```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

#
# fichier: polynome.py
# version: 0.5.0
#  auteur: Pascal CHAUVIN
#    date: 2014/10/28
#
# (tous les symboles non internationaux sont volontairement omis)
#

import sys
sys.path.append('../entier_mod')
sys.path.append('../monome_mod')
sys.path.append('../monome_mod/joli_mod')
sys.path.append('../rationnel_mod')
sys.path.append('../utile_mod')

import entier as ent
import monome as mo
import joli
import rationnel as ra
import utile

class polynome(object):

  def __init__(self, monome =mo.monome(), gauche =None, droite =None, valide =True):
    """ - """
    if valide:
      self.__monome = monome
    else:
      self.__monome = None

    self.__valide = valide and monome.est_valide()
    self.__gauche = gauche
    self.__droite = droite



  def plat(self):
    """ - """
    acc = ""
    if self.__droite is not None:
      acc = " + " + self.__droite.plat()

    acc = str(self.__monome) + acc

    if self.__gauche is not None:
      acc = self.__gauche.plat() + " + " + acc

    return acc



  def __repr__(self):
    """ - """
    gauche = None
    if self.__gauche is not None:
      gauche = id(self.__gauche)

    droite = None
    if self.__droite is not None:
      droite = id(self.__droite)

    return "[polynome:\n__monome={},\n__gauche={},\n__droite={},\n__valide={}\n]\n"\
      .format(self.__monome, gauche, droite, self.__valide)
  def __str__(self):
    return self.plat()


  def joli(self):
    """ - """
    acc = ""
    if self.__droite is not None:
      acc = " + " + self.__droite.joli()

    acc = self.__monome.joli() + acc

    if self.__gauche is not None:
      acc = self.__gauche.joli() + " + " + acc

    return acc


  def est_valide(self):
    """ accesseur """
    return self.__valide


  def fixer_valide(self, v):
    """ accesseur """
    self.__valide = v


  def lire_monome(self):
    return self.__monome


  def __inserer(self, k):
    if self.__monome < k:
      if self.__gauche is None:
        self.__gauche = polynome(k)
      else:
        self.__gauche.__inserer(k)
      return

    if self.__droite is None:
      self.__droite = polynome(k)
    else:
      self.__droite.__inserer(k)


  def inserer(self, k):
    if self.__monome.lire_coeff().est_zero():
      self.__monome = k
    else:
      self.__inserer(k)


  def __iterateur(self, acc):
    """ - """
    if self.__gauche is not None:
       self.__gauche.__iterateur(acc)

    acc.append(self.__monome)
    if self.__droite is not None:
      self.__droite.__iterateur(acc)



  def iterateur(self):
    """ - """
    acc = []
    self.__iterateur(acc)
    return acc



  def joindre(self, k):
    """ ajouter un monome donne un nouveau polynome """
    if not k.est_valide:
      p = polynome()
      p.fixer_valide(False)
      return p

    if k.lire_coeff().est_zero():
      p = polynome()
      u = self.iterateur()
      for i in u:
        p.inserer(i)
      return p

    p = polynome()
    u = self.iterateur()
    trouve = False

    for i in u:
      if i.lire_indet() == k.lire_indet():
        trouve = True
        c = i.lire_coeff() + k.lire_coeff()
        if not c.est_zero():
          p.inserer(mo.monome(c, k.lire_indet()))
      else:
        p.inserer(i)

    if not trouve:
      p.inserer(k)

    return p



  def contient(self, k):
    """ - """
    if self.__monome == k:
      return True

    if self.__monome < k:
      if self.__gauche is None:
        return False
      else:
        return self.__gauche.contient(k)

    if self.__droite is None:
      return False
    else:
      return self.__droite.contient(k)



  def debut(self):
    """ debut = monome le plus a gauche """
    if self.__gauche is None:
      return self
    else:
      return self.__gauche.debut()



  def degre(self):
    """ donne le degre du polynome """
    m = self.debut().lire_monome()
    if m is None:
      return (-1) # concession a la def. math. du degre du pol. nul

    if m.est_degre_nul():
      return 0

    return len(m.lire_indet())



  def fin(self):
    """ fin = monome le plus a droite """
    if self.__droite is None:
      return self
    else:
      return self.__droite.fin()



  def valuation(self):
    """ - """
    m = self.fin().lire_monome()
    if m:
      return m.lire_coeff()

    return rat.rationnel()



  def nombre_monomes(self):
    """ nombre de monomes """
    n = 1

    if self.__gauche is not None:
      n += self.__gauche.nombre_monomes()

    if self.__droite is not None:
      n += self.__droite.nombre_monomes()

    return n



  def __add__(self, autre):
    """ addition """
    if isinstance(autre, polynome):
      if (self.__valide) and (autre.__valide):
        p = polynome()

        for m in self.iterateur():
          p = p.joindre(m)

        for m in autre.iterateur():
          p = p.joindre(m)

        return p

    return polynome(mo.monome(), None, None, False)
  def __neg__(self):
    """ polynome oppose (inverse pour l'addition) """
    if self.__valide:
      p = polynome()

      for m in self.iterateur():
        t = mo.monome(-m.lire_coeff(), m.lire_indet(), m.est_valide())
        p = p.joindre(t)

      return p

    return polynome(mo.monome(), None, None, False)



  def oppose(self):
    """ polynome oppose """
    return self.__neg__()



  def __sub__(self, autre):
    """ difference de deux polynomes """
    return (self + (-autre))



  def __mul__(self, autre):
    """ produit de deux polynomes """
    if isinstance(autre, polynome):
      if (self.__valide) and (autre.__valide):
        p = polynome()

        for m in self.iterateur():
          m_coeff = m.lire_coeff()
          m_indet = m.lire_indet()

          for n in autre.iterateur():
            n_coeff = n.lire_coeff()
            n_indet = n.lire_indet()

            i = utile.reduction(m_indet + n_indet)

            k = mo.monome(m_coeff * n_coeff, i)
            p = p.joindre(k)

        return p

    return polynome(mo.monome(), None, None, False)



  def __exponentiation(self, n):
    """ exponentiation (exposant entier naturel) """
    p = polynome()

    a = self
    p = p.joindre(mo.monome(ra.rationnel(1)))

    while n > 0:
      if n % 2 == 1:
        p *= a
      n //= 2
      a *= a
    return p



  def __pow__(self, autre):
    """ exponentiation """
    if isinstance(autre, int):
      autre = polynome(mo.monome(ra.rationnel(autre)))

    if isinstance(autre, polynome):
      if (self.__valide) and (autre.__valide):
        v = autre.valuation()

        if self.est_polynome_nul() and v.lire_num().est_zero():
          return polynome(mo.monome(), None, None, False)

        n = v.lire_num().lire_valeur()

        if (self.degre() != 0) and (n < 0):
          return polynome(mo.monome(), None, None, False)

        return self.__exponentiation(n)

    return polynome(mo.monome(), None, None, False)



  def est_degre_nul(self):
    """ - """
    return (self.degre() == 0)



  def est_polynome_nul(self):
    """ - """
    return self.valuation().lire_num().est_zero()



  def est_polynome_unite(self):
    """ - """
    return (self.degre() == 0) and \
      self.valuation().lire_num().est_un()



  def liste_decroissante_monomes(self):
    """ - """
    liste = []

    it = self.iterateur()
    for mono in it:
      liste.append(mono) # i.e. liste.append(repr(mono))

    return sorted(liste, reverse = True)



  def pgcd_numerateurs(self):
    """ - """
    l = []
    if self.__valide:
      for m in self.liste_decroissante_monomes():
        e = m.lire_coeff().lire_num().lire_valeur()
        if not (e in l):
          l.append(e)
    return ent.pgcd_liste(l)
```

```
 def ppcm_denominateurs(self):
  """ - """
  l = []
  n = 1
  if self.__valide:
   for m in self.liste_decroissante_monomes():
    """ les denominateurs sont positifs """
    e = m.lire_coeff().lire_denom().lire_valeur()
    if not (e in l):
     l.append(e)
    n *= e
  return (n // ent.pgcd_liste(l))



def polynome_err():
 """ polynome nul obtenu par un calcul avec erreur """
 p = polynome()
 p.fixer_valide(False)
 return p



def polynome_nul():
 """ polynome nul """
 return polynome()



def polynome_un():
 """ polynome unite """
 return polynome().joindre(mo.monome(ra.rationnel(1)))



if __name__ == "__main__":
 pass
```