M2207 - Introduction à la programmation orientée objet : TD-TP1

9 février 2017

1 Compte bancaire

Un compte bancaire est défini par les attributs suivants : nom de la banque, nom du titulaire et solde du compte. Deux opérations de base sont possibles sur un compte : `déposer` et `retirer` de l'argent. Aucune facilité de caisse n'est autorisée (i.e. le solde d'un compte ne peut pas être négatif). La classe `CompteBancaire` modélise un compte.

1. Comment faut il déclarer les trois attributs d'un compte : des attributs de classe ou des attributs d'instance ?

   Les trois attributs sont des attributs d'instance .

2. Donner la méthode constructeur de cette classe. La méthode doit vérifier que les valeurs fournies en entrée sont recevables. Dans le cas contraire aucun objet ne devrais être créé.

```
def __init__(self,banque,titulaire,solde):
        assert isinstance(banque,str) and len(banque)>0
        assert isinstance(titulaire,str) and len(titulaire)
            >0
        assert isinstance(float(solde),float) and solde>=0

        self.banque=banque
        self.titulaire=titulaire
        self.solde=solde
```

3. Définir une méthode `deposer` qui permet de déposer une somme positive d'argent.

```
def deposer(montant):
        assert isinstance(montant,int) and montant >=0
        self.solde = self.solde+montant
```

4. Définir une méthode `retirer` qui permet de retirer une somme positive si le solde du compte le permet.

```
def retirer(montant):
        assert isinstance(montant,int) and montant <=self.
            solde and montant>=0
        this.solde=this.solde-montant
```

5 Définir une méthode qui permet d'afficher un l'état d'un compte sous le forme suivant :

```
Titulaire : nom du titulaire
Banque : nom de la banque
Solde    : solde du compte
```

```
def __str__(self):
        res="Titualire \t: %s \n"%self.titulaire
        res=res+"Banque \t\t: %s\n"%self.banque
        res=res+"Solde \t\t: %f \n "%self.solde
        return res
```

6 On considère une liste `listComptes` de comptes bancaires. Comment faire pour permettre de trier cette liste facilement en fonction de l'ordre alphabétique des noms des titulaires ? (Indice pour trier une liste n utilise la méthode *sort*() de la classe *list*.

```
def __cmp__(self,other):
        assert isinstance(other,CompteBancaire)

        if self.titulaire < other.titulaire:
            return -1
        elif self.titulaire==other.titulaire:
            return 0
        return 1
```

7 Soit *c* un objet instancié à partir de la classe `CompteBancaire`. Exécuter les instructions suivantes :

```
c.solde = -300
print c
setattr(c,solde,-1000)
print c
```

Que remarquez vous ? Comment résoudre le problème constaté ?
(*Indice : Penser à redéfinir la méthode* `__setattr__`)

On arrive à modifier l'attribue sans tenir comptes des contraintes. On peut imposer la vérification des contraintes en modifiant la méthode `__setattr__`

```
def __setattr__(self,nom,value):
        if nom in ["banque","titulaire"]:
            if isinstance(value,str) and len(value)>0:
                self.__dict__[nom]=value
        elif nom=="solde":
            if isinstance(float(value),float) and value
                >=0:
                self.__dict__[nom]=value
        else:
            raise ValueException
```

2 Dates

Une date est définie par **trois entiers** `y, m, d` qui représentent respectivement l'année, le mois et le jour. Proposer une simple classe `Date` qui ne permet d'instancier que de **dates valides**. Une date est valide si y est positif et $m \in [1, 12]$ et $d \in [1, nbJour(m)]$ où $nbJour(m)$ est le nombre de jours dans le mois m. Une année y est bissextile (i.e. le mois de février a 29 jours) si :

- y est divisible par 4 et non divisible par 100, ou
- y est divisible par 400.

1 Dans la classe `Date`, définir une **méthode de classe** `estBissextile` qui permet de tester si une année est bissextile ou non.

```
def estBissextile(year):
        return (((year % 4 ==0 )and (year%100 !=0)) or (
            year %400==0))
    estBissextile=staticmethod(estBissextile)
```

2 Donner la méthode constructeur de la classe `Date`

```
def __init__(self,day,month,year):
        assert isinstance(day,int)  and day in range(1,32)
        assert isinstance(month,int) and month in range
            (1,13)
        assert isinstance(year,int) and  year >0

        dayInMonth=[31,28,31,30,31,30,31,31,30,31,30,31]
        bis=Date.estBissextile(year)
```

```
8           if bis :
9               dayInMonth[1]=29
10          if day <=dayInMonth[month-1]:
11              self.day=day
12              self.month=month
13              self.year=year
14          else:
15              raise ValueError
```

3 Définir une méthode qui étant donnée une date d permet de comparer la date locale avec d pour déterminer si la date est antérieur ou postérieur à d

```
1  def __cmp__(self,other):
2          assert isinstance(other,Date)
3
4          if (self.year == other.year):
5              if (self.month==other.month):
6                  if (self.day==other.day):
7                      return 0
8                  else:
9                      if (self.day<other.day):
10                         return -1
11                     else:
12                         return 1
13             else:
14                 if self.month<other.month:
15                     return -1
16                 else:
17                     return 1
18         else:
19             if self.year < other.year:
20                 return -1
21             else:
22                 return 1
```

4 Définir une méthode qui permet d'afficher la date soue la forme : `DD - nom du mois - Année` (Ex. pour une date $d = 2, m = 2, y = 2017$ on affiche 02 Février 2017)

```
1  def __str__(self):
2          months_names={1:"Jan",2:"Fev",3:"Mar",4:"Avr",5:"
               Mai",6:"Jun",7:"Jul",
3                        8:"Aou",9:"Sep",10:"Oct",11:"Nov",12:"
                             Dec"}
4          res="%s - %s - %s"% (self.day,months_names[int(self
               .month)],self.year)
```

```
5         return res
```

3 Calcul de moyenne

Définir une classe `Etudiant` Un considère le cas d'une groupe d'étudiants qui suivent une formation en réseaux et télécommunications. Le tableau 1 donne coefficients appliqués pour chaque module dans la formation et le code pour désigner chaque module.

TABLE 1 – Tableau de coefficients

Module	code	Coefficient
Réseaux	res	3
Administration système	admin	1.5
Base de données	bd	1.5
Web Dynamique	web	1.5
Transmission numérique	tn	3
Anglais	ang	3
Communication	com	2
Programmation	prog	1.5

Chaque étudiant doit avoir une note dans chaque module de la formation. En plus des notes un étudiant est caractérisé par les attributs suivants : Numéro d'étudiant, nom et prénom.

1. On veut développer une classe `Etudiant` qui modélise un étudiant. Donner les attributs nécessaires à a définition d'une telle classe en précisant la nature de chaque attribut (attribut de classe, attribut d'instance).

```
1 class Etudiant:
2     modules ={"res":3; "admin":1.5; "bd":1.5; "web":1.5; "
          tn":3;"ang":3;"com":2;"prog":1.5}
3     cof=[1.5,2,3,4]
```

2. Proposer une méthode constructeur de la classe `Etudiant`.

```
1 def __init__(self,num,prenom,nom):
2         assert isinstance(num,str) and len(num)>0
3         assert isinstance(num,prenom) and len(prenom)>0
```

```
4           assert isinstance(nom,str) and len(nom)>0
5
6           self.num=num
7           self.nom=nom
8           self.prenom=prenom
9           self.notes=dict()
10          for k in Etudiant.modules.keys():
11              self.notes[k]=float(0)
```

3 Proposer une méthode `addNote` qui permet d'affecter la note d'un étudiant pour un module donnée.

```
1 def addNote(self,module,note):
2           assert module in Etudiant.modules.keys()
3           assert isinstance(float(note),float) and float(note
                )>=0 and float(note)<=20
4
5           self.notes[module]=note
```

4 Propose une méthode `setCoeff` qui permet de changer le coefficient d'un module. Le coefficient d'un module est obligatoirement une valeur parmi les valeurs suivantes : $\{1.5, 2, 3, 4\}$

```
1 def setCoeff(module,coeff):
2           assert module in Etudiant.modules.keys()
3           assert coeff in Etudiant.cof
4
5           Etudiant.modules[module]=coeff
6 setCoeff=staticmethod(setcoeff)
```

5 Proposer une méthode `moyenne` qui calcule la moyenne des notes d'un étudiant.

```
1 def moyenne(self):
2           res=0.0
3           for m in Etudiant.modules.keys():
4               res=res+(self.notes[m]*Etudiant.modules[m])
5           res=res/sum(Etudiant.modules.values())
6           return res
```

6 Soit $\mathbb{L}$ une liste d'étudiants. On veut afficher les noms d'étudiants dans cette liste selon l'ordre décroissant des moyennes. Les étudiants ayant la même moyenne seront triés en ordre alphabétique de leurs noms.

Comment faire pour faciliter le tri de cette liste ?

```
def __cmp__(self,other):
        assert isinstance(other,Etudiant)

        if slef.moyenne() < other.moyenne :
            return -1
        elif self.moyenne== other.moyenne:
            return cmp(self.nom,other.nom)
        else:
            return 1
```

4 Arithmétiques des fractions

On considère l'ensemble de nombres rationnels $\mathbb{Q}$ défini comme suit :

$$\mathbb{Q} = \{\frac{n}{d} : (n,d) \in \mathbb{Z} \times (\mathbb{Z}\{0\})\}$$

où $\mathbb{Z}$ est l'ensemble d'entiers relatifs. On souhaite définir une classe `Fraction` qui modélise les éléments de $\mathbb{Q}$.

1 On propose de décrire l'état d'une fraction par trois attributs : **les deux valeurs absolues** de n et d et **le signe** de la fraction. Donner la méthode constructeur de la classe `Fraction`. Il faut s'assurer qu'une fraction est toujours conforme à sa définition.

```
class Fraction:

    def __init__(self,n,d):
        assert isinstance(n,int)
        assert (isinstance(d,int) and d!=0)

        self.n=abs(n)  # numerateur
        self.d=abs(d)  # denominateur
        if (n*d)>=0:   # le signe
            self.s=1
        else:
            self.s=-1

    def __setattr__(self,nom,value):
        if nom not in ["n","d","s"]:
            raise ValueError
        if nom=="n":
            assert isinstance(value,int)
            self.__dict__[nom]=abs(value)
        if nom=="d":
            assert (isinstance(value,int) and value!=0)
```

```
22                 self.__dict__[nom]=abs(value)
23             if nom=="s":
24                 assert value in (-1,1)
25                 self.__dict__[nom]=value
```

2 Redéfinir la méthode `__str__` afin de pouvoir afficher une fraction sous la forme `n:d`.

```
1  def __str__(self):
2          res=""
3          if self.s==-1:
4              res="-"
5          res=res+str(self.n)+":"+str(self.d)
6          return res
```

3 Proposer une méthode `inverse` qui permet d'inverser une fraction. Par exemple l'inverse de $\frac{5}{7}$ est $\frac{7}{5}$

```
1  def inverse(self):
2          if self.n !=0:
3              self.n,self.d=self.d,self.n
4          else:
5              raise ValueError
```

4 Proposer une méthode `inverseSigne` qui permet d'inverser le signe d'une fraction.

```
1  def inverseSigne(self):
2          self.s=self.s * -1
```

5 Proposer une méthode `add` qui permet d'additionner deux fractions.

```
1  def add(self,other):
2          assert isinstance(other,Fraction)
3
4          num=(self.s*self.n*other.d)+(other.s*other.n*self.d
               )
5          den=self.d*other.d
6          return Fraction(num,den)
```

6 Proposer une méthode `sub` qui permet de faire la soustraction de deux fractions.

```
def sub(self,other):
        assert isinstance(other,Fraction)

        aux=Fraction(other.n,other.d)
        aux.s=other.s
        aux.inverseSigne()
        return self.add(aux)
```

7 Proposer une méthode `mul` qui permet de faire la multiplication de deux fractions.

```
def mul(self,other):
        assert isinstance(other,Fraction)

        res=Fraction(self.n*other.n,self.d*other.d)
        res.s=self.s*other.s
        return res
```

8 Proposer une méthode `div` qui permet de faire la division de deux fractions.

```
def div(self,other):
        assert isinstance(other,Fraction)

        aux=Fraction(other.n,other.d)
        aux.s=other.s
        aux.inverse()

        return(self.mul(aux))
```

9 Proposer une méthode `simplifier` qui permet de simplifier une fraction. Exemple : $\frac{9}{27}$ peut être simplifié à $\frac{1}{3}$, $\frac{16}{20}$ peut être simplifié à $\frac{4}{5}$. (Indice : penser à calculer le PGCD de n et d.)

```
def pgcd(n,m):
        assert isinstance(n,int) and isinstance(m,int) and
            n>0 and m >0
        # Euclide algorithm
        if n<m:
            n,m=m,n
        while m >0:
```

```
            n,m=m,n%m
        return n
    pgcd=staticmethod(pgcd)

    def simplify(self):
        diviseur=Fraction.pgcd(self.n,self.d)
        if diviseur !=1:
            self.n=self.n/diviseur
            self.d=self.d/diviseur
```

10 Proposer une méthode `irreductible` qui renvoie : `True` si la fraction ne peut pas être simplifiée, `False` sinon.

```
def irreductible(self):
        return (Fraction.pgcd(self.n,self.d)==1)
```

5 Calcul de percentiles

Soit X une liste de n mesures observées sur une période de temps. On désigne pr x_i la mesure prise au temps i. Afin de caractériser un ensemble de mesure, on fait souvent appel à des fonctions statistiques. On définit ci-après quelques fonctions statistiques de base :

- La moyenne (moy) : $moy(X) = \bar{X} = \frac{1}{n}\sum_{i=1}^{i=n} x_i$
- La variance (var) : $var(X) = V(X) = \frac{1}{n}\sum_{i=1}^{i=n}(x_i - \bar{X})^2$
- La fréquence empirique de dépassement (EDF) :

$$EDF(x, X) = \frac{1}{n}||\{x_i \in X : x_i \leq x\}||$$

- I^{me} percentile : est donné par la plus petite valeur x_i de X tel que $EDF(x_i) \geq I\%$

Illustration : Soit $X = \{-2, 7, 7, 4, 18, -5\}$ nous avons :

x	-5	-2	4	7	18
$EDF(x)$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{5}{6}$	1

Le 50^{me} percentile est donc 4 et le 25^{me} est -2.

Proposer une classe `Stat` qui permet de fournir les fonctions statistiques données ci-haut. Comment faut il définir les différentes méthodes de cette classe ?

```
class Stat:

    def validList(l):
        assert isinstance(l,list) and len(l)>0
        assert all([isinstance(float(x),float) for x in l])
        return True
    validList=staticmethod(validList)

    def moy(l):
        assert Stat.validList(l)
        return sum(l)/len(l)
    moy=staticmethod(moy)

    def var(l):
        assert Stat.validList(l)
        m=Stat.moy(l)
        aux=0.0
        for x in l:
            aux=aux+(x-m)*(x-m)
        return aux/len(l)
    var=staticmethod(var)

    def edf(v,l):
        assert Stat.validList(l)
        assert isinstance(float(v),float)

        v=float(v)
        l.sort()
        aux=[x <=v for x in l]
        nb=aux.count(True)

        return float(nb)/len(l)
    edf=staticmethod(edf)

    def percentile(i,l):
        assert Stat.validList(l)
        assert i in range(0,101)

        l.sort()
        i=0
        while Stat.edf(l[i],l) <=float(i)/100:
            i=i+1
        return l[i]
    percentile=staticmethod(percentile)
```

6 Définition de constantes

Le langage *Python* ne fournit pas une primitive de définition de constantes (des variables qui ne peuvent pas changer de valeurs). proposer une classe

qui permet de définir des constantes.

Indice : penser à la méthode `__setattr__`

```
class CONST(object):
    A = 1
    B= 2
    def __setattr__(self, *_):
        pass
```