```python
# Sudoku

import itertools
import re
from functools import reduce

from .csp import CSP

def flatten(seqs):
    """flatten(seqs)
    Flattens objects in
    """
    return sum(seqs, [])


easy1 = '..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9..5.1.3..'
harder1 = '4173698.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'



def different_values_constraint(_A, a, _B, b):
    """A constraint saying two neighboring variables must differ in value."""
    return a != b



class Sudoku(CSP):
    """A Sudoku problem.
    The box grid is a 3x3 array of boxes, each a 3x3 array of cells.
    Each cell holds a digit in 1..9. In each box, all digits are
    different; the same for each row and column as a 9x9 grid.
    >>> e = Sudoku(easy1)

    Method infer_assignment shows the puzzle with all of the variables
    that are currently assigned.  Since we haven't inferred anything,
    this shows the initial puzzle assignments that are given in the problem.
    >>> e.display(e.infer_assignment())
    . . 3 | . 2 . | 6 . .
    9 . . | 3 . 5 | . . 1
    . . 1 | 8 . 6 | 4 . .
    ------+-------+------
    . . 8 | 1 . 2 | 9 . .
    7 . . | . . . | . . 8
    . . 6 | 7 . 8 | 2 . .
    ------+-------+------
    . . 2 | 6 . 9 | 5 . .
    8 . . | 2 . 3 | . . 9
    . . 5 | . 1 . | 3 . .

    AC3 will mutate the state of the puzzle to reduce variable domains as
    much as possible by constraint propagation.
    We see that the easy puzzle is solved by AC3.
    >>> AC3(e); e.display(e.infer_assignment())
    True
    4 8 3 | 9 2 1 | 6 5 7
    9 6 7 | 3 4 5 | 8 2 1
    2 5 1 | 8 7 6 | 4 9 3
    ------+-------+------
    5 4 8 | 1 3 2 | 9 7 6
    7 2 9 | 5 6 4 | 1 3 8
    1 3 6 | 7 9 8 | 2 4 5
    ------+-------+------
    3 7 2 | 6 8 9 | 5 1 4
    8 1 4 | 2 5 3 | 7 6 9
    6 9 5 | 4 1 7 | 3 8 2

    We could test if it was solved using Soduko's parent class goal_test method
    s.goal_test(s.curr_domains)
    True

    This one is harder and AC3 does not help much at all:


    >>> h = Sudoku(harder1)
    Initial problem:
    4 1 7 | 3 6 9 | 8 . 5
    . 3 . | . . . | . . .
```

```
    . . . | 7 . . | . . .
    ------+-------+------
    . 2 . | . . . | . 6 .
    . . . | . 8 . | 4 . .
    . . . | . 1 . | . . .
    ------+-------+------
    . . . | 6 . 3 | . 7 .
    5 . . | 2 . . | . . .
    1 . 4 | . . . | . . .

    After AC3 constraint propagation

    4 1 7 | 3 6 9 | 8 2 5
    . 3 . | . . . | . . .
    . . . | 7 . . | . . .
    ------+-------+------
    . 2 . | . . . | . 6 .
    . . . | . 8 . | 4 . .
    . . . | . 1 . | . . .
    ------+-------+------
    . . . | 6 . 3 | . 7 .
    5 . . | 2 . . | . . .
    1 . 4 | . . . | . . .

    To solve this, we need to use backtracking_search which also mutates
    the object given to it.
    >>> solved = backtracking_search(h, select_unassigned_variable=mrv,
            inference=forward_checking) is not None
    If solved is True, the puzzle can be displayed with as above.
    """


    R3 = list(range( )) # All Sudoku puzzles use 3x3 grids, one side

    # Generate board of fixed size 3x3 sets of 3x3 boxes
    # Use Cell to generate integers for each box (variables are numbers)
    Cell = itertools.count().__next__


    def __init__(self, grid):
        """Build a Sudoku problem from a string representing the grid:
        the digits 1-9 denote a filled cell, '.' or '0' an empty one;
        other characters are ignored."""


        # Build a grid of variables. Variables are numbered
        # and the grid is 4 dimensional.
        # Grid looks like the following:
        #    00 01 02 | 09 10 11 | 18 19 20
        #    03 04 05 | 12 13 14 | 21 22 23
        #    06 07 08 | 15 16 17 | 24 25 26
        #    -------------------------------
        #    27 28 29 | 36 ...   | 45 ...
        #    30 31 32 |
        #    33 34 35 |
        #    -------------------------------
        #    54 55 56 | 63 64 65 | 72 73 74
        #    57 58 59 | 66 67 68 | 75 76 77
        #    60 61 62 | 69 70 71 | 78 79 80
        #
        #  self.bgrid[i][j] is a double list for a box.
        #  In the above variable set, the bottom right
        #  is self.bgrid[2][2]
        #     [[72, 73, 74], [75, 76, 77], [78, 79, 80]]
        #  The final two dimensions are the row and column
        #  within the box.  self.bgrid[2][2][0][1] = 73
        self.bgrid = [[
                       # one box
                       [[self.Cell() for _x in self.R3] for _y in self.R3]
                       # series of boxes bx, by
                       for _bx in self.R3
                      ]
                      for _by in self.R3
                     ]
        # list of variables in each box, self.boxes[0] = [0, 1, ... 8]
        self.boxes = flatten([list(map(flatten, brow)) for brow in self.bgrid])
        # list of variables in each row
```

```python
        # self.rows[0] = [0, 1, 2, 9, 10, 11, 18, 19, 20]
        self.rows = flatten([list(map(flatten, zip(*brow))) for brow in self.bgrid])
        # list of variables in each column
        self.cols = list(zip(*self.rows))

        # Build the neighbors list
        # It should be implemented as a dictionary.
        # Keys are the variables names (numbers) and values are a set
        # Each variable should have a set associated with it containing
        #    all of the variables that have constraints.  As an example,
        #    if variable 100 had constraints between itself and variables
        #    103 and 104, self.neighbors[100] would contain a set with members
        #    103, and 104.
        #
        # See Python library reference if you are not familiar with sets
        # Tutorial:  https://www.learnpython.org/en/Sets

        # Build dictionary of list of variables
        self.neighbors = {v: set() for v in flatten(self.rows)}
        # Populate with all variables that are neighbors of the
        # unit.
        for unit in map(set, self.boxes + self.rows + self.cols):
            for v in unit:
                self.neighbors[v].update(unit - {v})

        squares = iter(re.findall(r'\d|\.', grid))
        domains = {var: [ch] if ch in '123456789' else '123456789'
                   for var, ch in zip(flatten(self.rows), squares)}
        for _ in squares:
            raise ValueError("Not a Sudoku grid", grid)  # Too many squares
        CSP.__init__(self, None, domains, self.neighbors, different_values_constraint)

        self.support_pruning()

    def display(self, assignment):
        def show_box(box): return [' '.join(map(show_cell, row)) for row in box]

        def show_cell(cell): return str(assignment.get(cell, '.'))

        def abut(lines1, lines2): return list(
            map(' | '.join, list(zip(lines1, lines2))))
        print('\n------+-------+------\n'.join(
            '\n'.join(reduce(
                abut, map(show_box, brow))) for brow in self.bgrid))
```