notes sur le programme "calc3"
==========

(tous les symboles non internationaux sont volontairement omis)

* version 0.4.3 (2011/05/07)

  - dans le module "entier.py", remplacement de la fonction "pgcd_naturel()" par une version iterative (cout d'execution sur les tres grands entiers).

* version 0.4.2 (2011/05/04)

  - premiere version publique.

P. CHAUVIN

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

#
# fichier: calc3.py
#    date: 2011/05/07
#
# (tous les symboles non internationaux sont volontairement omis)
#

from os import *

from expression import *

class calc3(object):
  """ une classe qui represente le calculateur calc3 """

  def __init__(self):
    """ constructeur """
    self.__fichier = file("calculs.txt", "w")
    self.__compteur = 1



  def exemples(self):
    """ tests automatiques """
    print "*** quelques exemples ***\n"

    e = expression("(3 + 4) * 5 ^ (1 + 1) - 7")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne 168

    e = expression("2 ^ 3 ^ 2 ^ 2")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne 24178516392292583494123 52

    e = expression("-(5 - 1) : 5^3 + ( 9 + 1 ) * ( 7 + 2 * 5 )")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne 169.968

    e = expression("3/7 - 2/7 : ( 5 : 14 )")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne -13/35

    e = expression("((a + b)/2)^2 - ((a - b)/2)^2")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne a*b

    e = expression("(x+2/3)^(-3)")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne 27/(27*x^3 + 54*x^2 + 36*x + 8)

    e = expression("(8*(16/10)*10^8)/((4/10)*10^10)")
    print "exemple de calcul :", e.get_source()
    print e, '\n' # donne 0.32



  def notice(self):
    """ notice """
    print
    print "*** calc3: programme de calcul formel (corps de fractions) ***"
    print
    print "               \"Python pour le lyc\\'{e}e et la pr\\'{e}pa\""
    print "                                Alexandre CASAMAYOU-BOUCAU"
    print "                                          Pascal CHAUVIN"
    print "                                       Guillaume CONNAN"
    print
```

```
    print "    (version 0.4.3)"
    print



  def remarque(self):
    """ remarque """
    print "*** remarque importante ***"
    print
    print "    Les calculs sont enregistres dans un fichier (en format texte pur)"
    print "    nomme \"calculs.txt\" dans le repertoire courant d'execution du pro-"
    print "    gramme. Il est donc indispensable d'executer le programme \"calc3\""
    print "    depuis un repertoire ou l'utilisateur possede le droit d'ecriture."
    print



  def lecture(self):
    """ saisie d'une expression """
    invite = "calc3:" + str(self.__compteur) + "> "
    s = str(raw_input(invite))
    self.__fichier.write(invite + s + "\n")
    if len(s) > 0:
      self.__compteur += 1
    return s



  def boucle(self):
    """ boucle d'evaluation """
    print "*** boucle interactive ***"
    print
    print "    Entrer une expression mathematique a evaluer (ou laisser vide pour"
    print "    finir) puis valider."
    print
    entree = self.lecture()
    while len(entree) > 0:
      e = expression(entree)
      print e
      self.__fichier.write(str(e) + "\n\n")
      print
      entree = self.lecture()
    print
    self.__fichier.close()



  def executer(self):
    """ execution du programme """
    self.notice()
    self.exemples()
    self.remarque()
    self.boucle()



if __name__ == "__main__":
  c = calc3()
  c.executer()
```

```
calc3:1> 1/3*(1/x + 1/y + 1/z)
(x * y + x * z + y * z)/(3 * x * y * z)

calc3:2>
```

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

#
# fichier: entier.py
#    date: 2011/05/07
#
# (tous les symboles non internationaux sont volontairement omis)
#

#def pgcd_naturel(a, b):
# """ pgcd de deux nombres entiers naturels """
# if b == 0:
#   return a
# else:
#   return pgcd_naturel(b, a % b)



def pgcd_naturel(a, b):
  """ pgcd (iteratif) de deux nombres entiers naturels """
  if b == 0:
    return a
  r = long(a) % long(b)
  while r > 0:
    a = b
    b = r
    r = long(a) % long(b)
  return b



def pgcd_liste(l):
  """ pgcd d'une liste d'entiers """
  if l == []:
    return 1
  if len(l) == 1:
    return abs(l[0])
  g = pgcd_naturel(abs(l[0]), abs(l[1]))
  if len(l) == 2:
    return g
  for n in l[1:]:
    g = pgcd_naturel(g, abs(n))
  return g



def ppcm_naturel(a, b):
  """ pgcd de deux nombres entiers naturels """
  if a == 0 or b == 0:
    return 0
  else:
    return (a*b) / pgcd_naturel(a, b)



class entier(object):
  """ classe pour un nombre entier """

  def __init__(self, valeur =long(0), valide =True):
    """ constructeur """
    self.__valide = valide
    if self.__valide:
      self.__valeur = long(valeur)
    else:
      self.__valeur = long(0)
```

```
  def __str__(self):
    """ representation en chaine de caracteres """
    if self.__valide:
      return str(self.__valeur)
    else:
      return "(nombre entier invalide)"



  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
    self.__valeur = long(0)



  def valeur(self):
    """ donne la valeur (type natif long) de l'entier """
    return long(self.__valeur)



  def __add__(self, autre):
    """ addition """
    if self.__valide and autre.__valide:
      a = long(self.__valeur)
      b = long(autre.__valeur)
      return entier(a + b)
    else:
      return entier(0, False)



  def __neg__(self):
    """ nombre oppose """
    if self.__valide:
      a = long(self.__valeur)
      return entier(-a)
    else:
      return entier(0, False)



  def __sub__(self, autre):
    """ soustraction (par addition de l'oppose) """
    return self.__add__(autre.__neg__())



  def __mul__(self, autre):
    """ multiplication """
    if self.__valide and autre.__valide:
      a = long(self.__valeur)
      b = long(autre.__valeur)
      return entier(a * b)
```

```
    else:
      return entier(0, False)



  def __div__(self, autre):
    """ division (si le quotient est entier) """
    if not(self.__valide and autre.__valide):
      return entier(0, False)
    if autre.__valeur == 0:
      return entier(0, False)
    a = long(self.__valeur)
    b = long(autre.__valeur)
    if (a % b) != 0:
      return entier(0, False)
    return entier(a / b)



  def __pow__(self, autre):
    """ exponentiation (si exposant entier naturel) """
    if not(self.__valide and autre.__valide):
      return entier(0, False)
    a = self.__valeur
    n = autre.__valeur
    if a == 0:
      if n <= 0:
        return entier(0, False)
      else:
        return entier()
    if n < 0:
      return entier(0, False)
    p = 1
    while n > 0:
      if n % 2 == 1:
        p *= a
      n //= 2
      a *= a
    return entier(p)



  def est_nul(self):
    """ l'entier est-il nul ? """
    if self.__valide:
      return self.__valeur == 0
    else:
      return False



  def est_positif(self):
    """ l'entier est-il positif ? """
    if self.__valide:
      return self.__valeur > 0
    else:
      return False



def pgcd_entier(a, b):
  """ pgcd pour le type entier """
  if isinstance(a, entier) and isinstance(b, entier):
    if a.est_valide() and b.est_valide():
      return entier(pgcd_naturel(abs(a.valeur()), abs(b.valeur())))
  return entier(0, False)
```

```
def ppcm_entier(a, b):
  """ ppcm (positif) pour le type entier """
  if isinstance(a, entier) and isinstance(b, entier):
    if a.est_valide() and b.est_valide():
      return entier(ppcm_naturel(abs(a.valeur()), abs(b.valeur())))
  return entier(0, False)



if __name__ == "__main__":

  a = entier(-5)
  b = entier(-20, False)
  x = pgcd_entier(a, b)

  print x.est_valide()
  print x
  print x.valeur()

  a = entier(-5)
  b = entier(-20)
  x = pgcd_entier(a, b)

  print x.est_valide()
  print x
  print x.valeur()

  x = ppcm_entier(a, b)

  print x.est_valide()
  print x
  print x.valeur()
```

```
#!/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
```

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

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

import string

from polynome import *

class fraction(object):

  def __init__(self, num =polynome_nul(), denom =polynome_un(), valide =True):
    """ constructeur """
    self.__valide = valide and num.est_valide() and denom.est_valide() and (not denom.est_n
ul())
    if self.__valide:
      self.__num = num
      self.__denom = denom
      self.reduire()
    else:
      self.__num = polynome_nul()
      self.__denom = polynome_un()



  def __str__(self):
    """ representation en chaine de caracteres """
    if self.__valide:
      if self.__denom.est_unite():
        return str(self.__num)
      else:
        a = str(self.__num)
        if self.__num.nombre_monomes() == 1:
          while a.startswith("(") and a.endswith(")"):
            a = a[1:-1]
        b = str(self.__denom)
        if self.__denom.nombre_monomes() == 1:
          while b.startswith("(") and b.endswith(")"):
            b = b[1:-1]
        return "(" + a + ")/(" + b + ")"
    else:
      return "(fraction invalide)"



  def simplifier_coefficients(self):
    """ simplification des coefficients du numerateur et du denominateur """
    if self.__valide:
      n = self.__num.pgcd_numerateurs()
      m = self.__denom.pgcd_numerateurs()
      d = pgcd_naturel(n, m)
      r = rationnel(d)

      p = polynome()
      for k in self.__num.liste_decroissante():
        c = k.get_coefficient()
        s = k.get_indeterminee()
        c /= r
        p.ajouter_monome(monome(c, s))

      q = polynome()
      for k in self.__denom.liste_decroissante():
        c = k.get_coefficient()
```

```
      s = k.get_indeterminee()
      c /= r
      q.ajouter_monome(monome(c, s))

    self.__num = p
    self.__denom = q



  def reduire(self):
    """ reduction des coefficients du numerateur et du denominateur """
    if self.__valide:
      n = self.__num.ppcm_denominateurs()
      m = self.__denom.ppcm_denominateurs()
      d = pgcd_naturel(n, m)
      r = rationnel(m*n/d)

      p = polynome()
      for k in self.__num.liste_decroissante():
        c = k.get_coefficient()
        s = k.get_indeterminee()
        c *= r
        p.ajouter_monome(monome(c, s))

      q = polynome()
      for k in self.__denom.liste_decroissante():
        c = k.get_coefficient()
        s = k.get_indeterminee()
        c *= r
        q.ajouter_monome(monome(c, s))

      if q.degre() == 0:
        n = q.valuation().get_num().valeur()
        m = q.valuation().get_denom().valeur()
        r = rationnel(m, n)
        t = polynome()
        for k in p.liste_decroissante():
          c = k.get_coefficient()
          s = k.get_indeterminee()
          c *= r
          t.ajouter_monome(monome(c, s))
        p = t
        q = polynome_un()

      self.__num = p
      self.__denom = q

      self.simplifier_coefficients()



  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 __add__(self, autre):
    """ addition """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return fraction(a*q + b*p, b*q)
    else:
      return fraction(polynome_nul(), polynome_un(), False)



  def __neg__(self):
    """ fraction opposee """
    if self.__valide:
      a, b = self.__num, self.__denom
      return fraction(-a, b)
    else:
      return fraction(polynome_nul(), polynome_un(), False)



  def __sub__(self, autre):
    """ addition de la fraction opposee """
    return self.__add__(autre.__neg__())



  def __mul__(self, autre):
    """ multiplication """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return fraction(a*p, b*q)
    else:
      return fraction(polynome_nul(), polynome_un(), False)



  def __div__(self, autre):
    """ division """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return fraction(a*q, b*p)
    else:
      return fraction(polynome_nul(), polynome_un(), False)



  def __pow__(self, autre):
    """ exponentiation (si exposant entier relatif) """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom

      if (p.degre() != 0) or (q.degre() != 0):
        return fraction(polynome_nul(), polynome_un(), False)

      r = p.valuation() / q.valuation()
      if r.get_num().valeur() < 0:
        r, a, b = -r, b, a

      if r.est_entier():
        t = polynome()
        t.ajouter_monome(monome(r))
        return fraction(a**t, b**t)
```

```
    return fraction(polynome_nul(), polynome_un(), False)



def fraction_depuis_lettre(c):
  """ construire une fraction (rationnelle) depuis une lettre """
  if c in string.letters:
    p = polynome()
    p.ajouter_monome(monome(rationnel(1), str(c)))
    return fraction(p, polynome_un())
  else:
    return fraction(polynome_nul(), polynome_un(), False)



def fraction_depuis_naturel(x):
  """ construire une fraction (rationnelle) depuis un entier naturel """
  p = polynome()
  p.ajouter_monome(monome(rationnel(x)))
  return fraction(p, polynome_un())



def fraction_nulle_erronee():
  """ fraction nulle (erreur) """
  return fraction(polynome_nul(), polynome_un(), False)



#
# tests unitaires
#
def test_01():
  print "\n*** test 1 ***"
  x = fraction()
  print x
  print x.est_valide()



def test_02():
  print "\n*** test 2 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(1), "x"))
  a.ajouter_monome(monome(rationnel(2)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(1), "xx"))
  b.ajouter_monome(monome(rationnel(2, 3)))

  u = fraction(a, b)
  print u

  c = polynome()
  c.ajouter_monome(monome(rationnel(1), "x"))
  c.ajouter_monome(monome(rationnel(1)))

  d = polynome()
  d.ajouter_monome(monome(rationnel(1), "x"))
  d.ajouter_monome(monome(rationnel(-1)))

  v = fraction(c, d)
  print v

  x = u + v
  print x
```

```
def test_03():
  print "\n*** test 3 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(2, 3), "x"))
  a.ajouter_monome(monome(rationnel(1, 7)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(-1, 24)))

  u = fraction(a, b)
  print u



def test_04():
  print "\n*** test 4 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(2, 3), "x"))
  a.ajouter_monome(monome(rationnel(1, 7)))

  b = polynome_un()

  u = fraction(a, b)
  print u



def test_05():
  print "\n*** test 5 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(2, 3), "x"))
  a.ajouter_monome(monome(rationnel(1, 7)))

  b = polynome_un()

  b = polynome()
  b.ajouter_monome(monome(rationnel(4, 15), "y"))
  b.ajouter_monome(monome(rationnel(-1, 24)))

  u = fraction(a, b)
  print u

  c = polynome()
  c.ajouter_monome(monome(rationnel(1), "x"))
  c.ajouter_monome(monome(rationnel(1)))

  d = polynome()
  d.ajouter_monome(monome(rationnel(1), "x"))
  d.ajouter_monome(monome(rationnel(-1)))

  v = fraction(c, d)
  print v

  x = u + v
  print x

  x = u - v
  print x

  x = u * v
  print x

  x = u / v
  print x
```

```
def test_06():
  print "\n*** test 6 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(1), "x"))
  a.ajouter_monome(monome(rationnel(-1)))

  b = polynome_un()

  b = polynome()
  b.ajouter_monome(monome(rationnel(1), "y"))
  b.ajouter_monome(monome(rationnel(2)))

  u = fraction(a, b)
  print u

  e = polynome()
  e.ajouter_monome(monome(rationnel(-8, -2)))
  n = fraction(e, polynome_un())
  print n

  x = u ** n
  print x



def test_07():
  print "\n*** test 7 ***"
  f = fraction_depuis_lettre("x")
  print f

  g = fraction_depuis_naturel(3)
  print g

  s = f + g
  print s



def test_08():
  print "\n*** test 8 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(1), "x"))
  a.ajouter_monome(monome(rationnel(-1)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(1), "y"))
  b.ajouter_monome(monome(rationnel(2)))

  u = fraction(a, b)
  print u

  c = polynome()
  c.ajouter_monome(monome(rationnel(4), "x"))

  d = polynome()
  d.ajouter_monome(monome(rationnel(1), "x"))

  v = fraction(c, d)
  print v

  x = u ** v
  print x



def test_09():
  print "\n*** test 9 ***"
  a = polynome()
```

```
  a.ajouter_monome(monome(rationnel(1), "x"))
  a.ajouter_monome(monome(rationnel(-1)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(1), "y"))
  b.ajouter_monome(monome(rationnel(2)))

  u = fraction(a, b)
  print u

  c = polynome()
  c.ajouter_monome(monome(rationnel(4)))

  d = polynome()
  d.ajouter_monome(monome(rationnel(1)))

  v = fraction(c, d)
  print v

  x = u ** v
  print x



def test_10():
  print "\n*** test 10 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(6), "x"))
  a.ajouter_monome(monome(rationnel(-24)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(12), "y"))
  b.ajouter_monome(monome(rationnel(-6)))

  u = fraction(a, b)
  print u



def test_11():
  print "\n*** test 11 ***"
  a = polynome()
# a.ajouter_monome(monome(rationnel(6), "x"))
# a.ajouter_monome(monome(rationnel(-24)))
  a.ajouter_monome(monome(rationnel(-24), "x"))

  b = polynome()
  b.ajouter_monome(monome(rationnel(12), "y"))
  b.ajouter_monome(monome(rationnel(-6)))

  u = fraction(a, b)
  print u



def test_12():
  print "\n*** test 12 ***"
  a = polynome()
  a.ajouter_monome(monome(rationnel(2, 3), "x"))
  a.ajouter_monome(monome(rationnel(1, 7)))

  b = polynome()
  b.ajouter_monome(monome(rationnel(4, 15), "y"))
  b.ajouter_monome(monome(rationnel(-1, 24)))

  u = fraction(a, b)
  print u
```

```
  c = polynome()
  c.ajouter_monome(monome(rationnel(4)))

  d = polynome_un()

  v = fraction(c, d)
  print v

  x = u ** v
  print x



def tests_unitaires():
  test_01()
  test_02()
  test_03()
  test_04()
  test_05()
  test_06()
  test_07()
  test_08()
  test_09()
  test_10()
  test_11()
  test_12()



if __name__ == "__main__":
  tests_unitaires()
```

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

#
# fichier: monome.py
#    date: 2011/05/02
#
# (tous les symboles non internationaux sont volontairement omis)
#

import string

from rationnel import *

class monome(object):
  """ classe pour un monome (rationnel et indeterminee) """

  def __init__(self, coefficient =rationnel(0), indeterminee ="", valide =True):
    """ constructeur """
    self.__valide = valide and coefficient.est_valide()
    s = ""
    if self.__valide:
      l = []
      for i in indeterminee:
        self.__valide = self.__valide and (i in string.letters)
        if self.__valide:
          l.append(i)
      if self.__valide:
        l.sort()
        for c in l:
          s += c
      else:
        self.__coefficient = rationnel()
    self.__coefficient = coefficient
    self.__indeterminee = s



  def __formater_indeterminee(self):
    """ formate l'indeterminee pour l'impression """
    t = str(self.__indeterminee)
    if len(t) > 1:
      v = []
      n = 0
      i = 0
      for ch in t:
        if i == 0:
          v.append(ch)
          n = 1
        else:
          if ch == v[-1]:
            n += 1
          else:
            if n > 1:
              v.append("^")
              v.append(str(n))
              n = 1
            v.append(" * ")
            v.append(ch)
        i = i + 1
      if n > 1:
        v.append("^")
        v.append(str(n))
      t = "".join(v)
    return t
```

```
  def __str__(self):
    """ representation en chaine de caracteres """
    if not self.__valide:
      return "(monome invalide)"
    t = str(self.__coefficient)
    if not self.__coefficient.est_positif():
      t = "(" + t + ")"
    if len(self.__indeterminee) == 0:
#     return "(" + str(self.__coefficient) + ")"
      return t
    else:
      s = self.__formater_indeterminee()
#     return "(" + str(self.__coefficient) + ") * " + s
      if self.__coefficient.est_unite():
        return s
      else:
        return t + " * " + s



  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 __cmp__(self, autre):
    """ comparaison (ordre sur les indeterminees) """
    u = len(self.__indeterminee)
    v = len(autre.__indeterminee)
    if u != v:
      return -cmp(u, v)
    else:
      return cmp(self.__indeterminee, autre.__indeterminee)



  def get_coefficient(self):
    """ accesseur coefficient """
    return self.__coefficient



  def get_indeterminee(self):
    """ accesseur indeterminee """
    return self.__indeterminee



  def __neg__(self):
    """ monome oppose """
    return monome(-(self.__coefficient), self.__indeterminee, self.__valide)



  def produit(self, autre):
```

```
    """ produit de deux monomes """
    if self.__valide and autre.__valide:
      c = self.__coefficient * autre.__coefficient
      s = str(self.__indeterminee) + str(autre.__indeterminee)
      return monome(c, s)
    else:
      return monome(rationnel(0, 1, False), "")



if __name__ == "__main__":

  x = monome(rationnel(8, -14),"$")
  print x
  print x.est_valide()

  y = monome(rationnel(8, 0),"a")
  print y
  print y.est_valide()

  z = -x
  print z
  print z.est_valide()

  x = monome(rationnel(8, -14),"aaaxxaacccddxxddx")
  print x
  print x.est_valide()
```

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

#
# fichier: noeud.py
#    date: 2011/05/02
#
# (tous les symboles non internationaux sont volontairement omis)
#

from monome import *

class noeud(object):
  """ classe pour un noeud d'ABR representant un polynome """

  def __init__(self, monome, gauche =None, droite =None):
    """ constructeur """
    self.__monome = monome
    self.__gauche = gauche
    self.__droite = droite



  def __str__(self):
    """ afficher le monome """
    s = ""
    if self.__droite:
      s = " + " + str(self.__droite)
    s = str(self.__monome) + s
    if self.__gauche:
      s = str(self.__gauche) + " + " + s
    return s



  def chercher(self, monome):
    """ chercher un monome """
    if monome == self.__monome:
      return True
    if monome < self.__monome:
      if self.__gauche:
        return self.__gauche.chercher(monome)
      else:
        return False
    else:
      if self.__droite:
        return self.__droite.chercher(monome)
      else:
        return False



  def inserer(self, monome):
    """ inserer un monome (non existant) """
    if monome != self.__monome:
      if monome < self.__monome:
        if self.__gauche:
          self.__gauche.inserer(monome)
        else:
          self.__gauche = noeud(monome)
      else:
        if self.__droite:
          self.__droite.inserer(monome)
        else:
          self.__droite = noeud(monome)
```

```
  def plus_grand(self):
    """ plus grand monome (a droite i.e. plus faible degre) """
    if self.__droite:
      return self.__droite.plus_grand()
    else:
      return self.__monome



  def plus_petit(self):
    """ plus petit monome (a gauche i.e. plus grand degre) """
    if self.__gauche:
      return self.__gauche.plus_petit()
    else:
      return self.__monome



  def get_monome(self):
    """ accesseur du monome contenu dans le noeud """
    return self.__monome



  def set_monome(self, monome):
    """ mutateur du monome contenu dans le noeud """
    self.__monome = monome



  def fils_gauche(self):
    """ accesseur fils gauche """
    return self.__gauche



  def fils_droite(self):
    """ accesseur fils droit """
    return self.__droite



  def nombre_monomes(self):
    n = 1
    if self.__gauche:
      n += self.__gauche.nombre_monomes()
    if self.__droite:
      n += self.__droite.nombre_monomes()
    return n



  def chercher_parent(self, monome):
    """ rechercher le noeud parent d'un monome """
    if monome == self.__monome:
      return None
    if monome < self.__monome:
      if self.__gauche.__monome == monome:
        return self
      else:
        return self.__gauche.chercher_parent(monome)
    else:
      if self.__droite.__monome == monome:
        return self
      else:
        return self.__droite.chercher_parent(monome)
```

```
  def __supprimer_noeud(self, monome, parent):
    """ supprimer le noeud contenant un monome """
    if monome < self.__monome:
      self.__gauche.__supprimer_noeud(monome, parent)
    else:
      if monome > self.__monome:
        self.__droite.__supprimer_noeud(monome, parent)
      else:
        if self.__gauche is None and self.__droite is None:
          """ aucun fils """
          if parent.__gauche is self:
            parent.__gauche = None
          else:
            parent.__droite = None
        else:
          if self.__gauche is None or self.__droite is None:
            """ fils unique """
            if self.__gauche:
              t = self.__gauche
            else:
              t = self.__droite
            if parent.__gauche is self:
              s = parent.__gauche
              parent.__gauche = t
            else:
              s = parent.__droite
              parent.__droite = t
            s.__gauche = None
            s.__droite = None
          else:
            """ deux fils """
            if self.__gauche.nombre_monomes() > self.__droite.nombre_monomes():
              x = self.__gauche.plus_grand()
            else:
              x =  self.__droite.plus_petit()
            self.supprimer(x)
            self.__monome = x



  def supprimer(self, monome):
    """ supprimer un monome (existant) """
    parent = self.chercher_parent(monome)
    self.__supprimer_noeud(monome, parent)



  def retrouver(self, monome):
    """ retrouver un monome (existant) """
    if monome == self.__monome:
      return self.__monome
    if monome < self.__monome:
      if self.__gauche:
        return self.__gauche.retrouver(monome)
    else:
      if self.__droite:
        return self.__droite.retrouver(monome)



# def montrer_ordre_croissant(self):
#   """ montrer le polynome suivant les puissances croissantes """
#   if self.__gauche:
#     self.__gauche.montrer_ordre_croissant()
#   print self.__monome
#   if self.__droite:
#     self.__droite.montrer_ordre_croissant()
```

```
# def montrer_ordre_decroissant(self):
#   """ montrer le polynome suivant les puissances decroissantes """
#   if self.__droite:
#     self.__droite.montrer_ordre_decroissant()
#   print self.__monome
#   if self.__gauche:
#     self.__gauche.montrer_ordre_decroissant()



# def __liste_ordre_croissant(self, liste):
#   """ liste des monomes suivant les puissances croissantes """
#   if self.__gauche:
#     self.__gauche.__liste_ordre_croissant(liste)
#   liste.append(self.__monome)
#   if self.__droite:
#     self.__droite.__liste_ordre_croissant(liste)



# def liste_croissante(self):
#   """ accesseur liste des monomes suivant les puissances croissantes """
#   liste = []
#   self.__liste_ordre_croissant(liste)
#   return liste



  def __liste_ordre_decroissant(self, liste):
    """ liste des monomes suivant les puissances decroissantes """
    if self.__droite:
      self.__droite.__liste_ordre_decroissant(liste)
    liste.append(self.__monome)
    if self.__gauche:
      self.__gauche.__liste_ordre_decroissant(liste)



  def liste_decroissante(self):
    """ accesseur liste des monomes suivant les puissances decroissantes """
    liste = []
    self.__liste_ordre_decroissant(liste)
    return liste
```

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

#
# fichier: polynome.py
#    date: 2011/05/03
#
# (tous les symboles non internationaux sont volontairement omis)
#

from string import *

from entier import *
from monome import *
from noeud import *
from rationnel import *

class polynome(object):
  """ classe pour un polynome (structure d'arbre binaire de recherche) """

  def __init__(self, valide =True):
    """ constructeur (polynome nul) """
    self.__valide = valide
    self.__tete = noeud(monome())



  def __str__(self):
    """ representation en chaine de caracteres """
    if self.__valide:
      return str(self.__tete)
    else:
      return "(polynome invalide)"



  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 chercher(self, monome):
    """ rechercher un monome """
    if self.__tete:
      return self.__tete.chercher(monome)
    else:
      return False



  def inserer(self, monome):
    """ inserer un nouveau monome """
    self.__valide = self.__valide and monome.est_valide()
    if self.__valide:
      if self.__tete:
```

```
        self.__tete.inserer(monome)
      else:
        self.__tete = noeud(monome)



  def supprimer(self, monome):
    """ supprimer un monome """
    if self.chercher(monome):
      if self.__tete.get_monome() == monome:
        if self.__tete.fils_gauche():
          fils = self.__tete.fils_gauche()
        else:
          fils = self.__tete.fils_droite()
        if fils is None:
          self.__tete = None
        else:
          if self.__tete.fils_gauche():
            x = fils.plus_grand()
          else:
            x = fils.plus_petit()
          self.__tete.supprimer(x)
          self.__tete.set_monome(x)
      else:
        self.__tete.supprimer(monome)
    else:
      print "(monome inexistant)"



  def retrouver(self, monome):
    """ retrouver un monome (existant) """
    if self.chercher(monome):
      return self.__tete.retrouver(monome)
    else:
      return None



  def degre(self):
    """ donne le degre du polynome """
    if self.__tete:
      return len(self.__tete.plus_petit().get_indeterminee())
    else:
      """ concession a la definition mathematique du degre du polynome nul """
      return (-1)



  def valuation(self):
    """ donne la valuation du polynome """
    if self.__tete:
      return self.__tete.plus_grand().get_coefficient()
    else:
      return rationnel()



  def ajouter_monome(self, m):
    """ adjoindre un nouveau monome au polynome """
    if m.get_coefficient().est_nul():
      return
    if self.chercher(m) == True:
      x = self.retrouver(m)
      a = x.get_coefficient()
      b = m.get_coefficient()
      self.supprimer(x)
      r = a + b
```

```
      if not r.est_nul():
        self.inserer(monome(r, m.get_indeterminee()))
    else:
      self.inserer(m)
    """ suppression du monome nul (si besoin) """
    if self.valuation().est_nul() and self.degre() > 0:
      self.supprimer(monome())



  def monome_plus_grand_degre(self):
    """ donne le monome de plus haut degre """
    if self.__tete:
      """ plus petit == plus a gauche dans l'ABR """
      return self.__tete.plus_petit()
    else:
      return None



  def monome_plus_petit_degre(self):
    """ donne le monome de plus petit degre """
    if self.__tete:
      """ plus grand == plus a droite dans l'ABR """
      return self.__tete.plus_grand()
    else:
      return None



  def liste_decroissante(self):
    """ liste des monomes en ordre decroissant """
    if self.__tete:
      return self.__tete.liste_decroissante()
    else:
      return []



  def nombre_monomes(self):
    """ nombre de monomes du polynome """
    if self.__tete:
      return self.__tete.nombre_monomes()
    else:
      return 0



  def __add__(self, autre):
    """ addition """
    if self.__valide and autre.__valide:
      p = polynome()
      a = self.liste_decroissante()
      for m in a:
        p.ajouter_monome(m)
      b = autre.liste_decroissante()
      for m in b:
        p.ajouter_monome(m)
      return p
    else:
      return polynome(False)



  def __neg__(self):
    """ oppose du polynome """
    if self.__valide:
      p = polynome()
```

```
      a = self.liste_decroissante()
      for m in a:
        p.ajouter_monome(-m)
      return p
    else:
      return polynome(False)



  def __sub__(self, autre):
    """ soustraction """
    return self.__add__(autre.__neg__())



  def __mul__(self, autre):
    """ produit de deux polynomes """
    if self.__valide and autre.__valide:
      p = polynome()
      a = self.liste_decroissante()
      b = autre.liste_decroissante()
      for m in a:
        for n in b:
          p.ajouter_monome(monome.produit(m, n))
      return p
    else:
      return polynome(False)



  def __pow__(self, autre):
    """ exponentiation d'un polynome (exposant entier naturel) """
    if not(self.__valide and autre.__valide):
      return polynome(False)

    if autre.degre() < 0:
      return polynome(False)

    v = autre.valuation()
    if not v.est_entier():
      return polynome(False)

    n = v.get_num().valeur()
    if self.valuation().est_nul():
      if n < 0:
        return polynome(False)
      else:
        p = polynome()
        p.ajouter_monome(monome(rationnel(1)))
        return p

    if n < 0:
      return polynome(False)

    p = polynome()
    a = self
    p.ajouter_monome(monome(rationnel(1)))
    while n > 0:
      if n % 2 == 1:
        p *= a
      n //= 2
      a *= a
    return p



  def est_nul(self):
    """ le polynome est-il nul ? """
```

```
    if self.__valide:
      return (self.degre() == 0) and (self.valuation().est_nul())
    else:
      return False



  def est_unite(self):
    """ le polynome est-il egal a 1 ? """
    if self.__valide:
      return (self.degre() == 0) and self.valuation().est_unite()
    else:
      return False



  def pgcd_numerateurs(self):
    """ pgcd des numerateurs des coefficients des monomes """
    l = []
    if self.__valide:
      for m in self.liste_decroissante():
        e = abs(m.get_coefficient().get_num().valeur())
        if not (e in l):
          l.append(e)
    return pgcd_liste(l)



  def ppcm_denominateurs(self):
    """ ppcm des denominateurs des coefficients des monomes """
    l = []
    n = 1
    if self.__valide:
      for m in self.liste_decroissante():
        """ les denominateurs sont positifs """
        e = m.get_coefficient().get_denom().valeur()
        if not (e in l):
          l.append(e)
        n *= e
    return n / pgcd_liste(l)



def polynome_nul():
  """ le polynome nul """
  p = polynome()
  return p



def polynome_un():
  """ le polynome unite """
  p = polynome()
  p.ajouter_monome(monome(rationnel(1)))
  return p



if __name__ == "__main__":

  print "polynome a (invalide)"
  a = polynome()
  print a
  a.ajouter_monome(monome(rationnel(1), "xx"))
  t = monome(rationnel(4, -6, False), "x")
  a.ajouter_monome(t)
  a.ajouter_monome(monome(rationnel(-5)))
  print a
```

```
  print "\n"

  print "polynome a"
  a = polynome()
  print a
  a.ajouter_monome(monome(rationnel(1), "xx"))
  t = monome(rationnel(4, -6), "x")
  a.ajouter_monome(t)
  a.ajouter_monome(monome(rationnel(-5)))
  print a
  m = monome(rationnel(2), "x")
  a.ajouter_monome(m)
  print a
  a.ajouter_monome(monome(rationnel(4, -3), "x"))
  print a
  print a.degre()
  print a.valuation()
  print "\n"

  print "polynome b"
  b = polynome()
  b.ajouter_monome(monome(rationnel(1), "a"))
  print b
  b.ajouter_monome(monome(rationnel(7, -5), "a"))
  print b
  b.ajouter_monome(monome(rationnel(-5)))
  print b
  print "\n"

  print "polynome s (somme)"
  s = a + b
  print s
  print "\n"

  print "polynome d (difference)"
  d = a - b
  print d
  print "\n"

  print "polynome p (produit)"
  p = a * b
  print p
  print "\n"

# print p.monome_plus_grand_degre()
# print p.monome_plus_petit_degre()

# for c in p.liste_coefficients():
#   print c

# for c in p.liste_denominateurs():
#   print c
# print

# print p.ppcm_denominateurs()
# print

  print "polynome a"
  a = polynome()
  a.ajouter_monome(monome(rationnel(560), "x"))
  a.ajouter_monome(monome(rationnel(120)))
  print a
  print "polynome n"
  n = polynome()
  n.ajouter_monome(monome(rationnel(4)))
  print n
  print "polynome p (puissance)"
  p = a ** n
```

```
  print p
  print "\n"

# print "polynome n"
# n = polynome()
# n.ajouter_monome(monome(rationnel(-4)))
# print n
# print "polynome p (puissance)"
# p = a ** n
# print p
# print "\n"

# print "polynome n"
# n = polynome()
# print n
# print "polynome p (puissance)"
# p = a ** n
# print p
# print "\n"
```

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

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

from entier import *

def produit_2_5(n):
  """ n est-il de la forme (2^p)*(5^q) ? """
  if n == 1:
    return True
  else:
    if n % 2 == 0:
      return produit_2_5(n/2)
    if n % 5 == 0:
      return produit_2_5(n/5)
    return False



class rationnel(object):
  """ classe pour un nombre rationnel """

  def __init__(self, num =long(0), denom =long(1), valide =True):
    """ constructeur """
    valide = valide and (type(num) == int or type(num) == long or isinstance(num, entier))
    valide = valide and (type(denom) == int or type(denom) == long or isinstance(denom, ent
ier))
    if valide:
      if type(num) == int or type(num) == long:
        num = entier(num)
      if type(denom) == int or type(denom) == long:
        denom = entier(denom)
    valide = valide and num.est_valide() and denom.est_valide()
    self.__valide = valide and (denom.valeur() != 0)
    if self.__valide:
      if denom.valeur() < 0:
        num, denom = -num, -denom
      d = pgcd_entier(num, denom)
      self.__num, self.__denom = num / d, denom / d
    else:
      self.__num, self.__denom = entier(0), entier(1)



  def __str__(self):
    """ representation en chaine de caracteres """
    if self.__valide:
      if self.__denom.valeur() == 1:
        return str(self.__num)
      if produit_2_5(self.__denom.valeur()):
        return str(self.__float__())
      else:
        return str(self.__num) + "/" + str(self.__denom)
    return "(nombre rationnel invalide)"



  def __float__(self):
    """ donne (si possible) la representation decimale, sinon 0.0 """
    if self.__valide and produit_2_5(self.__denom.valeur()):
      a, b = self.__num.valeur(), self.__denom.valeur()
      return (float(a)/float(b))
```

```
    return float(0)


  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 __add__(self, autre):
    """ addition """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return rationnel(a*q + b*p, b*q)
    else:
      return rationnel(0, 1, False)


  def __neg__(self):
    """ rationnel oppose """
    if self.__valide:
      a, b = self.__num, self.__denom
      return rationnel(-a, b)
    else:
      return rationnel(0, 1, False)


  def __sub__(self, autre):
    """ addition de l'oppose """
    return self.__add__(autre.__neg__())


  def __mul__(self, autre):
    """ multiplication """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return rationnel(a*p, b*q)
    else:
      return rationnel(0, 1, False)


  def __div__(self, autre):
    """ division """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      return rationnel(a*q, b*p)
    else:
      return rationnel(0, 1, False)
```

```
  def __pow__(self, autre):
    """ exponentiation (si exposant entier relatif) """
    if self.__valide and autre.__valide:
      a, b = self.__num, self.__denom
      p, q = autre.__num, autre.__denom
      if (a.valeur() == 0) and (p.valeur() <= 0):
        return rationnel(0, 1, False)
      if p.valeur() < 0:
        p, a, b = -p, b, a
      if b.valeur() < 0:
        a, b = -a, -b
      if q.valeur() == 1:
        return rationnel(a**p, b**p)
    return rationnel(0, 1, False)



  def est_nul(self):
    """ le rationnel est-il nul ? """
    if self.__valide:
      return self.__num.valeur() == 0
    else:
      return False



  def est_positif(self):
    """ le rationnel est-il positif ? """
    if self.__valide:
      return self.__num.est_positif()
    else:
      return False



  def est_entier(self):
    """ le rationnel est-il entier ? """
    if self.__valide:
      return (self.__denom.valeur() == 1)
    else:
      return False



  def est_unite(self):
    """ le polynome est-il egal a 1 ? """
    if self.__valide:
      return (self.__num.valeur() == self.__denom.valeur())
    else:
      return False



  def get_num(self):
    """ accesseur du numerateur """
    return self.__num



  def get_denom(self):
    """ accesseur du denominateur """
    return self.__denom
```

```
if __name__ == "__main__":

  x = rationnel(-10, -40)
  print x.est_valide()
  print x
  print float(x)

  y = rationnel(-10, -41)
  print y
  print float(y)

  z = rationnel(10, -45)
  print z.est_valide()
  print z

  z = rationnel(1, 560)
  n = rationnel(-4)

  p = z ** n
  print p.est_valide()
  print p
```