```
# -*-coding: UTF-8 -*-

## Data: 20-04-2011

from abc import ABCMeta, abstractmethod

class Expressao(object):
    ''' Classe abstrata que tem a define uma expressao comum. '''

    __metaclass__ = ABCMeta

    def __ne__(self, other):
        ''' Sobrecarga do operador != para que expressoes possam ser
            devidamente comparadas '''
        return self.__str__() != other.__str__() or self.eval() != other.eval()

    def __eq__(self, other):
        ''' Sobrecarga do operador == para que expressoes possam ser
            devidamente comparadas '''
        return self.__str__() == other.__str__() and self.eval() == other.eval()

    @abstractmethod
    def eval(self):
        ''' Avalia e retorna o valor logico da expressao '''
        pass

class Unaria(Expressao):
    ''' Classe que representa uma operacao unaria '''
    __metaclass__ = ABCMeta

    def __init__(self, exp):
        ''' exp :: Expression '''
        self.exp = exp

class Binaria(Expressao):
    ''' Classe que representa uma expressao binaria '''
    __metaclass__ = ABCMeta

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        self.exp1 = exp1
        self.exp2 = exp2

class Variavel(Expressao):
    ''' Classe que representa uma variavel unaria logica que armazena um
    identificador, que no mundo das pessoas comuns e "trabalhadeiras" chamamos
    de "nome", e um valor logico muito simpatico. '''

    def __init__(self, nome, valor):
        ''' nome :: String, valor :: Boolean '''
        self.valor = valor
        self.nome = nome
```

```
    def __str__(self):
        return self.nome

    def eval(self):
        ''' -> Boolean '''
        return self.valor

class NOT(Unaria):
    ''' Classe que representa uma operacao NOT, que inverte o valor logico de
    uma sentenca ou variavel logica. '''

    def __init__(self, exp):
        ''' exp :: Expression '''
        Unaria.__init__(self, exp)

    def __str__(self):
        if str(self.exp).startswith("!"):
            return str(self.exp)[1:]
        else:
            return "!" + str(self.exp)

    def eval(self):
        ''' -> Boolean '''
        return not self.exp.eval()

class AND(Binaria):
    ''' Classe que representa o dono do Woody e do Buzz Lightyear. '''

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        Binaria.__init__(self, exp1, exp2)

    def __str__(self):
        return "(" + str(self.exp1) + " & " + str(self.exp2) + ")"

    def eval(self):
        ''' -> Boolean '''
        p = self.exp1.eval()
        q = self.exp2.eval()
        return p and q

class OR(Binaria):
    ''' Classe que representa a forma de falar "ou" em Minas Gerais. '''

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        Binaria.__init__(self, exp1, exp2)

    def __str__(self):
        return "(" + str(self.exp1) + " | " + str(self.exp2) + ")"

    def eval(self):
        ''' -> Boolean '''
```

```
        p = self.exp1.eval()
        q = self.exp2.eval()
        return p or q

class XOR(Binaria):
    ''' Classe que presenta a forma de falar "xo" em Minas Gerais '''

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        Binaria.__init__(self, exp1, exp2)

    def __str__(self):
        return "(" + str(self.exp1) + " * " + str(self.exp2) + ")"

    def eval(self):
        ''' -> Boolean '''
        p = self.exp1.eval()
        q = self.exp2.eval()
        return p ^ q

class IMPLIES(Binaria):
    ''' Classe que representa seu irmao IMPLICANTE... Entendeu? Implica!
    Implicante! hahahaha '''

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        Binaria.__init__(self, exp1, exp2)

    def __str__(self):
        return "(" + str(self.exp1) + " -> " + str(self.exp2) + ")"

    def eval(self):
        ''' -> Boolean '''
        p = self.exp1.eval()
        q = self.exp2.eval()

        if p == True and q == False:
            return False
        else:
            return True

class BIIMPLIES(Binaria):
    ''' Classe que representa uma expressao de bi-implicacao. '''

    def __init__(self, exp1, exp2):
        ''' exp1, exp2 :: Expression '''
        Binaria.__init__(self, exp1, exp2)

    def __str__(self):
        return "(" + str(self.exp1) + " <-> " + str(self.exp2) + ")"

    def eval(self):
        ''' -> Boolean '''
```

```
        p = self.exp1.eval()
        q = self.exp2.eval()

        if (p and q) or (not p and not q):
            return True
        else:
            return False
```