Assignment Description: Implement a brute force (backtracking) SudokuSolver (50 points) Sample code has been provided that implements a structure of a Sudoku Solver in Python. The student is not required to use any of this code, but what must remain is that the class must include a *solve()* function, and a *loader(board)* function. The solve function should return True or False OR optionally return the solution (if solvable) and None (if not). Finally, the loader function should accept a 2D Numpy Array containing a representation of a Sudoku puzzle. The provided *test_sudoku_solve()* function can be used to ensure that your solver returns the proper values. It does NOT test that your solution is CORRECT. It is up to the student to ensure that their solver produces a correct answer to the given puzzle.

Submission Instructions Please submit a single python file (.py) with a class defined as described blow. Please name the file:

`SudokuSolver.py`

The starter code contains some useful code for instantiating a board, loading previously designed board, generating a sample board, and for printing the solution to a board. These can be modified if desirable but it not required. The following functions are **required** to be implemented by the student. They are implemented in the sample code as a *stub* including only a single pass statement. Note that **can_place** has been implemented with a suggested signature of *self, board, row, col, value* but this is not a **required** signature. The **_solve(board)** function should use can_place as desired. I have left my saple code in as a suggestion only.

```
class SudokuSolver:
    """ A Python implementation of a SudokoSolver, uses a backtracking method to solve a Sudoku Puzzle
    with bruteforce.  For each empty cell, the algorithm will attempt to place a digit and them move to
    the next cell.  If a dead cell is found (no digits can be placed). The previously placed digit will
    be changed."""

    def solve(self):
        """
        Uses the board's base as the initial board to solve recursively.
        :return: True if the board is solved, false otherwise.
        """

    def _solve(self, board):
        """
        Recursive Function to solve a provided board.
        :param board: The current iteration of the board in order to solve.
        :return: True if the board is completed/solved, False otherwise.
        """

    def can_place(self, board, row, col, value):
        """
        Accepts a current board, and verifies if the provided value can be placed in the indicated row a

        :param board: The board to evaluate.  This is generally a partially complete board.
        :param row: The row of which the digit is being placed.
        :param col: The column of which the digit is being placed.
        :param value: The digit being placed in the cell.
        :return: False if there exists a matching digit in the same row, column or section.  True otherv
        """
```