fiche 1

Notion de classes

Version avec preuves.

Le but du TD est de se familiariser avec la notion de classe.

Exercice 1 : Créer la classe nommée `Exemple` qui possède deux attributs :

- Un entier n.
- Une chaine de caractère nommé `nom`.

1. Écrire les méthodes accesseurs.
2. Ecrire la méthode mutateur pour l'entier.
3. Ecrire la méthode d'affichage sous la forme :
   Son nom est ... , sa valeur est ...

Correction

```
class Exemple :
    def __init__(self , n , nom ) :
        """ methode constructeur """
        self.n = n
        self.nom = nom

    def get_n (self) :
        """ accesseur ( getter )  de n """
        return self.n

    def get_nom(self) :
        """ renvoie le nom """
        return self.nom

    def set_n (self , n_new ) :
        """ change la valeur de n """
        self.n = n_new

    def __str__ (self) :
        """ renvoie la chaine de caractere """
        chaine = "Son␣nom␣est␣" + self.nom +\
                 "␣,␣et␣sa␣valeur␣est␣" + str(self.n)
        return chaine
```

Exercice 2 : Le but de l'exercice est de créer une classe `Ensemble` qui gèrera un ensemble d'entiers sans doublon.
Les éléments de l'ensemble seront stockés dans une liste (attribut `liste` qui sera vide par défaut.)

1. Écrire la méthode `nb_element` qui renvoie le nombre d'élément de l'ensemble.
2. Écrire la méthode `cumul` qui prend en argument un élément et qui rajoute l'élément à la liste s'il n'est pas déjà présent.
3. Écrire la méthode `union` qui renvoie un nouvel ensemble qui est l'union de l'ensemble et de l'argument.
4. Écrire la méthode `inter` qui renvoie un nouvel ensemble qui est l'intersection de l'ensemble et de l'argument.

Correction

```
class Ensemble :
    def __init__(self , liste = [] ) :
        """ Methode constructeur """
        self.liste = liste

    def nb_element (self) :
        """ renvoie le nb d'element de l'ensemble"""
        return len(self.liste)

    def cumul ( self , element ) :
        """ cumule l'element dans la liste s'il n'est pas present"""
        if not element in self.liste :
            self.liste.append(element)

    def union (self, ens_2):
        """ union de deux ensemble """
        ens_3 = Ensemble(self.liste)
        for i in ens_2.liste :
            ens_3.cumul(i)
        return ens_3

    def __add__(self, ens_2):
        """ union de deux ensemble """
        ens_3 = Ensemble(self.liste)
        for i in ens_2.liste :
            ens_3.cumul(i)
        return ens_3
```

Exercice 3 : Créer la classe `Vecteur` qui a pour but de gérer les vecteurs de l'espace. On donnera ainsi trois attributs qui sont l'abscisse, l'ordonnée et la cote.

1. Écrire la méthode renvoyant la norme du vecteur.
2. Écrire la surcharge de l'opérateur +, qui renvoie la somme de deux vecteurs.
3. Écrire la méthode `mult_scal` qui renvoie le vecteur par multiplication du scalaire en argument.
4. Écrire la surcharge de l'opérateur *, qui renvoie le produit scalaire de deux vecteurs.

Correction

```
from math import *

class Vecteur :
    def __init__(self, x,y,z) :
        """ construction du vecteur """
        self.x = x
        self.y = y
        self.z = z

    def norme(self) :
        """ Retourne la norme du vecteur """
        return sqrt(self.x **2 + self.y **2 +self.z **2 )

    def __add__(self,vect_2) :
        """ surcharge de +. retourne la somme de deux vecteurs """
        vect_3 = Vecteur(self.x + vect_2.x , \
                         self.y + vect_2.y , \
                          self.z + vect_2.z )
        return vect_3

    def mult_scal(self, alpha ) :
        """ multiplication par alpha """
        vect_2 = Vecteur(self.x * alpha , \
                         self.y * alpha , \
                          self.z * alpha )
        return vect_2

    def __mul__(self,vect_2) :
        """ surcharge de *. retourne le produit scalaire de deux vecteurs """
        ps = self.x * vect_2.x + self.y * vect_2.y +self.z * vect_2.z
        return ps

    def __str__(self) :
        affiche_x = "␣|␣" + str(self.x) + "␣|␣\n␣"
        affiche_y = "␣|␣" + str(self.y) + "␣|␣\n␣"
        affiche_z = "␣|␣" + str(self.z) + "␣|␣"
        return affiche_x +affiche_y + affiche_z
```

Exercice 4 : Créer la classe `Fraction` qui a pour but de gérer le calcul fractionnaire.
Les arguments seront naturellement le numérateur et le dénominateur.

1. Écrire la méthode `simplifie` qui simplifie la fraction. (On écrira au préalable la méthode `pgcd` qui renvoie le pgcd du numérateur et du dénominateur.)
2. Écrire la surcharge de l'opérateur -, qui renvoie l'opposé d'une fraction.
3. Écrire la surcharge de l'opérateur +, qui renvoie la somme de deux fractions.
4. Écrire la surcharge de l'opérateur -, qui renvoie la différence de deux fractions.
5. Écrire la surcharge de l'opérateur *, qui renvoie le produit de deux fractions.
6. Écrire la surcharge de l'opérateur /, qui renvoie la division de deux fractions.

Correction

```
class Fraction :
    def __init__(self,numerateur,denominateur) :
        """ methode constructeur """
        self.num  =numerateur
        self.den = denominateur

    def pgcd(self) :
        """ retourne le pgcd de num et dem """
        a = self.num
        b=  self.den
        while not b == 0 :
            r = a%b
            a = b
            b=r
        return a


    def simplifie(self) :
        """ simplifie le num et de den """
        p = self.pgcd()
        self.num = self.num // p
        self.den = self.den // p

    def __neg__(self) :
        """ renvoie l'oppose """
        f = Fraction( - self.num , self.den )
        f.simplifie()
        return f

    def __add__(self,f_2) :
        """ surcharge de l'operateur +
            entree : objet et une fration f_2
            sortie : fraction, somme des deux """
        f_3 = Fraction( self.num * f_2.den + self.den * f_2.num ,\
                        self.den * f_2.den )
```

```
        f_3.simplifie()
        return f_3

    def __str__(self) :
        mot = str(self.num) + "␣\n␣" + "--␣\n" + str(self.den)
        return mot

    def __sub__(self,f_2) :
        """ surcharge de l'operateur -
            entree : objet et une fration f_2
            sortie : fraction, difference des deux """
        return  self + (- f_2)

    def __mul__(self,f_2) :
        """ surcharge de l'operateur *
            entree : objet et une fration f_2
            sortie : fraction, produit des deux """
        f_3 = Fraction( self.num * f_2.num ,\
                        self.den * f_2.den )
        f_3.simplifie()
        return f_3

    def __div__(self,f_2) :
        """ surcharge de l'operateur /
            entree : objet et une fration f_2
            sortie : fraction, produit des deux """
        f_3 = Fraction( self.num * f_2.den ,\
                        self.den * f_2.num )
        f_3.simplifie()
        return f_3
```