```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#
# fichier: expression.py
#    date: 2011/05/04
#
# (tous les symboles non internationaux sont volontairement omis)
#

import string

from fraction import *

class expression(object):

  def __init__(self, source ="", valide =True):
    """ constructeur """
    s = ""
    for x in source:
      if (x in string.digits) or (x in string.letters): s += x
      if x in "{[": x = "("
      if x in "}]": x = ")"
      if x in "+-*^:/()": s += x

    self.__valide = valide and (len(s) > 0)
    if self.__valide:
      self.__source = "{" + s + "}"
      self.__n = len(self.__source)
      self.__i = 0
      self.__err = 0
      self.__val = fraction()
      self.evaluer()
    else:
      self.__err = 3 # syntaxe non conforme
      self.__val = fraction_nulle_erronee()



  def __str__(self):
    """ representation en chaine de caracteres """
    if self.__err == 0:
      return str(self.__val)
    else:
      return "erreur: " + self.message_erreur()



  def valeur(self):
    """ fraction egale a l'expression evaluee """
    if self.__valide:
      return self.__val
    else:
      return fraction_nulle_erronee()



  def est_valide(self):
    """ indique l'etat de validite """
    return self.__valide



  def valider(self):
    """ valider l'objet """
    self.__valide = True
```

```
  def invalider(self):
    """ invalider l'objet """
    self.__valide = False



  def get_source(self):
    """ donne l'expression initiale """
    return self.__source[1:-1]



  def rapporter_erreur(self, e):
    """ relever une erreur (si aucune erreur precedente) """
    if self.__err == 0:
      self.__err = e



  def erreur_existe(self):
    """ indique si une erreur s'est produite (derniere evaluation) """
    if self.__err == 0:
      return False
    else:
      return True



  def aucune_erreur(self):
    """ indique s'il n'y a pas eu d'erreur (derniere evaluation) """
    return not self.erreur_existe()



  def message_erreur(self):
    """ indique le type d'erreur """
    if self.__err == 0:
      return "aucune erreur"

    if self.__err == 1:
      return "il manque une accolade ouvrante \"{\""

    if self.__err == 2:
      return "il manque une accolade fermante \"}\""

    if self.__err == 3:
      return "la syntaxe de l'expression n'est pas conforme"

    if self.__err == 4:
      return "tentative de diviser par 0"

    if self.__err == 5:
      return "exposant invalide"

    if self.__err == 6:
      return "il manque un nombre ou une parenthese ouvrante \"(\""

    if self.__err == 7:
      return "il manque une parenthese fermante \")\""

    return "erreur de type inconnu"



  def evaluer(self):
    """ effectuer l'evaluation de l'expression """
    self.__ch = self.suivant()
```

```
    if self.__ch == '{':
      self.__ch = self.suivant()
      self.__val = self.expr()
      self.__ch = self.suivant()
      if self.__ch != '}':
        self.rapporter_erreur(2) # il manque une "}"
    else:
      self.rapporter_erreur(1) # il manque une "{"



  def suivant(self):
    """ lecture du caractere suivant """
    while self.__i < self.__n:
      t = self.__source[self.__i]
      self.__i += 1
      if str.isspace(t):
        continue
      else:
        return t
    return '\0'



  def prochain(self):
    """ observer le prochain caractere """
    while self.__i < self.__n:
      t = self.__source[self.__i]
      if str.isspace(t):
        self.__i += 1
      else:
        return t
    return '\0'



  def prochain_est(self, t):
    """ comparer avec le prochain caractere """
    if self.prochain() == t:
      return True
    else:
      return False



  def expr(self):
    """ expr ::= expr1 '+' expr1 | expr1 '-' expr1 | expr1 """
    if self.erreur_existe():
      return fraction_nulle_erronee()
    t = self.expr1()
    while self.prochain_est('+') or self.prochain_est('-'):
      self.__ch = self.suivant()
      if self.__ch == '+':
        self.__ch = self.suivant()
        t += self.expr1()
      else:
        if self.__ch == '-':
          self.__ch = self.suivant()
          t -= self.expr1()
    return t



  def expr1(self):
    """ expr1 ::= expr2 '*' expr2 | expr2 '/' expr2 | expr2 """
    if self.erreur_existe():
      return fraction_nulle_erronee()
    t = self.expr2()
```

```
    while self.prochain_est('*') or self.prochain_est('/') or self.prochain_est(':'):
      self.__ch = self.suivant()
      if self.__ch == '*':
        self.__ch = self.suivant()
        t *= self.expr2()
      else:
        if self.__ch == '/' or self.__ch == ':':
          self.__ch = self.suivant()
          t /= self.expr2()
          if not t.est_valide():
            self.rapporter_erreur(4) # tentative de diviser par 0
    return t



  def expr2(self):
    """ expr2 ::= '-' expr3 | expr3 """
    if self.erreur_existe():
      return fraction_nulle_erronee()
    oppose = False
    while self.__ch == '-':
      oppose = not oppose
      self.__ch = self.suivant()
    t = self.expr3()
    if oppose:
      return -t
    else:
      return t




  def expr3(self):
    """ expr3 ::= expr4 '^' expr2 | expr4 """
    if self.erreur_existe():
      return fraction_nulle_erronee()
    t = self.expr4()
    if self.prochain_est('^'):
      self.__ch = self.suivant()
      self.__ch = self.suivant()
      k = self.expr2()
      if not k.est_valide():
        self.rapporter_erreur(5) # exposant invalide
      t = t ** k
    return t



  def expr4(self):
    """ expr4 ::= <entier naturel> | <lettre> | '(' expr ')' """
    if self.erreur_existe():
      return fraction_nulle_erronee()
    if str.isdigit(self.__ch):
      t = self.naturel()
      return fraction_depuis_naturel(t)
    if (self.__ch in string.letters):
      t = self.lettre()
      return fraction_depuis_lettre(t)
    if self.__ch == '(':
      self.__ch = self.suivant()
      t = self.expr()
      self.__ch = self.suivant()
      if self.__ch == ')':
        return t
      else:
        self.rapporter_erreur(7) # il manque une ")"
    else:
      self.rapporter_erreur(6) # il manque un nombre ou une "("
```

```
    return fraction_nulle_erronee()



  def naturel(self):
    """ naturel ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')* """
    n = ord(self.__ch) - ord('0')
    x = self.prochain()
    while str.isdigit(x):
      n = n * 10 + int(x)
      self.__ch = self.suivant()
      x = self.prochain()
    return n



  def lettre(self):
    """ lettre ::= 'A' | ... | 'Z' | 'a' | ... | 'z' """
    return str(self.__ch)



  def __add__(self, autre):
    """ somme """
    return self.valeur() + autre.valeur()



  def __sub__(self, autre):
    """ difference """
    return self.valeur() - autre.valeur()



  def __mul__(self, autre):
    """ produit """
    return self.valeur() * autre.valeur()



  def __div__(self, autre):
    """ quotient """
    return self.valeur() / autre.valeur()



  def __pow__(self, autre):
    """ exponentiation """
    return self.valeur() ** autre.valeur()



if __name__ == "__main__":

  a = expression("3 * ( x + 1 ) / (x - 20)")
  print a

  b = expression("x - 1")
  print b

  x = a + b
  print x

  x = a - b
  print x

  x = a * b
  print x
```

```
  x = a / b
  print x

  a = expression("x + 1")
  print a

  n = expression("4")
  print n

  x = a ** n
  print x
```