```python
class CaesarCipher:
    """Class for doing encryption and decryption using a Caesar cipher."""

    def __init__(self, shift):
        """Construct Caesar cipher using given integer shift for rotation."""
        encoder = [None] * 26  # temp array for encryption
        decoder = [None] * 26  # temp array for decryption
        for k in range(26):
            encoder[k] = chr((k + shift) % 26 + ord('A'))
            decoder[k] = chr((k - shift) % 26 + ord('A'))
        self._forward = ''.join(encoder)  # will store as string
        self._backward = ''.join(decoder)  # since fixed

    def encrypt(self, message):
        """Return string representing encrypted message."""
        return self._transform(message, self._forward)

    def decrypt(self, secret):
        """Return decrypted message given encrypted secret."""
        return self._transform(secret, self._backward)

    def _transform(self, original, code):
        """Utility to perform transformation based on given code string."""
        msg = list(original)
        for k in range(len(msg)):
            if msg[k].isupper():
                j = ord(msg[k]) - ord('A')  # index from 0 to 25
                msg[k] = code[j]  # replace this character
        return ''.join(msg)


if __name__ == '__main__':
    cipher = CaesarCipher(3)
    message = "THE EAGLE IS IN PLAY; MEET AT JOE'S."
    coded = cipher.encrypt(message)
    print('Secret: ', coded)
    answer = cipher.decrypt(coded)
    print('Message:', answer)

    assert coded == "WKH HDJOH LV LQ SODB; PHHW DW MRH'V."
    assert answer == "THE EAGLE IS IN PLAY; MEET AT JOE'S."
```

```python
def draw_line(tick_length, tick_label=''):
    """Draw one line with given tick length (followed by optional label)."""
    line = '-' * tick_length
    if tick_label:
        line += ' ' + tick_label
    print(line)


def draw_interval(center_length):
    """Draw tick interval based upon a central tick length."""
    if center_length > 0:  # stop when length drops to 0
        draw_interval(center_length - 1)  # recursively draw top ticks
        draw_line(center_length)  # draw center tick
        draw_interval(center_length - 1)  # recursively draw bottom ticks


def draw_ruler(num_inches, major_length):
    """Draw English ruler with given number of inches and major tick length."""
    draw_line(major_length, '0')  # draw inch 0 line
    for j in range(1, 1 + num_inches):
        draw_interval(major_length - 1)  # draw interior ticks for inch
        draw_line(major_length, str(j))  # draw inch j line and label


if __name__ == '__main__':
    draw_ruler(2, 4)
    print('=' * 30)
    draw_ruler(1, 5)
    print('=' * 30)
    draw_ruler(3, 3)
```

```python
def is_matched_html(raw):
    """Return True if all HTML tags are properly match; False otherwise."""
    S = ArrayStack()
    j = raw.find('<')  # find first '<' character (if any)
    while j != -1:
        k = raw.find('>', j + 1)  # find next '>' character
        if k == -1:
            return False  # invalid tag
        tag = raw[j + 1:k]  # strip away < >
        if not tag.startswith('/'):  # this is opening tag
            S.push(tag)
        else:  # this is closing tag
            if S.is_empty():
                return False  # nothing to match with
            if tag[1:] != S.pop():
                return False  # mismatched delimiter
        j = raw.find('<', k + 1)  # find next '<' character (if any)
    return S.is_empty()  # were all opening tags matched?
```

```python
class TicTacToe:
    """Management of a Tic-Tac-Toe game (does not do strategy)."""

    def __init__(self):
        """Start a new game."""
        self._board = [[' '] * 3 for j in range(3)]
        self._player = 'X'

    def mark(self, i, j):
        """Put an X or O mark at position (i,j) for next player's turn."""
        if not (0 <= i <= 2 and 0 <= j <= 2):
            raise ValueError('Invalid board position')
        if self._board[i][j] != ' ':
            raise ValueError('Board position occupied')
        if self.winner() is not None:
            raise ValueError('Game is already complete')
        self._board[i][j] = self._player
        if self._player == 'X':
            self._player = 'O'
        else:
            self._player = 'X'

    def _is_win(self, mark):
        """Check whether the board configuration is a win for the given player."""
        board = self._board  # local variable for shorthand
        return (mark == board[0][0] == board[0][1] == board[0][2] or   # row 0
                mark == board[1][0] == board[1][1] == board[1][2] or   # row 1
                mark == board[2][0] == board[2][1] == board[2][2] or   # row 2
                mark == board[0][0] == board[1][0] == board[2][0] or   # column 0
                mark == board[0][1] == board[1][1] == board[2][1] or   # column 1
                mark == board[0][2] == board[1][2] == board[2][2] or   # column 2
                mark == board[0][0] == board[1][1] == board[2][2] or   # diagonal
                mark == board[0][2] == board[1][1] == board[2][0])  # rev diag

    def winner(self):
        """Return mark of winning player, or None to indicate a tie."""
        for mark in 'XO':
            if self._is_win(mark):
                return mark
        return None

    def __str__(self):
        """Return string representation of current game board."""
        rows = ['|'.join(self._board[r]) for r in range(3)]
        return '\n-----\n'.join(rows)


if __name__ == '__main__':
    game = TicTacToe()
    # X moves:            # O moves:
    game.mark(1, 1)
    game.mark(0, 2)
    game.mark(2, 2)
    game.mark(0, 0)
    game.mark(0, 1)
    game.mark(2, 1)
    game.mark(1, 2)
    game.mark(1, 0)
    game.mark(2, 0)

    print(game)
    winner = game.winner()
    if winner is None:
        print('Tie')
    else:
        print(winner, 'wins')
```

```python
def parenthesize(T, p):
    """Print parenthesized representation of subtree of T rooted at p."""
    print(p.element(), end='')  # use of end avoids trailing newline
    if not T.is_leaf(p):
        first_time = True
        for c in T.children(p):
            sep = ' (' if first_time else ', '  # determine proper separator
            print(sep, end='')
            first_time = False  # any future passes will not be the first
            parenthesize(T, c)  # recur on child
        print(')', end='')  # include closing parenthesis
```

```python
"""Tests if a string of nested braces, parentheses, and brackets are all
matched and nested correctly.
"""


def is_matched(expr):
    """Return True if all delimiters are properly match; False otherwise."""
    lefty = '({['  # opening delimiters
    righty = ')}]'  # respective closing delims
    S = ArrayStack()
    for c in expr:
        if c in lefty:
            S.push(c)  # push left delimiter on stack
        elif c in righty:
            if S.is_empty():
                return False  # nothing to match with
            if righty.index(c) != lefty.index(S.pop()):
                return False  # mismatched
    return S.is_empty()  # were all symbols matched?
```