```
'''
    Matrix.py

    Matrix class - Calculates the Minimum Edit Distance and generates the MED
    Matrix, Pointer Matrix, and Edit String Alignment.
'''

from .Util import BTMATRIX_SYMBOLS, CellFormat
from .Cell import Cell
import os

class Matrix(object):
    '''
    Minimum Edit Distance Matrix Calculator
    '''

    def __init__(self, source_word, dest_word):
        '''
        Create a Minimum Edit Distance Matrix.
            source_word - word to edit from
            dest_word - word to edit to
        '''
        self._source_word = source_word
        self._dest_word = dest_word
        self._edit_string = str()

        self._matrix_width = len(dest_word) + 1
        self._matrix_height = len(source_word) + 1

        # The matrix is stored as a list of lists.
        # It is a list or rows, so the first coordinate
        # is the row number and the second coordinate is
        # the column number, resulting in the coordinate pair
        # of (y, x).  Noting this because it is counter to
        # what is expected.
        self._matrix = list()

        # Now do the MED calculation and backtrace
        self._doMEDCalculation()
        self._generateEditString()


    def _doMEDCalculation(self):
        '''
        Initialize the matrix and do the Minimum Edit Distance Calculation
        For internal use only - Called by Matrix.__init__()
        '''
        # Initialize matrix with cells
        for row in range(0, self._matrix_height):
            self._matrix.append(list())      # Add a new row
            for col in range(0, self._matrix_width):
                self._matrix[row].append(Cell())

                # If the current cell is the origin or on the top or left
                # boarder, initialize the value.
                if row == 0:    #initialize the first row
                    self._matrix[row][col].distance = col
                elif col == 0:    # initialize the first column
                    self._matrix[row][col].distance = row

                # If not a boarder cell, calculate the cell value
                else:
                    upvalue = self._matrix[row-1][col].distance + 1
                    leftvalue = self._matrix[row][col-1].distance + 1
                    diagvalue = self._matrix[row-1][col-1].distance

                    # Check to see if diagonal value is a substitution or not
                    # If it is a substitution, add 2 to diagonal value
                    if self._source_word[row-1] == self._dest_word[col-1]:
                        self._matrix[row][col].equivalent = True
                    else:
                        diagvalue += 2

                    # Calculate the minimum value and assign the distance to
                    # the cell
                    minvalue = min(upvalue, leftvalue, diagvalue)
                    self._matrix[row][col].distance = minvalue

                    # Now set the backtrace flags for the cell
                    if minvalue == upvalue:
                        self._matrix[row][col].up = True
                    if minvalue == leftvalue:
                        self._matrix[row][col].left = True
                    if minvalue == diagvalue:
                        self._matrix[row][col].diag = True


    def _generateEditString(self):
        '''
        Preform the backtrace calculation and store the edit string
        For internal use only - Called by Matrix.__init__()
        '''
        cell_row = self._matrix_height - 1
        cell_col = self._matrix_width - 1
        edit_string = str()
        current_cell = self._matrix[cell_row][cell_col]

        while (cell_row, cell_col) != (0, 0):
            if current_cell.diag:
                if current_cell.equivalent:
                    edit_string += " "
                else:
                    edit_string += "s"
                cell_row -= 1
                cell_col -= 1
            elif current_cell.up:
                edit_string += "d"
                cell_row -= 1
            elif current_cell.left:
                edit_string += "i"
                cell_col -= 1
            else:
                if cell_row == 0:
                    edit_string += "i"
                    cell_col -= 1
                elif cell_col == 0:
                    edit_string += "d"
                    cell_row -= 1
            current_cell = self._matrix[cell_row][cell_col]

        # Now reverse the string generated during the backtrace and store it
        # String reversed using annoying python list extended slice syntax
        self._edit_string = edit_string[::-1]

    def _genFormattedMatrix(self, fun):
        '''
        Generate a formatted matrix based upon the return of fun, which is a
        lambda.  For internal use only.
        returns a string
        '''
        # Note: As the formatted matrix looks like this:
        #   #   #   B   R   I   E   F
        #   #   -   -   -   -   -   -
        #   D   -   -   -   -   -   -
        #   R   -   -   -   -   -   -
        #   I   -   -   -   -   -   -
        #   V   -   -   -   -   -   -
        #   E   -   -   -   -   -   -
        #
        # We have to play with off-by-one coordinates into the distance matrix.
        # Also, as the matrix does not start printing the characters of the
        # words until the 3rd row or column, word accesses are i-2
        #
        # Also, using lambdas!  Learning Erlang was actually good for
        # something!

        return_string = str()
        for row in range(0, self._matrix_height + 1):
            for col in range(0, self._matrix_width + 1):
                # Print the # symbol to start the matrix
                if (row, col) == (0, 0) or (row, col) == (0, 1) or (row, col) == (1,
    0):
                    return_string += CellFormat('#')
                elif row == 0:
                    return_string += CellFormat(self._dest_word[col - 2])
                elif col == 0:
                    return_string += CellFormat(self._source_word[row - 2])
                else:
                    return_string += CellFormat(fun(row-1, col-1))
            if row != self._matrix_height:
                return_string += os.linesep

        return return_string

    @property
    def MED(self):
        '''
        Returns the calculated Minimum Edit Distance
        '''
        return self._matrix[self._matrix_height-1][self._matrix_width-1].distance


    def getDistMatrixFormatted(self):
        '''
        Returns the Distance Matrix as a formatted string for printing
        '''
        return self._genFormattedMatrix(lambda y, x:
                self._matrix[y][x].distance)


    def getBTMatrixFormatted(self):
        '''
        Returns the Backtrace Matrix as a formatted string for printing
        '''
        return self._genFormattedMatrix(lambda y, x:
                BTMATRIX_SYMBOLS[self._matrix[y][x].backtrace_flags])


    def getStringAlignmentFormatted(self):
        '''
        Returns the String alignment as a formatted string
        '''
        source_word_counter = dest_word_counter = 0
        aligned_source_word = str()
        aligned_dest_word = str()
        bar_string = "|" * len(self._edit_string)

        for op in self._edit_string:
            if op == "d":
                aligned_source_word += self._source_word[source_word_counter]
                source_word_counter += 1
                aligned_dest_word += "*"
            elif op == "i":
                aligned_source_word += "*"
                aligned_dest_word += self._dest_word[dest_word_counter]
                dest_word_counter += 1
            else:
                aligned_source_word += self._source_word[source_word_counter]
                source_word_counter += 1
                aligned_dest_word += self._dest_word[dest_word_counter]
                dest_word_counter += 1

        return (aligned_source_word + os.linesep
                + bar_string + os.linesep
                + aligned_dest_word + os.linesep
                + self._edit_string)
```