

```

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

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

import string

import sys
sys.path.append('../math_mod')
sys.path.append('../math_mod/fraction_mod')
sys.path.append('../math_mod/monome_mod')
#sys.path.append('../math_mod/monome_mod/joli_mod')
sys.path.append('../math_mod/polynome_mod')
sys.path.append('../math_mod/rationnel_mod')
sys.path.append('../math_mod/utile_mod')
sys.path.append('../erreur_mod')

import math_mod.fraction_mod.fraction as frac
import math_mod.monome_mod.monome as mo
import math_mod.polynome_mod.polynome as po
import math_mod.rationnel_mod.rationnel as ra
import math_mod.utile_mod.utile as util
import erreur_mod.erreur as err

class expression(object):

    def __init__(self, formule="", valide =True):
        """ constructeur """
        correction = util.correction_math(formule)
        self.__valide = valide and (len(correction) > 0)

        self.__valeur = frac.fraction()

        if self.__valide:
            self.__formule = "{" + correction + "}"
            self.__erreur = err.ERREUR.AUCUNE_ERREUR
            self.__valeur = frac.fraction()

            self.__i = 0
            self.__longueur = len(self.__formule)
            self.evaluer()
        else:
            self.__erreur = err.ERREUR.SYNTAXE_NON_CONFORME
            self.__valeur = frac.fraction_err()

    def __repr__(self):
        """ - """
        return "[expression:\
\n__formule={0},\
\n__valeur={1},\
\n__erreur={2},\
\n__valide={3}\
\n]\n".\
format(self.__formule, \
        self.__valeur, \
        self.__erreur, \
        self.__valide
)

```

```
def __str__(self):
    """ _ """
    if self.__erreur == err.ERREUR.AUCUNE_ERREUR:
        return self.__valeur.joli()
    else:
        return "[erreur: {}]".format(self.message())

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

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

def message(self):
    """ indique le type d'erreur """
    if self.__erreur == err.ERREUR.AUCUNE_ERREUR:
        return "aucune erreur"

    if self.__erreur == err.ERREUR.ACC_OUVRANTE_MANQUANTE:
        return "il manque une accolade ouvrante \"{\"}"

    if self.__erreur == err.ERREUR.ACC_FERMANTE_MANQUANTE:
        return "il manque une accolade fermante \"}\""

    if self.__erreur == err.ERREUR.SYNTAXE_NON_CONFORME:
        return "la syntaxe de l'expression n'est pas conforme"

    if self.__erreur == err.ERREUR.DIVISION_PAR_0:
        return "tentative de diviser par 0"

    if self.__erreur == err.ERREUR.EXPOSANT_INVALIDE:
        return "exposant invalide"

    if self.__erreur == err.ERREUR.MANQUE_NOMBRE_OU_PER_OUVRANTE:
        return "il manque un nombre ou une parenthese ouvrante \"(\""

    if self.__erreur == err.ERREUR.MANQUE_PAR_FERMANTE:
        return "il manque une parenthese fermante \")\""

    return "erreur de type inconnu"

def afficher_erreur(self):
    """ _ """
    print("erreur: {}\\n".format(self.message()))

def lire_valeur(self):
    """ fraction egale a l'expression evaluee """
    if self.__valide:
        return self.__valeur
    else:
        return frac.fraction_err()
```

```
def valider(self):
    """ _ """
    self.__valide = True

def invalider(self):
    """ _ """
    self.__valide = False

def lire_formule(self):
    """ donne l'expression initiale (textuelle) """
    return self.__formule[1:-1]

def rapporter_erreur(self, e):
    """ relever une erreur (si aucune erreur avant) """
    if self.__erreur == err.ERREUR.AUCUNE_ERREUR:
        self.__erreur = e

def erreur_existe(self):
    """ indique si une erreur s'est produite (derniere evaluation) """
    return (self.__erreur != err.ERREUR.AUCUNE_ERREUR)

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

def evaluer(self):
    """ evaluation mathematique de l'expression """
    self.__ch = self.suivant()
    if self.__ch == '{':
        self.__ch = self.suivant()
        self.__valeur = self.expr()
        self.__ch = self.suivant()
        if self.__ch != '}':
            self.rapporter_erreur(err.ERREUR.ACC_FERMANTE_MANQUANTE)
    else:
        self.rapporter_erreur(err.ERREUR.ACC_OUVRANTE_MANQUANTE)

def suivant(self):
    """ lecture du caractere suivant """
    while self.__i < self.__longueur:
        t = self.__formule[self.__i]
        self.__i += 1
        if not str.isspace(t):
            return t
    return '\0'

def prochain(self):
    """ observer le prochain caractere """
    while self.__i < self.__longueur:
        t = self.__formule[self.__i]
        if str.isspace(t):
            self.__i += 1
```

```

    else:
        return t
    return '\0'

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

def expr(self):
    """ expr ::= expr1 '+' expr1 | expr1 '-' expr1 | expr1 """
    if self.erreur_existe():
        return fraction_err()

    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_erreur()

    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()
                e = self.expr2()
                if e.lire_num().est_polynome_nul():
                    self.rapporter_erreur(err.ERREUR.DIVISION_PAR_0)
                    self.invalider()
                    t = frac.fraction_err()
                else:
                    t /= e
    return t

def expr2(self):
    """ expr2 ::= '-' expr3 | expr3 """
    if self.erreur_existe():
        return fraction_nulle_erreur()

    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 frac.fraction_nulle_erreur()

    t = self.expr4()
    if self.prochain_est('^'):
        self.__ch = self.suivant()
        self.__ch = self.suivant()
        k = self.expr2()
        if k.est_valide() and k.est_un_entier():
            t = t ** k
        else:
            self.rapporter_erreur(err.ERREUR.EXPOSANT_INVALIDE)
            self.invalider()
            t = frac.fraction_err()
    return t

def expr4(self):
    """ expr4 ::= <entier naturel> | <lettre> | '(' expr ')' """
    if self.erreur_existe():
        return fraction_nulle_erreur()

    if str.isdigit(self.__ch):
        t = self.naturel()
        return frac.fraction_depuis_naturel(t)

    if (self.__ch in string.ascii_letters):
        t = self.lettre()
        return frac.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(err.ERREUR.MANQUE_PAR_FERMANTE)
    else:
        self.rapporter_erreur(err.ERREUR.MANQUE_NOMBRE_OU_PER_OUVRANTE)

    return frac.fraction_nulle_erreur()

def naturel(self):
    """ naturel ::= ('0' | '1' | '2' | ... | '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.lire_valeur() + autre.lire_valeur()

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

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

def __truediv__(self, autre):
    """ quotient """
    return self.lire_valeur() / autre.lire_valeur()

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

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

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

#
# fichier: expression_tests.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('../calc_mod')
sys.path.append('../calc_mod/math_mod')
sys.path.append('../calc_mod/math_mod/entier_mod')
sys.path.append('../calc_mod/math_mod/rationnel_mod')
sys.path.append('../calc_mod/math_mod/monome_mod')
sys.path.append('../calc_mod/math_mod/polynome_mod')
sys.path.append('../calc_mod/math_mod/fraction_mod')
sys.path.append('../calc_mod/math_mod/utile_mod')
sys.path.append('../calc_mod/expression_mod')
sys.path.append('../calc_mod/calcul_mod')

import expression as ex

import fraction as frac
import rationnel as ra

def test_unitaire_0(visible =False):
    print("*** expression: test_unitaire_0 ***")

    e = ex.expression("{5 * {x / 3} + [x - 1]}")
    if visible:
        print(e)
        print(repr(e))

    f = frac.fraction()
    if visible: print(f)

    ok = e.est_valide() and f.est_valide()
    return ok

def test_unitaire_1(visible =False):
    print("*** expression: test_unitaire_1 ***")

    a = frac.fraction_depuis_lettre('a')
    if visible: print(a)

    b = frac.fraction_depuis_lettre("b")
    if visible: print(b)

    ok = (b.lire_num().valuation() == ra.rationnel(1))
    return ok

def test_unitaire_2(visible =False):
    print("*** expression: test_unitaire_2 ***")

    n = frac.fraction_depuis_naturel(20 + 1)
    if visible: print(n)

    ok = (n.lire_num().valuation() == ra.rationnel(21))
    return ok
```

```
def test_unitaire_3(visible =False):
    print("*** expression: test_unitaire_3 ***")

    e = ex.expression("3")
    if visible: print(e)

    ok = (e.lire_valeur().lire_num().valuation() == ra.rationnel(3))
    return ok


def test_unitaire_4(visible =False):
    print("*** expression: test_unitaire_4 ***")

    e = ex.expression("3 * ( 1/2 + 1 )")
    if visible: print(e)

    ok = e.est_valide()
    return ok


def test_unitaire_5(visible =False):
    print("*** expression: test_unitaire_5 ***")

    e = ex.expression("a + b")
    if visible: print(e)

    ok = e.est_valide()
    return ok


def test_unitaire_6(visible =False):
    print("*** expression: test_unitaire_6 ***")

    e = ex.expression("(a + b)/(3-1)")
    if visible: print(e)

    ok = e.est_valide()
    return ok


def test_unitaire_7(visible =False):
    print("*** expression: test_unitaire_7 ***")

    e = ex.expression("(a + b)/(x+x)")
    if visible: print(e)

    ok = e.est_valide()
    return ok


def test_unitaire_8(visible =False):
    print("*** expression: test_unitaire_8 ***")

    e = ex.expression("(a + b)/(x+1)")
    if visible: print(e)

    ok = e.est_valide()
    return ok
```



```
def test_unitaire_9(visible =False):
    print("*** expression: test_unitaire_9 ***")

    e = ex.expression("(a + b)/(x)")
    if visible: print(e)

    ok = e.est_valide()
    return ok


def test_unitaire_10(visible =False):
    print("*** expression: test_unitaire_10 ***")

    a = ex.expression("3 * ( x + 1 ) / (x - 20)")
    if visible: print("a =", a)

    b = ex.expression("x - 1")
    if visible: print("b =", b)

    x = a + b
    if visible: print("a + b = {}".format(x))

    x = a - b
    if visible: print("a - b = {}".format(x))

    x = a * b
    if visible: print("a * b = {}".format(x))

    x = a / b
    if visible: print("a / b = {}".format(x))

    a = ex.expression("x + 1")
    if visible: print("a =", a)

    n = ex.expression("4")
    if visible: print("n =", n)

    x = a ** n
    if visible: print("a ** n = {}".format(x))

    ok = True
    return ok


def test_unitaire_11(visible =False):
    print("*** expression: test_unitaire_11 ***")

    e = ex.expression("3 / ( (19 + 1) - 20 )")
    if visible:
        print("e =", e)
        e.afficher_erreur()

    ok = (not e.est_valide())
    return ok


def test_unitaire_12(visible =False):
    print("*** expression: test_unitaire_12 ***")

    e = ex.expression("(2+1)/(x*x*x*x*x+1)")
    if visible: print(e)

    ok = e.est_valide()
    return ok
```

```
def test_unitaire_13(visible =False):
    print("*** expression: test_unitaire_13 ***")

    ok = True
    return ok

def test_unitaire_14(visible =False):
    print("*** expression: test_unitaire_14 ***")

    ok = True
    return ok

def test_unitaire_15(visible =False):
    print("*** expression: test_unitaire_15 ***")

    ok = True
    return ok

def test_unitaire_16(visible =False):
    print("*** expression: test_unitaire_16 ***")

    ok = True
    return ok

def test_unitaire_17(visible =False):
    print("*** expression: test_unitaire_17 ***")

    ok = True
    return ok

def test_unitaire_18(visible =False):
    print("*** expression: test_unitaire_18 ***")

    ok = True
    return ok

def test_unitaire_19(visible =False):
    print("*** expression: test_unitaire_19 ***")

    ok = True
    return ok

def test_unitaire_(visible =False):
    print("*** expression: test_unitaire_ ***")

    ok = True
    return ok

def tests_unitaires():
    return (
```

```
test_unitaire_0() and \
test_unitaire_1() and \
test_unitaire_2() and \
test_unitaire_3() and \
test_unitaire_4() and \
test_unitaire_5(True) and \
test_unitaire_6(True) and \
test_unitaire_7(True) and \
test_unitaire_8(True) and \
test_unitaire_9(True) and \
test_unitaire_10(True) and \
test_unitaire_11(True) and \
test_unitaire_12(True) and \
test_unitaire_13() and \
test_unitaire_14() and \
test_unitaire_15() and \
test_unitaire_16() and \
test_unitaire_17() and \
test_unitaire_18() and \
test_unitaire_19()
)

if __name__ == "__main__":
    ok = tests_unitaires()
    if ok:
        print("*** expression: tests unitaires OK ***")
```

