""
# fichier : point_m.py
# auteur : Bob Cordeau
# import
from math import hypot

# classe
class Point(object):
    """Definit les points de l'espace 2D."""
    def __init__(self, x=3.0, y=4.0):
        "Constructeur avec valeurs par defaut."
        self.x, self.y = x, y

    def distance(self):
        "Retourne la distance euclidienne du point à l'origine."
        return hypot(self.x, self.y)

    def __str__(self):
        "Modele d'affichage des attributs d'un Point."
        return "[x = {:g}, y = {:g} ; d = {:g}]" .format (self.x,
                self.y, self.distance())

# Auto-test ------------------------------------------------------------
if __name__ == '__main__':
    help(Point)
    p = Point()
    print (p)
```

```
p2 = Point(3.7, 8.1)
print (p2)
```

Bien remarquer l'utilisation de la méthode spéciale `__init__()` qui permet de créer automatiquement des objets `Point` dans un état initial connu.

Dans l'auto-test, l'opération fondamentale est l'***instanciation***, c'est-à-dire la création d'un objet de la classe `Point`. À la ligne suivante, on affiche l'objet `p` grâce à la méthode spéciale `__str__()` qui fournit un modèle à la fonction `print`.

Vous noterez enfin l'utilisation des « docstrings » dans les classes (l'appel à la documentation par `help(Point)` affiche bien ces informations).

Héritage

Comme nous l'avons vu en cours, l'un des principaux atouts de la *POO* (Programmation Orientée Objet) est l'*héritage* (ou la *dérivation*) : c'est la possibilité de se servir d'une classe préexistante pour en créer une nouvelle qui possédera quelques fonctionnalités différentes ou supplémentaires. Ce procédé permet de créer toute une hiérarchie de classes allant du général au particulier.

À partir de l'exemple suivant, instanciez trois objets un `Mammifere`, un `Carnivore` et un `Chien`, et imprimez leurs attributs. Commentez.

```
class Mammifere(object):
    def __init__(self) :
        self.caractMam = "il allaite ses petits ;"

class Carnivore(Mammifere):
    def __init__(self) :
        Mammifere.__init__(self)
        self.caractCar = "il se nourrit de la chair de ses proies ;"

class Chien(Carnivore):
    def __init__(self) :
        Carnivore.__init__(self)
        self.caractCh = "son cri se nomme aboiement ;"
```

Bien noter l'appel, dans une sous-classe, du `__init__()` de la classe parente, afin que celle-ci procède aussi à ses initialisations.

7.1.2 - Un module de classes

Tout comme dans le cas déjà vu des fonctions, on peut faire un module de classes. Un exemple est donné ci-dessus avec la classe `Point`.

7.2 -Programmation

7.2.1 - Classes parallèlépipède et cube

Module de classe TP7_1_m

Sur le modèle de la classe `Point`, écrivez un module « auto-testable » ***TP7_1_m***.

Ce module doit implémenter la classe `Parallelepipede` :

- prévoir un constructeur avec des valeurs par défaut pour les longueur, largeur et hauteur ;
- affectez le nom de cette figure (`'parallèlépipède'`) comme attribut d'instance ;

- définissez la méthode `volume()` qui retourne le volume d'un parallélépipède rectangle ;
- munir la classe d'une méthode de représentation permettant que l'affichage d'une instance de `Parallelepipede` produise :

```
Le parallèlépipède de côtés 12, 8 et 10 a un volume de 960.
```

Testez votre module, puis ajoutez une classe `Cube` qui hérite de la classe `Parallelepipede`. Son constructeur aura une valeur d'arête par défaut et, dans sa définition :

- appellera le constructeur de la classe `Parallelepipede` ;
- surchargera son propre nom (attribut d'instance) : `'cube'`.

L'affichage d'un objet `Cube` de côté 10 doit produire le message :

```
Le cube de côtés 10, 10 et 10 a un volume de 1000.
```

7.2.2 - Classe Fraction

Module de classe TP7_2_m

Implémentez la classe `Fraction` représentant les fractions rationnelles réduites.

Cette classe possède :

Un **constructeur** qui possède les propriétés suivantes :

- gestion de valeurs par défaut : `numerateur` et `denominateur` initialisés à 1,
- interdiction d'instancier une fraction ayant un dénominateur nul (dans ce cas on lèvera une exception `ValueError` contenant un message indiquant l'erreur),
- définition de trois attributs d'instance :
  - **num** la valeur absolue du numérateur ;
  - **den** la valeur absolue du dénominateur ;
  - **signe** le signe de la fraction[39], qui vaudra `1` si numérateur * dénominateur ≥0, `-1` sinon.
- enfin rendre une fraction simplifiée, si c'est possible, grâce à l'appel à la méthode `simpliFrac()`.

Une **méthode spéciale `__str__()`**, permettant d'afficher par exemple :

```
>>> print (Fraction(6, -9))
(-2,3)
```

Les méthodes de **surcharge d'opérateurs** :

- `__neg__(self)` retourne la Fraction opposée (`- f`)
- `__add__(self,other)` retourne la Fraction somme (`f + g`)
- `__sub__(self,other)` retourne la Fraction différence (`f - g`)
- `__mul__(self,other)` retourne la Fraction produit (`f * g`)
- `__truediv__(self,other)` retourne la Fraction rapport (`f / g`)
- `__floordiv__(self,other)` retourne la Fraction rapport (`f // g`)

Une **méthode `simpliFrac()`** faisant appel à une méthode `PGCD()` détaillée plus bas.

Le rôle de cette procédure est de changer les attributs d'instance de la fraction sur laquelle elle travaille afin que celle-ci soit réduite.

---

39 Mettre le signe à part permet de gérer élégamment la représentation des fractions du type $\frac{a}{-b}$.

Voici les tests que vous devez réaliser et commenter :

```
# Auto-test ------------------------------------------------------------
if __name__ == '__main__':
    f = Fraction()
    print ("Constructeur par défaut : f =", f)
    #~ print "\nFraction interdite :" # à tester une fois !
    #~ frac = Fraction(2, 0)
    g = Fraction(10, 30)
    print ("\ng =", g)
    print ("\topposée :", -g)
    print ("\tg.num =", g.num)
    print ("\tg.den =", g.den)
    h = Fraction(7, -14)
    print ("\nh =", h)
    print ("\ng + h =", g+h)
    print ("g - h =", g-h)
    print ("g * h =", g*h)
    print ("g / h =", g/h)
```

Algorithme d'Euclide - calcul du PGCD (Plus Grand Commun Diviseur)

Pour vous aider à simplifier vos fractions, voici un codage de l'algorithme d'Euclide, suivi d'explications.

```
class Fraction(object) :
    …
    def PGCD(self) :
        "Algorithme d'Euclide"
        x,y = self.num,self.den
        while y :
            x,y = y, x%y
        return x
```

Si *a* et *b* sont deux nombres entiers naturels (*non nuls*) qui ne sont pas premiers entre eux, l'ensemble de leurs diviseurs communs comporte d'autres éléments que le nombre 1. Cet ensemble admet un plus grand élément appelé le PGCD.

Il y a deux façons de procéder pour rechercher le PGCD de deux nombres.

Par exemple pour 360 et 108 :

1. par l'utilisation de la décomposition en produits de facteurs premiers :

   $$360 = 2^3.3^2.5$$
   $$108 = 2^2.3^3$$

   Le PGCD est obtenu en faisant le produit de tous les facteurs qui figurent à la fois dans les deux décompositions avec leur exposant le plus bas :

   $$pgcd = 2^2.3^2 = 36$$

2. Par l'algorithme d'Euclide. On prend le reste précédent le reste nul :

   $360/108=3$, **reste 36**

   $108/36=3$, reste 0

On rappelle la propriété suivante :

$$PGCD(x,y).PPCM(x,y)=x.y$$